Abstract

This paper presents the concept of role-based abstraction and a distributed shell to help prototype it. Role-based abstraction is the idea that a user can interact with a set of computers of a similar role in the same way that they would interact with only one computer. The end goal is to abstract away the distributed nature of the system to the developer or administrator working on it. JDShell is the first direction to accomplish this. It runs user commands in parallel on all servers in a peer group and handles differences in environments and server failures. The performance of JDShell is then analyzed to show relatively low overhead and the ability to scale linearly.

1 Introduction

Past research in operating systems has largely been a discussion of abstraction. Many people argue, such as Edsger Dijkstra[1] and Andrew Birrell[2], that ease of use for the developer is paramount for operating system technology to become adopted. Others, such as Dawson Engler[3] and Brian Bershad[4], argue that abstraction causes too much performance overhead and loss of specialization. Both sides of the argument present compelling research to back up their claims but most of these discussions happened years ago before the rise of “cloud computing”.

“Cloud computing” is a term that has been thrown around a lot – this paper defines it as a set of services hosted on the internet built on an infrastructure composed of many commodity machines (often virtualized) that can scale out very inexpensively. The reason that cloud computing negates Engler and Bershad’s argument is how cheap hardware has gotten and how easy it is to achieve good performance by scaling out additional computers. In the mean time, the cost of specialized labor forces, such as system administrators and developers has been steadily growing. Therefore, paying a computer scientist to spend his time creating optimized low-level hardware drivers or debugging complex un-abstracted protocols is not worth it when you can just add on another machine or two to get the performance boost.

With the current state of distributed systems and how application infrastructures are today, it is time to address the abstraction question again. However the question has changed: we no longer need to answer whether or not abstraction is good, the increased cost of labor has already answered that. Now we need to ask whether we can do more and how much is too much. In this paper we will present a new layer of abstraction for distributed systems based on server role. We will present one tool that can work toward this goal, a distributed shell called JDShell. Then the paper will analyze the performance of JDShell to show that it is not entirely a piece of junk. Finally we will conclude with improvements and future work.

2 Role-Based Abstraction

In cloud computing it is very common to see N-tiered architectures, as seen in Figure 1, supporting websites that process millions of requests. In
this architecture multiple computers scale out to provide good performance and redundancy for one role.

**Figure 1:** A standard web architecture with multiple sets of machines fulfilling different roles such as "memory cache" and "application server"

Consider a website that has 100 application servers with MapReduce, 20 database servers that with mysql, and 40 front end web servers with apache. The 40 front end web servers should have the same software installed, same users, directory structure and should have only some differences in what’s on memory, or in the "tmp" directory for example. Each tier of the website must have consistent environments otherwise errors will arise and the code will be unnecessarily difficult to maintain. For the rest of the paper, we will refer these tiers as peer groups. So how can we make maintaining an infrastructure like this easier?

One solution that has already been proposed is through server administration scripts. Tools such as puppet and chef have become wildly popular because they enable users to install the same software on several computers at the same time. However the tools themselves can be very burdensome to bootstrap and set up initially. They work best when running server provisioning scripts on fresh new servers, but cause headaches when trying to do maintenance or server updates on existing servers. A better answer is to abstract away the fact that you are working on several servers at the operating system level.

The idea behind **role-based abstraction** is that a user can interact with a set of computers of a similar role in the same way that they would interact if there were only one computer. By abstracting away the distributed nature of the system, a user of these systems can modify each tier as an atomic unit to apply patches, change passwords or update settings. By applying the same commands and sharing resources between machines, we hope to achieve the following goals:

1. Reduced time administering multiple servers
2. Increased amounts of consistency between servers and
3. Reduced time spent debugging distributed applications.

**Figure 2:** Role-based Abstraction: several computers members of a role based peer group that is controlled through the operating system

In Figure 2, we can see how a peer group of computers is united and controlled within the operating system. This peer group has the same applications installed enabling the possibility of shared interfaces and shared resources. However, this type of abstraction is limited to machines with a similar environment. Consider trying to administer a database server as if it were a web server – that would be quite silly.

3 JDShell: A Distributed Shell

In order to test our theories that role-based abstraction will help increase the productivity of
users, we must build a prototype operating system that enables use of a shared interface and shared resources. Due to time constraints we cannot implement a full operating system and will have to address shared resources in a later paper. In this paper we focus on building an interface that allows users to interact with all of the nodes of a peer group at the same time.

**JDShell** is distributed shell that was written in Java in the form of a client-server application that replaces the default bash shell in Linux. Java was chosen as the development language (despite the fact that c++ integrates better with Linux) for two reasons. 1.) Java enables minimal effort in porting between different operating systems. With little work, this application could be written to support both Unix and Windows operating systems. 2.) Robust multithreaded server/socket programming in C++ is really hard and I didn’t think there was enough time for it. Java is better for prototyping because it has a large networking and IO library which can be used to expedite development time.

JDShell accepts commands from the user who is logged into any peer. Those commands will then be applied to all members of the peer group so that an administrator can save time by not logging in to each peer individually. JDShell knows who the other members of the peer group are based on a **peer list file** that lives in the operating system. For the success of this prototype we must assume that the shell is running on all members of the peer group and that there is a process in place that updates the peer list when servers are added or removed to the peer group. JDShell can also support individual machines that are not in a peer group by having their peer list consist of only the localhost address.

When a command is issued, a new thread is created for each member in the peer group which handles communication between that peer and the client. The command is then issued to all peers in parallel over the transport layer (TCP sockets). The **JDShellServer** on the peer accepts the command and creates a thread to process it. The server is multi-threaded so it can receive commands from multiple peers in parallel. Once the command has been processed, the results from each server are sent back to the originating user and aggregated. This process is shown in Figure 3.

![Architecture of JDShell](image)

**Figure 3:** Architecture of JDShell: Threads are created to facilitate communication between peers, peer executes command, client aggregates the results.

It should be noted that the change directory command (cd) is not an external application and is native inside the shell code. Because of this cd functionality was reimplemented within JDShell. It can support relative and absolute paths, as well as navigating to the parent directory ("cd .."). However, because it is a distributed shell, we have added checks to prevent the users from navigating to a directory that does not exist on ALL of the member machines. If you need to execute commands on one peer only, you can use the "![@peeraddress]" primitive. Add this to the end of any command issue any it will only execute on that peer. This lets you navigate to non-uniform directories so that you can fix them and make them more consistent. An example usage would be "rm tmp/full_file.txt !@192.168.1.5"

### 3.1 Error handling

In order for JDShell to be robust, it must handle the usual shell errors and also be prepared to deal with consistency issues and network failure issues. This section discusses some of those errors and how JDShell processes them.

- **CD to nonexistent directory.** As men-
tioned above, JDShell will not update to a new directory if it does not exist on all peers. It will instead give you a message indicating which peer does not have that directory and keep you in your current directory so you can make changes.

- **Server Failure** JDshell will return a message stating which peer it cannot connect to, but will execute the command on the other peers. This could be made more atomic by checking all of the thread connections before issuing the command. However, an example use cases where this is not desirable would be a nightly maintenance where half of the servers are taken down and patched then the other half.

- **Process Failure** JDshell will return a message stating which peer the Process Failed on, but will execute on the other peers. This can be made more atomic as well, but is much more difficult compared to server failures. It would involve an undo log that is maintained for each command so that whatever the command was can be rolled back. Since the behavior of Linux processes varies so much, this is a daunting task.

4 Evaluation

In order to prove that role-based abstraction works successfully we must show that it takes less time to issue one command to several computers in parallel then it does to log into each computer manually. However the effort it takes to measure this is more than the effort it takes to use common sense, therefore the author will take the path of least resistance in regard to this specific success metric. However the tool we have developed to support role-based abstraction can be measured in greater detail. In this section we evaluate the JDShell on two metrics: scalability and overhead.

4.1 Test Environment

In order to test JDShell, we installed it on 10 virtual machines running Ubuntu 12.04. Each virtual machine had 1.0gb of memory, 20gb of hard drive space. These virtual machines were running on a workstation with 16gb of ram, one 8 core Intel x86-64 processor, and 2TB hard drive in RAID5 array. The virtual machines stayed running through out all tests to ensure that extra memory wasn’t somehow freed up to make performance faster. Each VM had its internal clock synchronized with the others using the network time protocol server (ntpd) to ensure timing is consistent across all servers.

4.2 Scalability

The core contribution of the JDShell interface is that it is distributed. Like other distributed systems, to be valuable its performance must scale linearly as additional peers get added. To measure this concept we issued two commands to JDShell, "pwd" and "cd /home/sam/src", and measured the time it took in milliseconds to return the output as we increased the number of peers in the peer list. Figure 4 shows the results of this experiment. It appears to scale linearly for up to 10 peers in the peer list however in practicality we need to test at the 100-200 scale. Another metric that is commonly used to measure how distributed systems scale is to measure the maximum load N servers can handle. We do not need to perform this test because, in most cases, the shell will be under secure access and limited to the public.

![Figure 4: Scalability of JDShell](image-url)
4.3 Overhead

The other metric that needs to be examined is the overhead that JDShell incurs when running a single process as it is compared to the regular non-distributed shell. We already know that a non-distributed shell will outperform JDShell because there is no network communication involved. Since network communication is determined by bandwidth, current load, and distance it can be sporadic. So to measure just the overhead of the shell without the network communication we placed a start and stop timer on the execution of command passed in and measured just that. The command we used was a simple script that just sleeps for 10 seconds and it was executed 20 times and the average overhead was calculated. The bash shell was able to execute this with an average overhead of 3ms. The JDShell was able to execute this with an average overhead of 12.4ms. This slowdown is probably due to two major factors: Java is slower than C and JDShell has an extra thread to create that the bash shell does not. Despite this factor of 4 slowdown, these performance is actually a speed up because it does not include all the time logging into other servers that would be involved in a non-distributed shell.

5 Future Work

JDshell seems to be a decent prototype of a distributed shell, that allows users to issue commands to multiple peers at the same time. However it is still very far from a complete shell. Here is a list of improvements that can be made to the shell as it is in its present state.

- **XTerm support.** The shell does not support xterm applications, and this is the biggest shortcoming in the authors opinion. Some examples of xterm processes that don’t work are top, more, vi.

- **Atomic commands** In order to truly act as a consistent, distributed shell, JDshell needs to support full atomicity within the peer group. This needs to be implemented with some sort of transaction control and rollback algorithms to ensure that A.) All commands executed are either all or nothing and B.) When multiple clients issue commands, they are executed in some consistent, set ordering.

- **Non-redundant output.** Currently JDShell displays the output of all of the commands from each peer in the peer group. This can get really ugly and involve a lot of scrolling for the user when there are a lot of peers. Ideally JDShell would only display one copy of the output, by merging the output of all peers. If one peer has different output, then that is displayed distinctly to the user so that they know they need to address it.

- **Tab completion.** It would be really nice to have tab completion and command history for an improved user experience.

- **Syntax highlighting.** For example, listed directories are bold and a different color for an improved user experience.

6 Conclusion

There is a lot of room to perform further research on role-based abstraction as well as improve the JDshell tool. This paper only investigated the shared interface aspect of role-based abstraction. It did so by proposing a distributed shell that allows users to issue one command across several peer computers in parallel. Many more questions arise when we consider the fact that we could also share resources as well. How can we construct a full operating system that is fine-tuned for role-based abstraction? What would the file system look like? The process manager? Will this new operating system scale well? Will it ever get adopted over the tried-and-true operating systems that have been used for the past 20 years? -fin
References


