Shell over a Cluster (SHOC): Towards Achieving Single System Image via the Shell

C.M Tan, C.P. Tan and W.F. Wong
Department of Computer Science
National University of Singapore
3 Science Drive 2, Singapore 117543
{tancherm, tanchung, wongwf}@comp.nus.edu.sg

Abstract

With dramatic improvements in cost-performance, the use of clusters of personal computers is fast becoming widespread. For ease of use and management, a Single System Image (SSI) is highly desirable. There are several approaches that one can take to achieve SSI. In this paper, we discuss the achievement of SSI via the use of the user login shell. To this end, we describe `shoc` *(Shell over a Cluster) – an implementation of the standard Linux-GNU* `bash` *shell that permits the user to utilize a cluster as a single resource. In addition,* `shoc` *provides for transparent pre-emptive load balancing without requiring the user to rewrite, recompile or even relink of existing applications. Running at user-level,* `shoc` *does not require any kernel modification and currently runs on any Linux cluster fulfilling a minimal set of requirements. We will also present results on the performance of* `shoc` *and show that the load balancing feature gives rise to better overall cluster utilization as well as improvement in response time for individual processes.*

1. Introduction

Recent advances in both microprocessor and network technologies have resulted in the widespread use of cluster of personal computers (PCs) [4]. Dramatic improvements in cost-performance and their off-the-shelf availability make it an attractive approach to provide cheap computing cycles to computer users.

With the use of networked nodes or cluster, computing resources are no longer centralized. This gives rise to the situation of unbalanced utilization where certain resources (for example, CPU or memory) on one node are heavily used, while those of some others are under-utilized. Hence, the availability of a single system image (SSI) [12] for the entire cluster provides the opportunities for load balancing. This allows a user's program to execute transparently on any node in the cluster. The program is unaware of the individual nodes making up the cluster and how they are interconnected. To the user, the cluster appears and operates like a single node with much more computing power. Thus an important SSI feature is transparent, cluster-wide load distribution. This attempts to share all available resources in the cluster, such as CPU cycles and memory, among all executing processes.

In this paper, we describe a prototypical user-level shell we called `shoc` (Shell Over a Cluster) that provides SSI over a cluster of PCs. The shell is responsible for accepting the commands entered by the user and submitting them to the operating system (OS) for execution. Hence, the shell acts as an interface between the user and the OS. Thus the shell gives us a point of entry for implementing SSI features at the application level.

Our cluster consists of dedicated Linux systems, interconnected with Ethernet LAN. `Shoc` is a modified version of the Linux-GNU `bash` shell extended to provide the clustering capabilities. Currently, we have achieved the following:

- *No modifications, no relinking and no recompiling to existing program.*
  The use of dynamic library preloading allows us to provide SSI feature.
- *Cluster-wide load distribution.*
  We had modified the shell to provide for load distribution functionality to achieve effective resource utilization. The load distribution uses a Load Balancing (LB) policy in order to even out the load among the nodes in the cluster.

2. Related work on SSI

There are several approaches to realizing SSI. They can be classified in two main categories, those that perform it at the application or at the kernel level. LSF [6][11] provided support for remote execution of interactive sequential jobs and load balancing but does not provide process migration. Parallel processing libraries such as PVM [7] provided for means to perform parallel processing on a cluster. Extensions of these systems such as dynamicPVM [9] (using Condor [2]) and tmPVM [8] provided for dynamic load balancing in a PVM environment. GLUnix [5] supports both interactive and batch-style remote execution of both parallel and sequential jobs. All the above systems lack transparency as either special commands are introduced, or the user is require to rewrite, recompile and relink existing code with special libraries.

Another approach that is totally transparent is to solve the problem at the kernel level. Both Mosix [1] and SSI for Linux [12] are such solutions. However, one needs to commit to the OS and there is a whole set of administrative considerations.

Cluster Starter Kit for Linux [14] is an application for the management and monitoring of a Linux cluster. It allows administrators to monitor for certain conditions and to take automated responses. However the Cluster Starter Kit works with only one server per cluster. The Jxta command shell [10][15] is an application that provides interface to the Jxta platform. The Jxta platform implements a core set of basic services such as communication and group membership. This allows other services and applications to be built on top of the basic components.

Scyld Beowulf [18] is another software solution for cluster OS. It includes an enhanced Linux kernel, libraries and utilities designed for clustering. The SSI is provided through bproc, the Beowulf cluster process management kernel enhancement. Scyld Beowulf works on the concept of a front-end "Master node" and cluster "Computation node". Processes are started on the front-end node and migrate to a cluster node. The bproc makes the processes running on cluster nodes visible and manageable on the front-end node. Currently, Scyld Beowulf does not support load balancing.

Table 1 gives a summary of the comparison between various approaches to achieving SSI. In summary, `shoc` distinguishes itself from the other approaches by the following:

- *It is implemented at the user level*. This gives the user flexibility in changing the choice of OS as an after-thought.
- *It is transparent to the user*. The user is presented a familiar `bash` shell interface. Applications do not need to be touched to run in this environment.
- *It performs load balancing.* This allows for the true sharing of cluster-wide resources.

3. Design and Implementation

3.1. Overview

`Shoc` is a variant of the standard Linux-GNU `bash` shell [16][19], implemented at user-level to support cluster-wide load distribution. The current version of `shoc` works in a homogeneous environment, where all the nodes in the cluster have the same architecture and running the same version of the Linux kernel. This ensures binary compatibility among all nodes in the cluster. Our cluster is built upon a common Network File System (NFS) [3] that allows homogeneous file access from all nodes.

When the user submits a command at the command-line, `shoc` will process the command just as `bash` would and in addition will perform initial process placement. The decision about where to execute the command is made by the *Load Manager* (LM) based on the load information of individual nodes in the cluster. Hence, both the shell and LM cooperate to determine where a command should execute. The command is not executed directly on the remote node. Rather it is spawned off as a child process of the *LSERV* process. We will describe the details of this in section 3.3. In summary, the three components to achieve SSI are listed below:

- Shell
- LSERV
- Load Manager (LM)

The overview of the interactions between the above components is shown in Figure 1. In the following subsections, more detailed description of each component will be given.

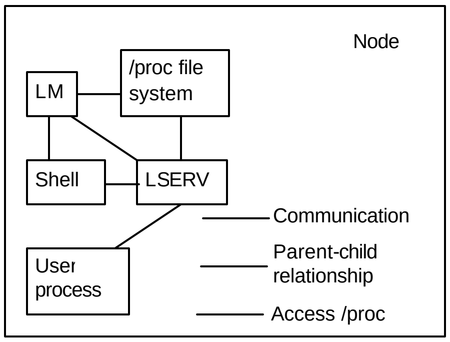


Figure 1. Components to achieve SSI

System	Category	Implementation level	Load balancing	Process migration
Shoc	SSI	User (shell)	Yes	Yes
LSF	Resource management suite	User	Yes	No
PVM	Software package	User	Yes	No
Condor	High throughput computing environment	User	Load sharing	Yes
TmPVM	Extension to PVM	User	Yes	Yes
DynamicPVM	Extension to PVM	User	Yes	Yes
GLUnix	SSI	User	Yes	No
Mosix	SSI	Kernel	Yes	Yes
SSI for Linux	SSI	Kernel	Yes	Yes
Cluster Starter Kit for Linux	SSI	User (application)	-	-
Jxta command shell	Interface to Jxta platform	User (shell)	-	-
Scyld Beowulf	software solution	Kernel and User	No	Yes

Table 1. Comparison of approaches to achieve SSI

3.2. The Shell

We shall now describe the internal structure of `shoc`. Our description will be based on `bash`. However, we believe our ideas can easily be implemented in other shells as well.

As an interface between the user and the operating system, the shell handles the submission of user process. This allows the shell to communicate with the LM to determine where it should execute the process, either locally or remotely. Figure 2 shows a simplified version of how the shell reads in a user command and executes it. It consists of a loop where the user command is read in by `read_command()`. The function `execute_command()` will then determine the type of the command supplied by the user, execute shell internal commands or it will calls `execute_disk_command()` to spawn a new process to execute executable programs.

For the shell to know where to execute a user command, the function `execute_disk_command()` in `bash` is modified to include the following function:

```
RequestSingleHostForRemoteExec(program)
```

The above function will send a message to LM to request for the node to execute the process. Together with the message, the estimated memory requirement (based on executable file size) of the process is sent. This is to allow the LM to check whether the execution of the process will exhaust the available free memory. The reply from LM contains a host name for the process to be executed. This task is performed by the LSERV component.

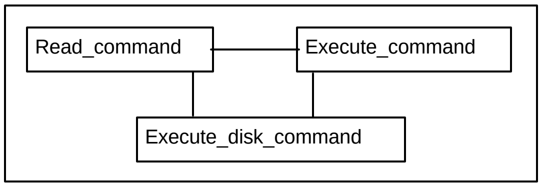


Figure 2. Execution of a user command

Using LSERV to start the process for both local and remote execution requires modification to the command submitted by the user. Initially, the shell will have the command as entered by the user given as follow:

```
program argument …
```

Modifications to `execute_disk_command()` is needed to replace the user program with LSERV. The steps to achieve this are given below:

1. Set the program to be executed by shell as LSERV.
2. Pass the host name for execution as an argument to LSERV.
3. The user process and its arguments are then appended as arguments to LSERV.

To the shell, it will look as though the user had typed the following at the command line:

```
LSERV hostname program argument …
```

By performing this simple modification, a user program can be run on any cluster node. Figure 3 shows the exchange of messages between the shell and LM to perform initial task placement (ITP).

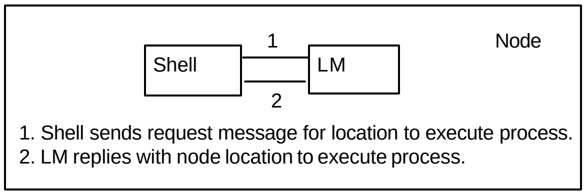
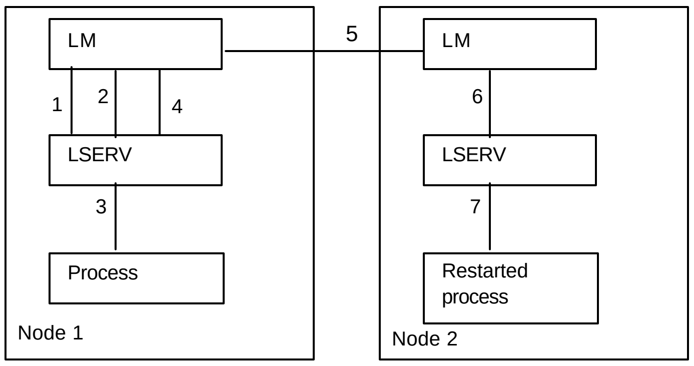

Shell
LM
Node
1
2
1. Shell sends request message for location to execute process.
2. LM replies with node location to execute process.

Figure 3. Exchange of messages between shell and LM.

3.3. LSERV

LSERV is an application level program started by the shell. Its responsibility is to start the user program on



1. LSERV sends resource usage message of process to LM.
   This is done when the process is eligible for migration.
2. LM has chosen process for migration by sending a migration message to LSERV.
3. LSERV sends a SIGUSR2 signal to process to freeze the process to be migrated.
4. LSERV informs LM to restart process on another node.
5. LM forwards restart message to destination node LM.
6. Destination node LM starts LSERV.
7. LSERV restarts the frozen process to allow it to continue execution on a new node.

Figure 4. Resource usage update and process migration

either a local or remote node. Hence, for each executing user process, there is a copy of LSERV started. The rationale for using LSERV is to minimize the amount of modifications to be made to the shell. This allows a more modular development and permits additional functionalities to be added in an easier manner. This is possible since LSERV is just another program that is independent from the shell.

Currently, the LSERV is responsible for performing a number of tasks to achieve the overall goal of load distribution. One of its important tasks is to set the environment variable (LD_PRELOAD) to load the dynamic library for migration and signal handling before the process starts to execute. When the user process is executing, LSERV will sample the process resource usage at periodic interval. This information is obtained from `/proc/PID/status`, where PID denotes the process identifier of the user process. The information that is of interest to us is the amount of memory used by the process as well as its execution time. When a process becomes eligible for migration, its execution time must be at least the cost of performing the migration. This condition is needed to prevent short-lived processes from being chosen for migration.

When a migration message is received by LSERV from LM, the LSERV is responsible for carrying out the migration of the user process to a new node. LSERV achieved this by sending a signal SIGUSR2 to its child process and restart the process on another node. Figure 4 shows both the resource usage update (Step 1) and the process migration (Steps 2-7).

3.4. Load Manager (LM)

A copy of the LM is executed on each node. It performs regular sampling of load information to determine if load re-distribution is necessary. LM also takes care of the dissemination of the local load information to other nodes. The local load information is obtained from /proc/loadavg, which gives both the system load average for the past one minute and the instantaneous run queue length. The file /proc/meminfo supplies us the amount of free memory of the local node. When the local node becomes heavily loaded, the LM will use the resource usage message (from LSERV) to formulate a reply (to LSERV) to carry out migration, pushing those eligible processes to node with lighter load. The LM component does not record the names of processes that are eligible for migration in order to keep the management overhead low.

3.5. Achieving migration

A migration facility consists of two parts, freezing (or suspending) and restarting of the process. The freezing stage is used to prepare the necessary transfer of memory pages of the migrating process. The freeze stage preserves the current execution context as well as its environment for later use. During restart, the previous execution environment of the process as well as its memory pages are restored so that it may continue execution.

Checkpointing method used in Condor is an example of a migration facility. It consists of a library that must be linked with the application code. Part of this library is a signal handler that is used to create the checkpoint file of the process. This checkpoint file is then used to restart the process on a node of compatible architecture and operating system.

Our migration facility is similar to the one used by Condor. Instead of using the linker to incorporate the necessary signal handler for performing migration, we use the dynamic library preloading. This achieves the installation of the signal handler as well as the migration code without any modifications (compilation and relinking) to user executables.

The dynamic library preloading is achieved by setting the environmental variable LD_PRELOAD. This allows our migration library to be loaded before the user process is executed. Our migration library will install the signal handler for SIGUSR2 before executing the actual user process.

The process can be frozen during its execution when the signal SIGUSR2 is raised by LSERV. The associated signal handler will invoke the migration code to extract the memory segments of the executing process from the /proc filesystem [17]. The extracted contents are then written to a segment file. Once this is done, the current process will exit.

The restart of the process is achieved by using the text-segment from the process binary and restoring the memory segments of the frozen process from the segment file. The stack of the frozen process is also restored so that the execution is continued immediately where it received SIGUSR2 signal.

3.6. Limitations

Our current implementation works with only single process that does not use inter-process communication (IPC), sockets and pipes. However, the standard file descriptors are handled by setting up socket connections to the local LSERV. This enables us to redirect any input and output to the remote process.

The following is a list of other limitations:

- The migration mechanism assumes that the user directories are mounted identically on all nodes.
- Timer routines (such as sleep()) may not be honoured.
- Sufficient disk space must be available to store the segment file during migration.
- Both LM and LSERV must be present in order for migration to be successful.
- Support for parallel jobs are currently not present.

4. Load balancing

Load Balancing (LB) is the attempt by the system to even out the differences in the load of the nodes. This has the aim of ensuring that all the available resources are shard equally among all running processes. The following subsections will describe more about the considerations during load distribution and the algorithms used.

4.1 Load distribution considerations

Load distribution comes with overhead. Hence, care must be taken when capturing load information. Scalability issues related to how load information is disseminated need to be considered too. These will affect the accuracy of the load information and the quality of the load distribution decision. There is a tradeoff between the accuracy of the required information and the overhead. Care must be taken to prevent the system from becoming unstable when it is heavily loaded. The reason is that the extra overhead may aggravate the load problem instead of alleviating it.

Capturing of information. When load distribution is performed at the application level, there are limitations on the amount of information we could obtain. Take disk I/O monitoring as an example. Intercepting system calls would slow down the overall process and thus affect the significance of the recorded information. Another issue is the frequency at which both the load information as well as the resource usage of process can be obtained. In Linux, both of the above information can be obtained from the `/proc` file system. This involves file access that can be costly especially when the operation is carried out frequently.

In our design, we have chosen to keep the overhead cost of load distribution as low as possible by sacrificing the accuracy in gathering both the load and process resource usage information. This is achieved by performing the sampling at intervals of 1s. This sampling interval is adjustable in terms of value and scale depending on the system used.

Scalability issue. Two obstacles to the scalability of any system are the use of a centralized node to perform certain duties and the use of a broadcast algorithm. When the node becomes heavily loaded, a centralized design will experience a bottleneck due to its inability to service the incoming requests fast enough. The use of broadcast can aggravate the network congestion problem.

In our design, each LM is responsible for handling execution requests from processes originating from the local host. This will eliminate any communication between a process and a remote LM. The communication between any two LMs is peer to peer. This avoids the use of broadcast or any centralized server.

4.2 Algorithms used

Dissemination of load information. The algorithm used in the dissemination of load information is given by Mosix [1], Algorithm 8.1. This involves updating the local load value and randomly sending to another node half of the local load vector entries. When the local node receives load information vector from another node, the former will merge the received information into the local load vector.

Triggering of load distribution. The load distribution algorithm is triggered when the amount of swap space used exceeds the amount of free memory. This is to avoid the occurrence of disk thrashing when the memory demand by all the processes exceeds the amount of physical memory available on the node. Another trigger condition is when the minimum of instantaneous run queue length and average queue length for the last one minute hits a certain threshold. The rationale for using the minimum value is given below:

- For a lightly loaded node, the instantaneous run queue length reflects a sudden increase in the number of short-lived processes. The value for the average queue length, however, will grow more slowly. Taking the smaller of the two values, we try to delay load distribution to allow the local processor to clear the run queue length.
- For a heavily loaded node, the instantaneous run queue length reflects a sudden drop in the number of processes. The value for the average queue length, however, will decrease more slowly. Taking the smaller of the two values, we try to avoid unnecessary load distribution because the average queue length shows a high value.

The metrics used to derive trigger conditions for load distributions contain three key values, namely the amount of free memory, the instantaneous run queue length and the average queue length. When no trigger condition is present, a newly arrived process will execute on the local node, and no process migration will be carried out.

5. Performance of load distribution

Using our implemented system, we performed experiments to determine its performance in load distribution using LB. The experiments are aimed at testing two aspects of the system. The first experiment tests how the system responds to an increasing number of hosts while the workload is kept constant. The second experiment tests how the system responds to the increased in workload while the number of hosts is fixed.

In the experiment environment, a 100Mbps Ethernet switch connects 11 personal computer nodes (Pentium II 400 MHz single processor system). Each computer runs the Redhat v2.2.16-22 Linux operating system. The `bash` shell (version 2.04.0(1)) was modified and used in our experiments. In all the experiments, a single stream of workload is injected into a node. The workload consists of uniform CPU-bound processes that require 60 seconds of CPU time each. Once the workload is injected into the system, they are free to migrate to any node. Each experiment is carried out four times and the average execution times are recorded.

Varying the number of hosts. In this experiment, 30 CPU-bound processes are injected into a node. The number of hosts is varied from 7 to 11 and we want to observe whether the additional nodes result in better wall-clock time. Table 2 shows the wall-clock time of the processes. The column 'Max' refers to the longest wall-clock time recorded for a process while the column 'Average' refers to the average of the wall-clock time of all 30 processes. The column 'Average (opt)' refers to the best possible average wall-clock time with prior knowledge about the workload. The percentage value in column three represents the performance of the system with respect a system with prior knowledge about the workload. As we can see from Table 2, the adding of more hosts results in better performance (both in terms of Max and Average). The percentage value also shows a corresponding improvement. Therefore with load distribution, it gives rise to better overall cluster utilization as well as improvement in response time for individual processes.

Number of hosts	Wall-clock time (s)		
	Max	Average	Average (opt)
7	414.67	290.14 (31.7%)	220.33
9	261.06	207.70 (16.7%)	178.00
11	226.65	171.11 (15.5%)	148.20

Table 2. Varying the number of hosts

Varying the workload. In this experiment, a total number of 11 hosts are used to study how the increase in workload affects the ability of the system to distribute the workload. Table 3 shows that the average wall-clock time for 10 to 35 processes is within 17% of the optimal load distribution. This shows that the system is able to balance the range of increasing workload by utilizing the available nodes.

Number of processes	Wall-clock time (s)		
	Max	Average	Average (opt)
10	62.15	60.48 (0.8%)	60.00
15	111.32	94.13 (9.3%)	86.10
20	158.66	118.14 (13.5)	104.10
25	207.04	147.16 (16.2%)	126.60
30	209.42	164.23 (10.8%)	148.20
35	282.50	190.70 (12.6%)	169.40

Table 3. Varying the workload

From the above two experiments, the implemented system shows that it is able to use the increase in the number of available hosts as well as carrying out effective load distribution as the workload is increased.

6. Inter-process Communication (IPC)

6.1. Overview of distributed IPC (DIPC)

We have implemented a prototype System V IPC that is able to run over a cluster. This is achieved by intercepting the IPC function calls. Figure 5 shows the implementation overview of the developed DIPC.

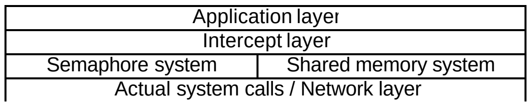


Figure 5. Implementation overview of DIPC

The current system provides for only semaphore and shared memory usage. Each node will run a DIPC daemon that is responsible for handling IPC requests. The DIPC daemon (DIPCd) residing on the login node of the user is called the master DIPC daemon (mDIPCd). mDIPCd is responsible for handling semaphore requests and perform the required operation locally. This centralized control is to ensure the correctness of the semaphore operation. The control for the shared memory is distributed, handled by all DIPCd. This is to avoid the bottleneck when all memory requests are directed to a single node.

The application layer consists of user programs making use of IPC calls. The intercept layer serves to determine the nature of the IPC calls made, either local or remote operation. Local operation is performed locally while remote operation is carried out by sending a message to the remote node. The remote node will then perform the operation on behalf of the sender, and sending the result back as a reply. Figure 6 shows the actions taken when semget() is called. For shared memory, the IPC calls are intercepted and the local DIPCd performs all the requested operations.

The use of DIPC is available without making any modifications to existing source or executables. This is achieved through the use of dynamic library preloading that adds in the necessary code to perform remote operations for DIPC.

6.2. Cost of basic operations

The following tests are performed to determine the cost of DIPC calls as well as the time taken to read and write a shared memory page (of size 4096 bytes). All tests are performed on a cluster of PCs (450 MHz K6-2 with 128 MB memory) networked together by a 100 Mbps Ethernet switch. Table 4 shows the performance results.

IPC operation	**Cost (usec)**
Semaphore	
Semget (local)	48
Semop (local)	59
Semctl (local)	49
Semget (remote)	239
Semop (remote)	301
Semctl (remote)	239
Shared memory	
Shmget	131
Shmctl	11
Shmat	451
Shmdt	516
Read (single page)	1396
Write (single page)	10042

Table 4. Cost of basic operation for DIPC

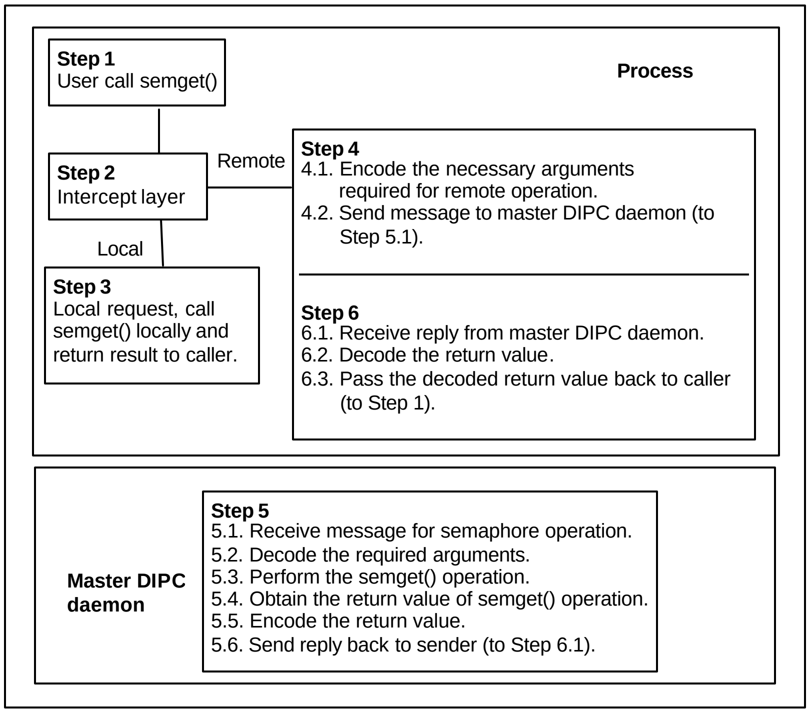

Step 1
User call semget()
Process
Step 2
Intercept layer
Remote
Local
Step 3
Local request, call semget() locally and return result to caller.
Step 4
4.1. Encode the necessary arguments required for remote operation.
4.2. Send message to master DIPC daemon (to Step 5.1).
Step 6
6.1. Receive reply from master DIPC daemon.
6.2. Decode the return value.
6.3. Pass the decoded return value back to caller (to Step 1).
Master DIPC daemon
Step 5
5.1. Receive message for semaphore operation.
5.2. Decode the required arguments.
5.3. Perform the semget() operation.
5.4. Obtain the return value of semget() operation.
5.5. Encode the return value.
5.6. Send reply back to sender (to Step 6.1).

Figure 6. Performing intercepted semaphore call

7. Distributed execution (forall)

7.1. Overview

We have extended the shell to recognize the `forall` construct. The `forall` is an extension of the `for` construct from `bash`. This allows the iterations within the `for` loop to execute concurrently and in a distributed manner.

We assume that the iterations within the loops are independent of one another. Hence, only statements within the loop are executed in a sequential manner.

The distribution of the loop iterations to various nodes for processing is based on the load information gathered by the Load Manager (LM). This ensures a better utilization of computing resources.

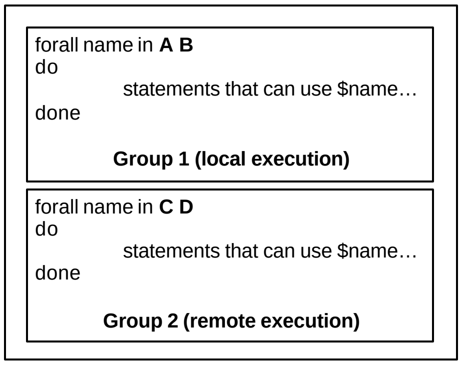
The local node will wait for all remote executions to complete before proceeding with the next statement after the `forall` loop. This ensures the correctness of the script execution. The following subsection describes about the modifications made to provide the `forall` feature.

7.2. Modifications to provide forall construct

The grammar rules for the `forall` construct are added to allow the shell to recognize the keyword `forall`. When the forall keyword is encountered during execution, the shell will requests from the LM a list of available nodes for remote execution. When no available nodes are available, forall will executes the iterations sequentially. For remote executions, the loop iterations are split such that each node will executes at least one iteration. Figures 7 and 8 show how the splitting of the loop iterations is done. Once the splitting is done, both the local and remote executions are able to proceed independently of one another.

```
forall name in A B C D
do
        statements that can use $name…
done
```

Figure 7. forall statements (before splitting)



```
forall name in A B
do
        statements that can use $name…
done
```

```
forall name in C D
do
        statements that can use $name…
done
```

Figure 8. forall statements (after splitting)

8. Conclusion and future work

Our implementation demonstrates that SSI can be achieved through a simple shell mechanism. It allows the user to execute existing binaries without any modifications, relinking or recompilation. The ability of the system to provide load distribution has the added advantage of utilizing both the cluster-wide resources and improving the response time for individual processes. With these attractive features and its ease of use, it will result in better user acceptance. Our current work involves the use of the implemented system to study the effects of different workload characteristics with respect to different load distribution policies.

Future work involves implementing a cluster-wide addressing scheme that will enable the user to access any resource in the cluster. This will also enable utilities like "ps" to list out all user's processes (both local and remote).

IPC calls could be handled in a distributed manner by intercepting the respective system calls. This concept is applicable to both sockets and pipes. However, to migrate processes that uses the above is much more tricky. As an example, we need to be able to access the data that is stored in the kernel for pipes during process migration. This will allow the restarted process to access the same data.

Signal handling is another area that requires special handling. As an example, a local SIGKILL signal generated by the user needs to be caught and send to the correct remote process.

9. References

[1] A. Barak, S. Guday, and R. Wheeler, The MOSIX Distributed Operating System, Load Balancing for UNIX. Lecture Notes in Computer Science, Vol. 672, Springer-Verlag, 1993.

[2] A. Bricker, M. Litzkow, and M. Livny, "Condor Technical Summary." University of Wisconsin-Madison Technical Report 1069. Oct 1991.

[3] T. Barr, N. Langfeldt, and S. Vidal, Network File System (NFS). http://www.linuxdoc.org/HOWTO/NFS-HOWTO/

[4] M. Baker, Cluster Computing White Paper. http://www.dcs.port.ac.uk/~mab/tfcc/WhitePaper/

[5] D.P. Ghormley, D. Petrou and S.H. Rodrigues, "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.

[6] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S.N. Zhou. Process migration survey. Collected papers, The Open Group Research Institute, March 1997. http://www.sdsc.edu/projects/production/NQE/SysAdmin/SysAdmin_Batch.html

[7] V. Sunderam, "PVM: A Framework for Parallel Distributed Computing." Concurrency: Practice and Experience, 2(4):315-339. Dec 1990.

[8] C. P. Tan, W.F. Wong, and C.K. Yuen, "tmPVM: Task Migratable PVM." Proc. of IPPS/SPDP 1999. pp. 196-202a. Apr 1999.

[9] Leen Dikken, Frank van der Linden, Joep J. J. Vesseur, and Peter M.A. Sloot, "DynamicPVM: Dynamic Load Balancing on Parallel Systems". Lecture Notes in Computer Science, High Performance Computing and Networking, Vol 797:273-277, Springer-Verlag, 1994

[10] R. Dornfest, Learning the JXTA Shell. http://www.openp2p.com/pub/a/p2p/2001/04/25/learning_jxta_shell.html

[11] S. Zhou, "LSF: Load Sharing in Large-scale Heterogeneous Distributed Systems." Proc. of Workshop on Cluster Computing, Dec 1992.

[12] Compaq Computer Corporation, Brian J. Watson and Bruce J. Walker, SSI for Linux, http://ssic-linux.sourceforge.net/index.shtml

[13] U. Vahalia. Unix Internals – The New Frontiers. Prentice Hall, 1996.

[14] Mark Ball, Sandy Bowers, Jackie Drane, Kevin Fought, Alice Gentry, Ron Goering, Daniel Nguyen, Susan Segura, and Johnny Shieh, Cluster Starter Kit for Linux. http://www.alphaWorks.ibm.com/tech/clusterstarterkit?open&l=TS040102,t=awfl

[15] Wrox Press, "Making P2P interoperable: The Jxta command shell", Sep 2001, http://www-106.ibm.com/developerworks/library/j-p2pint2/index.html

[16] GNU Project, "Bash shell". http://www.gnu.org/software/bash/bash.html

[17] Red Hat Inc, "Chapter 4: The /proc Filesystem", Red Hat Linux 7.2: The Official Red Hat Linux Reference Guide, http://www.redhat.com/docs/manuals/linux/RHL-7.2-Manual/ref-guide/ch-proc.html

[18] Scyld Computing Corporation, "Scyld Beowulf white paper", http://www.scyld.com/products/wpaper.pdf

[19] C. Newham and B. Rosenblatt, "Learning the bash shell." First Edition, O'Reilly & Associates, Inc, 1995.