A Job Brokering Shell for Interactive Single System Image Support

Cameron R. Kerr
University of Otago
New Zealand
ckerr@cs.otago.ac.nz

Paul Werstein
University of Otago
New Zealand
werstein@cs.otago.ac.nz

Zhiyi Huang
University of Otago
New Zealand
hzy@cs.otago.ac.nz

Abstract

When users interact with computers, programs are run on a local machine and display on the local machine. Single System Image (SSI) is about taking a cluster of computers, and making them appear and behave as a single machine. SSI systems are typically non-interactive, batch systems. This paper introduces a Job Brokering Shell model for SSI which allows running interactive processes on worker nodes while having them appear to run on the cluster's gateway. The assumed environment is a computer science student laboratory where students can log in remotely to a gateway and their programs are distributed evenly to the laboratory workstations.

1. Introduction

It is perhaps an ironic side effect of modern computing, that with faster workstations, computers are both idle for long periods, and overburdened at other times. Distributed computing helps to relieve this pressure by using lightly loaded workstations as computational facilities.

This form of clustering is an active area of research. Today we see it being used in projects such as SETI@home[4] and distributed.net[1]. Such clusters are often used for batch processing. It also is desirable to run processes on other machines *interactively*. This is especially the case on systems where you log in remotely (into a *gateway*), and then log into one of the interior workstations (*node*, in cluster terms) in order to do work.

The environment we envision is a computer science student laboratory. Students are not assigned to any particular machine and can use any available computer. They may remotely login. However it is not possible to determine which machines are in use prior to logging in remotely. Thus the loading of processes and users on any particular machine is not predictable or uniform.

For both remote logins and for some types of processing, it would be desirable to view the machines in the lab as a single computer. This concept is known as *Single System Image* (SSI)[6]. Ideally with SSI, the remote user has no knowledge of the machines in the cluster. Users interact with the gateway machine which assigns tasks to worker nodes. The key points of SSI systems are often given as listed below.

Transparency The user should not be able to discern that their jobs are being run on different computers.

Minimising load on the gateway This is a key goal because all programs, whether executed remotely or locally, take up resources on the gateway. In addition, remote jobs incur load by the transport mechanism to execute and interact with the programs.

Cost Increasingly clusters are being built with COTS (Commodity Off The Shelf) components, often meaning x86 based machines running a Linux operating system. This significantly reduces costs over other solutions such as a true massively parallel processor (MPP). However, something is needed to create the illusion of a single powerful machine. That bond is provided by an SSI solution.

Interconnect A high speed interconnecting network is required.

A gateway based SSI system is beneficial to users because jobs are automatically sent to the most appropriate machine. In a true SSI environment, users do not need to be concerned about the existence of the cluster. Instead, the gateway appears as a single powerful machine. The environment is also beneficial to the other machines in the cluster since the workload is spread more evenly.

In the next section, SSI models are introduced. Section 3 describes our job brokering shell model. Its implemntation is given in Section 4. Section 5 give the results of performance tests on the prototype. Finally we give some areas of future research and conclusions. In the remainder of the paper, we use the term *gateway* to refer to the gateway machine which is the computer accessed by remote users of the

cluster. *Worker node* refers to machines in the cluster which are available for user jobs, but users do not directly log into them.

2. Single system image models

The design of a SSI system is a tradeoff between transparency and time/effort costs. Not all operating systems are designed to facilitate clustering, and it can take a significant effort to implement such a system. Issues to consider in the design of SSI include:

- Degree of transparency
- Shared resources
- Portability
- Scalability

2.1. Degree of transparency

A SSI system can be implemented at many layers, depending upon how transparent the system is to the user and the cost of implementation. On systems with weak transparency, a programmer must be aware of the system's true nature, else programs may not execute as expected.

Listed below are layers where transparency could be implemented. Higher level implementations generally have weaker transparency, while lower level implementations have more robust transparency.

Program Each program that is to be run remotely is contained in a wrapper program or script that starts the program on a remote machine. This is a simple technique for a few programs that must run on another host, but does not scale well.

For example, using the following command at work allows me to check and read my news at home.

```
ssh -2 -t cameron@myhome.net rtin
```

`rtin` is a script that starts news on my home computer.

Shell The shell receives the command from the user (for example to start Netscape). Then the shell logs into an appropriate worker node and starts the program on that node.

API The programmer uses a special API (application programming interface) to distribute the tasks to worker nodes. Two examples are MPI (Message Passing Interface)[2] and DSM (Distributed Shared Memory)[8] systems.

System Library The application is compiled with (or linked at run–time to) a library that implements RPC-like techniques. This can be network intensive since all system calls get routed through local/remote stubs. The Condor System[9] uses this technique.

Kernel A kernel implementation uses a modified kernel such as the Mosix kernel[5] (modified Linux kernel). Kernel level SSI is fully transparent to applications, but it requires a homogeneous environment.

In environments where the machines may be used interactively, it may be impractical to go lower than the system library level since kernel level implementations tend to turn the machines into a computational facility rather than a workstation.

2.2. Shared resources

Shared resources are the filesystem, I/O devices, memory, and the process space. The filesystem must be shared by all machines in a SSI cluster to provide a uniform environment. In the case of a heterogeneous set of machines, a common filesystem layout needs to be emulated, depending upon the level of transparency desired. I/O devices refer to devices such as terminals and printers which are often shared in a networked environment and are not a concern in an SSI environment.

Shared memory is implemented in DSM (distributed shared memory) systems, where each node shares part of its memory space with other nodes, forming a block of shared memory. Software DSM requires programs to be written in a special way or to be recompiled in order to use DSM. Making the process space transparent requires any processes, started by the gateway on a worker node, to be mirrored in the gateway's process table. Thus the remote process appears to be running on the gateway itself.

As well as appearing as though a process is running on the gateway, it must also respond to local events such as signals. Not all signals can be caught (SIGKILL, for example) so the SSI system must be able to monitor the mirrored process entries and send fatal signals to the real remote process.

2.3. Portability

Ideally, a SSI system should work well on a cluster of heterogeneous machines, though how important that is, is a matter for the local policy makers.

2.4. Scalability

If everyone uses the same gateway machine as the point of entry, the gateway could get heavily loaded and become a bottleneck. It may be possible to use more than one point

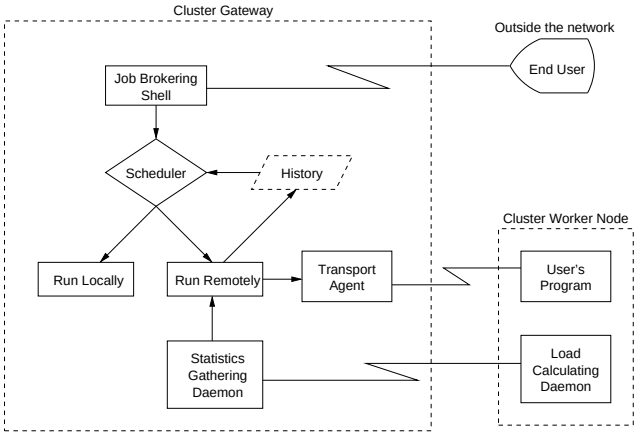

Figure 1. Job Brokering System Model

of entry. Load balancing solutions, such as Round Robin DNS[6], could be used to log the user onto an appropriate gateway.

3. Job brokering shell model

The Job Brokering Shell model is shown in Figure 1. The user interacts with a modified shell, called a Job Brokering Shell, on the gateway system. Ideally, the user should not be aware that commands they enter may be run on other machines. Thus the Job Brokering Shell acts as an interactive job submission agent.

Not all programs can be run on another machine. For example, the `logout` shell command must be run locally. In addition, the system administrator may elect to assign some programs to run on only certain machines. Determining the best machine for a given command is done by the *Scheduler*.

The user's command may be a command pipeline which is a string of connected processes. Ideally, the pipeline should be executed on a single host to reduce network traffic due to the Unix pipe. However, it is possible that a command pipeline has to be run on different machines due to application assignment. The Job Brokering Shell must be able to start such a command.

When the Scheduler decides to run a command remotely, it must choose the best suited worker node. The model has monitoring software consisting of a *Load Calculating Daemon* and a *Statistics Gathering Daemon*. Load Calculating Daemons, running on each worker node, determine local loading. They report the loads to the Statistics Gathering Daemon running on the gateway. Using these statistics, the scheduler determines the most appropriate worker node. It would also be possible to maintain some type of history to aid in the determination.

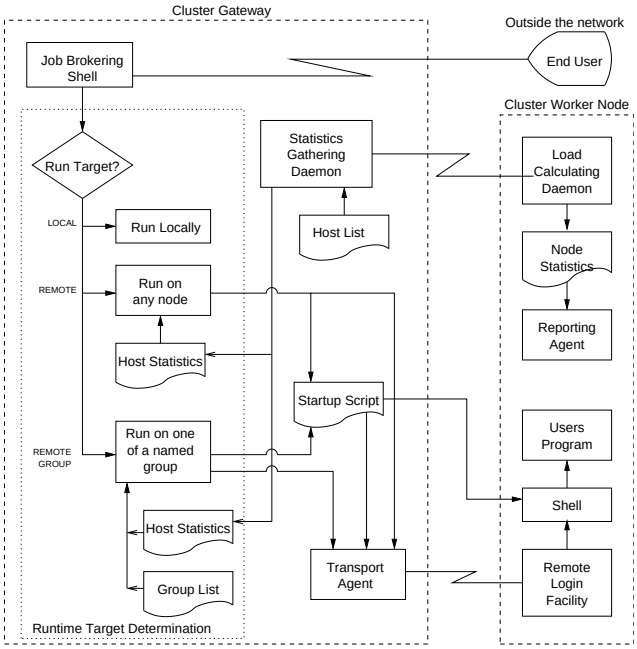

Figure 2. Overview of the Job Brokering System

Once the Scheduler determines the target worker node, the Job Brokering Shell uses the *Transport Agent* to log into the worker node and execute the user's program. Before the command starts execution, the environment on the worker node is made to match the user's current environment on the gateway.

4. Implementation

The implementation of our Job Brokering Shell model is shown in Figure 2. We modified the popular `bash`[7] command-line shell to provide basic job brokering capabilities. In addition, load determination and reporting were developed. With the exception of the modified shell (called `jbash`), which is written in C, all other code is presently implemented as Bourne shell scripts. Each of the components is described below.

4.1. Worker–node services and daemons

Load calculating daemon The Load Calculating Daemon, called *jbstatpd*, determines the system and network utilisation. The system load is defined as some absolute value representing the performance of the system, where memory and CPU utilisation are the main factors.

In the prototype implementation, the load average given by the kernel is used. This load average is the number of processes waiting in the kernel ready queue, averaged over the last one minute. As an aside, it might be a worthwhile to include some metric for the maximum performance expected from the system, such as the "bogomips"[10] rating that Linux gives a CPU, the number of CPUs in the system, and the amount of memory.

The network load is characterised by four parameters: actual traffic into and out of the interface, maximum capacity of the interface, and whether the interface is half-duplex or full-duplex. Determining these parameters from the network interface is fairly complicated. For the prototype, they were manually set based on an observed average.

The system and network load is written to a file so that the *Reporting Agent* can return the statistics to the gateway on request. The contents of the file are:

```
1.22 eth0:2,2/100000-HD
```

The first entry is the system load. Then, for each network interface, the entries are: interface name, incoming and outgoing data rate in kbps, followed by the maximum channel capacity of the interface in kbps. The last entry is the duplex setting of the interface.

We feel that it is more appropriate to give the network statistics in this form, rather than, say, the percent utilisation of an interface, since it results in the possibility of a more intelligent statistics gatherer.

Reporting agent The Reporting Agent, called *jbreport*, outputs the worker node's statistics to the gateway. It is activated throuth `inetd` or `xinetd`.

Remote login facility The Remote Login Facility uses OpenSSH in the prototype. OpenSSH provides X11 forwarding and tunnelling as well as non-interactive, secure user authentication. Thus the worker nodes can have private IP addresses and data appears to come from the gateway.

4.2. Gateway services and daemons

Job brokering shell The Job Brokering Shell is the program the user interacts with on the gateway machine. In the prototype, it is called *jbash* and is a modified version of `bash`.

The modification required for our prototype involves applying some logic to the requested command to determine where to run the command. This is done by matching the command line to some regular expressions to determine whether to run the program locally or on a worker node. In general, commands dealing with the local system such as `logout` are run locally. Otherwise any program with more overhead that the transport agent should be run on a worker node.

Statistics gathering daemon The Statistics Gathering Daemon, called *jbstatgd*, polls the *jbreport* programs on the worker nodes to gather their statistics. It uses the statistics to compute a desirability rating for each worker node. The desirability rating is a number from 0 to 7 where a seven means a node is very lightly loaded. The results are written to the *Host Statistics* file.

Runtime target determination The scheduler is responsible for determining which worker node(s) will run a user's request. This part of our implementation is shown inside the dotted box marked *Runtime Target Determination*. There are two modes in the prototype, REMOTE and REMOTE GROUP.

In the REMOTE mode, the program, *jbquery*, determines which of the active hosts has the highest desirability rating. That host's name is passed to the Transport Agent to run the command.

In the REMOTE GROUP mode, the program, *jbqueryg*, uses the Group list file along with the host statistics to determine the most desirable machine in a group of machines. Again, that host's name is passed to the Transport Agent.

Transport agent The Transport Agent is responsible for loging into the target worker node and running the user's command. The prototype uses ssh (OpenSSH[3], in particular), since it has the capability to authenticate without the use of a password, as well as being able to run both interactive and non-interactive programs.

4.3. Limitations

Some limitations exist in the prototype system, related to the tools that we used and design limitations of the system.

The load imposed by the transport agent will be one of the defining performance characteristics of the system. Given that `ssh` is the transport agent in the prototype, it only makes sense to off-load heavier programs, such as emacs and netscape, or those that need to run on an application server. In an isolated environment, the overhead of `ssh` could be reduced if it could be used without encryption. It is worth noting that the SSH2 product has a non-encrypted mode as a compile-time option but was not available for the prototype.

High-level SSI systems have practical limitations with regards to transparency. Some of these limitations are in

making the process space transparent between machines. However, the approach taken in the prototype is very useful because less code is modified, and it should work well on any POSIX based operating system.

The current implementation considers all worker nodes to be of equal capability. Issues related to worker nodes with different capabilities will be addressed in a future version.

5. Results

To measure the preliminary performance of our prototype, a set of eight identical PCs, incorporating an 800 MHz Pentium III processor with 128 MB memory, were configured as a SSI cluster. One of the machines was designated as the gateway.

A user was simulated by means of an `expect` script. The script runs a combination of `gcc`, `netscape`, and `latex`.

Two metrics were selected to be measured against the number of users in the cluster. The metrics are:

- Load on the gateway
- Network traffic

These metrics are discussed below.

5.1. Load of the gateway

To measure the load of the gateway, we use the same load calculating program used to measure the desirability of the worker nodes. For brevity, this metric is called Desirability.

The results are shown in Figure 3. The vertical bars show the minimum, average, and maximum desirability of each worker node, indicating a large variation in the loading on the worker nodes. This is a result of the simple scheduling algorithm presently used. Because the Linux kernel gives us a one minute sliding average of the system load, starting n programs within a few seconds of each other means the load statistics have not changed sufficiently to reflect the most recently used processor from the current scheduling decision. A more desireable scheduling algorithm is being explored.

Figure 3 shows the performance of the gateway (solid line) declining gracefully when the jobs are distributed to worker nodes. This is in contrast to the rapid decline (dashed line) when all users are running on the gateway machine.

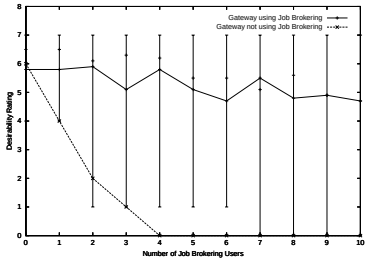

Figure 3. Gateway desirability vs number of users

5.2. Network traffic

The metric measures the amount of network traffic created by the job brokering system. For this metric, we determined the average network traffic of the cluster without

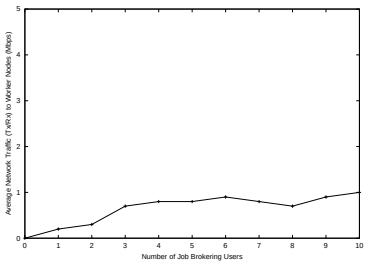

Figure 4. Network traffic vs number of users

any users on the worker nodes or the gateway. The user `expect` scripts were then run and the background network activity was subtracted. The result is the network traffic induced by the job brokering system sending jobs to worker nodes. It is shown in Figure 4. Only a small amount of traffic is caused by distributing the jobs to worker nodes.

5.3. Transparency

The prototype is not as transparent as a system implemented at a lower level. For example, the `emacs` program, when run in its X11 mode, will display the hostname as part of the title–bar since it uses the *gethostbyname* command. Other transparency leakages involve process names and process IDs of child processes not being visible on the gateway.

Currently, signals can not be passed through the transport agent to the remotely running program. These problems could be resolved using lower layer implementation techniques. One way would be to employ shadow processes, which would essentially be a proxy, and provide local PIDs for the remote processes. Since not all signals can be trapped, such as SIGKILL, the shell will need to monitor the shadow children and send these signals to the appropriate remote process.

5.4. Further work

There are ample avenues for further research and work on the prototype. A more intelligent scheduling algorithm is needed. Such an algorithm would look at a smaller time interval or perhaps take into account the fact that jobs were recently introduced and have not yet influenced the load average being reported. Alternatively, the scheduler could maintain a history of its allocation of jobs to worker nodes or distribute jobs to machines of equal desirability in a round robin fashion.

As explained above, work needs to be done so jobs actually appear to be running on the gateway. At the present time, users have no way of monitoring the status of their jobs on the worker nodes.

In addition, the file system needs to be made more uniformly available. The prototype is able to distribute jobs easily because its environment uses a NFS file server for the uniform copy of the `/home` file system.

6. Conclusions

This paper has presented a model for running processes (or rather, command pipelines), on the least loaded machines in an cluster. The implementation allows load sharing in an environment where many people log into a gateway machine to do their work.

While the current implementation is rather basic, it does have several advantages. Since it operates at a high level, changes and upgrades to the kernel have no effect on the implementation, and no software must be changed. It provides an environment with which users are familiar so training is not a concern.

It allows the administrator to more easily control external access to the lab machines. At the same time, it prevents the accessed machines from being heavily loaded while idle machines are available.

References

[1] distributed.net. http://www.distributed.net/.
[2] MPI Forum. http://www.mpi-forum.org/.
[3] OpenSSH. http://www.openssh.org/.
[4] SETI@home. http://setiathome.ssl.berkeley.edu/.
[5] The MOSIX Kernel. http://www.mosix.org/.
[6] R. Buyya. *High Performance Cluster Computing, vols 1 and 2*. Prentice Hall PTR, Upper Saddle River, New Jersey 07458, 1999.
[7] T. F. S. Foundation. GNU Bash. http://www.gnu.org/software/bash/bash.html/.

[8] Z. Huang, C. Sun, S. Cranefield, and M. Purvis. View-based Consistency and its Implementation. In *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2001)*, pages 74–81. IEEE Computer Society, May 2001.

[9] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor: A Hunter of Idle Workstations. In *Proceedings of the 8th Internation Conference on Distributed Computer Systems*, 1988.

[10] W. van Dorst. BogoMips mini HOWTO. http://www.linuxdoc.org/.