Developing Simple Load Sharing Utilities (SLSU) for SWARM

by

Yan Zhao

A Research Paper

submitted to

Oregon State University

in partial fulfillment of the

requirements for the degree of

Master of Science

Completed: August 26, 1998

Commencement: June, 1999

Abstract

With the strong shift toward network-centric cluster-computing, workload management has become increasingly important in both industrial and academic environments. To fully harness the distributed computing resources of the 32 PC's in our department, I developed a workload monitoring system together with a few utilities for users to submit either sequential or parallel jobs to the PC cluster. The system schedules and runs the jobs and returns the results to the users.

ACKNOWLEDGMENTS

Much of the work presented in this report originated from stimulating discussions with Prof. Michael J. Quinn. His encouragement and direction have been invaluable throughout my project.

My thanks also go to Prof. Toshimi Minoura and my officemates Bill Langford and Alexey Malishevsky. Their comments have helped me to implement my project.

I greatly appreciate the support of my fiancée Haiyan Wang. She has borne the stress of many nights and weekends instead of going to a park.

Contents

1 Introduction **1**

2 Design **3**

2.1 Desirable Properties 3

2.2 Design Decisions 4

2.3 Design Details 5

2.3.1 Overall Structure 5

2.3.2 Master Server Election Algorithm and Fault Tolerance 6

2.3.3 Master Server Functions 8

2.3.4 Data Structures 9

3 Implementation **12**

3.1 Server Side 12

3.1.1 Techniques Used For the Servers 12

3.1.2 Load Information Server (LIS) 15

3.1.3 Master Server 16

3.2 Client Side 17

3.2.1 Job Submission 18

3.2.2 Histogramming Load Information 19

3.2.3 List Jobs/Users and Kill Jobs 20

4 Example Runs **21**

4.1 Submit Batch Jobs 22

4.2 Run Jobs Interactively 24

4.3 Display Aggregate Load 27

4.4 Other Utilities 27

4.5 Fault Tolerance 28

5 Future Work **32**

A Usage of Various Utilities **37**

B Source Code of Test Programs **39**

B.1 Sort 39

B.2 Sort1 41

B.3 Longtime 42

B.4 Getproc 42

B.5 Floyd 43

C Source Code of Server **46**

C.1 Header Files 46

C.2 Load Information and Master Server 48

C.3 Networking Library 70

C.4 Server Administration Utilities 73

C.4.1 Fireserver 73

C.4.2 Showserver 73

C.4.3 Killserver 74

C.4.4 Clearlog 74

D Source Code of Client **75**

D.1 Myplace 75

D.2 Myrun 76

D.3 Mylogin 77

D.4 Mysub 78

D.5 Listjob 79

D.6 Killjob 80

D.7 Listuser 81

D.8 Showload 82

List of Figures

2.1 Overall Structure of SLSU 6

2.2 LIS Election State Transition Diagram 7

2.3 Multi-threaded Master Server 9

2.4 Master Server Job Scheduling 11

3.1 Load Information Server (LIS) Flowing Chart 15

3.2 Master Server Election Algorithm 16

3.3 Client and Server Message Format 17

4.1 Showload Utility Screen Dump 1 27

4.2 Showload Utility Screen Dump 2 28

4.3 Showload Utility Screen Dump 3 29

Chapter 1

Introduction

Many computationally intensive tasks have been moved from conventional parallel computers to clusters of high-end workstations or PC's. Due to the limited computing resources, we often have to share our systems with other users. In a network environment, the network operating system usually does not take care of the workload management. This results in an imbalance of CPU load in the network of machines. Some machines are heavily loaded while others remain idle. The phenomenon takes place almost everywhere, in an academic department or an industrial enterprise. We often hear the complaints from other users: "You have a big job running there for days. Please ensure they are not run-aways". Most of the time, it is the system administrator's job to manually rearrange some jobs across the network to avoid hot spots. The reason for all those inconveniences is the lack of a workload management system.

Workload management assumes many of the traditional operating system responsibilities at the network level. It dynamically allocates user activities across the cluster to fully harness distributed computing resources.

There are a few commercial solutions for workload management, including LSF Suite from Platform Computing Corporation and ESP from Cybermation. These packages help users yield greater productivity and increase processing efficiency.

In our department, we have a cluster of 32 PC's connected by fast Ethernet. Each node is powered with an Intel Pentium II processor and 128 Mbytes memory. How to fully take advantage of the computing resources of the PC's is an important issue. My project has been to build a simple system which takes care of the user job submission and scheduling based on the load information of the cluster. I also need to provide a few utilities for the users:

- log in to node with lightest load
- run sequential job on node with lightest load, either interactive or batch
- run parallel job on a number of nodes with lightest load
- list jobs currently running
- kill jobs
- list users
- show aggregate system load

My solution has been to build a client/server system. Client and server communicate through message passing. The server runs on each node of the cluster, monitors the load information, waits for and processes the client's requests. The client is a group of utilities which make requests to the server to complete tasks.

Chapter 2

Design

2.1 Desirable Properties

Based on the initial requirements of this project, the following properties have been identified.

- *General Purpose*: The system should be usable for any network environment. It should handle not only sequential jobs but also parallel jobs. It also needs to provide interactive and batch modes.

- *Efficiency*: The system itself should not incur much overhead, so that the benefits of sharing can be maintained.

- *Extendibility*: The design should be modular and extendible. It should be easy to plug in more functions and add more features for future maintenance.

- *Fault Tolerance*: A critical requirement of a workload management or load sharing system in a network environment is fault tolerance. We do not want to shut down the whole system just because one of the nodes is down.

- *Scalability*: If the system requires that the number of nodes in the cluster be fixed, the load sharing system is almost useless. That is where scalability kicks in. To make the system work for a larger number of nodes, we do not want to introduce too much overhead.

- *Easy to use*: Users do not like a complex system for just running a simple job, so the interface to the utilities should be simple and easy to use.

In the design of Simple Load Sharing Utilities (SLSU), substantial efforts were made to realize the above properties.

2.2 Design Decisions

There are a number of design decisions made at the early design phase. First, I decided to build the system on top of the network protocol level. For efficiency, C was chosen as the programming language for SLSU server and TCP and UDP as the network protocol. Unix has a full range of C API's for network programming on top of TCP/IP. To make code simple and maintainable, the client side utilities are implemented using a mix of languages such as JAVA, C, and Unix Shell Script.

Second, in order not to introduce much network traffic and overhead, UDP was selected as the server-server communication protocol. This is because all the servers reside in the local area network and are hooked up by fast Ethernet. The chance of losing data across the network is very small. In addition, a lot of broadcast operations take place among the servers. TCP does not have a low-level broadcast API. Finally, a UDP packet does not keep as much information as a TCP packet does, such as check-sum, time-out etc., so UDP introduces less overhead.

Third, client and server communication protocol uses TCP. I do not want to restrict the job submission within local area network. Potentially, a user can submit his or her job or browse the system information from anywhere across the Internet, even through a web browser. TCP provides reliable transportation of messages across the wide area of network.

Fourth, I assume all the files reside in a network-based file system, such as NFS or AFS. All the nodes access a file through a uniform name space. Without uniform file access, load sharing would be restrictive and expensive because remotely executing tasks may reference files (using file names possibly embedded in their executables) that would either have to be transferred upon reference, or at least have to have their names translated on the fly to those usable on the execution host. *Condor* [8] chose the former approach by intercepting all the UNIX file system calls and sending them back to a shadow process on the original host.

2.3 Design Details

2.3.1 Overall Structure

In order to collect load and other resource information, it is necessary to have a server running on each host in the cluster. This server is called the *load information server* (LIS). To make the system simple, I chose a central control scenario. There is a central master server which runs on one of the hosts and keeps all the information. The master server also takes care of all the client requests. The *remote execution server* (RES) is responsible for the execution of application on local host. The system architecture is shown in Figure 2.1.

The arrowed lines in Figure 2.1 illustrate the communications between client and server. The client sends requests to the master server and receives the job running results from the

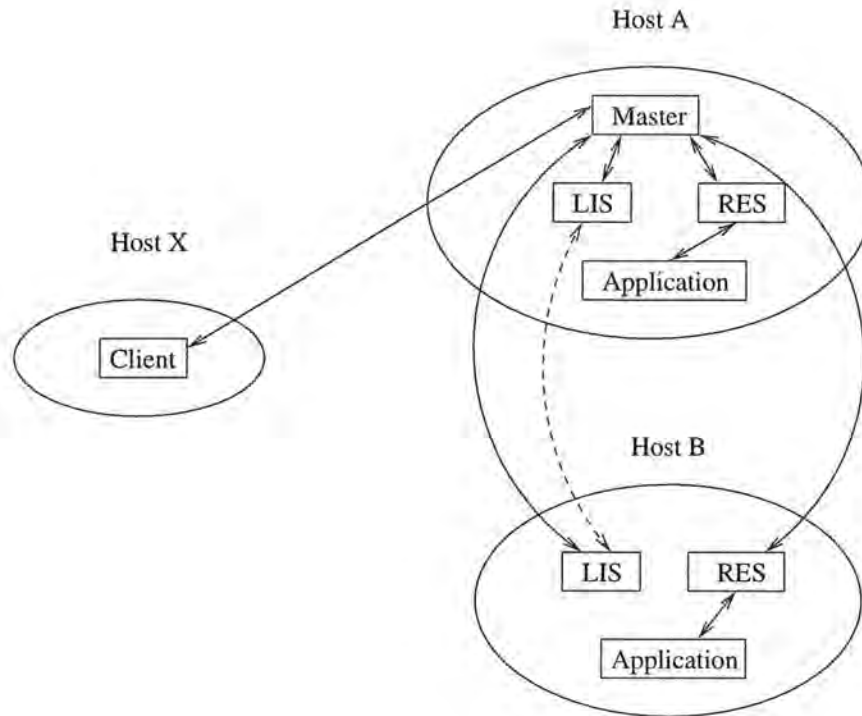

Figure 2.1: Overall Structure of SLSU

master server. LIS elects the master server upon initialization. The master server needs to get load information from LIS and broadcast its own heartbeat to LIS periodically. LIS listens to the master server's heartbeat, collects current system load information, and sends it to the master server.

2.3.2 Master Server Election Algorithm and Fault Tolerance

LIS elects the master server upon initialization or failure to receive the master server's heartbeat within a time interval. Theoretically, it can be shown that election in distributed systems with possible network and host failures cannot be guaranteed to succeed in any finite amount of time [7]. By studying related literatures, I chose an algorithm called the *bully* algorithm [3]. It is a simple algorithm used by many distributed systems when where a new copy of the coordinator process should be restarted.

Suppose an LIS does not receive master server's heartbeat within a time interval T1. In this situation, it assumes that master server has failed and tries to elect itself as the new master server. Upon initialization, each LIS gets its own ID from the configuration file. The ID is a unique integer number. If within a time interval T2 ($T2 > T1$), the LIS has not heard any election message from a host with larger ID than its own, it becomes the new master server and broadcasts a message to let the other LIS know that there is a new master server running. The state transition diagram of each LIS is shown in Figure 2.2. If a master server received a message from a host with larger ID, it knows that a higher priority host has joined in. It broadcasts a message saying that it is dying and kills itself. Then if the new host does not die in the next few moments, it will elect itself as the new master server. This is the drawback of this algorithm. The one with higher ID always bullies the one with lower ID. In the worst case, it can make the system paralyzed. However, this barely happens since it requires that the host with the largest ID keep crashing and rebooting.

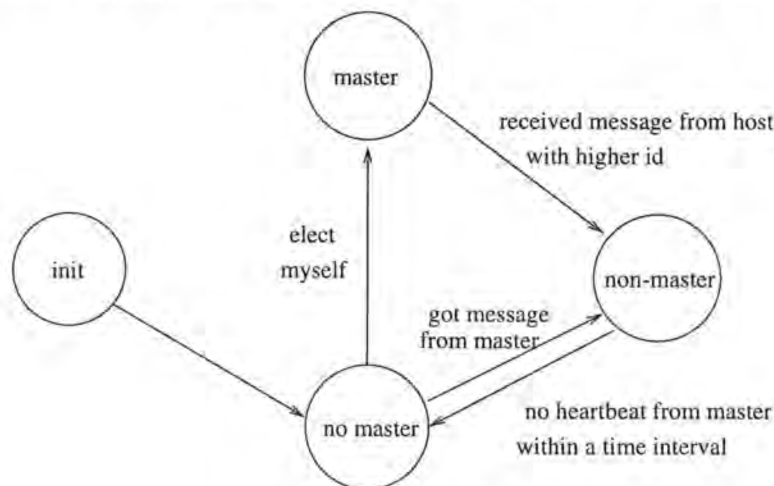

Figure 2.2: LIS Election State Transition Diagram

The *bully* algorithm ensures that there is no more than one master server running in the system at all times. To fulfill the fault tolerance property, the master server also has to

guarantee that it does not send jobs to any dead host. Actually the master server records the alive-dead information of each host in the cluster by periodically going through the load information sent from each LIS. Since each load information is time-stamped, the master server can check if any host did not send out its load information in the past time interval T3. If yes, it assumes that the host is dead and marks the host dead. Later on when the host comes back up, the master server will receive a load information message from the host eventually and mark it back to alive. When the master server fires up a job, it only sends it the host marked alive. If there are no enough hosts available, the server will inform the client of this.

2.3.3 Master Server Functions

The key functions on the server side are realized by the master server. There are three major functions, which are:

- process the client's requests;
- collect and record resource information from LIS;
- handle job start-up and return results to users.

The server can be designed as either iterative or concurrent server. An iterative server processes job submissions sequentially, while a concurrent server does the job in parallel. A concurrent server consumes more server resources, introducing more overhead. However, an iterative server performs poorly when there are many job submissions happening at the same time. I made a compromise during the design. Apparently, to make the system usable, those functions have to be done in parallel. Since Linux supports p-threads, I designed the

master server as multi-threaded. The three threads run in parallel and within each thread, things are done sequentially. The functions of each thread are shown in Figure 2.3.

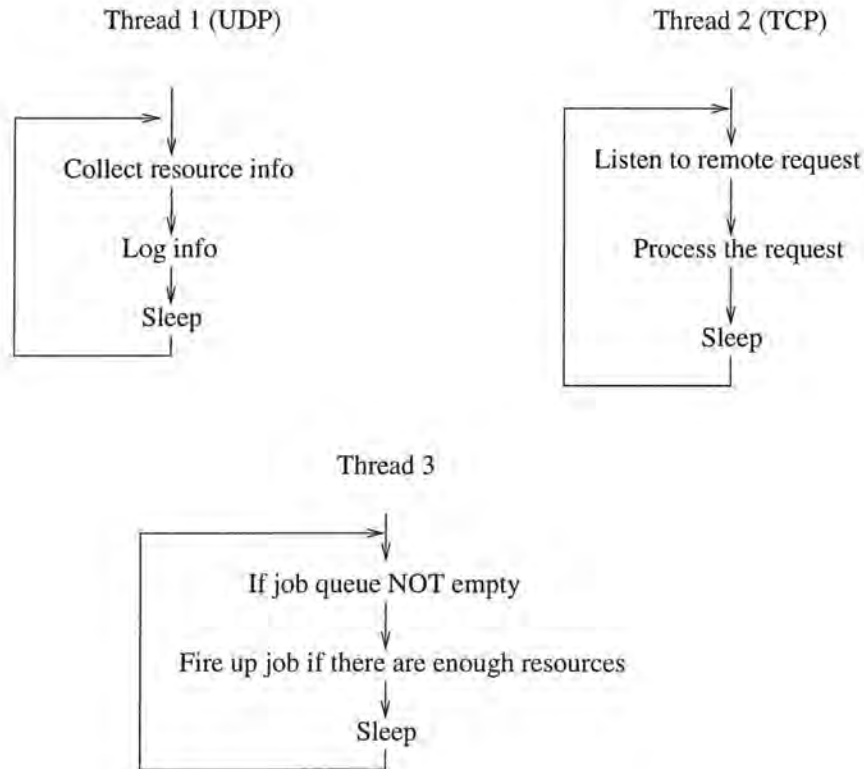

Figure 2.3: Multi-threaded Master Server

In the actual implementation, the third thread creates a separate thread dealing with each job and the main thread loops back to process the next job. Otherwise, one long running job will block the whole thread and the remaining jobs will not get assigned to processors.

2.3.4 Data Structures

There are three major data structures on the server side which deal with host status, job submission and scheduling, and user information, respectively. The status for each host is maintained in the following structure:

```
typedef struct Node {
    int id;           // node ID
    float aggregate_load[WINDOWSIZE];
    time_t lasttime;    // HAVE to use time() instead of clock()
    int available;      // 1 - available, 0 - unavailable
}NODE;
```

Field *lasttime* stores the most recent time when the master server heard from an LIS. The master server periodically scans the node list and checks this field to determine if a particular host is down. Only available hosts can be selected as hosts for the master server to send applications on to.

There are two queues inside the master server, which are the job waiting queue (JWQ) and the job running queue (JRQ). They are two single-linked lists. Each node contains all the information a job needs.

```
typedef struct Job {
    int id;
    int type;
    char email[LINELEN];
    int uid;
    int nproc;
    char currentdir[LINELEN];
    char cmdline[LINELEN];
    struct Job *next;
}JOB;
```

Field *type* shows if the job is sequential or parallel. Field *nproc* records how many processors a parallel job needs. For a sequential job, it is always 1. The detailed job

scheduling between these two queues is illustrated in Figure 2.4.

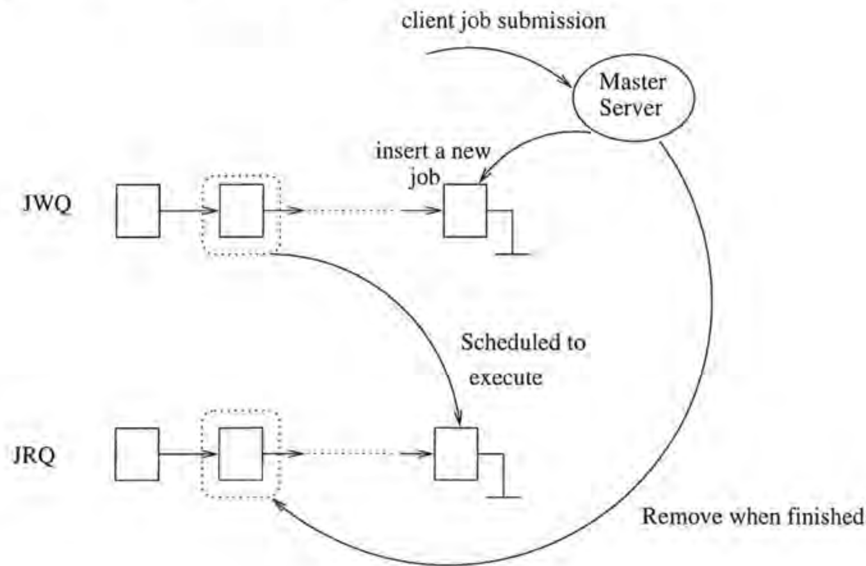

Figure 2.4: Master Server Job Scheduling

JWQ and JRQ are shared by a number of threads in the master server. One thread takes care of the job submission and inserts the new job into JWQ. Another thread scans JWQ and picks up a job which the system has enough resources to run and moves that node from JWQ to JRQ. Then another thread fires up this newly scheduled job and removes the node from JRQ when the job is finished. Access to JWQ and JRQ should be mutually excluded.

User information is used only for the *listuser* utility. It contains the user's name and the number of jobs currently in JWQ and JRQ submitted by the user.

The job queues and user list are shared by multiple threads. Therefore, each of them needs to have a mutex to guarantee mutual exclusion. The p-threads library provides *lock* and *unlock* operations on mutexes. To avoid deadlock, any thread can not apply for another mutex while holding a mutex.

Chapter 3

Implementation

3.1 Server Side

Due to the time constraint, I did not implement my own *remote execution server*. Instead, the master server makes a use of UNIX *rshd* as remote execution server. All the remote applications are started by calling *rsh* command. This only introduces overheads on the master server host because there might have many *rsh* processes running. However, taking advantage of UNIX's own server makes SLSU much more reliable.

3.1.1 Techniques Used For the Servers

To build a practical server on UNIX, I have learned a few techniques which make the server more reliable and closer to production software and applied them to my coding.

Programming a Daemon Server In UNIX, it can be done by having the server call *fork* to create a new process, and then arranging for the parent to exit. The advantage of doing so is the new process completely detaches from its parent. The *initial system*

process(init) will take over its parent's role. This will make the server exit cleanly, because *init* calls the UNIX system call *wait* to terminate each of its children.

Programming a Server to Avoid Multiple Copies Another commonly used technique in server programming is avoiding multiple copies running at the same time. One simple way to do this is creating a lock file. The following code does the trick:

```
/* acquire an exclusive lock, or exit, avoid multiple copies of LIS */
lockfile = open(LOCKLIS, O_RDWR|O_CREAT, 0640);
if (lockfile < 0) exit(1); /* error open lock file */
if (flock(lockfile, LOCK_EX|LOCK_NB)) {
printf("There is already a copy of LIS currently running\n");
/* could not obtain a lock, exit */
exit(0);
}
```

The mutual exclusion is realized by the system call *flock*. The nice thing is when the server crashes or the system reboots, the lock will be released. Otherwise, the second copy of the server cannot be started.

Broadcasting Message through UDP To make a UDP port broadcastable, we need to call the system functions *setsockopt* and *ioctl*. The following code segment shows how to turn a socket into a broadcast socket and associate it with a port and broadcast address.

```
int makeBroadcastCapable(int sock, struct sockaddr_in *bcastaddr, int port) {
   struct ifconf      ifc;
   struct ifreq       *ifr;
   int i = 1;
```

```
    /* turn the socket into a BROADCAST socket */
    setsockopt(sock, SOL_SOCKET, SO_BROADCAST, &i, sizeof(i));
    ... ...
    /* create a global broadcast address */
    ifc.ifc_len = sizeof(buffer);
    ifc.ifc_buf = buffer;
    ioctl(sock, SIOCGIFCONF, (char *) &ifc);
    ... ...
    /* Get and save interface address */
    ioctl(sock, SIOCGIFADDR, (char *) ifr);
    ... ...
    /* Get the interface broadcast address */
    ioctl(sock, SIOCGIFBRDADDR, (char *) ifr);
    ... ...
    bcopy((char *)&(ifr->ifr_broadaddr), (char *) bcastaddr, sizeof(ifr->ifr_broadaddr));
    /* Assign the port */
    bcastaddr->sin_port = htons(port);
    ... ...
}
```

The address stored in *bcastaddr* is the returned broadcast address. Any message sent to this address will be broadcast to all the nodes within the local area network. The UDP broadcast cannot cross the intra-net.

Besides the above techniques, all the timeout detections in the server are implemented using UNIX's synchronous I/O multiplexing, namely the system call *select*. For example, to make the server wait for a period of time for a message sent in through a socket, we can use the following C code:

```
fd_set readSet;
```

```
FD_ZERO(&readSet);
FD_SET(socketArray[i], &readSet);
... ...
somethingRead = select(biggestSocketNumber, &readSet, NULL, NULL, &theTime);
```

If within *theTime* period, the process has not heard anything from socket *socketArray[i]*, *select* will return 0. If the process heard anything, select will return immediately and return a positive value.

3.1.2 Load Information Server (LIS)

LIS is the server running on each host in the cluster. Its functions are shown in Figure 3.1.

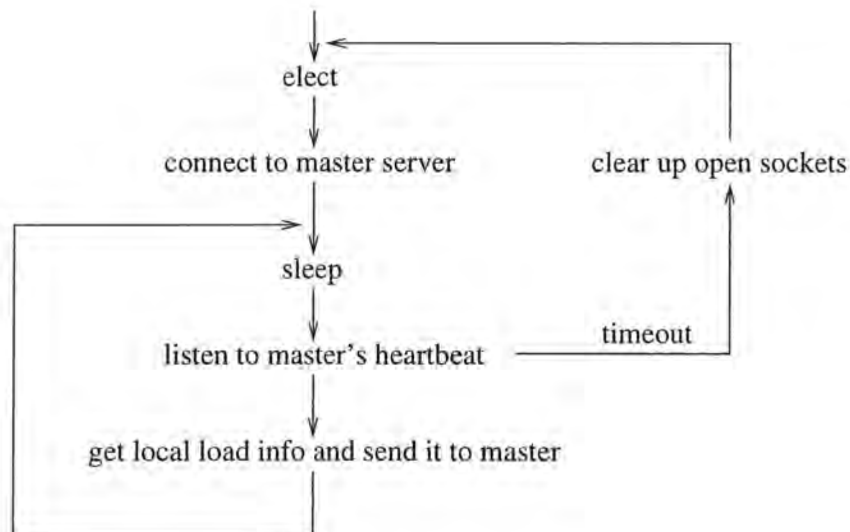

Figure 3.1: Load Information Server (LIS) Flowing Chart

The heart of LIS is the election algorithm. It is based on the *bully* algorithm discussed in the previous chapter. Every time the system starts up or any LIS detects that the master server died, the *election* subroutine is executed. It is illustrated in Figure 3.2.

Each LIS has a log file named as *log.hostname*. All the actions performed by LIS are

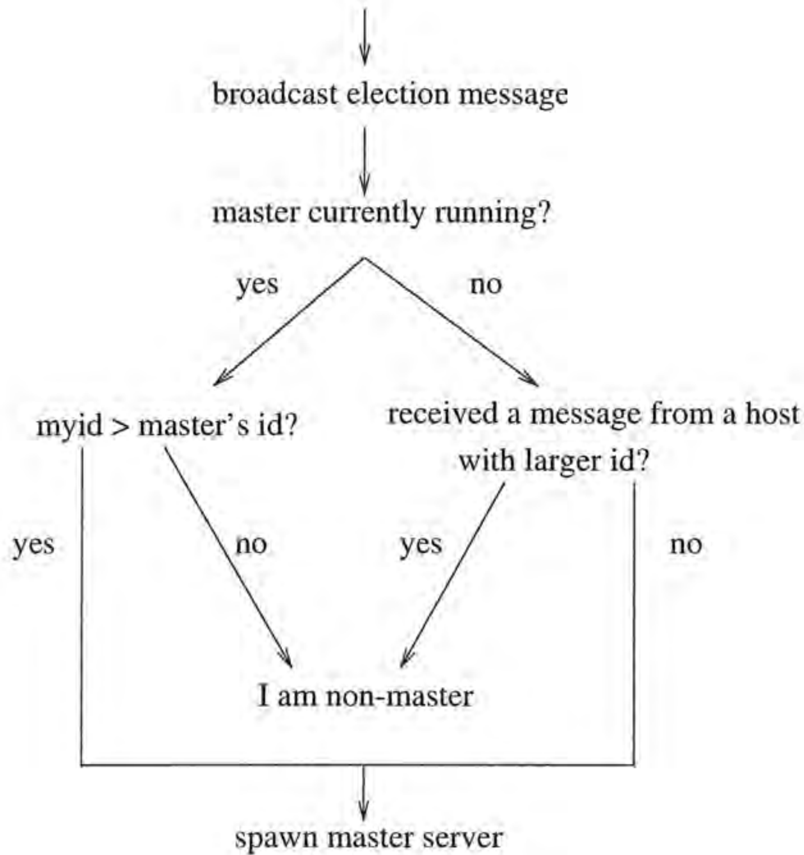

Figure 3.2: Master Server Election Algorithm

logged into this file.

3.1.3 Master Server

The master server maintains all the data structures discussed in Chapter 2 and takes care of all the server side responses to the client's requests. The thread which responds to the client's requests answers seven types of messages:

- *Sequential*: Sequential batch job submission. The master server inserts a new job node into the job waiting queue.

- *Parallel*: Parallel batch job submission. The master server inserts a new job node into the job waiting queue.
- *Myplace*: Request host(s) with lightest load.
- *Listjobs*: The master server returns the jobs currently in the job waiting queue and the job running queue.
- *Killjobs*: The master server removes a job (if exists) from the job waiting queue.
- *Listusers*: The master server returns the users information including the number of jobs currently in the queues for each user.
- *Showload*: The master server sends back to the client the aggregate load of the host that the client requests.

The master server logs all its actions with LIS into a log file called *master.log* and its actions with clients into a log file called *master.service*.

3.2 Client Side

The message format sent between the master server and the client side utilities is a character string. It is illustrated in Figure 3.3.

Job Type	E-mail	Uid	# of processors	Current Dir	Cmdline

Figure 3.3: Client and Server Message Format

Each client utility collects necessary information from the client machine and the user's account and packs them into a message and sends it to the master server. The seven types of

job are listed in the previous master server section. Field "*Uid*" is the real user ID returned by the system call *getuid*. This is the user's unique ID. All the user's other information is obtained from his/her *uid*. Field "*# of processors*" is the number of hosts needed for a parallel job. For a sequential job, it is always 1. Field "*Current Dir*" is the current directory where the user submits the job. Field "*Cmdline*" contains all the user's job name and its arguments.

3.2.1 Job Submission

This client part is implemented using C and UNIX Shell scripts. Batch job submission is written in C, since it involves packing up strings and communications with server. Interactive job submission is written in UNIX Shell script. It calls one of the utilities *myplace* to get the the number of processors available from the master server. It is illustrated as below:

```
set -- 'getopt pn: $*'
... ...
case $# in
0) echo 'Usage: myrun [-p -n num] job arg ...' 1>&2; exit 1
esac
... ...
for i
do
 case $i in
 -p) PFLAG=1; shift;;
 -n) nump=$2; shift 2;;
 --) shift; break;;
 esac
done
```

```
if test $PFLAG = 0
then
# sequential job
 cpath='pwd'
 host='myplace'
 ... ...
 rsh $host $cpath/$@
 ... ...
else
 ... ...
 if test $? = 1
 then
  cat $machiefile
 else
  ... ...
  mpirun -nolocal -np $nump -machinefile $machinefile $cpath/$@
  ... ...
```

Currently the server only recognizes MPI jobs. It fires up a parallel job using the utility *mpirun* provided by any MPI implementation. The detailed usage of the job submission utilities is shown in Appendix A.

3.2.2 Histogramming Load Information

This part is implemented using Java. It provides a user three types of information: master host name, available and unavailable hosts and the aggregate load from each available host. The program is multi-threaded. The main thread deals with the graphical user interface and the other thread connects to the master server and collects those three types of information from the server on the fly and passes them to the main thread.

3.2.3 List Jobs/Users and Kill Jobs

These three utilities are implemented very similarly. The only difference is they pass different job types and arguments to the master server. One issue I want to address here is the *killjob* function. Actually, the master server can only remove a job which has not been scheduled to execute. This is because on UNIX, at the process level, when the master server fires up a job, it only knows the job's main thread id. If the job, thereafter, spawns any other jobs or processes, there is no easy way for the master server to detect that. In this sense, killing a currently running job is nothing but killing only one of the processes of this job and it is much different from killing a process. Therefore, the current *killjob* utility can only remove a job which is not yet scheduled to run. I also tested one of the commercial workload management systems, LSF, running on SWARM for this function. I notice that LSF only gets rid of the main thread of my job when I try to kill my running job. The other threads of my jobs are still running. In my opinion, there is no good way to solve this problem above process level on UNIX. *Condor* [8, 9] gives one solution to this problem at the system level. However, as indicated in section 2.2, it is quite complicated.

Chapter 4

Example Runs

SLSU was tested on SWARM, which consists of 32 PC's connected by fast Ethernet to a CISCO 5505 switch. Each host is powered by a 300MHz Intel Pentium II processor and 128 Mbytes primary memory. The operating system is Red Hat Linux release 5.0.

Five test programs are used throughout this chapter. They are:

- *sort*: A sequential program which sorts a randomly generated integer array using quicksort and heapsort.

- *sort1*: A sequential program which does the same job as *sort* does. It contains interactive input and output. Its usage is: *sort1 <number of integers>*. If no command line argument is specified, it asks the user to enter a number. The heapsort result is optional.

- *longtime*: A dummy sequential program which runs about 30 seconds.

- *getproc*: A parallel program which runs on a number of hosts and prints out which process is run on which host.

- *floyd*: A parallel program which finds the all-pairs shortest paths of a given graph using Floyd's algorithm.

4.1 Submit Batch Jobs

A sample run of *sort* and sequential batch job submission are shown below:

```
bee19 ~/research/swarm/util 151% sort 20
Original list:
3 6 17 15 13 15 6 12 9 1 2 7 10 19 3 6 0 6 12 16
Sorted list: - Quicksort
0 1 2 3 3 6 6 6 6 7 9 10 12 12 13 15 15 16 17 19
Sorted list: - Heapsort
0 1 2 3 3 6 6 6 6 7 9 10 12 12 13 15 15 16 17 19
bee19 ~/research/swarm/util 152% mysub sort 20
```

After a while, I receive the following e-mail message from the server:

```
To: zhao@CS.ORST.EDU
Date: Fri, 24 Jul 1998 03:53:18 GMT
Job was run on bee20.CS.ORST.EDU, output looks like:
Original list:
3 6 17 15 13 15 6 12 9 1 2 7 10 19 3 6 0 6 12 16
Sorted list: - Quicksort
0 1 2 3 3 6 6 6 6 7 9 10 12 12 13 15 15 16 17 19
Sorted list: - Heapsort
0 1 2 3 3 6 6 6 6 7 9 10 12 12 13 15 15 16 17 19
```

A sample run of parallel job, *getproc*, is shown as follows:

```
bee19 ~/research/swarm/util 155% mpirun -np 5 getproc
0 is on bee19.CS.ORST.EDU
1 is on bee01.CS.ORST.EDU
2 is on bee02.CS.ORST.EDU
3 is on bee03.CS.ORST.EDU
4 is on bee04.CS.ORST.EDU
bee19 ~/research/swarm/util 156% mysub -p -n 5 getproc
bee19 ~/research/swarm/util 157%
```

The message received from the server later on is:

```
To: zhao@CS.ORST.EDU
Date: Fri, 24 Jul 1998 04:16:57 GMT
Job executed on host(s):
bee14.CS.ORST.EDU
bee28.CS.ORST.EDU
bee08.CS.ORST.EDU
bee09.CS.ORST.EDU
bee10.CS.ORST.EDU

Your job looked like:
------------------------------
/nfs/phantom/u1/zhao/research/swarm/util/getproc
------------------------------

The output (if any) follows:

3 is on bee09.CS.ORST.EDU
2 is on bee08.CS.ORST.EDU
4 is on bee10.CS.ORST.EDU
1 is on bee28.CS.ORST.EDU
0 is on bee14.CS.ORST.EDU
```

The average load information on the hosts which *mpirun* scheduled on before *getproc* was submitted is:

bee19	up 31 days, 8:22, load average: 1.09 1.09 1.01
bee01	up 31 days, 8:21, load average: 2.12 2.12 2.09
bee02	up 31 days, 8:21, load average: 2.53 2.54 2.50
bee03	up 31 days, 8:23, load average: 1.38 1.29 1.16
bee04	up 31 days, 8:22, load average: 1.04 1.04 1.05

The average load information on the hosts which *SLSU* scheduled on before *getproc* was submitted is:

bee14	up 11 days, 11:05, load average: 1.00 1.00 1.00
bee28	up 11 days, 11:05, load average: 1.00 1.00 1.00
bee08	up 9 days, 6:59, load average: 1.00 1.00 1.00
bee09	up 31 days, 8:22, load average: 1.00 1.00 1.00
bee10	up 31 days, 8:22, load average: 1.03 1.01 1.00

From the above two boxes, we can easily see that using workload management makes a better use of the network resources.

4.2 Run Jobs Interactively

This is an example run of a sequential job interactively:

```
bee08 ~/research/swarm/util 879% myplace
bee21.CS.ORST.EDU
bee08 ~/research/swarm/util 880% myrun sort1
Job executed on bee21.CS.ORST.EDU...
Your job looked like:
--------------------------------------------
/nfs/phantom/u1/zhao/research/swarm/util/sort1
--------------------------------------------

Enter the number of integers you want to sort: 20
Original list:
3 6 17 15 13 15 6 12 9 1 2 7 10 19 3 6 0 6 12 16
Sorted list: - Quicksort
0 1 2 3 3 6 6 6 6 7 9 10 12 12 13 15 15 16 17 19
Do you want to see the heapsort result (y/n)? y

Sorted list: - Heapsort
0 1 2 3 3 6 6 6 6 7 9 10 12 12 13 15 15 16 17 19
bee08 ~/research/swarm/util 881%
```

An interactive parallel job run is shown as below:

```
bee08 ~/research/swarm/util 916% myplace 1
bee21.CS.ORST.EDU
bee08 ~/research/swarm/util 917% myrun -p -n 1 floyd
Job executed on host(s):
bee21.CS.ORST.EDU

Your job looked like:

/nfs/phantom/u1/zhao/research/swarm/util/floyd


Enter the graph size: 8
Initial Adjacency Matrix D:
0 2 3 4 5 6 7 8
2 0 4 5 6 7 8 9
3 4 0 6 7 8 9 10
4 5 6 0 8 9 10 1
5 6 7 8 0 10 1 2
6 7 8 9 10 0 2 3
7 8 9 10 1 2 0 4
8 9 10 1 2 3 4 0
Final Results:
0 2 3 4 5 6 6 5
2 0 4 5 6 7 7 6
3 4 0 6 7 8 8 7
4 5 6 0 3 4 4 1
5 6 7 3 0 3 1 2
6 7 8 4 3 0 2 3
6 7 8 4 1 2 0 3
5 6 7 1 2 3 3 0
Do you want to see the elapsed time (y/n)? y
Elapsed Time: 0.00 sec
bee08 ~/research/swarm/util 918%
```

The user inputs are in italic. From the above two example runs, we can see that SLSU handles jobs with interactive inputs and outputs very well.

4.3 Display Aggregate Load

Screen dumps of running *showload* on one of the machines, *bee08*, are shown in Figure 4.1 - 4.3. I killed the LIS server on *bee19*. So in Figure 4.1, *bee19* is in different color, indicating that *bee19* is "dead". Pulling down the *File* menu in the main window of *showload* and clicking on the *master host* submenu, the user will get the window shown in Figure 4.2. Clicking on the *showload* submenu, the user will see the window shown in Figure 4.3.

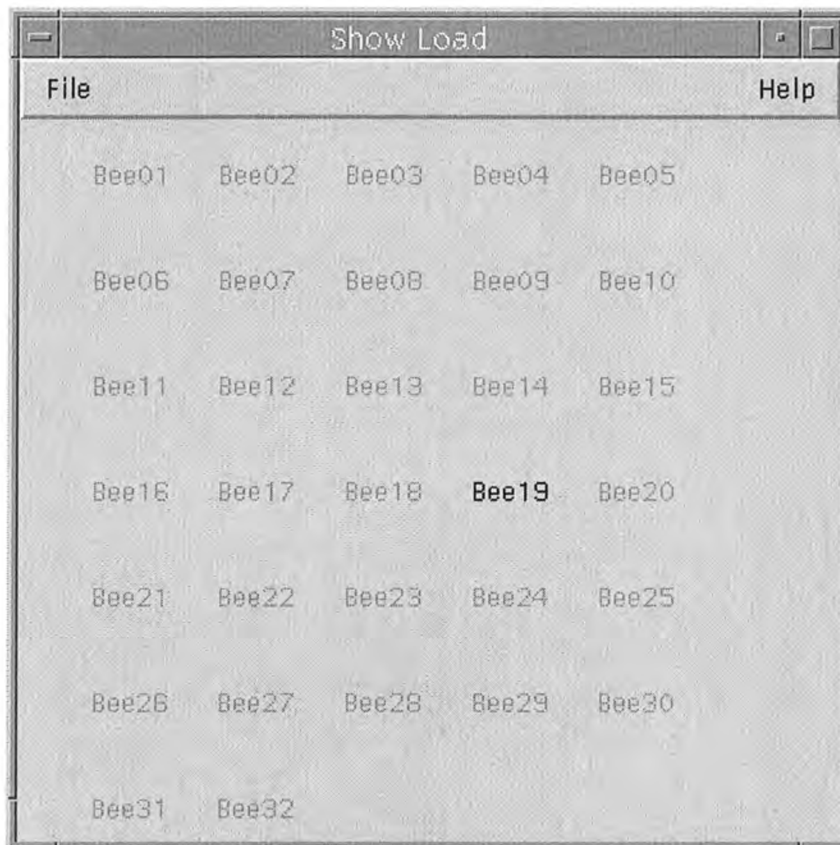

Figure 4.1: Showload Utility Screen Dump 1

4.4 Other Utilities

The following three boxes show a few example runs of *listjobs*, *listusers* and *listjobs*.

Figure 4.2: Showload Utility Screen Dump 2

```
bee08 ~/research/swarm/util 546% listjob
Jobs in the waiting queue
JOBID  UID  E-mail
2  11045  zhao@bee08.CS.ORST.EDU

Jobs in the running queue:
JOBID  UID  E-mail
0  11045  zhao@bee19.CS.ORST.EDU
1  11045  zhao@bee32.CS.ORST.EDU
```

```
bee08 ~/research/swarm/util 548% listuser
USER      NJOBS  PEND  RUN
godin       0     0     0
malishal    0     0     0
quinn       0     0     0
seelam      0     0     0
zhao        3     1     2
```

```
bee08 ~/research/swarm/util 549% killjob 5
Job 5 is not in the waiting queue
bee08 ~/research/swarm/util 550% killjob 2
Job 2 was removed from the waiting queue
```

4.5 Fault Tolerance

As shown in Figure 4.3, the master server can detect a host crash and mark the dead host so that it will not schedule any job on that host until that host reboots.

Figure 4.3: Showload Utility Screen Dump 3

Dealing with a master server crash is a little more complicated. A copy of the LIS (*bee03*) log file and master log file in the following figure shows when the master server (*bee32*) is down and the other hosts re-elect a new master server on host *bee31*.

```
bee19 ~/research/swarm/etc 182% cat log.bee03.CS.ORST.EDU
Log for LIS:
my id = 3
... ...
Master host name is: bee32.CS.ORST.EDU
master (bee32.CS.ORST.EDU) died, re-elect...
There is a master server currently running
I'm waiting for master to tell me its id
Master host name is: bee31.CS.ORST.EDU

bee19 ~/research/swarm/etc 183% cat master.id
bee31.CS.ORST.EDU

bee19 ~/research/swarm/etc 184% cat master.log
.... ...
master started on bee32.CS.ORST.EDU
I'm broadcasting my host name
master started on bee31.CS.ORST.EDU
I'm broadcasting my host name
```

Later on when *bee32* kicks back in, the master server on *bee31* detects there is a host with larger ID that has come back to live and quits. The LIS on *bee32* finds no other hosts have larger ID than its ID and then elects itself as the new master server according to the *bully* algorithm. The following is a part copy of the LIS log file on *bee31* and the master log file which shows the new change of master server.

```
bee08 ~/research/swarm/etc 556% cat master.log
master started on bee32.CS.ORST.EDU
I'm broadcasting my host name
master started on bee31.CS.ORST.EDU
I'm broadcasting my host name
I'm dying ...

master started on bee32.CS.ORST.EDU
I'm broadcasting my host name
bee08 ~/research/swarm/etc 557% cat log.bee31.CS.ORST.EDU
Log for LIS:
my id = 31
started master, pid = 23745
I'm waiting for master to tell me its id
Master host name is: bee31.CS.ORST.EDU
master changed
I'm waiting for master to tell me its id
Master host name is: bee32.CS.ORST.EDU
bee08 ~/research/swarm/etc 558%
```

Chapter 5

Future Work

There are a number of ways in which SLSU can be improved. These improvements will make it a better system more suitable for practical use.

- *Implement my own remote execution server instead of using rsh server.* The current implementation brings a little more overhead to the master server host when the master server fires up a sequential job. For a parallel job, the MPI daemon process acts as a remote execution server on each host. The remote execution server is another daemon process started with LIS and communicating with the master server through a port. Whenever the master server processes a job, it passes the job and its running environment to the remote execution server and lets it handle the job start-up and collecting of running results locally. The remote execution server notifies the master server upon the completion of the job and returns the running results to the submitted user through e-mail. Another field should be added into the message (see Figure 3.3) passed from a client to the master server. It records the user's environment when a user submits a job. Then the master server needs to pass this information to the

remote execution server to let it set up the user's environment before the job is started on this host by the remote execution server.

- *Adding more queues in the server.* Usually a job scheduling system has a number of queues for users to submit their jobs. One queue does not meet the needs for practical use. To make a better use of the cluster resources, we may want to schedule the computation intensive jobs during off-work hours, for example, after midnight. Some of the jobs may be urgent; the system should schedule those jobs to be executed right away. Adding more waiting queues is very easy in the current system design. One field, *queue name*, should be added into the message format communicated between master server and clients. On the master server side, more queues are initialized and jobs are inserted into different queues according to its queue name. The job scheduling thread will scan different queues in different ways. For example, it only scans the *midnight* queue after midnight and scans the *urgent* queue before everytime it scans the other queues. All the normal jobs are submitted into the *normal* queue. The system still keeps one job running queue.

- *Filling in more error processing modules.* There is not much error handling in the current implementation. Most of the time, the system just prints a message and exits. The system does not deal with the network message lost or errors. All the networking errors are the return messages from the C networking API. They are very hard to understand. The system would be better if it associated an error number with each error and dealt with network timeout.

- *Decreasing the network traffic between master server and LIS:* The *dead reckoning* [10] algorithm can be used to decrease the communication cost between the master server

and LIS. If the load of a host has not changed, the host does not need to send its load information to the master server. LIS only sends its load to the master when its current load is different from the last one it sent to the master. This brings up another problem: the master server does not know if a host is alive if it has not heard from a LIS for a period of time. The current design of the master server needs to be changed. When the master server collects alive-dead status of a host, it needs to *ping* the host to see if it is dead, rather than scanning the node list as in the current implementation. However, as a drawback, this slows down the master server's response time of starting up a job and replying to the clients.

- *Adding more client side utilities and GUI.* To make the system easier to maintain, it is better to provide graphical user interfaces (GUI) for the system administrator to monitor the system, view the system logs, restart and shutdown hosts. In the current implementation, only monitoring load information and available hosts have a GUI. I expect to have a Web interface for users in the future so that users even do not have to log on to our system to make a use of our resources. To implement this ambitious idea, the server has to take care of the security issues, and a CGI program needs to be added to act as a server to communicate with the Web browser and pass messages back and forth between the browser and the master server.

The testing of this system has been very limited. More usage may result in the discovery of more bugs. Bug fixing is absolutely always one of the main future tasks for any software system, including SLSU.

Bibliography

[1] D. Comer, *Internetworking with TCP/IP*, Vol. 3, Prentice Hall, 1993.

[2] Platform Computing Corp., *LSF Batch Administrator's Guide*, 5th Ed., 1997.

[3] A. Silberschatz and P. Galvin, *Operating System Concepts*, 4th Ed., Addison Wesley, 1994.

[4] B. Lewis and D. Berg, *Threads Primer*, Prentice Hall, 1996.

[5] D. Flanagan, *JAVA in a Nutshell*, 2nd Ed., O'Reilly, 1997.

[6] S. Zhou et.la, *UTOPIA: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems*, Technical Report CSRI-257, University of Toronto, April 1992.

[7] H. Garcia-Molina, *Elections in a distributed computing system*, IEEE Transactions on Computers, 32:48-59, January 1982.

[8] M. Litzkow, M. Livny, and M. Mutka, *Condor - a hunter of idle workstations*, Proceedings of the 8th International Conference on Distributed Computing Systems, p.104-111, San Jose, California, June 1988.

[9] A. Bricker, M. Litzkow and M. Livny, *CONDOR Technical Summary*, Technical report, Computer Sciences Department, University of Wisconsin-Madison, 1992.

[10] R. Pausch, et al, *An Introductory Tutorial for Developing Multi-User Virtual Environments*, Technical report, Computer Science Department, University of Virginia.

Appendix A

Usage of Various Utilities

In this appendix, all the utilities and servers are illustrated. They can all be executed from the command line on any machine in SWARM. The server start-up, display, kill and log file clear-up can be run from any host presumably.

udpserver — LIS and master server

mylogin — Log in to the host with the lightest load

myrun — Run sequential/parallel jobs interactively

Usage: *myrun [-p -n num] job arg ...*

mysub — Submit batch sequential/parallel jobs, the results will be e-mailed back

Usage: *mysub [-p -n num] job arg ...*

myplace — Show the host(s) with the lightest load

Usage: *myplace [number of hosts]*

showload — Show the aggregate load of the system graphically

listjob — List all the jobs in the waiting queue and the running queue

killjob — Kill job by its id. Only jobs in the waiting queue can be killed

Usage: *killjob <jobid>*

listuser — List the users and their job running status

fireserver — Fire up all the servers on SWARM

showserver — Show if the servers are running on each node

killserver — Shut down all the servers

clearlog — Remove all the log files created by the servers

Appendix B

Source Code of Test Programs

B.1 Sort

```
// sort.cc - test different sorting algorithms
//
// Author: Yan Zhao (zhao@cs.orst.edu)
// Date: Oct. 3, 1997
// Revised on: August 9, 1998

#include <iostream.h>
#include "sort.h"

int main(int argc, char **argv)
{
  int N, i;
  char ch;

  if (argc < 2) {
      cout << "usage: sort <# of integers>\n";
      exit(1);
  }
  else
      N = atoi(argv[1]);

  vector<int> a(N), b(N);
  for(i = 0; i < N; i++)
    a[i] = rand()%N;

  b = a;

  // print original list
  cout << "Original list:\n";
  print_vals(a);

  quicksort(a, 0, N-1);
  heapsort(b);

  // print sorted list
  cout << "\nSorted list: -- Quicksort\n";
  print_vals(a);

  cout << "\nSorted list: -- Heapsort\n";
  print_vals(b);

  return 0;
}

// sort.h - Various sort templates
//
```

```
// Author: Yan Zhao (zhao@cs.orst.edu)
// Date: Oct 3, 1997
// Revised: August 26, 1998

#include <stdlib.h>
#include <iostream.h>
#include <vector>

// swap two entries in a vector
template<class T>
void swap(vector<T>& a, int i, int j)
{
  T tmp;

  tmp = a[i];
  a[i] = a[j];
  a[j] = tmp;
}

// quicksort - originated from Sedgewick's Algorithm book
template<class T>
void quicksort(vector<T>& a, int l, int r)
{
  int i, j;
  T tmp;

  // partitioning
  if (r > l) {
    tmp = a[r];
    i = l - 1;
    j = r;
    for (;;) {
      while(a[++i] < tmp);
      while((a[--j] > tmp) && j > l); // add a left guard j > l to
                                      // make sure j does not go out of bound
                                      // Thank Prof. Budd for pointing out this
                                      // bug

      if (i >= j) break;
      swap(a, i, j);
    }

    swap(a, i, r);
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
  }
}

int heap_size;

// make sure the heap satisfies the heap property
template<class T>
void heapify(vector<T>& a, int i)
{
  int l, r, largest;

  l = 2*i;          // left child of i
  r = l + 1;        // right child of i

  if (l < heap_size && a[l] > a[i])
    largest = l;
  else
    largest = i;
```

```
  if (r < heap_size && a[r] > a[largest])
    largest = r;

  if (largest != i) {
    swap(a, i, largest);
    heapify(a, largest);
  }
}
// build heap
template<class T>
void build_heap(vector<T>& a)
{
  heap_size = a.size();

  for (int i = a.size()/2 - 1; i >= 0; i--)
    heapify(a, i);
}
// heapsort
template<class T>
void heapsort(vector<T>& a)
{
  build_heap(a);

  for (int i = heap_size - 1; i > 0; i--) {
    swap(a, 0, i);
    heap_size--;
    heapify(a, 0);
  }
}

// print out each entry in a vector
template<class T>
void print_vals(vector<T>& a)
{
  vector<T>::iterator i;

  for(i = a.begin(); i < a.end(); i++)
    cout << *i << ' ';
  cout << '\n';
}
```

B.2 Sort1

```
// sort1.cc - test different sorting algorithms interactively
//
// Author: Yan Zhao (zhao@cs.orst.edu)
// Date: Oct. 3, 1997
// Revised on: August 9, 1998

#include <iostream.h>
#include "sort.h"

int main(int argc, char **argv)
{
  int N, i;
  char ch;

  if (argc < 2) {
      cout << "Enter the number of integers you want to sort: ";
```

```
    cin >> N;
  }
  else
    N = atoi(argv[1]);

  vector<int> a(N), b(N);
  for(i = 0; i < N; i++)
    a[i] = rand()%N;

  b = a;

  // print original list
  cout << "Original list:\n";
  print_vals(a);

  quicksort(a, 0, N-1);
  heapsort(b);

  // print sorted list
  cout << "\nSorted list: -- Quicksort\n";
  print_vals(a);

  cout << endl << "Do you want to see the heapsort result (y/n)? ";
  cin >> ch;
  if (ch == 'y') {
     cout << "\nSorted list: -- Heapsort\n";
     print_vals(b);
  }

  return 0;
}
```

B.3 Longtime

```
/* a program runs a little longer */
#include <stdio.h>

int main()
{
   sleep(30);
   printf("Finally, I'm awake after a long sleep!\n");

   return 0;
}
```

B.4 Getproc

```
/* A simple MPI program which prints out which process is run on which
 * host
 *
 * $Author: zhao $
 * $Date: 1998/08/11 05:41:18 $
 * $Revision: 1.1 $
 */

#include <mpi.h>
#include <stdio.h>

void main(int argc, char **argv)
{
 int myid, nump, len = 32;
 char name[32];
```

```
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &nump);

  MPI_Get_processor_name(name, &len);

  printf("%d is on %s\n", myid, name);

  MPI_Finalize();
}
```

B.5 Floyd

```
/*
 * floyd.c - Floyd's all-pairs shortest paths algorithm
 *           Parallel Version using MPI
 *
 * Author: Yan Zhao (zhao@cs.orst.edu)
 * Date: July 9, 1997
 * Revised on: August 10, 1998
 */

#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>

#define MIN(a,b) ((a)<=(b)?(a):(b))

int **D, **D1, *tmpD, N, myid, nump;
int firstrow, lastrow, chunk;

void init(int);
void print_init();
void floyd();
void print_result();

int main(int argc, char **argv)
{
  double t;
  char buf[2], ch;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &nump);

  if (argc < 2) {
     if (myid == 0) {
     printf("Enter the graph size: ");
     fflush(stdout);
     scanf("%d", &N);
     scanf("%c", &ch);
     }
     MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
  }
  else
     N = atoi(argv[1]);

  init(N);

  print_init();

  MPI_Barrier(MPI_COMM_WORLD);
  t = MPI_Wtime();
```

```
  floyd();

  MPI_Barrier(MPI_COMM_WORLD);
  t = MPI_Wtime() - t;

  print_result();

  if (myid == 0) {
      printf("Do you want to see the elapsed time (y/n)? ");
      fflush(stdout);
      scanf("%c", &ch);
      if (ch == 'y')
      printf("Elapsed Time: %7.2f sec\n", t);
  }

  free(D);
  free(D1);
  free(tmpD);
  MPI_Finalize();
  return 0;
}
void print_init()
{
  int i, j;

  printf("\nInitial Adjacency Matrix D:\n");
  for(i = 0; i < chunk; i++) {
    for(j = 0; j < N; j++)
      printf("%3d", D[i][j]);
    printf("\n");
  }
  printf("\n");
}

void init(int n)
{
  int *tmpArray, i, j, seed;

  seed = myid + 1;
  chunk = n / nump;
  firstrow = myid * chunk;
  lastrow = firstrow + chunk;

  D = (int**)malloc(chunk*sizeof(int*));
  tmpArray = (int*)malloc(chunk*n*sizeof(int));
  for(i = 0; i < chunk; i++)
    D[i] = &tmpArray[i*n];

  D1 = (int**)malloc(chunk*sizeof(int*));
  tmpArray = (int*)malloc(chunk*n*sizeof(int));
  for(i = 0; i < chunk; i++)
    D1[i] = &tmpArray[i*n];

  tmpD = (int*)malloc(n*sizeof(int));

  for(i = 0; i < chunk; i++)
    for(j = 0; j < N; j++) {
      if(i+firstrow == j) D[i][j] = 0;
      else
    D[i][j] = (i + firstrow + j)%10 + 1;
    }
```

```
}
void floyd()
{
  int i, j, k, bcast_id;

  for(k = 0; k < N; k++) {
    /* broadcast kth row */
    bcast_id = k / chunk;
    if(bcast_id == myid) {
      MPI_Bcast(&D[k-firstrow][0], N, MPI_INT, bcast_id, MPI_COMM_WORLD);

      for(i = 0; i < chunk; i++)
    for(j = 0; j < N; j++)
      D1[i][j] = MIN(D[i][j], D[i][k]+D[k-firstrow][j]);
    } else {
      MPI_Bcast(tmpD, N, MPI_INT, bcast_id, MPI_COMM_WORLD);

      for(i = 0; i < chunk; i++)
    for(j = 0; j < N; j++)
      D1[i][j] = MIN(D[i][j], D[i][k]+tmpD[j]);
    }

    for(i = 0; i < chunk; i++)
      for(j = 0; j < N; j++)
    D[i][j] = D1[i][j];
  }
}
void print_result()
{
  int i, j;

  printf("Final Results:\n");
  for(i = 0; i < chunk; i++) {
    for(j = 0; j < N; j++)
      printf("%3d", D[i][j]);
    printf("\n");
  }
  printf("\n");
}
```

Appendix C

Source Code of Server

C.1 Header Files

```
/*
 * udpserver.h - header file for udpserver
 *
 * $Author: zhao $
 * $Date: 1998/07/21 04:42:01 $
 * $Revision: 1.4 $
 */

#define UNIXEPOCH 2208988800 /* UNIX epoch, in UCT sec */
#define LINELEN 128          /* default buffer size */
#define TIMEOUT -1           /* time out event */
#define MTIMEOUT 10.0        /* master timeout */
#define ELECTTIMEOUT 1.0     /* time master waits to check if there is a
            reelection */
#define LISDIEDTIMEOUT 30.0  /* if master hasn't heard anything from
            a LIS in this interval, it assumes
            the LIS node died */
#define BCAST_SOCKET 0       /* used in makeBroadcastPossible */
#define ELECT      100       /* message type for ELECT */
#define CHECKLISINTERVAL 30.0 /* interval master checks who are alive */
#define QLEN        10       /* queue length of master service port */

#define WINDOWSIZE 100
#define TOTALNODES 32        // total number of nodes
#define TOTALUSERS 5         // total number of users

// various job types
#define SEQUENTIAL 0
#define PARALLEL   1
#define MYPLACE    2
#define LISTJOBS   3
#define KILLJOBS   4
#define LISTUSERS  5
#define SHOWLOAD   6
//#define MYRUN    7

// LIS lock file
const char *LOCKLIS = "/tmp/LIS.lock";

const int AVAILABLE = 1;
const int UNAVAILABLE = 0;
const int BCAST_PORT1 = 10000;  /* master broadcast port # */
const int BCAST_PORT2 = 10001;  /* LIS broadcast port # */
const char *loadavg = "/proc/loadavg";
const char *logfilestub = "/nfs/phantom/u1/zhao/research/swarm/etc/log."; /* LIS log file */
```

```
const char *logmessage = "Log for LIS:\n";
const char *errormsg1 = "Error collect the load info\n";
const char *logmessage2 = "started master";
const char *mlogmessage = "master started on ";
const char *heartbeat = "I'm alive";
const char *mhostfile = "/nfs/phantom/u1/zhao/research/swarm/etc/master.id";
const char *mlogfile = "/nfs/phantom/u1/zhao/research/swarm/etc/master.log";
const char *configfile = "/nfs/phantom/u1/zhao/research/swarm/etc/config";
const char *hostidfile = "/nfs/phantom/u1/zhao/research/swarm/etc/hostid";
const char *mservicefile = "/nfs/phantom/u1/zhao/research/swarm/etc/master.service";
const char *myrun = "/nfs/phantom/u1/zhao/research/swarm/util/myrun";

const char *mport = "10000"; /* port used by master to send out
            heartbeat */
const char *lisport = "10001"; /* LIS broadcast port */
const char *serviceport = "10002"; /* master service port (TCP) */
const char *port1 = "12345"; /* port used by master to receive message from
           LIS */

extern int errno;

/* Linux defines sys_errlist as extern const char *const */
extern const char *const sys_errlist[];

typedef struct Node {
   int id;              // node id
   float aggregate_load[WINDOWSIZE];
   //clock_t lasttime; // last time heard from a LIS
   time_t lasttime;     // HAVE to use time() instead of clock()
   //int currentptr;    // 0 - WINDOWSIZE - 1
   int available;       // 1 - available
   //struct Node *prev;
   //struct Node *next;
}NODE;

NODE *NodeList[TOTALNODES];
int NumAvailableNodes;          /* total # of available nodes */
int AvailableNode[TOTALNODES]; /* available node's ids */

typedef struct Job {
   int id;
   int type;
   char email[LINELEN];
   int uid;
   int nproc;
   char currentdir[LINELEN];
   char cmdline[LINELEN];
   struct Job *next;
}JOB;

JOB *JobQHead = NULL; /* head and tail of job submission queue */
JOB *JobQTail = NULL;
JOB *JobRunQHead = NULL; /* head and tail of job
                   running queue */
JOB *JobRunQTail = NULL;
const int MaxJobs = 10000;   /* maximum number of batch jobs */
int CurrentJobId = 0;        /* current job id */

typedef struct User {        /* used by list user function */
   char name[LINELEN];
   int totaljobs;
   int runningjobs;
```

```
}USER;
USER UserList[TOTALUSERS];
static char *UserNames[] = {"godin", "malishal", "quinn", "seelam", "zhao"};
```

C.2 Load Information and Master Server

```
/*------------------------------------------------------------
 * udpserver - Load Information Server and Master Server
 *
 * $Author: zhao $
 * $Date: 1998/08/03 04:27:34 $
 * $Revision: 2.2 $
 *------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <netdb.h>
#include <assert.h>

#include <net/if.h>     /* for UDP broadcast */
#include <sys/ioctl.h>
#include <sys/time.h>     /* struct timeval */

#include <pwd.h>        /* with sys/types.h, for LISTUSERS */
#include <sys/file.h>  /* for flock() */

#include <pthread.h>
#include "udpserver.h"

static char rcsid[] = "$Id: udpserver.c,v 2.2 1998/08/03 04:27:34 zhao Exp zhao $";

struct sockaddr_in BroadcastAddress1, BroadcastAddress2;    /* address on which to broadcast */

char masterhost[LINELEN];
char myhostname[LINELEN];  /* host name */

int fd;
FILE *fplog, *mfplog;              /* LIS and master log file */
char logfile[LINELEN];

int msock, lisbsock, msock_recv, msock_service, sock;
int myid;                   /* my id, unique for each host */
int masterid;
//int alen = sizeof(BroadcastAddress);

/* declare mutexes */
pthread_mutex_t job_sub_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t job_run_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t user_q_mutex = PTHREAD_MUTEX_INITIALIZER;

/* function prototype */
void elect();
void master();
int makeBroadcastCapable(int, struct sockaddr_in*, int);
int checkAndProcess(int*, int, float);
int iAmFirst();
int getmyid(char*);
```

```
int idToHostname(int, char*);
int connectToMaster();
void master_init();
void masterCheckLis();
void update_node_list(int, float);
void master_finalize();
void job_handler();
void job_process();
void job_fire(void*);
int filter(char*);

/* concurrent TCP/UDP server for SWARM */
int main(int argc, char *argv[])
{
    struct sockaddr_in fsin;      /* the from address of a client */
    char *service = "12345";
    static char *greeting = "\n               Welcome to server!\n\n";
    time_t now;                   /* current time */
    int alen;                     /* from-address length */
    char *pts;
    FILE *fp, *fphost;
    char buf[LINELEN], reply[LINELEN];
    int i, msocArray[1];
    float curload;
    struct hostent *hent;
    int lockfile;

    /* acquire an exclusive lock, or exit, avoid multiple copies of
      LIS */
    lockfile = open(LOCKLIS, O_RDWR|O_CREAT, 0640);
    if (lockfile < 0) exit(1); /* error open lock file */
    if (flock(lockfile, LOCK_EX|LOCK_NB)) {
    printf("There is already a copy of LIS currently running\n");
    /* could not obtain a lock, exit */
    exit(0);
    }

    switch(argc) {
    case 1:
    break;
    case 2:
    service = argv[1];
    break;
    default:
    errexit("usage: udpserver [port]\n");
    }

    i = fork();
    if (i < 0)
    errexit("error when forking: %s\n", sys_errlist[errno]);

    if (i) exit(0); /* parent exit and child becomes a daemon */

    /*sock = passiveUDP(service);*/

    strcpy(logfile, logfilestub);
    gethostname(myhostname, LINELEN);
    strcat(logfile, myhostname);

    fplog = fopen(logfile, "w");
    if (fplog == NULL) {
    fprintf(stderr, "Error open log file\n");
```

```
exit(1);
}
fprintf(fplog, "%s", logmessage);
fflush(fplog);

/* get my unique id */
myid = getmyid(myhostname);
fprintf(fplog, "my id = %d\n", myid);
fflush(fplog);

msock = passiveUDP(mport); /* master broadcast port */
if (makeBroadcastCapable(msock, &BroadcastAddress1, BCAST_PORT1) == -1)
exit(1);

msocArray[0] = msock;

lisbsock = passiveUDP(lisport); /* LIS broadcast port */
if (makeBroadcastCapable(lisbsock, &BroadcastAddress2, BCAST_PORT2) == -1)
exit(1);

elect(); /* start up master */

sleep(5); /* wait for server to start up */

/*
fphost = fopen(mhostfile, "r");
if (fphost == NULL) {
fprintf(stderr, "error open master host file\n");
exit(1);
}
fscanf(fphost, "%s", masterhost);
fclose(fphost);
*/
/* connect to master */
sock = connectToMaster();

//fprintf(fplog, "msock = %d sock = %d\n", msock, sock);
while (1) {

sleep(2);

/* check if the master is still alive */
if (checkAndProcess(msocArray, 1, MTIMEOUT) == TIMEOUT) {
   // master died, log into log file
   fprintf(fplog, "master (%s) died, re-elect...\n", masterhost);
   fflush(fplog);

   // re-elect
   close(sock);
   elect();
   sleep(2);        /* wait for new server to start up */
   sock = connectToMaster();
}
   alen = sizeof(fsin);
/*bcopy(&BroadcastAddress, &fsin, alen);*/
//fprintf(fplog, "I'm receiving a message from %s\n", masterhost);
//fflush(fplog);

   if (recvfrom(msock, buf, LINELEN, 0, (struct sockaddr
                                    *)&fsin, &alen) < 0)
      errexit("recvfrom: %s\n", sys_errlist[errno]);
```

```
/*
if ((i = read(msock, buf, LINELEN)) > 0)
    buf[i] = '\0';
*/
//fprintf(fplog, "got a message %s from %s\n", buf, masterhost);
//fflush(fplog);

/* get the master's host name and log it into the log file
 */
hent = gethostbyaddr((char*)&fsin.sin_addr, 4, AF_INET);
if (hent != NULL) {
    //fprintf(fplog, "Master(%s) said %s\n", hent->h_name, buf);
    //fflush(fplog);
}

/* check if master changed */
if (strcmp(hent->h_name, masterhost)) {
    fprintf(fplog, "master changed\n");
    close(sock);
    //reconnect to new master
    sock = connectToMaster();
}

fp = fopen(loadavg, "r");
if (fp == NULL)
    /*write(fd, errormsg1, strlen(errormsg1));*/
    fprintf(fplog, "%s", errormsg1);
else {
    fscanf(fp, "%f", &curload);
    bzero(reply, LINELEN);
    sprintf(reply, "%d %.2f", myid, curload);
}

fclose(fp);

//fprintf(fplog, "Sending out load info\n");

// ***** fsin already stored the BCAST address, so the
// following line will do a bcast to msock, instead of sock
//sendto(sock, reply, strlen(reply), 0, (struct sockaddr*)&fsin,
//      sizeof(fsin));

write(sock, reply, strlen(reply));

fflush(fplog);
}

fclose(fplog);
return 0;
}
/* connect to master */
int connectToMaster()
{
    FILE *fphost;
    char buf[LINELEN];
    struct sockaddr_in fsin;
    int alen;
    struct hostent *hent;

    fprintf(fplog, "I'm waiting for master to tell me its id\n");
    fflush(fplog);
```

```
    //printf("%d: ======> 1\n", myid); fflush(stdout);
    if (recvfrom(msock, buf, LINELEN, 0, (struct sockaddr
                *)&fsin, &alen) < 0)
    errexit("recvfrom: %s\n", sys_errlist[errno]);

    /* get the master's host name and log it into the log file
     */
    hent = gethostbyaddr((char*)&fsin.sin_addr, 4, AF_INET);
    if (hent != NULL) {
    fprintf(fplog, "Master host name is: %s\n", hent->h_name);
    fflush(fplog);
    }

    /* record master's id */
    strcpy(masterhost, hent->h_name);

    /*
    fphost = fopen(mhostfile, "r");
    if (fphost == NULL) {
    fprintf(stderr, "error open master host file\n");
    exit(1);
    }
    fscanf(fphost, "%s", masterhost);
    fclose(fphost);
    */

    //printf("%d: ======> 2\n", myid); fflush(stdout);
    /* connect to master */
    return connectUDP(masterhost, port1);
}
/* get my unique id based on my hostname */
int getmyid(char *hostname)
{
    FILE *fp;
    int id;
    char host[LINELEN];

    fp = fopen(hostidfile, "r");
    assert(fp != NULL);

    while(fscanf(fp, "%d %s", &id, host) != EOF) {
    if (strcmp(host, hostname) == 0) break;
    bzero(host, LINELEN);
    }

    fclose(fp);

    return id;
}
/* id to hostname */
int idToHostname(int myid, char *hostname)
{
    FILE *fp;
    int id;
    char host[LINELEN];

    fp = fopen(hostidfile, "r");
    assert(fp != NULL);

    while(fscanf(fp, "%d %s", &id, host) != EOF) {
```

```
    if (myid == id) {
        strcpy(hostname, host);
        break;
    }
    bzero(host, LINELEN);
    }

    fclose(fp);

    if (strcmp(host, ""))
    return 0;
    else
    return 1;   // not find the hostname
}
/*-----------------------------------------------------------
 * checkAndProcess - check if there is any message came in. If no,
 *                   sleep until timeout
 *
 * timeout < 0 : blocks
 * timeout = 0 : returns immediately
 * timeout > 0 : waits until signal or time out
 *-----------------------------------------------------------*/
int checkAndProcess(int *socketArray, int arrayLength, float timeout)
{
    int stillRunning = 1;
    int i , biggestSocketNumber, somethingRead;
    fd_set readSet;
    struct timeval theTime;

    /*
     * clear the selection set, put in all the sockets and initialize the counter
     */
    FD_ZERO(&readSet);
    biggestSocketNumber = 0;

    for (i = 0; i < arrayLength; i++) {
    if (socketArray[i] > 0) {
        FD_SET(socketArray[i], &readSet);
        if (socketArray[i] > biggestSocketNumber)
        biggestSocketNumber = socketArray[i];
    }
    }
    biggestSocketNumber++;

    /*
     * set up to block on the select, or timeout
     */
    if (timeout < 0.0)  /* block */
        somethingRead = select(biggestSocketNumber, &readSet, NULL, NULL, NULL);

    else {
      theTime.tv_sec = (long) timeout;  /* long will truncate the float */
      theTime.tv_usec = (long)((timeout - (float) theTime.tv_sec)*1000000);

      somethingRead = select(biggestSocketNumber, &readSet, NULL, NULL, &theTime);
    }

    if (somethingRead > 0) {
    if (FD_ISSET(socketArray[BCAST_SOCKET], &readSet))
        stillRunning = 0; /* got a bcast message */

    /* else, other sockets are handled */
```

```
    }
    else {
    stillRunning = TIMEOUT;
    }
    return stillRunning;
}
/* check if I'm the first node in the node configuration file */
/*
int iAmFirst()
{
    FILE *fp;
    char node[LINELEN];
    fp = fopen(configfile, "r");
    assert(fp != NULL);
    fscanf(fp, "%s", node);
    fclose(fp);
    return (!strcmp(node, myhostname));
}
*/
/* use bully algorithm to elect a master */
int iAmFirst()
{
    char buf[LINELEN], body[LINELEN];
    int socArray1[1], socArray2[1], alen, id, msgtype, retvalue,
    check, len;
    struct sockaddr fsin;
    FILE *fphost;
    socArray1[0] = lisbsock;
    socArray2[0] = msock;
    /* broadcast election message */
    sprintf(buf, "%d %d %s", myid, ELECT, "election");
    sendto(lisbsock, buf, strlen(buf), 0, (struct
                sockaddr*)&BroadcastAddress2,
      sizeof(BroadcastAddress2));
    /* listen if there is a master already running and get its id */
    if (checkAndProcess(socArray2, 1, MTIMEOUT) != TIMEOUT) {
    // ***** BETTER USE message passing to determine master's id
    /* check the master id and compare with mine */
    fphost = fopen(mhostfile, "r");
    if (fphost == NULL) {
        fprintf(stderr, "error open master host file\n");
        exit(1);
    }
    fscanf(fphost, "%s", masterhost);
    fclose(fphost);
    /* I have higher priority than the current master */
    if (myid > getmyid(masterhost)) retvalue = 1;
    else retvalue = 0;
    }
    else { // no master currently running
    /* wait for any response from higher prioty node */
    while((check = checkAndProcess(socArray1, 1, MTIMEOUT)) != TIMEOUT) {
```

```
        //printf("%d: I'm here1\n", myid); fflush(stdout);
        bzero(buf, LINELEN);
        /*if (recvfrom(lisbsock, buf, LINELEN, 0, (struct sockaddr
                        *)&fsin, &alen) < 0)
        errexit("recvfrom: %s\n", sys_errlist[errno]);
        */
        len = read(lisbsock, buf, LINELEN);
        buf[len] = '\0';
        sscanf(buf, "%d %d %s", &id, &msgtype, body);
        if(id > myid) {
        retvalue = 0;
        break;
        }
    }
    if (check == TIMEOUT) retvalue = 1; /* I have the highest
                        priotiry */
    }
    //printf("%d: I'm here2\n", myid); fflush(stdout);
    return retvalue;
}

/* elect a new master server */
void elect()
{
    /*
    char *execArgv[3];
    FILE *fp;
    */
    int childid;

    /* no master running or I have the highest priority */
    if (iAmFirst()) {
    childid = fork();

    if (childid != 0) {
        /*fprintf(stderr, "I'm returning\n");*/
        fprintf(fplog, "%s, pid = %d\n", logmessage2, childid);
        fflush(fplog);
        return; /* parent return */
    }
    /*
      masterid= getpid();
      execArgv[0] = "master";
      execArgv[1] = "10000";
      execArgv[2] = 0;
      execvp("master", execArgv);
    */

    fclose(fplog);
    master();
    }
    /* there is a master server already running */
    fprintf(fplog, "There is a master server currently running\n");
    fflush(fplog);
    return;
}

/* master server */
void master()
{
```

```
struct sockaddr_in fsin;      /* the from address of a client */
int alen;                     /* from-address length */
char buf[LINELEN], reply[LINELEN], msgbody[LINELEN];
float curload;                /* current load */
struct hostent *hent;
FILE *fphost;
int socArray[1], id, msgtype, i;
//clock_t time_start, stop;
time_t t_start, t_stop;
pthread_t tid1, tid2;         /* service thread id's */
int return_val;           /* return value of the pthread
        functions*/

socArray[0] = lisbsock;
msock_recv = passiveUDP(port1);
msock_service = passiveTCP(serviceport, QLEN);
/* write down the master host id */
fphost = fopen(mhostfile, "w");
gethostname(masterhost, LINELEN);
fprintf(fphost, "%s", masterhost);
fclose(fphost);

mfplog = fopen(mlogfile, "a");
if (mfplog == NULL) {
fprintf(stderr, "Error open a file under /tmp\n");
exit(1);
}

fprintf(mfplog, "%s %s\n", mlogmessage, masterhost);
fflush(mfplog);

/* broadcast its id */
fprintf(mfplog, "I'm broadcasting my host name\n");
fflush(mfplog);
alen = sizeof(BroadcastAddress1);
sendto(msock, heartbeat, strlen(heartbeat), 0, (struct
                    sockaddr*)&BroadcastAddress1, alen);

//time_start = clock();
// *** WHY clock() returns 0???? Have to use time() which is not a
// ANSI C function:-(
t_start = time(&t_start);
master_init();

// create three threads, one deals with LIS, one with job
// submission and another with job processing
return_val = pthread_create(&tid1, (const pthread_attr_t *) NULL,
        (void*) job_handler, (void*) NULL);
assert(return_val == 0);
return_val = pthread_detach (tid1);
assert(return_val == 0);

return_val = pthread_create(&tid2, (const pthread_attr_t *) NULL,
        (void*) job_process, (void*) NULL);
assert(return_val == 0);
return_val = pthread_detach (tid2);
assert(return_val == 0);

while (1) {

sleep(2);
```

```
//fprintf(mfplog, "I'm sending out heartbeat\n");
//fflush(mfplog);
/* broadcast heartbeat */
alen = sizeof(BroadcastAddress1);
   sendto(msock, heartbeat, strlen(heartbeat), 0, (struct
                    sockaddr*)&BroadcastAddress1, alen);

/* listen if there is an election message */
if (checkAndProcess(socArray, 1, ELECTTIMEOUT) != TIMEOUT) {
   // need to avoid master eats its own message
   if (recvfrom(lisbsock, buf, LINELEN, 0, (struct sockaddr
                *)&fsin, &alen) < 0)
   errexit("recvfrom: %s\n", sys_errlist[errno]);
   sscanf(buf, "%d %d %s", &id, &msgtype, msgbody);
   //printf("I'm here\n");
   if (myid < id) { // a higher priority node kicks in
   fprintf(mfplog, "I'm dying ...\n\n");
   break;  // should kill myself here, current way to
         // deal with will leave a zombie
   }
}
// receive NumAvailableNodes times of load info
for (i = 0; i < NumAvailableNodes; i++) {
   //fprintf(mfplog, "I'm waiting for load info\n");
   //fflush(mfplog);

   /* receive load info */
   bzero(buf, LINELEN);
   if (recvfrom(msock_recv, buf, LINELEN, 0, (struct sockaddr
                     *)&fsin, &alen) < 0)
   errexit("recvfrom: %s\n", sys_errlist[errno]);

   // peel off client's id and load
   sscanf(buf, "%d %f", &id, &curload);

   /* get the LIS' host name and log it into the log file
    */
   hent = gethostbyaddr((char*)&fsin.sin_addr, 4, AF_INET);
   if (hent != NULL) {
   //fprintf(mfplog, "%s load: %.2f\n", hent->h_name, curload);
   //fflush(mfplog);
   }

   // update the client's load and sort the node list by its load
   // in ascending order
   update_node_list(id, curload);
}
//printf("hello\n");
//fflush(stdout);

// check who are alive at an interval
t_stop = time(&t_stop);
//printf("stop = %d, time_start = %d\n", t_stop, t_start);
if (t_stop - t_start > CHECKLISINTERVAL) {
   //printf("master checking LIS...\n"); fflush(stdout);
   masterCheckLis();

   // log the available nodes
   /*
   fprintf(mfplog, "Available nodes: ");
```

```
        for(i = 0; i < NumAvailableNodes; i++)
        fprintf(mfplog, "%d(%.2f) ", AvailableNode[i],
            NodeList[AvailableNode[i] - 1]->aggregate_load[WINDOWSIZE-1]);

        fprintf(mfplog, "\n");
        */
        //time_start = clock();
        t_start = t_stop;
    }
    fflush(mfplog);
    }

    master_finalize();
    _exit(1);
}
/* convert + to space */
int filter(char* str)
{
    int i;

    for(i = 0; i < strlen(str); i++)
    if (str[i] == '+') str[i] = ' ';

    return 0;
}
// job handling routine
// for easy debugging, implemented as iterative TCP server
// Better change to concurrent server
void job_handler()
{
    int jsock;
    struct sockaddr_in fsin;
    int alen, n;
    FILE *fpservice;
    char buf[LINELEN], message[8*LINELEN], reply[8*LINELEN];
    struct hostent *hent;
    int jobtype, i;
    float load;
    JOB *new_job, *onejob;

    fpservice = fopen(mservicefile, "w");
    assert(fpservice != NULL);

    //printf("job_handler: hello1\n");
    //fflush(stdout);

    while (1) {
    jsock = accept(msock_service, (struct sockaddr*)&fsin, &alen);
    assert(jsock >= 0);

    /* get the client's host name and log it into the log file */
    /*
    hent = gethostbyaddr((char*)&fsin.sin_addr, 4, AF_INET);
    assert(hent != NULL);
    */
    //printf("job_handler: hello2\n"); fflush(stdout);
    bzero(message, 8*LINELEN);
    //while ((n = read(jsock, buf, LINELEN)) > 0) {
    n = read(jsock, buf, LINELEN);
        buf[n] = '\0';
```

```
   strcat(message, buf);
   //}
/*
fprintf(fpservice, "Got message from %s: %s\n", hent->h_name,
   message);
*/
fprintf(fpservice, "Got message: %s\n", message);
fflush(fpservice);

sscanf(message, "%d", &jobtype);
switch(jobtype) {
case SEQUENTIAL: {
   new_job = (JOB*)malloc(sizeof(JOB));
   assert(new_job != NULL);

   // fill in the new job entry
   new_job->id = CurrentJobId++;
   if (CurrentJobId == MaxJobs) CurrentJobId = 0;

   sscanf(message, "%d %s %d %s %s", &new_job->type,
      new_job->email, &new_job->uid, new_job->currentdir,
      new_job->cmdline);
   new_job->nproc = 1;
   filter(new_job->cmdline);
   new_job->next = NULL;

   // insert the request into job submission queue
   pthread_mutex_lock(&job_sub_mutex);

   if (JobQTail != NULL) {
   JobQTail->next = new_job;
   JobQTail = new_job;
   } else // first node
   JobQHead = JobQTail = new_job;

   // release lock
   pthread_mutex_unlock(&job_sub_mutex);

   break;
}
case PARALLEL: {
   new_job = (JOB*)malloc(sizeof(JOB));
   assert(new_job != NULL);

   // fill in the new job entry
   new_job->id = CurrentJobId++;
   if (CurrentJobId == MaxJobs) CurrentJobId = 0;

   sscanf(message, "%d %s %d %d %s %s", &new_job->type,
      new_job->email, &new_job->uid, &new_job->nproc,
      new_job->currentdir, new_job->cmdline);
   filter(new_job->cmdline);
   new_job->next = NULL;

   // insert the request into job submission queue
   pthread_mutex_lock(&job_sub_mutex);

   if (JobQTail != NULL) {
   JobQTail->next = new_job;
   JobQTail = new_job;
   } else // first node
   JobQHead = JobQTail = new_job;

   // release lock
```

```
    pthread_mutex_unlock(&job_sub_mutex);

    break;
}
case MYPLACE: {
    int nump;
    char err[] = "no enough nodes available\n";

    sscanf(message, "%d %d", &jobtype, &nump);
    //printf("nump = %d NumAvailableNodes = %d\n", nump,
    //      NumAvailableNodes);
    //fflush(stdout);
    if (nump > NumAvailableNodes)
    write(jsock, err, strlen(err));
    else {
    bzero(message, 8*LINELEN);
    for(i = 0; i < nump; i++) {
        bzero(buf, LINELEN);
        idToHostname(AvailableNode[i], buf);
        strcat(message, buf);
        strcat(message, "\n");
    }
    write(jsock, message, strlen(message));
    }
    break;
}
case LISTJOBS: {
    bzero(reply, LINELEN*8);

    // list waiting queue
    pthread_mutex_lock(&job_sub_mutex);
    if (JobQHead == NULL)
    sprintf(reply, "No job in the waiting queue\n");
    else {
    sprintf(reply, "Jobs in the waiting queue:\n JOBID UID E-mail\n");
    onejob = JobQHead;
    while (onejob != NULL) {
        sprintf(buf, "%5d %3d %s\n", onejob->id,
            onejob->uid, onejob->email);
        strcat(reply, buf);
        onejob = onejob->next;
    }
    }
    pthread_mutex_unlock(&job_sub_mutex);

    // list running queue
    pthread_mutex_lock(&job_run_mutex);
    if (JobRunQHead == NULL)
    strcat(reply, "\n\nNo job in the running queue\n");
    else {
    strcat(reply, "Jobs in the running queue:\n JOBID UID E-mail\n");
    onejob = JobRunQHead;
    while (onejob != NULL) {
        sprintf(buf, "%5d %3d %s\n", onejob->id,
            onejob->uid, onejob->email);
        strcat(reply, buf);
        onejob = onejob->next;
    }
    }
    pthread_mutex_unlock(&job_run_mutex);
```

```
        // send results back to client
        write(jsock, reply, strlen(reply));

        break;
    }
    case KILLJOBS: {
        int jobid;
        JOB *prevjob;

        sscanf(message, "%d %d", &jobtype, &jobid);
        pthread_mutex_lock(&job_sub_mutex);
        onejob = JobQHead;
        while (onejob != NULL && onejob->id != jobid) {
        prevjob = onejob;
        onejob = onejob->next;
        }

        if (onejob == NULL) // not found this job
        sprintf(reply, "Job %d is not in the waiting queue\n",
            jobid);
        else { // remove this job
        if (onejob != JobQHead)
            prevjob->next = onejob->next;
        else {
            JobQHead = onejob->next;
            if (JobQHead == NULL)
            JobQTail = NULL;
        }
        free(onejob);
        sprintf(reply, "Job %d was removed from the waiting queue\n", jobid);
        }

        pthread_mutex_unlock(&job_sub_mutex);

        write(jsock, reply, strlen(reply));
        break;
    }
    case LISTUSERS: {
        struct passwd *password;

        // scan waiting queue
        pthread_mutex_lock(&job_sub_mutex);

        onejob = JobQHead;
        while (onejob != NULL) {
        password = getpwuid(onejob->uid);
        for (i = 0; i < TOTALUSERS; i++)
            if (strcmp(UserList[i].name, password->pw_name) ==
            0) {
            UserList[i].totaljobs++;
            break;
            }
        onejob = onejob->next;
        }
        pthread_mutex_unlock(&job_sub_mutex);

        // scan running queue
        pthread_mutex_lock(&job_run_mutex);

        onejob = JobRunQHead;
        while (onejob != NULL) {
        password = getpwuid(onejob->uid);
```

```
        for (i = 0; i < TOTALUSERS; i++)
            if (strcmp(UserList[i].name, password->pw_name) ==
            0) {
            UserList[i].totaljobs++;
            UserList[i].runningjobs++;
            break;
            }
        onejob = onejob->next;
        }
        pthread_mutex_unlock(&job_run_mutex);

        // reply to client
        bzero(reply, LINELEN*8);
        sprintf(reply, "    USER  NJOBS PEND RUN\n");
        for(i = 0; i < TOTALUSERS; i++) {
        bzero(buf, LINELEN);
        sprintf(buf, "%10s%4d  %4d  %4d\n", UserList[i].name,
            UserList[i].totaljobs, UserList[i].totaljobs -
            UserList[i].runningjobs,
            UserList[i].runningjobs);
        strcat(reply, buf);
        UserList[i].totaljobs = UserList[i].runningjobs = 0;
        }
        write(jsock, reply, strlen(reply));

        break;
    }
    case SHOWLOAD:
        bzero(reply, LINELEN*8);
        for (i = 0; i < TOTALNODES; i++) {
        bzero(buf, LINELEN);
        if (NodeList[i]->available == AVAILABLE)
            load = NodeList[i]->aggregate_load[WINDOWSIZE-1];
        else
            load = -1;
        sprintf(buf, "%.2f\n", load);
        //strcat(reply, buf);
        write(jsock, buf, strlen(buf));
        }
        //write(jsock, reply, strlen(reply));
        break;
    default:
    }

    close(jsock);
    }
}

// job processing thread
void job_process()
{
    JOB *onejob, *prevjob;
    pthread_t tid;
    int return_val;

    while (1) {
    sleep(2);
    // acquire the mutex of job submission queue
    pthread_mutex_lock(&job_sub_mutex);
    if (JobQHead == NULL) { // job submission queue empty
```

```
        // yield to other thread, POSIX.1b, not Pthreads:-((
        pthread_mutex_unlock(&job_sub_mutex);
        sched_yield();
        sleep(2);
    }
    else {
        //pthread_mutex_lock(&job_sub_mutex);

        onejob = JobQHead;

        // go thru the job list to find a job which requires the
        // resources currently system can provide
        while (onejob != NULL && onejob->type == PARALLEL &&
          onejob->nproc > NumAvailableNodes) {
        prevjob = onejob;
        onejob = onejob->next;
        }

        if (onejob != NULL) { // there is a job can be run
        // remove the job from the submission queue
        prevjob->next = onejob->next;

        if (onejob == JobQHead)
            JobQHead = JobQHead->next;
        if (JobQHead == NULL) JobQTail = NULL;
        }

        // release the mutex of job submission queue
        pthread_mutex_unlock(&job_sub_mutex);

        // add the job into job running queue
        if (onejob != NULL) {
        // acquire job running queue mutex
        pthread_mutex_lock(&job_run_mutex);

        onejob->next = NULL;
        if (JobRunQTail != NULL) {
            JobRunQTail->next = onejob;
            JobRunQTail = onejob;
        }
        else
            JobRunQHead = JobRunQTail = onejob;

        // release job running queue mutex
        pthread_mutex_unlock(&job_run_mutex);

        // create a thread which fires up this job
        return_val = pthread_create(&tid, (const pthread_attr_t *) NULL,
                    (void*)job_fire, (void*)onejob);
        assert(return_val == 0);
        return_val = pthread_detach (tid);
        assert(return_val == 0);
        }
    }
    }
}
// job fire up thread
void job_fire(void* arg)
{
   JOB *onejob = (JOB*) arg;
   JOB *prevjob, *currentjob;
```

```
char hosts[TOTALNODES][LINELEN], msg[LINELEN*6], ofile[LINELEN],
mfile[LINELEN], rcmd[LINELEN*2], buf[LINELEN];
const char *job_output = "/tmp/slsu_job";
const char *machine_file = "/tmp/machinefile";
int uid, i;
FILE *fp;

if (onejob->type == SEQUENTIAL) { // sequential job
idToHostname(AvailableNode[0], hosts[0]);

bzero(msg, LINELEN*6);
strcat(msg, "Job was run on ");
strcat(msg, hosts[0]);
strcat(msg, ", output looks like:\n\n");

//system("rm -f /tmp/seq.job");
sprintf(ofile, "%s.%d", job_output, pthread_self());
fp = fopen(ofile, "w");
assert(fp != NULL);
fprintf(fp, "%s", msg);
fclose(fp);

// *** NEED to set uid here
bzero(rcmd, LINELEN*2);
strcat(rcmd, "rsh ");
strcat(rcmd, hosts[0]);
strcat(rcmd, " ");
strcat(rcmd, onejob->currentdir);
strcat(rcmd, "/");
strcat(rcmd, onejob->cmdline);
strcat(rcmd, " >> ");
strcat(rcmd, ofile);
system(rcmd);
bzero(rcmd, LINELEN*2);
strcat(rcmd, "mail ");
strcat(rcmd, onejob->email);
strcat(rcmd, " < ");
strcat(rcmd, ofile);
system(rcmd);

// remove output file from /tmp
bzero(rcmd, LINELEN*2);
strcat(rcmd, "rm -f ");
strcat(rcmd, ofile);
system(rcmd);
}
else { // Parallel job
// generate machine file for mpirun
sprintf(mfile, "%s.%d", machine_file, pthread_self());
fp = fopen(mfile, "w");
assert(fp != NULL);
for (i = 0; i < onejob->nproc; i++) {
   idToHostname(AvailableNode[i], hosts[i]);
   fprintf(fp, "%s\n", hosts[i]);
}
fclose(fp);

sprintf(ofile, "%s.%d", job_output, pthread_self());
fp = fopen(ofile, "w");
assert(fp != NULL);
```

```
    bzero(msg, LINELEN*6);
    strcat(msg, "Job executed on host(s):\n");
    for (i = 0; i < onejob->nproc; i++) {
        strcat(msg, hosts[i]);
        strcat(msg, "\n");
    }
    strcat(msg, "\nYour job looked like:\n");
    sprintf(buf, " %s/%s\n", onejob->currentdir, onejob->cmdline);
    strcat(msg, "-----------------------------------------------------\n");
    strcat(msg, buf);
    strcat(msg, "-----------------------------------------------------\n");
    strcat(msg, "\nThe output (if any) follows:\n");

    fprintf(fp, "%s", msg);
    fclose(fp);

    bzero(rcmd, LINELEN*2);
    sprintf(rcmd, "mpirun -nolocal -np %d -machinefile %s %s/%s >> %s", onejob->nproc, mfile, onejob->currentdi
onejob->cmdline, ofile);

    // *** NEED to set uid here
    system(rcmd);

    // main results back
    bzero(rcmd, LINELEN*2);
    strcat(rcmd, "mail ");
    strcat(rcmd, onejob->email);
    strcat(rcmd, " < ");
    strcat(rcmd, ofile);
    system(rcmd);

    // remove output file and machine file from /tmp
    bzero(rcmd, LINELEN*2);
    strcat(rcmd, "rm -f ");
    strcat(rcmd, ofile);
    strcat(rcmd, " ");
    strcat(rcmd, mfile);
    system(rcmd);
    }

    // remove the job from running queue
    pthread_mutex_lock(&job_run_mutex);

    currentjob = JobRunQHead;
    while (currentjob != onejob) {
    prevjob = currentjob;
    currentjob = currentjob->next;
    }
    if (currentjob != JobRunQHead)
    prevjob->next = currentjob->next;
    else
    JobRunQHead = currentjob->next;
    if (JobRunQHead == NULL) JobRunQTail = NULL;
    free (currentjob);

    pthread_mutex_unlock(&job_run_mutex);
}

// master cleans up before exits
void master_finalize()
```

```
{
   int i;
   JOB *onejob;

   // ************ need to release memory here
   // release node list memory
   for(i = 0; i < TOTALNODES; i++)
   free(NodeList[i]);

   // job submission queue
   while (JobQHead != NULL) {
   onejob = JobQHead;
   JobQHead = JobQHead->next;
   free(onejob);
   }

   // job running queue
   while (JobRunQHead != NULL) {
   onejob = JobRunQHead;
   JobRunQHead = JobRunQHead->next;
   free(onejob);
   }

   // destroy mutexes
   pthread_mutex_destroy(&job_sub_mutex);
   pthread_mutex_destroy(&job_run_mutex);
   pthread_mutex_destroy(&user_q_mutex);

   // close files and sockets
   fclose(mfplog);
   close(msock);
   close(msock_recv);
   close(msock_service);
   return;
}
// update the client's load and sort the node list by its load
// in ascending order
void update_node_list(int id, float curload)
{
   int i;
   time_t t;

   //printf("update0, id = %d, curload = %.2f\n", id, curload); fflush(stdout);

   /*
   while (node) {
   printf("%d ", node->id);
   if (node->id != id) node = node->next;
   else break;
   }

   printf("\n"); fflush(stdout);
   assert(node != NULL);

   printf("update1\n"); fflush(stdout);
   */
   // shift load vector and add new load
   for(i = 0; i < WINDOWSIZE - 1; i++)
   NodeList[id - 1]->aggregate_load[i] = NodeList[id - 1]->aggregate_load[i+1];
   NodeList[id - 1]->aggregate_load[WINDOWSIZE - 1] = curload;

   // update lasttime
```

```
    t = time(&t);
    NodeList[id − 1]→lasttime = t;

    //printf("update2\n"); fflush(stdout);
    /* **** Postpone the sorting to the job handling routine
    // if no load change, just return
    if (curload == NodeList[id - 1]->aggregate_load[WINDOWSIZE - 2])
    return;

    // load changed, sort the node list
    if (curload > node->aggregate_load[WINDOWSIZE - 2]) {
    while(node->next != NULL && curload >
        node->next->aggregate_load[WINDOWSIZE - 1]) {
      if (node->next->next != NULL)
      node->next->next->prev = node;
      if (node != NodeHead) {
      node->prev->next = node->next;
      node->next->prev = node->prev;
      }
      else {
      node->next->prev = node->next;
      }
      node->prev = node->next;
      node->next = node->next->next;
    }
    }
    else { // new load < current load, shift left
    while(curload < node->prev->aggregate_load[WINDOWSIZE - 1]) {
      node->prev->next = node->next;
      node->next->prev = node->prev;
      node->next = node->prev;
      node->prev = node->prev->prev;
    }
    }
    printf("update3\n"); fflush(stdout);
    */
}

/* master initialization */
void master_init()
{
    int i;
    time_t t_start = 0;
    int return_val;

    //t_start = time(&t_start);
    for(i = 0; i < TOTALNODES; i++) {
    NodeList[i] = (NODE*)malloc(sizeof(NODE));
    assert(NodeList[i] != NULL);

    //node->next = NULL;
    NodeList[i]→id = i + 1;
    //node->currentptr = 0;
    NodeList[i]→lasttime = t_start;
    NodeList[i]→available = UNAVAILABLE;
    bzero(NodeList[i]→aggregate_load, WINDOWSIZE*sizeof(float));
    }
    NumAvailableNodes = 1; // since master started up, at least one
                        // one is available
    AvailableNode[0] = masterid = getmyid(masterhost);
```

```
    // initialize mutexes
    return_val = pthread_mutex_init(&job_sub_mutex, (const
                        pthread_mutexattr_t*) NULL);
    assert(return_val == 0);
    return_val = pthread_mutex_init(&job_run_mutex, (const
                        pthread_mutexattr_t*) NULL);
    assert(return_val == 0);
    return_val = pthread_mutex_init(&user_q_mutex, (const
                        pthread_mutexattr_t*) NULL);
    assert(return_val == 0);

    // initialize user list
    for (i = 0; i < TOTALUSERS; i++) {
    strcpy(UserList[i].name, UserNames[i]);
    UserList[i].totaljobs = UserList[i].runningjobs = 0;
    }
}
/*-----------------------------------------------------------
 * masterCheckLis - master checks who are alive and records the
 *                  information
 *-----------------------------------------------------------*/
void masterCheckLis()
{
    int i, j, temp;
    //clock_t current_clk = clock();
    time_t current_t;

    temp = 0;
    current_t = time(&current_t);
    for(i = 0; i < TOTALNODES; i++) {
    if ((current_t - NodeList[i]->lasttime) > LISDIEDTIMEOUT)
        NodeList[i]->available = UNAVAILABLE;
    else {
        NodeList[i]->available = AVAILABLE;
        AvailableNode[temp++] = NodeList[i]->id;
    }
    }
    NumAvailableNodes = temp;

    // sort the available node list
    for(i = 0; i < NumAvailableNodes; i++)
    for(j = 0; j < NumAvailableNodes - i - 1; j++)
        if (NodeList[AvailableNode[j] - 1]->aggregate_load[WINDOWSIZE
                        - 1] >
        NodeList[AvailableNode[j+1] - 1]->aggregate_load[WINDOWSIZE
                        - 1]) {
        // swap
        temp = AvailableNode[j];
        AvailableNode[j] = AvailableNode[j+1];
        AvailableNode[j+1] = temp;
        }
}
/*-----------------------------------------------------------
 * Turn a socket into a broadcast socket. We need to have
 * a global destination address on which to broadcast (a global var called
 * "BroadcastAddress"). That global variable is initalized here.
 *
 * On ERROR, print a message and return -1
```

```
*-------------------------------------------------------*/
int makeBroadcastCapable(int sock, struct sockaddr_in *bcastaddr, int port)
{
   char buffer[BUFSIZ];
   struct ifconf      ifc;
   struct ifreq       *ifr;
   int i;

   /*
    * turn the socket into a BROADCAST socket
    */
   i = 1;
   if (setsockopt(sock, SOL_SOCKET, SO_BROADCAST, &i, sizeof(i)) < 0)
   {
      fprintf (stderr, "MakeBroadcastCapable ERROR(): ");
      fprintf (stderr, "Could not turn the new socket into BROADCAST\n");
      return -1;
   }
   /*
    * voodoo socket magic to create a global broadcast address
    */
   ifc.ifc_len = sizeof(buffer);
   ifc.ifc_buf = buffer;
   if (ioctl(sock, SIOCGIFCONF, (char *) &ifc) < 0)
   {
      fprintf (stderr, "MakeBroadcastCapable ERROR(): ");
      fprintf (stderr, "Could not get IFC configuration\n");
      return -1;
   }

   /*
    * set a pointer into the ifc structure so that we can get the
    * interface structure
    */
   ifr = ifc.ifc_req;
   for (i = 0; i < (ifc.ifc_len / sizeof(struct ifreq)); i++, ifr++)
   {
      if (ifr->ifr_addr.sa_family != AF_INET)   /* Internet only */
      continue;

      /*
       * Get the interface flags
       */
      if (ioctl(sock, SIOCGIFFLAGS, (char *) ifr) < 0)
      {
         fprintf (stderr, "MakeBroadcastCapable ERROR(): ");
         fprintf (stderr, "Could not get the interface flags\n");
         return -1;
      }

      /*
       * Skip these flags
       */
      if ((ifr->ifr_flags & IFF_UP) == 0 ||          /* interface down? */
      (ifr->ifr_flags & IFF_LOOPBACK) ||             /* local loopback? */
      (ifr->ifr_flags & IFF_BROADCAST) == 0)         /* no broadcast? */
      {
```

```
            continue;
        }
        /*
         * Get and save interface address
         */
        if (ioctl(sock, SIOCGIFADDR, (char *) ifr) < 0)
        {
            fprintf (stderr, "MakeBroadcastCapable ERROR(): ");
            fprintf (stderr, "Could not get the interface address\n");
            return -1;
        }

        /*
         * Get the interface broadcast address
         */
        if (ioctl(sock, SIOCGIFBRDADDR, (char *) ifr) < 0)
        {
            fprintf (stderr, "MakeBroadcastCapable ERROR(): ");
            fprintf (stderr, "Could not get the broadcast address\n");
            return -1;
        }
        bcopy((char *) &(ifr->ifr_broadaddr), (char *) bcastaddr,
          sizeof(ifr->ifr_broadaddr));

        /*
         * Assign the port
            */
            bcastaddr->sin_port = htons(port);
        }
    return 1;
}
```

C.3 Networking Library

```
/* sock.c - interface to socket
 *
 * $Author: zhao $
 * $Date: 1998/04/10 21:54:32 $
 * $Revision: 1.1 $
 */
#include <sys/types.h>
#include <sys/socket.h>

#include <netinet/in.h>
#include <netdb.h>

#include <varargs.h>          /* for errexit() */
#include <stdio.h>

#ifndef INADDR_NONE
#define INADDR_NONE 0xffffffff
#endif

extern int errno;

/*extern char *sys_errlist[];*/
extern const char *const sys_errlist[]; /* fix for LINUX */
```

```
/*u_short htons(), ntohs();*/

u_short portbase = 0;    /* port base, for non-port servers */

/* print an error message and exit */
int errexit(format, va_alist)
char *format;
va_dcl
{
   va_list args;

   va_start(args);
   /*_doprnt(format, args, stderr);*/
   fprintf(stderr, format, args);    /* fix for LINUX */
   va_end(args);
   exit(1);
}

/* passivesock - allocate & connect a socket using TCP or UDP */
int passivesock(char *service, char *protocol, int qlen)
{
   struct servent *pse;
   struct protoent *ppe;
   struct sockaddr_in sin;
   int s, type;

   bzero((char*)&sin, sizeof(sin));
   sin.sin_family = AF_INET;
   sin.sin_addr.s_addr = INADDR_ANY;

   /* map service name to port number */
   if (pse = getservbyname(service, protocol))
      sin.sin_port = htons(ntohs((u_short)pse->s_port) + portbase);
   else if ((sin.sin_port = htons((u_short)atoi(service))) == 0)
      errexit("can't get \"%s\" service entry\n", service);

   /* map protocol name to protocol number */
   if ((ppe = getprotobyname(protocol)) == 0)
      errexit("can't get \"%s\" protocol entry\n", protocol);

   /* use protocol to choose a socket type */
   if (strcmp(protocol, "udp") == 0)
      type = SOCK_DGRAM;
   else
      type = SOCK_STREAM;

   /* allocate a socket */
   s = socket(PF_INET, type, ppe->p_proto);
   if (s < 0)
      errexit("can't create socket: %s\n", sys_errlist[errno]);

   /* bind the socket */
   if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
      errexit("can't bind to %s port: %s\n", service,
      sys_errlist[errno]);
   if (type == SOCK_STREAM && listen(s, qlen) < 0)
   errexit("can't listen on %s port: %s\n", service,
      sys_errlist[errno]);

   return s;
}

/* passiveTCP - create a passive socket for use in a TCP server */
int passiveTCP(char *service, int qlen)
```

```
{
    return passivesock(service, "tcp", qlen);
}

/* passiveUDP - create a passive socket for use in a UDP server */
int passiveUDP(char *service)
{
    return passivesock(service, "udp", 0);
}

/* connectsock - allocate & connect a socket using TCP or UDP */
int connectsock(char *host, char *service, char *protocol)
{
    struct hostent *phe;
    struct servent *pse;
    struct protoent *ppe;
    struct sockaddr_in sin;
    int s, type;

    bzero((char*)&sin, sizeof(sin));
    sin.sin_family = AF_INET;

    /* map service name to port number */
    if (pse = getservbyname(service, protocol))
    sin.sin_port = pse->s_port;
    else if ((sin.sin_port = htons((u_short)atoi(service))) == 0)
    errexit("can't get \"%s\" service entry\n", service);

    /* map host name to IP address, allowing for dotted decimal */
    if (phe = gethostbyname(host))
    bcopy(phe->h_addr, (char*)&sin.sin_addr, phe->h_length);
    else if ((sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE)
    errexit("can't get \"%s\" host entry\n", host);

    /* map protocol name to protocol number */
    if ((ppe = getprotobyname(protocol)) == 0)
    errexit("can't get \"%s\" protocol entry\n", protocol);

    /* use protocol to choose a socket type */
    if (strcmp(protocol, "udp") == 0)
    type = SOCK_DGRAM;
    else
    type = SOCK_STREAM;

    /* allocate a socket */
    s = socket(PF_INET, type, ppe->p_proto);
    if (s < 0)
    errexit("can't create socket: %s\n", sys_errlist[errno]);

    /* connect the socket */
    if (connect(s, (struct sockaddr*)&sin, sizeof(sin)) < 0)
    errexit("can't connect to %s.%s: %s\n", host, service,
        sys_errlist[errno]);
    return s;
}

/* make a TCP connecttion */
int connectTCP(char *host, char *service)
{
    return connectsock(host, service, "tcp");
}

/* make a UDP connecttion */
```

```
int connectUDP(char *host, char *service)
{
   return connectsock(host, service, "udp");
}
```

C.4 Server Administration Utilities

C.4.1 Fireserver

```
#!/bin/sh
#
# $Author: zhao $
# $Date: 1998/07/16 04:24:10 $
# $Revision: 1.2 $
#
# fire up servers in all 32 nodes

echo "fire up LIS servers on SWARM ..."

# make sure bee32 starts
xrsh bee32 $HOME/research/swarm/util/udpserver
sleep 30
echo "started on bee32"

for i in 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01
do
#    rsh bee$i $HOME/research/swarm/util/udpserver 6666
# rsh does not return. I have to use xrsh ??!!
   xrsh bee$i $HOME/research/swarm/util/udpserver
   echo "started on bee$i"
done
echo "finished"
```

C.4.2 Showserver

```
#!/bin/sh
#
# $Author: zhao $
# $Date: 1998/07/16 04:23:54 $
# $Revision: 1.2 $
#
# list the LIS servers running on SWARM

echo "SWARM LIS servers currently running..."

for i in 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
do
#    pid=`rsh bee$i ps x | grep "udpserver" | grep "?" | awk '{print $1}'`
   pid=`rsh bee$i ps x | grep "udpserver" | awk '{print $1}' | head -n 1`

# if there is multiple udpservers running, only display the pid of the first one
   if [ x$pid != "x" ] ; then
   echo "bee$i : ok ($pid)"
   else
   echo "bee$i : died"
   fi
done
```

C.4.3 Killserver

```
#!/bin/sh
#
# Kill all the LIS servers on SWARM
# $Author: zhao $
# $Date: 1998/07/16 04:19:38 $
# $Revision: 1.2 $

echo "SWARM LIS servers being killed..."

for i in 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
32
do
#    pid=`rsh bee$i ps x | grep "udpserver" | grep "?" | awk '{print $1}'`
   pid=`rsh bee$i ps x | grep "udpserver" | awk '{print $1}' | head -n 1`
   if [ x$pid != "x" ]; then
   rsh bee$i kill -9 $pid
   echo "bee$i : $pid killed"
   fi
done
```

C.4.4 Clearlog

```
#!/bin/sh

rm -f ../etc/log.* ../etc/master.*
```

Appendix D

Source Code of Client

D.1 Myplace

```
/* report the node with lightest load
 *
 * $Author: zhao $
 * $Date: 1998/07/16 22:55:52 $
 * $Revision: 1.4 $
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <assert.h>

#include "udpserver.h"

int main(int argc, char *argv[])
{
   int nump, jobtype, n, sock, retvalue = 0;
   char buf[LINELEN], reply[8*LINELEN], masterhost[LINELEN];
   char err[] = "no enough nodes available\n";
   FILE *fp;

   switch(argc) {
   case 1:
   nump = 1;
   break;
   case 2:
   nump = atoi(argv[1]);
   if (nump > TOTALNODES) {
      fprintf(stderr, "Usage: myplace [number of nodes (<=%d)]\n", TOTALNODES);
      exit(1);
   }
   break;
   default:
   break;
   }
   // get master host id
   fp = fopen(mhostfile, "r");
   assert(fp != NULL);

   fscanf(fp, "%s", masterhost);
   fclose(fp);

   //printf("masterhost: %s\n", masterhost);
```

```
    // connect to master server
    sock = connectTCP(masterhost, serviceport);

    // send request to server
    bzero(buf, LINELEN);
    jobtype = MYPLACE;
    sprintf(buf, "%d %d", jobtype, nump);
    //printf("buf = %s\n", buf);
    write(sock, buf, strlen(buf));

    // wait for reply
    bzero(reply, 8*LINELEN);
    while ((n = read(sock, buf, LINELEN)) > 0) {
    buf[n] = '\0';
    strcat(reply, buf);
    }

    printf("%s", reply);

    close(sock);

    if (strcmp(reply, err) == 0)
    retvalue = 1;
    return retvalue;
}
```

D.2 Myrun

```
#!/bin/sh

# $Author: zhao $
# $Date: 1998/05/31 22:36:08 $
# $Revision: 1.1 $

# my own job running script thru command line for SWARM
#
# Written by Yan Zhao (zhao@cs.orst.edu)
# Date: May 4, 1998

set -- 'getopt pn: $*'

if test $? != 0
then
   echo 'Usage: myrun <-p -n num> job args'
   exit 2
fi

case $# in
0) echo 'Usage: myrun [-p -n num] job arg ...' 1>&2; exit 1
esac

PFLAG=0
pid='getpid'
output=/tmp/myrunoutput.$pid
machinefile=/tmp/myrunmachines.$pid

for i
do
   case $i in
   -p) PFLAG=1; shift;;
   -n) nump=$2; shift 2;;
   --) shift; break;;
   esac
done
```

```
if test $PFLAG = 0
then
# sequential job
   cpath='pwd'
   host='myplace'
   echo "Job executed on $host..."
   echo ""
   echo "Your job looked like:"
   echo ""
   echo "---------------------------------------------------------"
   echo " $cpath/$@"
   echo "---------------------------------------------------------"
   echo ""

   rsh $host $cpath/$@ >& $output

   echo "The output (if any) follows:"
   echo ""
   cat $output
   rm -f $output
else
   cpath='pwd'
   host='myplace $nump > $machinefile'
   if test $? = 1
   then
   cat $machiefile
   else
      echo "Job executed on host(s):"
      cat $machinefile
      echo ""
      echo "Your job looked like:"
      echo ""
      echo "---------------------------------------------------------"
      echo " $cpath/$@"
      echo "---------------------------------------------------------"
      echo ""

      mpirun -nolocal -np $nump -machinefile $machinefile $cpath/$@ >& $output

      echo "The output (if any) follows:"
      echo ""
      cat $output
   rm -f $output
   fi
   rm -f $machinefile
fi
```

D.3 Mylogin

```
#!/bin/sh

# $Author: zhao $
# $Date: 1998/07/15 03:29:11 $
# $Revision: 1.2 $

# login into the node with the lightest load
#
# Written by Yan Zhao (zhao@cs.orst.edu)
# Date: June 14, 1998

host='rsh bee01 myplace'
rlogin $host
```

D.4 Mysub

```
/*
 * mysub - submit a batch job (sequential or parallel)
 *
 * $Author: zhao $
 * $Date: 1998/05/20 05:54:27 $
 * $Revision: 1.1 $
 */

#include <stdio.h>
#include <unistd.h>
#include <pwd.h>
#include <sys/types.h>
#include <assert.h>

#define BUFSIZE 512
#define STRINGLEN 96
#define SEQUENTIAL 0
#define PARALLEL 1

char masterhost[STRINGLEN];
const char *hostidfile = "/nfs/phantom/u1/zhao/research/swarm/etc/master.id";
const char *serviceport = "10002";

int filter(char*);

int main(int argc, char *argv[])
{
    int jobtype, uid, nproc, newid, c, sock;
    char buf[BUFSIZE], hostname[STRINGLEN], email[STRINGLEN],
    currentdir[STRINGLEN], cmdline[2*STRINGLEN],
    newaddr[STRINGLEN];
    struct passwd *pwd;
    extern char *optarg;
    extern int optind, optopt;
    FILE *fp;

    // get effective user id
    uid = geteuid();

    pwd = getpwuid(uid); /* get passwd entry for id */
    //printf("User name: %s\n", pwd->pw_name);

    gethostname(hostname, STRINGLEN);
    //printf("Host name: %s\n", hostname);

    /* compose a user's email address */
    bzero(email, STRINGLEN);
    strcat(email, pwd->pw_name);
    strcat(email, "@");
    strcat(email, hostname);
    //printf("email addr: %s\n", email);

    /* get current directory */
    getcwd(currentdir, STRINGLEN);

    jobtype = SEQUENTIAL;
    nproc = 1;

    while ((c = getopt(argc, argv, "pn:")) != -1)
    switch(c) {
    case 'p': jobtype = PARALLEL; break;
    case 'n': nproc = atoi(optarg); break;
```

```
    }
    // the rest of the command line
    bzero(cmdline, STRINGLEN);
    for(; optind < argc; optind++) {
    strcat(cmdline, argv[optind]);
    strcat(cmdline, "+");
    }
    //printf("cmdline = %s\n", cmdline);
    //filter(cmdline);
    // pack up the buffer
    if (jobtype == SEQUENTIAL)
    sprintf(buf, "%d %s %d %s %s", jobtype, email, uid,
        currentdir, cmdline);
    else
    sprintf(buf, "%d %s %d %d %s %s", jobtype, email, uid, nproc,
        currentdir, cmdline);
    //printf("buf = %s\n", buf);
    // get master host id
    fp = fopen(hostidfile, "r");
    assert(fp != NULL);
    fscanf(fp, "%s", masterhost);
    fclose(fp);
    //printf("masterhost: %s\n", masterhost);
    // connect to master server
    sock = connectTCP(masterhost, serviceport);
    // send request to server
    write(sock, buf, strlen(buf));
    close(sock);
    return 0;
}
/* convert space to + */
int filter(char* str)
{
    int i;
    for(i = 0; i < strlen(str); i++)
    if (str[i] == ' ') str[i] = '+';
    return 0;
}
```

D.5 Listjob

```
/*
 * listjob - list jobs currently in the submission queue and running
 *           queue
 *
 * $Author: zhao $
 * $Date: 1998/07/15 04:23:18 $
 * $Revision: 1.1 $
 */
#include <stdio.h>
```

```
#include <unistd.h>
#include <pwd.h>
#include <sys/types.h>
#include <assert.h>

#define BUFSIZE 2048
#define STRINGLEN 96
#define LISTJOBS 3

char masterhost[STRINGLEN];
const char *hostidfile = "/nfs/phantom/u1/zhao/research/swarm/etc/master.id";
const char *serviceport = "10002";

int main(int argc, char *argv[])
{
   int sock, n;
   char buf[BUFSIZE];
   FILE *fp;

   sprintf(buf, "%d", LISTJOBS);

   // get master host id
   fp = fopen(hostidfile, "r");
   assert(fp != NULL);

   fscanf(fp, "%s", masterhost);
   fclose(fp);

   // connect to master server
   sock = connectTCP(masterhost, serviceport);

   // send request to server
   write(sock, buf, strlen(buf));

   // read results from server
   n = read(sock, buf, BUFSIZE);
   buf[n] = '\0';

   printf("%s\n", buf);

   close(sock);

   return 0;
}
```

D.6 Killjob

```
/*
 * killjob - kill jobs currently in the submission queue
 *
 * $Author: zhao $
 * $Date: 1998/07/15 04:47:16 $
 * $Revision: 1.1 $
 */

#include <stdio.h>
#include <unistd.h>
#include <pwd.h>
#include <sys/types.h>
#include <assert.h>

#define BUFSIZE 2048
#define STRINGLEN 96
#define KILLJOBS 4
```

```
char masterhost[STRINGLEN];
const char *hostidfile = "/nfs/phantom/u1/zhao/research/swarm/etc/master.id";
const char *serviceport = "10002";

int main(int argc, char *argv[])
{
   int jobid, sock, n;
   char buf[BUFSIZE];
   FILE *fp;

   if (argc < 2) {
   printf("Usage: killjob <jobid>\n");
   exit(1);
   }

   jobid = atoi(argv[1]);
   sprintf(buf, "%d %d", KILLJOBS, jobid);

   // get master host id
   fp = fopen(hostidfile, "r");
   assert(fp != NULL);

   fscanf(fp, "%s", masterhost);
   fclose(fp);

   // connect to master server
   sock = connectTCP(masterhost, serviceport);

   // send request to server
   write(sock, buf, strlen(buf));

   // read results from server
   n = read(sock, buf, BUFSIZE);
   buf[n] = '\0';

   printf("%s\n", buf);

   close(sock);

   return 0;
}
```

D.7 Listuser

```
/*
 * listuser – list the users who currently have jobs in the queues
 *
 * $Author: zhao $
 * $Date: 1998/07/15 05:43:46 $
 * $Revision: 1.1 $
 */

#include <stdio.h>
#include <unistd.h>
#include <pwd.h>
#include <sys/types.h>
#include <assert.h>

#define BUFSIZE 2048
#define STRINGLEN 96
#define LISTUSERS 5

char masterhost[STRINGLEN];
const char *hostidfile = "/nfs/phantom/u1/zhao/research/swarm/etc/master.id";
const char *serviceport = "10002";
```

```
int main(int argc, char *argv[])
{
   int sock, n;
   char buf[BUFSIZE];
   FILE *fp;

   sprintf(buf, "%d", LISTUSERS);

   // get master host id
   fp = fopen(hostidfile, "r");
   assert(fp != NULL);

   fscanf(fp, "%s", masterhost);
   fclose(fp);

   // connect to master server
   sock = connectTCP(masterhost, serviceport);

   // send request to server
   write(sock, buf, strlen(buf));

   // read results from server
   n = read(sock, buf, BUFSIZE);
   buf[n] = '\0';

   printf("%s\n", buf);

   close(sock);

   return 0;
}
```

D.8 Showload

```
//----------------------------------------------------------------
// showload.java  - display aggregate load and node availability
//
// $Author: zhao $
// $Date: 1998/06/01 22:47:58 $
// $Revision: 1.2 $
//----------------------------------------------------------------
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.util.*;

// main program
class ShowLoad {
   private Frame mainWindow;
   private static final int FrameWidth = 400;
   private static final int FrameHeight = 400;
   private static final int TotalNodes = 32;
   private static final int WindowSize = 20;
   private static final int SHOWLOAD = 6;
   private String[] Nodes = {"Bee01", "Bee02", "Bee03", "Bee04",
            "Bee05", "Bee06", "Bee07", "Bee08",
            "Bee09", "Bee10", "Bee11", "Bee12",
            "Bee13", "Bee14", "Bee15", "Bee16",
            "Bee17", "Bee18", "Bee19", "Bee20",
            "Bee21", "Bee22", "Bee23", "Bee24",
            "Bee25", "Bee26", "Bee27", "Bee28",
```

```
            "Bee29", "Bee30", "Bee31", "Bee32"};
private String masterhost = new String();
private static final int masterport = 10002;
private Socket socket;
private float[] Load = new float[TotalNodes];
private int[][] xPoints = new int[TotalNodes][WindowSize];
private int[][] yPoints = new int[TotalNodes][WindowSize];

public static void main(String args[]) {
ShowLoad world = new ShowLoad();

//while (true) {
//   try {
//      Thread.sleep(1000);
//   } catch (InterruptedException e) {
//      System.err.println("Thread sleep err: " + e);
//   }
//   mainWindow.show();
//}
}

public ShowLoad() {
mainWindow = new ShowLoadWindow();
init();
mainWindow.show();
}

public void init() {
//   Vector
//System.out.println("Node[0] = " + Nodes[0]);
try {
   File fp = new
   File("/nfs/phantom/u1/zhao/research/swarm/etc/master.id");
   FileInputStream finstream = new FileInputStream(fp);
   DataInputStream instream = new
   DataInputStream(finstream);
   masterhost = instream.readLine();
   //System.out.println("master: " + masterhost);

   connectServer();

} catch (FileNotFoundException e) {
   System.err.println("File not found " + e);
} catch(IOException e) {
   System.err.println("File IO Exception " + e);
}
}

public void connectServer() {
try {
   InetAddress addr = InetAddress.getByName(masterhost);
   socket = new Socket(addr, masterport);
   try {
   //System.out.println("socket = " + socket);
   BufferedReader in = new BufferedReader(new
                     InputStreamReader(socket.getInputStream()));
   // output is automatically flushed by PrintWriter
   PrintWriter out = new PrintWriter(new
                BufferedWriter(new OutputStreamWriter(socket.getOutputStream())), true);

   // compose message
```

```
        out.println(SHOWLOAD + " java"); // print doesn't work
        //System.out.print("Message from server: ");
        for(int i = 0; i < TotalNodes; i++) {
            String str = in.readLine();
            Float load = new Float(str);
            Load[i] = load.floatValue();
            //System.out.print(load + " ");
        }
        //System.out.println();
        } finally {
        socket.close();
        }
    } catch(IOException e) {
        System.err.println("File IO Exception " + e);
    }
    //catch (UnknownHostException e) {
    //   System.err.println("Unknown Host Exception " + e);
    //}
    }

    public void reload() {
    connectServer();
    int k = 0;
    for (int i = 0; i < TotalNodes; i++)
        if (Load[i] >= 0.0) {
        for (int j = 0; j < WindowSize - 1; j++) {
            xPoints[i][j] = xPoints[i][j + 1] - 10;
            yPoints[i][j] = yPoints[i][j + 1];
        }

        xPoints[i][WindowSize-1] = 300;
        yPoints[i][WindowSize-1] = (int) (100 + 50*k -
                        Load[i] * 20);
        k++;
        }
    }

    private class ShowLoadWindow extends Frame implements ActionListener {
    public ShowLoadWindow() {
        setSize(FrameWidth, FrameHeight);
        setTitle("Show Load");
        //setBackground(Color.white);
        //addMouseListener (new MouseKeeper());

        // test scroll bar
        //Scrollbar sbar = new Scrollbar(Scrollbar.VERTICAL);
        //sbar.setValues(10, 10, 1, 100);
        //this.add("East", sbar);

        // add menu items
        MenuBar mbar = new MenuBar();
        this.setMenuBar(mbar);
        Menu file = new Menu("File");
        Menu help = new Menu("Help");
        mbar.setHelpMenu(help);
        mbar.add(file);
        mbar.add(help);

        MenuItem f0, f1, f2, f3;
        file.add(f0 = new MenuItem("Available Nodes"));
```

```
    file.add(f1 = new MenuItem("Show Load"));
    file.add(f2 = new MenuItem("Master Node"));
    file.addSeparator();
    file.add(f3 = new MenuItem("Quit"));

    // create and register action listener objects for the
    // menu items
    f0.setActionCommand("Node");
    f0.addActionListener(this);
    f1.setActionCommand("Load");
    f1.addActionListener(this);
    f2.setActionCommand("Master");
    f2.addActionListener(this);
    f3.setActionCommand("Quit");
    f3.addActionListener(this);
}
public void actionPerformed(ActionEvent e) {
    String command = e.getActionCommand();
    if (command.equals("Load")) {
    LoadWindow w = new LoadWindow();
    w.show();
    mainWindow.repaint();
    }
    else if (command.equals("Node")) {
    connectServer();
    mainWindow.repaint();
    }
    else if (command.equals("Master")) {
    Dialog d = new MasterDialog(this);
    d.show();
    }
    else if (command.equals("Quit")) {
    System.exit(0);
    }
}
public void paint(Graphics g) {
    int x;

    for(int i = 0; i < Nodes.length; i++) {
    if ((i%5) == 0) x = 40;
    else
        x = (i % 5) * 60 + 40;

    if (Load[i] < 0.0)
        g.setColor(Color.black);
    else
        g.setColor(Color.green);
    g.drawString(Nodes[i], x , 100 + 50 * (i/5));
    }
}
}
class MasterDialog extends Dialog {
public MasterDialog(Frame parent) {
    super(parent, "Master Host", false);
    setLayout(new GridLayout(1, 1));
    add(new Label(masterhost));
    resize(145, 100);
```

```
}
public boolean handleEvent(Event e) {
    if (e.id == Event.WINDOW_DESTROY)
    dispose();
    else
    return super.handleEvent(e);
    return true;
}
}
class LoadWindow extends Frame {
private Image buffer = null;
UpdateLoad thrd;
Button button1 = new Button("Reload");
Button button2 = new Button("Quit");
public LoadWindow() {
    super("Load Info");
    resize(FrameWidth, FrameHeight);
    add("North", button1);
    add("South", button2);

    // start a thread to update the load info
    thrd = new UpdateLoad(this);
    thrd.start();
}
public boolean action(Event e, Object arg) {
    if (e.target.equals(button1)) {
    reload();
    this.repaint();
    }
    else if (e.target.equals(button2))
    this.dispose(); // destroy the window
    else
    return super.action(e, arg);
    return true;
}
/*
public void show() {
    reload();
    super.show();
}
public void reload() {
    connectServer();
    int k = 0;
    for (int i = 0; i < TotalNodes; i++)
    if (Load[i] >= 0.0) {
        for (int j = 0; j < WindowSize - 1; j++) {
        xPoints[i][j] = xPoints[i][j + 1] - 10;
        yPoints[i][j] = yPoints[i][j + 1];
        }
        xPoints[i][WindowSize-1] = 300;
        yPoints[i][WindowSize-1] = (int) (100 + 50*k -
        Load[i] * 5);
        k++;
    }
}
```

```
    */
    // try to avoid flicker
    public void update (Graphics g)
    {
        if (buffer == null)
        buffer = createImage(FrameWidth, FrameHeight);
        Graphics secondBuffer = buffer.getGraphics();
        paint (secondBuffer);
        g.drawImage (buffer, 0, 0, this);
        //secondBuffer.dispose();
    }

    public void paint(Graphics g) {
        int k = 0;
        g.clearRect(0, 0, FrameWidth, FrameHeight);
        for(int i = 0; i < TotalNodes; i++) {
        if (Load[i] >= 0.0) {
            g.setColor(Color.red);
            g.drawString(Nodes[i], 10, 100 + 50 * k);
            //xPoints[i][WindowSize-1] = 300;
            //yPoints[i][WindowSize-1] = (int) (50 + 50*k +
            //                 Load[i] * 50);
            g.setColor(Color.white);
            g.fillRect(110, 60 + 50*k, 190, 45);
            g.setColor(Color.green);
            g.drawPolyline(xPoints[i], yPoints[i],
                    WindowSize);
            g.setColor(Color.black);
            g.drawString(" 2.0", 320, 70 + 50 * k);
            g.drawString(" 0.0", 320, 60 + 50 * (k+1));
            k++;
        }
        }
    }
    }

    class UpdateLoad extends Thread {
    private LoadWindow lwin;

    public UpdateLoad(LoadWindow w) {
        lwin = w;
    }

    public void run() {
        while (true) {
        lwin.repaint();
        yield();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.err.println("Thread sleep err: " + e);
        }
        reload();
        }
    }
    }
}
```