WISH: Interactive Host Administration Using Distributed Programming Within the Shell

Jude Nelson, Nick Jones, and Arman Suleimenov
Department of Computer Science
Princeton University
{jcnelson,najones,asuleime}@cs.princeton.edu

Abstract

This paper presents the wide area interactive shell (WISH), a novel wide area network service and toolkit that gives users the ability to interactively control large numbers of hosts via a single UNIX shell. WISH provides a global process, file, and environment variable namespace in which to spawn, join, signal, communicate, and synchronize with remote processes. Users harness these features to carry out complex distributed tasks which are difficult with normal remote shells. The main contribution of WISH is that it empowers users to control multiple hosts by extending familiar shell programming concepts into the wide area.

We motivate the need for WISH using our experience with managing hosts in PlanetLab, where some problems we faced could not be easily solved using remote shells or with existing scalable management solutions. From this experience, we describe the desired set of capabilities WISH should expose to users. We then automate multiple host management functions using WISH shell scripts in order to demonstrate the new paradigms WISH offers to users.

1 Introduction

In this paper we present our design of the wide area interactive shell (WISH), a distributed computing service which provides the ability to scalably execute shell commands across wide area networks. WISH is comprised of a daemon and a set of command-line tools which allow any standard UNIX shell to spawn and manage *jobs*[1] across a large set of hosts.

WISH is targeted at individuals who need more powerful job controls than those which are available in traditional remote shells such as SSH. For example, with WISH a user can quickly set up a publish-subscribe system by spawning a process on each host. This process will periodically query a global environment variable and execute commands based upon its value. As another example, a user can quickly distribute large datasets to hosts by scripting a subset of hosts to download the dataset from its origin, and then command disjoint subsets of hosts to download the dataset from peers that already have it. As a third example, a user can immediately identify a subset of hosts with WISH that are available to execute a computationally-intensive task, and then use their UNIX shell of choice to issue the task to them.

While existing remote shells such as SSH [1] may be leveraged to execute the above examples, the shell user must implement the inter-job and inter-host communication infrastructure necessary to ensure correct execution. WISH's contribution is that it provides this necessary infrastructure by creating a network-wide shell environment and by exposing synchronization primitives for a user's job to leverage. These allow a user to quickly write and run jobs in a familiar environment across many hosts, and empower users to leverage their knowledge of distributed programming techniques.

In the remainder of this paper, we describe the motivation for designing WISH based upon our experience with PlanetLab [11] slice administration. We

[1] We use the term "job" to refer to a sequence of one or more commands that a user submits to a conventional UNIX shell. A job executes in one process initially, but may fork others.

then present our design of the WISH system and compare it to existing parallel execution environments. We conclude with case studies and a discussion of related and future work.

2 Motivation

More often than not, distributed systems deployed on PlanetLab run in a *slice*, a collection of lightweight virtual machines spread across an equal number of nodes [11]. Within a slice, a single VM, or *sliver*, is subject to naturally occurring host faults (including complete VM re-instantiation), network faults, and resource scarcities, all of which can lead to unexpected system behavior. To counter these faults, developers devise means to monitor the health of their systems and to perform diagnostic and administrative tasks on their misbehaving slivers. These tasks often include reinstalling, reconfiguring, and upgrading their software [2]. While there are existing systems such as CoMon [14], SWORD [9], Stork [10], and Puppet [3] that can handle these tasks, they each present an additional deployment challenge to a PlanetLab user who is unfamiliar with them and who only needs a small subset of their functionality.

In the process of designing WISH, we considered what expectations we could reasonably have of the typical PlanetLab user. By their very nature, PlanetLab users should be familiar with remote shells and with UNIX shell programming. Since they use PlanetLab to deploy and test experimental distributed software, they can also be expected to have varying degrees of familiarity with distributed programming paradigms and design patterns. Since the default means of contacting and administrating PlanetLab slivers is via SSH [11], we are motivated to create a tool that lets a user extend familiar shell concepts such as job management, I/O redirection, and environment variables to operate on a collection of hosts instead of a single host.

The challenges PlanetLab users face in managing their slices are not specific to PlanetLab's architecture. Administering collections of hosts in both clouds and wide-area networks requires the ability to monitor the hosts' states and the ability to issue commands to hosts efficiently. While there are many tools that provide one or both abilities, none to our knowledge leverage both a user's shell programming and distributed computing experience as the primary means of performing administrative functions.

3 Design

We designed WISH to not hide the fact that there are multiple hosts within the WISH federation. There are existing systems such as Plush [8] and pssh [6] that provide an interface for interactively issuing shell commands to many hosts, but they are limited in their expressiveness because they issue commands synchronously, in lock-step. This approach simplifies shell programming by treating a collection of hosts as a single host, but it requires the user to issue the hosts the same commands. This is not always desirable.

A more flexible approach is to treat each host as a receptacle in which a user can place processes and data, so users can orchestrate host behavior however they desire. This can be partially achieved by iterating over a list of hosts and issuing the hosts commands in parallel via a remote shell. However, this solution offers no easy way for the processes to share data and offers no easy way for the user to control them once they start.

Given this approach, we designed WISH to provide users with the following capabilities.

Process Control: A user can spawn, signal, join, and synchronize with processes on any host within their WISH instance.

I/O Redirection: A user can redirect process input and output between any hosts in a WISH instance.

Resource Monitoring: A user can monitor the resource and network utilizations of hosts.

Environment Variables: A user can get, set, and atomically test-and-set globally-visible shell environment variables.

File Access: A user can globally expose files to all other hosts within the same WISH instance. Remote hosts can download files which are visible to it.

Below, we describe how WISH meets these requirements.

3.1 Overview

WISH uses a federated architecture for communication between its member hosts. There is no federa-

tion leader, but each member of the federation implements a *global shell environment* to coordinate execution and communication between the jobs it has been instructed to manage. The global shell environment exposes environment variables, processes, and files that are visible to jobs originating from the same daemon, wherever they are executing. In practice, each host runs a WISH daemon to participate in the federation, and its daemon assumes responsibility for processes spawned by local users.

To create a federation, a user distributes the WISH daemons and client programs to his/her hosts out-of-band, and generates a list of hostnames and port numbers for each daemon to use to contact the other daemons. In future work, we intend to allow hosts to authenticate with and securely join existing federations and to learn of other joining daemons via a distributed gossip protocol.

3.2 Client and Commands

Users do not directly interact with the WISH daemon. Instead, WISH provides shell commands and a shell client which communicate with the local WISH daemon via the loopback network interface. The WISH shell client is a program that runs as a child of a UNIX shell process that receives and prints output from WISH processes to the user's TTY. We plan to make the client configurable such that data from different remote hosts may be color-coded (if the TTY supports it), prefixed by the hostname or a host alias, and selectively filtered or redirected as the user sees fit.

3.3 Peers and Heartbeats

The WISH daemon attempts to monitor the health of all other hosts within the federation. When a host joins the WISH federation, its daemon attempts to connect to as many other daemons as possible in order to send and receive "heartbeat" packets. The heartbeat packets contain information about a host's resource utilization. Once it connects to its peers within the federation, it periodically re-sends heartbeat packets to them, and keeps a bounded log of the last heartbeats it has received from each other host connected to it. The heartbeat information may be queried by the user to select hosts with desirable resource availabilities. The information maintained by each daemon includes a remote host's average CPU load in the past 5, 10, and 15 minutes, the amount of free RAM and disk space, and the average round-trip times of the previously-received heartbeat packets from each other host.

WISH provides a command called `nget` to determine the host with the ith most free RAM, the ith most free disk space, the ith lowest CPU load in the past 5, 10, and 15 minutes, or the ith lowest network latency (for all $0 < i < N$, the number of hosts). This is useful for when the user does not care which specific hosts are used to process a job, but only needs a subset with available capacity. Additionally, the `nget` program lets a user query the total number of hosts in the federation, so the user can iterate through the hosts in the order specified.

3.4 Remote Files

Since WISH jobs run across multiple hosts, a user may need to distribute data between them. To facilitate this, WISH provides an embedded HTTP(S) server to serve files from a predetermined document root. As a security measure, it maintains a blacklist of directories within the document root from which no files may be served. Users can blacklist and unblacklist local files and directories with the `fhide` and `fshow` commands, respectively.

3.5 Process Management

When a user wishes to create a process to run a job, the local WISH daemon starts a *global process* by registering the job with its global shell environment and assigning it a globally-unique [2] GPID (global process identifier). It then contacts the user-indicated host in the federation on which to run the job and forwards the job's commands to that remote daemon. The remote daemon performs some bookkeeping, forks a child process to run the job, and acknowledges the local daemon's job execution request. When the child process terminates, the remote daemon forwards the exit code and last signal received of the child process back to the local daemon, completing the global process's life. We refer to the set of actions the local daemon takes to manage a process as *spawning* the process, and the set of actions taken by the remote daemon as *executing* the process.

The remote child process must know how to contact the local daemon to participate in its shell envi-

[2] In practice, a GPID is either a user-selected or randomly-generated 64-bit value, making it unique with high probability.

ronment. Part of the bookkeeping the remote daemon performs before forking the child includes setting a few variables in the child process's environment which tell the child the hostname and port number of the daemon that spawned it. This information will be preserved across future spawns performed by this child, so every descendent process of the child will access the global shell environment on the local daemon that spawned it (unless explicitly overridden by the user).

There are four commands to manage jobs in WISH. The `pspawn` command instructs the local daemon to spawn a process and execute it on a remote daemon of the user's choosing, and may either specify a list of shell commands to evaluate or a locally-hosted binary file to fetch, download, and then execute. The `psig` command routes signals from the local daemon to the remote daemon executing a process, which in turn signals the process itself. The `psync` command requests or acknowledges process barriers, and will not exit until all other processes named in the list of GPIDs passed to it perform the same barrier request. Finally, the `pjoin` command joins with an existing process by waiting until the local daemon has the process's exit code, and then retrieving and printing it out.

3.6 IO Redirection

When a WISH daemon executes a process, it directs its standard output and standard error files to local, temporary files. As the process executes the user's job, the daemon reads and sends back the data stored in the redirected output and error files to the daemon that spawned the process as quickly as it can. Once the data are received by the daemon that spawned the process, they are forwarded to the user's client, which prints them out as the user desires.

Sometimes, a user may need to redirect a remotely executing process's standard out and/or standard error. To do this, the user identifies a file for each stream in the `pspawn` command, which causes the daemon to redirect the process output accordingly. Also, a user may need to specify a file to serve as standard input for the remote process. To do this, the user identifies a file to `pspawn` to be used as standard input. If this file is visible in the local daemon's global shell environment, the remote daemon downloads the file to temporary storage and opens it as the process's standard input. It removes the file once the process terminates.

3.7 Environment Variables

A process running in WISH has access both to its local environment variables and the environment variables provided by its originating daemon's global shell environment. A user can get, set, and atomically test-and-set these global environment variables in the global shell environment via the `ggetenv`, `gsetenv`, and `taset` commands, respectively. These commands search their local environment for a WISH daemon's hostname and port number, and if present, contact the daemon to carry out the intended operation (if not present, they contact localhost at the port number in the local WISH configuration). This way, all descendents of a process started by the user's local daemon manipulate the same environment variable space as the user.

4 Evaluation

We present a preliminary evaluation of WISH by implementing two host management functions as case studies. Using these examples, we demonstrate many of WISH's unique features. We argue that these tasks are easier to perform using a standard UNIX shell in conjunction with WISH than with SSH alone.

The first case study is implemented in Figure 1. The script causes each host within a WISH federation to generate an x509 certificate signing request with `openssl`, and then determines which hosts failed to generate the certificate. The second case study is implemented in Figures 2 and 3. The script allows a user to alter the running state of a service on many hosts at once. Both functions may be implemented by means other than WISH, but WISH's capabilities make tasks like this easier to program.

4.1 Process Management

The difficulty in running `openssl` to generate x509 certificate signing requests *en masse* is that `openssl` requires user input to do so. In our solution, we put the user input into a local file called `/tmp/csr.dat`, and spawn a process on each host to invoke `openssl`, but redirect `/tmp/csr.dat` into `openssl` for standard input. We then join with each `openssl` process and report any hosts that failed to generate the certificate signing requests.

```
gpids=$(seq 1 $(nget -n))
keydir="/etc/pks/tls/private/"

for in in $gpids; do
  host=$(nget $i)

  pspawn -g $i -i /tmp/csr.dat \
        -o /dev/null                \
    -c "openssl req -new -key  \
    $keydir/localhost.key" $host
done

for i in $gpids; do
  if [[ $(pjoin $i) != 0]]; then
    echo "Key generation failed \
    on $(nget $i)"
  fi
done
```

Figure 1: OpenSSL CSR Generation

```
while true; do
  sleep 5
  val=$(ggetenv SYS_STATUS)

  if [[$val != "done"]]; then
    /etc/init.d/project $val
    ggetenv RC_$(hostname) $?
    psync $(seq 1 $(nget -n))
    gsetenv SYS_STATUS done
  fi
done
```

Figure 2: `mon.sh`: Process Watchdog

It is possible to use SSH and `expect` to issue the correct user inputs to `openssl`, but this requires the user to be familiar with `expect`. It is also possible to distribute `/tmp/csr.dat` to each host and then redirect the local copy to `openssl`'s standard input, but this introduces the requirement that the user (re)distribute `/tmp/csr.dat` if it changes. If the user ran `openssl` within SSH and wanted to know which hosts failed to execute it, the user would either need to wrap the invocation of `openssl` with a check on its exit code, or execute each invocation of `openssl` serially and check the exit code of SSH. With WISH, a user can instead run each invocation of `openssl` in parallel and not only capture its exit code, but block until all tasks have completed or failed.

```
for i in $(seq 1 $(nget -n)); do
  pspawn -g $i -f mon.sh $(nget $i)
done
```

Figure 3: Process Watchdog Launch Script

4.2 Global Environment Variables

Most Linux services are controlled via initscripts, which are used to start, stop, and restart them [4]. The difficulty of running many initscripts for the same service at once is verifying that each script performed as expected. Our WISH solution in Figure 3 deploys the long-running shell process called `mon.sh`, detailed in Figure 2, to address this problem. `mon.sh` periodically checks the global environment variable `SYS_STATUS` and invokes the operation stored as its value on the initscript `/etc/init.d/project`. It records the exit code of the initscript in a global environment variable prefixed with `RC_` and named after its hostname, waits for every other `mon.sh` instance to do the same, and then resets `SYS_STATUS` to "`done`".

It is possible to use SSH, Plush, or pssh to invoke an initscript across a set of nodes, but more difficult. The user must somehow make the initscript invocation report back its success or failure, and additionally have a means of knowing when all invocations have completed. WISH efficiently solves both problems by allowing the invocation to share status information with the user via global environment variables.

5 Related Work

WISH bears some similarity in function to batch-processing systems such as Condor [13], a distributed high-throughput batch computing system which schedules jobs in a non-interactive manner on a network-accessible computers. Like WISH, Condor provides an interface for an executing job to invoke I/O operations on the originating machine. However, the goal of WISH is to expose distributed job execution through a familiar shell environment in order to ease host administration, whereas the goal of Condor is to efficiently and opportunistically utilize computing resources across a grid for computationally-intensive workloads. Also, WISH does not concern itself with fault-tolerant job execution (including migrating and restarting failed jobs)

since shell commands can be host-specific and may not be idempotent or restartable.

WISH also bears some similarity in both function and usage to the class of programs containing Plush [8], vxargs [7], and pssh [6], as well as simple shell scripts that iterate through a list of hosts to remotely execute the same given command on each of them. While WISH and these programs provide a shell interface to control sets of hosts in both the local- and wide-area, WISH goes two steps further by exposing hosts as receptacles for processes and files and provides shell-level inter-process communication and synchronization primitives.

WISH partly fills the roles of monitoring and software deployment systems like Nagios [5], SmartFrog [12], and Puppet [3]. However, these systems implement their own specialized protocols and languages which a user must master before using them to their full potential. WISH, however, only requires knowledge of the UNIX shell and a primer on process communication and synchronization concepts.

6 Conclusions

In this paper, we have presented the design of our initial version of WISH. However, there is still significant future work remaining within the WISH project. The current version of WISH lacks per user access controls, so all hosts within the WISH federation have equal privileges. In a production ready version of WISH, we would like to provide full user level access controls when WISH is used in conjunction with a directory system such as LDAP. Additionally, we would like to provide more advanced tools for dealing with job output, so that users can more easily switch between streams of job output in real time. In the current version, all job output is either printed to the user's shell or sent to a file, but we'd like to support more varied forms of output.

Throughout this paper, we have described our design of the WISH system, and we have used our experience with PlanetLab as a motivating example. WISH introduces novel tools which allow users to use advanced parallel and distributed techniques within the UNIX shell of their choice. In this paper, we have described some initial uses for WISH, but we hope that many more will be found once the project is exposed to a wider audience.

References

[1] `http://www.openssh.com/`.

[2] `http://planet-lab.org/tools`.

[3] `http://www.puppetlabs.com/`.

[4] `http://www.linuxbase.org/`.

[5] `http://www.nagios.org/`.

[6] `http://code.google.com/p/parallel-ssh/`.

[7] `http://vxargs.sourceforge.net/`.

[8] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. Planetlab application management using plush. *SIGOPS Oper. Syst. Rev.*, 40:33–40, January 2006. ISSN 0163-5980.

[9] J. Albrecht, D. Oppenheimer, A. Vahdat, and D. A. Patterson. Design and implementation trade-offs for wide-area resource discovery. *ACM Trans. Internet Technol.*, 8:18:1–18:44, October 2008. ISSN 1533-5399.

[10] J. Cappos, S. Baker, J. Plichta, D. Nyugen, J. Hardies, M. Borgard, J. Johnston, and J. H. Hartman. Stork: package management for distributed vm environments. In *Proceedings of the 21st conference on Large Installation System Administration Conference*, pages 7:1–7:16, 2007. ISBN 978-1-59327-152-7.

[11] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33:3–12, July 2003. ISSN 0146-4833.

[12] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft. Smartfrog: Configuration and automatic ignition of distributed applications. In *In: HP Openview University Association Conference (HP OVUA*, pages 1–9, 2003.

[13] M. Litzkow, M. Livny, and M. Mutka. Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104 –111, jun 1988.

[14] K. Park and V. S. Pai. Comon: a mostly-scalable monitoring system for planetlab. *SIGOPS Oper. Syst. Rev.*, 40:65–74, January 2006. ISSN 0163-5980.