Abstract

The aim of this project was to develop a software program using the Python language to efficiently manage the processing of a given task that consists of multiple commands. These commands were to be run on the EEE Departmental Cluster, which has eight nodes, with two processors each.

The software that has been developed consists of three main programs: a node manager program that runs on each of the eight nodes, a central manager program running on Node 1 that acts as a server and a graphical user interface that runs on the client computer. The node managers collect statistics for each node which include number of processes running, CPU load, number of users running processes, free memory and process names and status. By means of socket connections, these statistics are sent over to the client computer which are then displayed. The graphs on the user interface update every three seconds and show cluster and node activity levels over the last three minutes.

This report outlines the implementation of the developed software. The logic and information flows have been determined for each main part of the project and explained by means of simple algorithms and diagrams. Although the project was expected to be completed approximately in sixty hours, the actual time spent was about forty hours over. The code written for the each of the three programs can be found in the appendices at the end.

Table of Contents

1.0 Introduction

The main purpose of the project was to make efficient use of the resources of the EEE Department computer cluster which contains eight Dual-CPU PCs. Each Dual-PC is called a node and each one of these nodes varies in performance. The eight PCs are connected in a private network to combine their computing power. At present, a package called MOSIX is used to manage the task of distributing processes between the different nodes, but this is quite inefficient for small tasks, thus better software needs to be developed.
The end-user of the developed software was to be given the option to load a task file from the Graphical User Interface (GUI), and the individual commands of this task were to be assigned to the nodes in an efficient manner. Monitoring the status of the commands and viewing general node statistics were also to be implemented.
This report briefly describes the breakdown of all the individual tasks involved and how each task was designed and implemented to meet the project specifications. It also includes algorithms developed to maximise efficiency of task distribution followed by pros and cons of the final solution. A rough timeline of the project schedule is also included.

2.0 Project Specifications

The programming language to be used was Python. The software was to include nine to ten running processes: eight node managers, a user interface on the client computer and an optional central manager. The end user was to be given the option to load a task file containing a list of commands to be executed and these were to be read in and assigned to nodes efficiently after considering task dependencies. Also the user interface was to display some real-time statistics of the node activities along with the status of the task once a task file was loaded.
An overview of the communication between various processes is shown below in Figure 01.

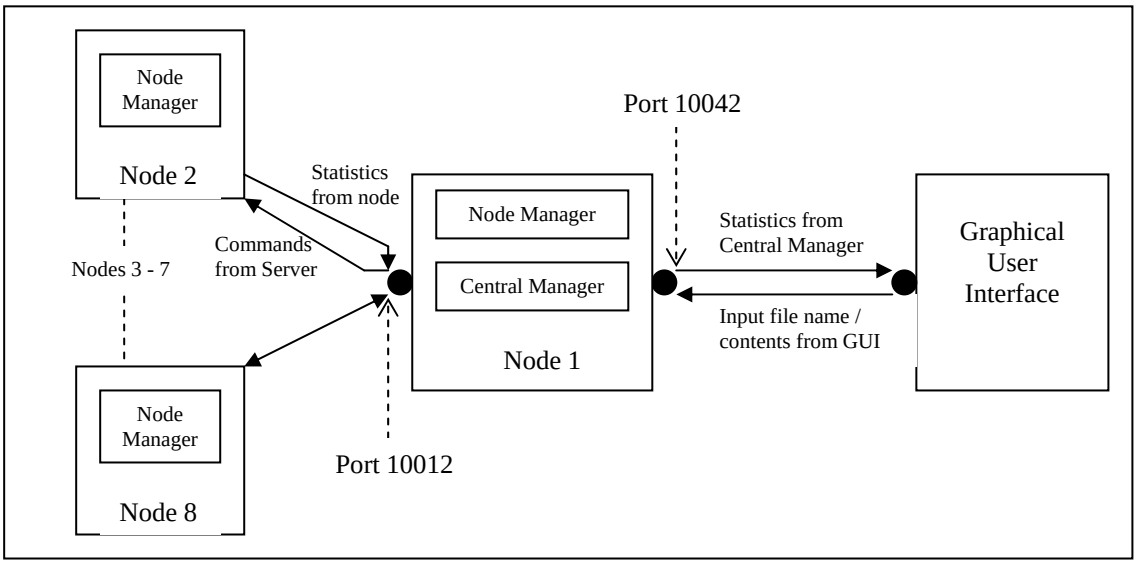


Figure 01: Communication between various processes.

3.0 Assumptions Made

- The task list has a finite number of commands
- The commands are in the order they are to be executed in.
- A command cannot start with a numeric character (the start of a command is identified by an alphabetic character).

- There is negligible delay between gathering statistics from each node.
- CPU usage and memory usage are equally important in determining node activity level.
- A node would not get inactive after a task has been assigned to it by the server.
- The IP addresses of the nodes are in sequence.
  Eg: First node will have the IP address 10.0.0.1, second one 10.0.0.2 and so on.

4.0 Development of the Design

The initial stage of the development process was deciding on a sensible breakdown of all the tasks involved and making a rough schedule with estimated time durations (Figure 04). This was done in coordination with the other group member Ms. Mythreyi Ragavan of Auckland University (UPI: mrag004). Since all the eight node managers are identical and run independent of each other, it was decided to have only three separate programs, one for the central manager, one for the node manager and the other for the user interface. Polling was to be used to gather statistics from the nodes and these statistics were to be stored in a 'class'. The server program was to consist of two independent processes running in parallel with each other, one to monitor node activities and gather statistics and the other to communicate with the GUI. The 'ssh' protocol was to be used for establishing connections between nodes and 'sftp' to upload and download files to and from the cluster. Although it was decided to use script files for automating the node connections, 'ssh –n –l node#' command was used in the final solution to achieve this.
The workload was split up and the GUI was to be developed by Ms. Ragavan while Mr. Mabotuwana was to develop the central manager and node manager programs and the GUI – Server connection so that the statistics could be sent over from the server.

5.0 Implementation of the Final Design

This section outlines the individual tasks involved along with a brief description of the actual implementation of each task. Figure 02 below shows a flowchart of the program logic flow.

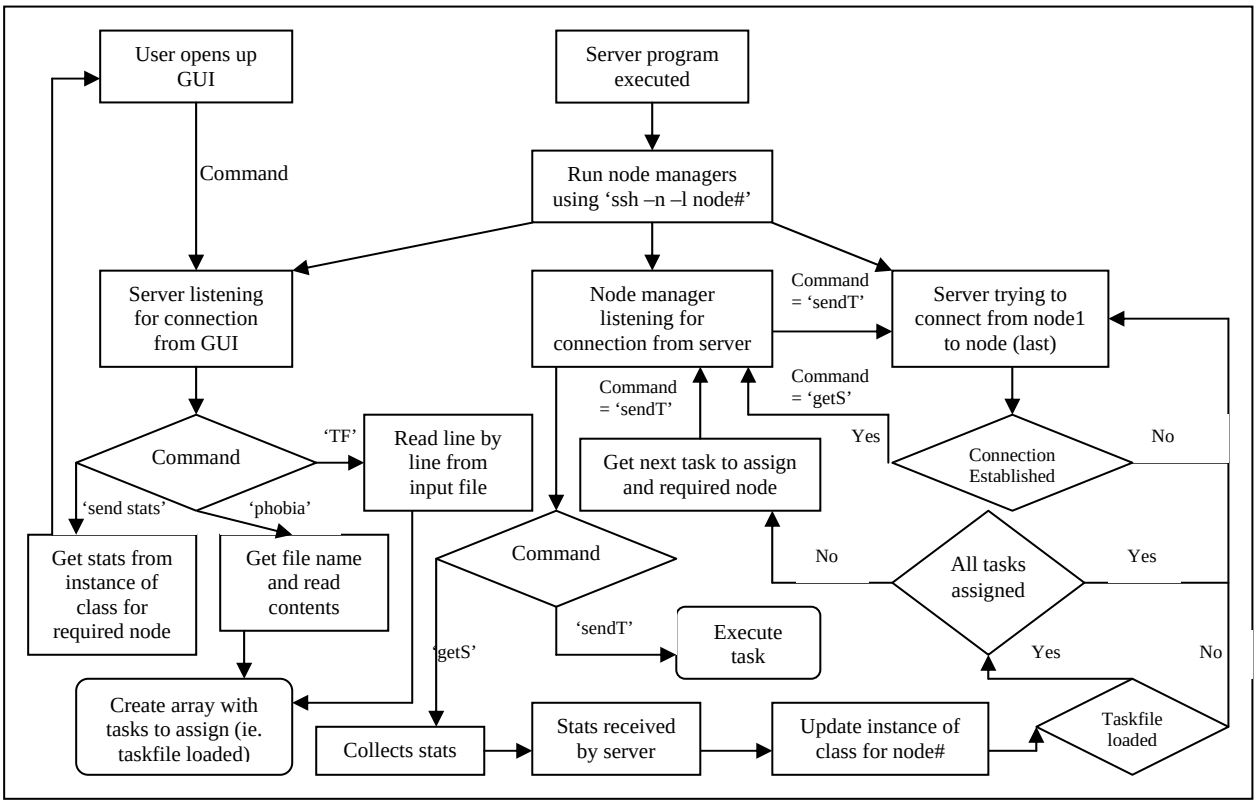


Figure 02: Flowchart of Program Logic Flow

Thusitha Mabotuwana
9790416

5.1

Node Manager Program

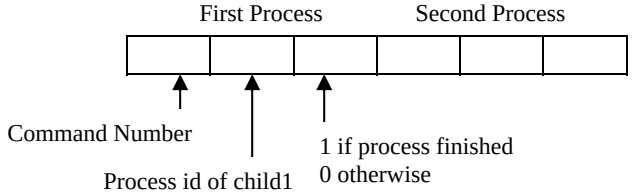
This program acts as a client and runs on each node. It listens on port 10012 for a connection from the server and when connection is established it receives data from the server and checks for the type of request sent. If received data contains 'getS' it sends the required statistics over for the node or else if data received is 'sendT', it responds to the server asking it to send over the command to be executed. Once the command to be executed is received, the program forks and makes the child process execute the command. The status of each of the two processes running on a given node are also sent over each time the server requests for statistics, so that the server knows whether an assigned task has finished or still running. If no commands are being executed by the child processes, status will show all zeros.

5.1.1 Collecting Statistics and Sending to Central Manager

The node managers collect load average, free memory, process names and status, number of processes running and number of users running processes on the node. These statistics are sent over to the server in the following format:
[No. of processes running\n + No. of users running processes on the node\n + load average\n + Free memory\n + (process name + ',' + process status) * x];
Where \n represents a new line and x represents the number of processes running on the node.

5.1.2 Assigning Tasks and Reporting Status

When a command has to be assigned, the server sends a request 'sendT' to which the node responds with 'sendTask'. The server then sends the command that has to be executed and this command is in the format:
[Command number, Command to execute], where command number is the command identification number as read from the input file loaded by the user.
The node program is capable of handling two child processes. It has an array 'tasksDone' with six elements, three for each process and stores information in the following format.



When a command has been assigned to a node, it forks a child process and stores the command number in the first field, process id of the forked child in the second and a '0' in the third before executing the command. Once a process has finished, the signal handler function in the node program automatically gets called and either element three or six of 'tasksDone' array is made '1' after comparing the process id of the child that finished. Whenever the node sends statistics over to the server, it adds four numbers to the end of the data stream which correspond to:
[Child 1 command number, Child 1 status (1 or 0), Child 2 command number, child 2 status].
If no process is being currently executed, the corresponding fields will be zero.
The server program can decide which processes have finished by checking these last four numbers.

5.2 Central Manager Program

This program runs on node1 and has two independent processes running in parallel with each other. The main process monitors node activities and gathers statistics while the other listens on port 10042 for a connection from the GUI.
When this program is first executed, it determines the number of nodes on the cluster and then runs the node manager program on each node. Hereon, the number of nodes the server has to get statistics from and consider when assigning tasks will depend on this number.

5.2.1 Gathering Statistics from Node Managers

The server gathers the required statistics by means of polling the nodes one by one. Whenever the statistics are required, the server connects to the node and sends a request with 'getS' asking the node to send over the statistics. These statistics are then stored in the appropriate instance of the class, which has an instance for each node.

The array 'runningNodes' has elements equal to the number of nodes in the cluster and keeps record of the nodes the server was able to connect to. It makes the appropriate element '1' or '0' depending on whether connection was established or not. This array is necessary so that the server would not try to assign a task to a node that was once alive and then became inactive.

5.2.2 Establishing Command Dependencies from the Task File

The task file containing a list of commands is received from the client. Each command is scanned into two parts: an array of integers that contains any required command numbers and a string that contains the actual command. A structure with these two parameters is used for each command.

The function 'getTask' is used to determine the command dependencies. When the main program requests for the task that has to be assigned next, the function goes through the array, 'tArray' which has all the commands to be assigned and picks up a command if the length is two (i.e. the structure for that command has only the command number and the actual command, thus it can be assigned immediately). This element is then deleted from 'tArray' and the command returned to the main program which will then be assigned to a node.

The Boolean array 'procsDone' stores the commands have been completed. It has all zeros initially and the respective index is made '1' when an assigned task has finished. If 'getTask' could not find a command with a length equal to zero (i.e. there is no command that can be assigned immediately), it goes through 'tArray' and compares all the pre-command numbers with 'procsDone' and deletes them, thus 'tArray' will get shorter and shorter as commands get completed.

Eg. Lets consider the command: 1,2,4, Calculate a.

The structure for this will have the following format:

1	2	4	Calculate a

Before and after command 2 has been completed the above mentioned arrays will look as follows (please note that only one element of 'tArray' is shown here).

	Before				After			
tArray:	1	2	4	Calculate a	1	4	Calculate a	
procsDone:	0	0	0	0	0	1	0	0

5.2.3 Reading Commands from the Task File

The end-user is given the option to load the task file either from the client computer or else from the cluster. If a file from the client computer is selected by pressing 'Select TaskFile' (Figure 04), the commands are transferred to the server program line by line and 'tArray' created. If a file from the cluster has to be used as the task file, the user has to enter the name of the file and then press 'Load Entered TaskFile'. The GUI then transfers the name of file to the server which in turn reads contents of the required file and creates 'tArray'

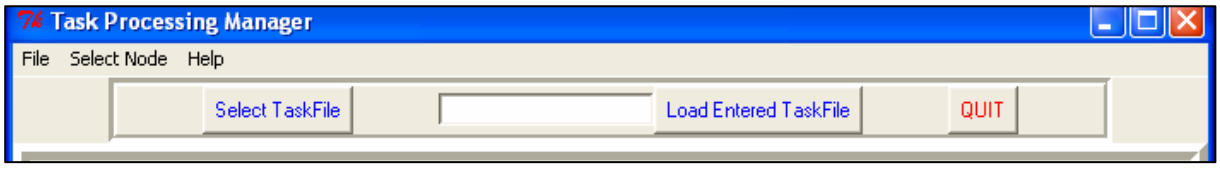


Figure 03: Selecting the task file from GUI

5.2.4 Sending Updated Statistics to the GUI for Display

The GUI tries to connect to the server every two seconds and get statistics for nodes unless the user is trying to send a new task file. The server keeps listening on port 10042 for a connection from GUI and when a connection is received, it sends the statistics over for the nodes, one by one, if the client request was 'send stats'. The server gets the statistics for the nodes by calling on the instance of the class for the required node and the information sent to the GUI has the following format:

[Node number\n + No. of processes running\n + No. of users running processes on the node\n + Load average\n + Free memory\n + (Process name + ',' + Process status) * x + Total number of tasks\n + Tasks completed\n + No. of tasks running\n + No. of tasks waiting];

Where \n represents a new line and x represents the number of processes running on the node. 'Total number of tasks' represents the number of commands run by the user. 'Tasks completed' is calculated by counting the number of '1's in 'procsDone' array and 'tasks waiting' is the length of 'tArray' since this array has the commands that are yet to be assigned. The number of commands currently being executed is calculated using the simple formula:

Commands running = total number of commands – commands finished – commands waiting.

5.2.5 Algorithm to Determine Nodes to Assign Tasks To

The CPU usage and free memory available for each node are the two factors used to determine which node a command can be assigned to. CPU usage is to be minimised and free memory maximised, thus the product of inverse of CPU usage and free memory (1/CPU * Free Memory) is considered when determining the required node. The maximum value will correspond to the node that is least busy.

The array 'prod' stores the above product for each node and gets updated everytime the server gets statistics from a node. Whenever a task has to be assigned, the node corresponding to the maximum of 'prod' is calculated and checked against the value for the same node in array 'procsOnNode' which stores the number of processes running on each node. If this number is less than two, the task is assigned to the corresponding node. If this number is equal to two, the second highest number in 'prod' is calculated and this process keeps repeating itself until all the nodes have been considered. If a suitable node is not found the command is held until a node becomes free again.

5.3 Graphical User Interface

The user interface runs on the client computer and gets the required statistics from the server. It displays a live summary of the whole cluster and also gives the user the option to select any node and view the node statistics.

5.3.1 Allowing the User to Load Task File(s)

The user is allowed to load a new task file through a 'File Dialog' window. This allows the user to load a task file from the local computer while typing the name of the task file and pressing 'Load Entered TaskFile' (Figure 03) allows the user to load a task file from the cluster.

5.3.2 Receiving and Displaying Statistics

An instance of the class 'EachNode' in the GUI program is used to store the statistics for each node. The statistics sent from the server are in a pre-determined order and the first number in the statistics stream corresponds to the node number the statistics are for. When the statistics are received, the node is determined and the corresponding instance of the class is called and values updated. The GUI tries to establish a connection with the server every two seconds but the refresh rate of graphs is set to three seconds. This assumes that the GUI will get the statistics for all the nodes within the next second after connecting to the server.

When the program first loads, the window with the cluster summary is shown (Figure 04) but the user is given the option to select a node at any given time. The cluster CPU load, free memory, number of users running processes and number of processes running on the cluster are determined by summing up the respective values for all the nodes.

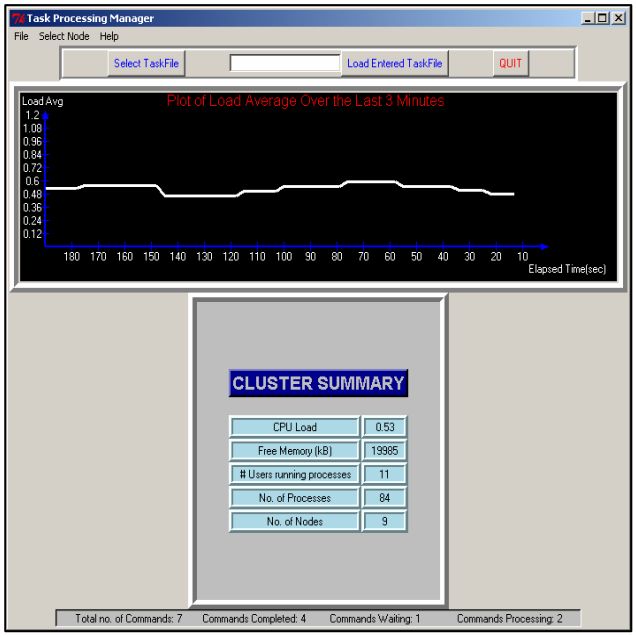


Figure 04: Window Showing Cluster Summary at Startup

When the user selects a node from the 'Select Node' dropdown menu, a separate window pops up with the node statistics. Apart from the CPU load, free memory, number of users running processes and number of processes running on the node, it lists all the process names along with their status (Figure 05). A stack with sixty values (sixty values are used since the values are updated every three seconds, and we want values over the last 180 seconds) is used to plot the 'Graph of Load Average Over the Last 3 Minutes' and each time the GUI gets statistics for the same node, the first value is popped and the new value is pushed into the stack. This helps to store the load average for the given node for the last three minutes. Also the Y-axis scale adjusts itself automatically depending on the highest CPU load value in the stack.

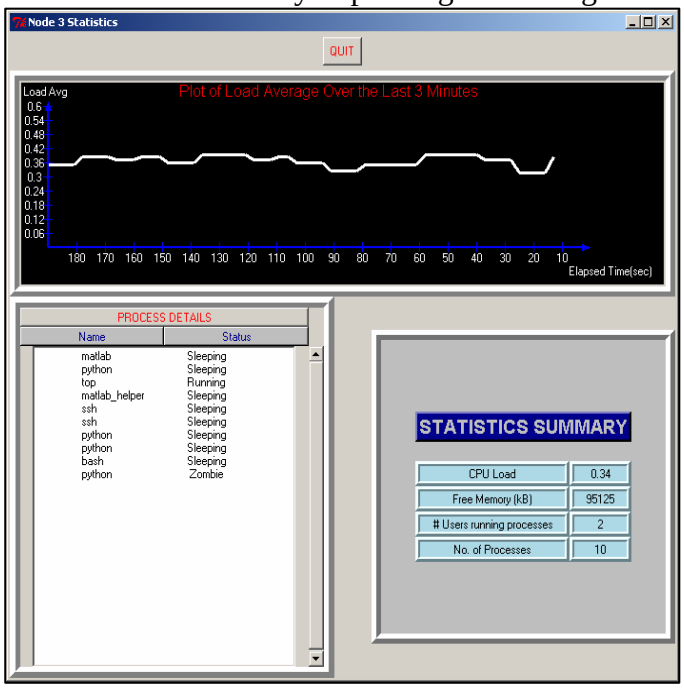


Figure 05: Pop-Up Window Showing Statistics for Selected Node

6.0 Special Features of the Design

In addition to the project specifications, the following special features were implemented in the final solution.

- Both, the server and the GUI are capable of handling any number of nodes. The server program gets the number of nodes on the cluster and tries to connect to all the nodes and gather statistics accordingly. It also sends the total number of nodes to the GUI so that the GUI can list all nodes in the menu 'Select Node' and also create the required number of instances of the class to store statistics.
- The user is given the option to load a taskfile either from the local drive or the cluster.
- All the process names and their status are listed in the GUI for each node in addition to CPU load, number of processes, free memory and number of users. The user identity (UPI) is not listed with the process name to maintain confidentiality.
- Although polling is used to get statistics from each node, when a task has to be assigned, a socket is created and the required (ie. least busy) node is connected to and the task assigned. Due to this, the server does not have to wait until it connects to the least busy node the next time thus increasing efficiency in task assignment.
- CPU usage and available free memory are considered when assigning a task to ensure that the task is assigned to the least busy node.

7.0 Project Time Durations – Estimated Vs. Actual

The Gantt chart below shows a comparison between the estimated duration and actual time spent, along with the sequence of activities for completion of the project

	Task Name	Duration	Actual Finish	Start	Finish
3	Uploading and running server program and running client program	3 hrs	Mon 12/05/03	Sun 11/05/03	Mon 12/05/03
4	**Write node mander programs**	**20 hrs**	**Tue 20/05/03**	**Wed 14/05/03**	**Fri 16/05/03**
5	Decide on informatin to gather	1 hr	Wed 14/05/03	Wed 14/05/03	Wed 14/05/03
6	Gather information and make each node manaher a standalone program	6 hrs	Fri 16/05/03	Wed 14/05/03	Wed 14/05/03
7	Send gathered statistics back to central manager (Node 1)	4 hrs	Tue 20/05/03	Fri 16/05/03	Fri 16/05/03
8	**Write Central manager program**	**4 hrs**	**Wed 21/05/03**	**Sat 17/05/03**	**Mon 19/05/03**
9	Get statistics from node managers at regular intervals	4 hrs	Wed 21/05/03	Sat 17/05/03	Mon 19/05/03
10	Pass statistics to the GUI	3 hrs	Wed 21/05/03	Sun 18/05/03	Mon 19/05/03
11	**Create the GUI**	**6 hrs**	**Sun 25/05/03**	**Sun 18/05/03**	**Mon 19/05/03**
12	Read commands from input and create data structures	6 hrs	Tue 20/05/03	Sun 18/05/03	Mon 19/05/03
13	Read statistics from Node 1	3 hrs	Fri 23/05/03	Mon 19/05/03	Mon 19/05/03
14	Display statistics in a logical manner	5 hrs	Sun 25/05/03	Mon 19/05/03	Mon 19/05/03
15	**Determine which node to assign tasks**	**26 hrs**	**Fri 30/05/03**	**Tue 20/05/03**	**Fri 23/05/03**
16	Develop an algorithm	4 hrs	Thu 22/05/03	Tue 20/05/03	Tue 20/05/03
17	Implement the algorithm to assign tasks	10 hrs	Fri 30/05/03	Thu 22/05/03	Fri 23/05/03

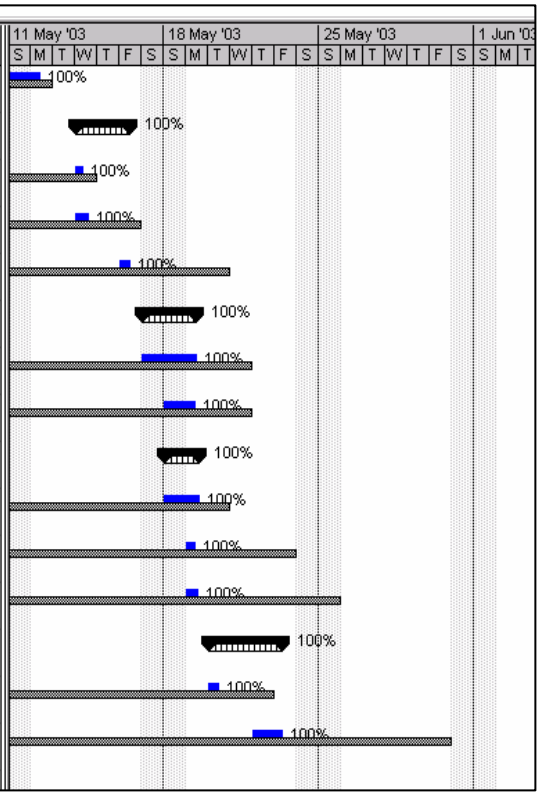


Figure 06: Gantt Chart showing estimated duration vs. actual time spent

8.0 Conclusions

The final design meets all the required specifications. The graphical user interface displays useful information about the cluster and nodes, such as number of processes running, CPU load, number of users running processes and free memory and updates every three seconds, displaying the activity levels of the cluster and its nodes over the last three minutes. It also lists all the process names along with their status.
The developed software is capable of handling any number of nodes and processing a given taskfile and reporting status back to the user.
All the algorithms have been developed based on efficiency and the feasibility of programming the algorithms in Python.

Bibliography

Harms, Daryl D., *The quick Python book*. Greenwich, CT : Manning, c2000
Brown, Martin C., *Python : the complete reference*. New York ; London : McGraw-Hill, 2001.
Python Reference Manual. Retrieved 20 May, 2003 from http://www.python.org/doc/current/ref/ref.html
Python Tutorial. Retrieved 21 May 2003 from http://www.python.org/doc/current/tut/tut.html
Cogliati, Gosh. *Non-Programmers Tutorial For Python*. Retrieved 21 May 2003 from http://honors.montana.edu/~jjc/easytut/easytut/

Thusitha Mabotuwana
9790416

Appendix i: Code for Node Manager Program

```
#   Course:         COMPSYS 302: Software Design 2E
#   Project 2:      Task Processing Manager
#   Authors:        Thusitha Mabotuwana (9790416)
#                   Mythreyi Ragavan    (9751564)
#   Purpose:        To collect statistics from the local node and send these to the central manager. Also, to execute
any commands received from the central manager
#   Last Modified:          June 3, 2003.

import os, sys, signal, string, socket, time, commands
from nodeStruct import *


#****************************************************************
# Function to check if a variable is an integer – for the directory names in 'proc'
def getVal(a):
    try:
        return int(a)
    except ValueError:                      # catch error if string cannot be converted to int
        return 0


#****************************************************************
# This function gets all the required stats to be displayed in the GUI
def getStat():
    stat = ""                               # concatenate all stats into a big string and send this
    userList = []
    x= os.listdir('/proc')                  # get stats from the 'proc' directory
    count=0

    info = open("/proc/loadavg", 'r')       # open load avg file in the proc directory
    temp = info.readline()
    stat = temp[:4] + "\n"                  # read first 4 characters (load avg in last 1 minute)
    info.close()

    data = open('/proc/meminfo','r')        # open meminfo file
    data1 =data.readline()
    while data1 != "":                      # scan through to the free memory line
        if data1[:4]=="MemF":
            stat = stat + (string.split(data1))[1] + "\n"       # get only the number part
        data1=data.readline()
    data.close()

    for i in range(0,len(x)):
      if (getVal(x[i])) != 0:               # if directory name is a number (i.e. process)
          try:
              count=count+1                 # no of processes running on each node
              os.chdir("/proc/" + x[i])     # go into each process directory
              info = open("status", 'r')    # get name and status of each process
              stat = stat + ',' + (string.split(info.readline()))[1]
              stat = stat + ',' + (string.split(info.readline()))[1]
              for j in range(0,5):          # get no of users running processes
                  u = info.readline()
              u = string.split(u)
              userList.append(u[1])         # make a list of all users
              info.close()
          except:                           # catch errors if the above files do not exist
              i =  i + 1
    userList.sort()                         # sort userlist before filtering
    i = 1
```

```
    while(i < len(userList)): # the same user can be running multiple processes, so list has to be filtered
        if userList[i] == userList[i-1]:
            del userList[i-1]
            i -= 1
        i+=1
    stat = str(count) + "\n" + str(len(userList)) + "\n" + stat        # add no. of users to stats string
    return stat


# ********************  In the MAIN code  ********************
# to keep track of commands assigned to the node
# [0] – task id for 1st process, [1] - pid for 1st process, [2] - 1st process finished or not, [3] – task id for 2nd process, [4] - pid for 2nd process, [5] - 2nd process finished or not
tasksDone = [0,0,0,0,0,0]


#*******************************************************************************************
# Signal handler function for child process. Gets called whenever a child dies
def sigchld_handler(signum, args):
    global tasksDone
    pid, returnVal = os.wait()

    # check which process finished using pid and update tasksDone array
    if pid == tasksDone[1]:
        tasksDone[2] = 1
        tasksDone[1] = 0
    if pid == tasksDone[4]:
        tasksDone[5] = 1
        tasksDone[4] = 0


#************************************     MAIN CODE    *********************************
signal.signal(signal.SIGCHLD, sigchld_handler)

PORT = 10012
s =  socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((' ', PORT))
s.listen(1)                    # only one incoming connections (from central manager)

# keep connecting and catch any errors
while(1):
    try:
        conn, addr = s.accept()
    except socket.error:
        pass
    except KeyboardInterrupt:
        break
    else:
        try:
            data = conn.recv(1024)
        except socket.error:
            pass
        except KeyboardInterrupt:
            break

        else:                                # when data received..
            if(data == 'getS'):              # send stats for this node to the server
                result = ""
                result = getStat() + ',' + str(tasksDone[0]) + ',' + str(tasksDone[2]) + ',' + str(tasksDone[3]) + ',' + str(tasksDone[5])
                # also send info on the progress of any commands/tasks
                if tasksDone[2] == 1:       # update tasksDone if any child processes have finished
                    tasksDone[2] = 0        # this shows if the task is finished or still running
```

```
            tasksDone[1] = 0        # has the pid of child
            tasksDone[0] = 0        # has task id which is returned to the server when the child dies
        if tasksDone[5] == 1:
            tasksDone[5] = 0
            tasksDone[4] = 0
            tasksDone[3] = 0
        try:
            conn.send(result)
        except:
            conn.close()

    # if the server wants to send a command to execute
    elif data == 'sendT':
        conn.send('sendTask')                   # indicate to server that node is listening
        try:
            task2Exe = conn.recv(1024)          # = 'task id' +  'command to execute'
            conn.close()
            task = string.split(task2Exe,",")   # split into command no. and actual command
            x=string.split(task2Exe,",")        # for debugging purposes
            second=string.replace(x[1]," ",",") #to split actual command into separate arguments
            second=string.split(second,",")
            first=second[0]                     # first argument for 'execv' command
            del second[0]                       # to give second argument
            st=""
            for i in range(0,len(second)):      # put back into a string for 'execv'
                st=st+second[i]+','
            st=st[:-1]

            if tasksDone[2] == 0:               # fork for 1st process
                pid1 = os.fork()                # store the pid so that it can be checked later
                tasksDone[0] = task[0]          # set parameters
                tasksDone[1] = pid1
                tasksDone[2] = 5
                if pid1 == 0:
                    os.execv(first,(first, string.replace(st, ',', ' ')))      # execute this command

            elif tasksDone[5] == 0:             # fork for 2nd process
                 pid2 = os.fork()
                 tasksDone[3] = task[0]
                 tasksDone[5] = 5
                 tasksDone[4] = pid2
                 if pid2 == 0:
                    os.execv(first,string.split(st, ','))

        # catch any errors when receiving the command
        except socket.error:
          print 'socket error'                 # for debugging
          pass

        except KeyboardInterrupt:
            break
```

Appendix ii: Code for Central Manager Program

```
#   Course:         COMPSYS 302: Software Design 2E
#   Project 2:      Task Processing Manager
#   Authors:        Thusitha Mabotuwana (9790416)
#                   Mythreyi Ragavan    (9751564)
#   Purpose:        To create a server program that will receive and transmit data from and to each of the nodes
and the GUI client. It manages the distribution of tasks on the nodes
#   Last Modified:  3 June, 2003.

import socket, copy, select
import time, os, sys, string, thread
from nodeStruct import *

# Global variables shared by the main program and the GUI thread
assgn2Node = 0
orderOfTasks = []
assignTasks = 0
procsDone = []
tArray = []
noOfNodes = os.listdir('/proc/mosix/nodes')     # get number of nodes on the cluster

#Starts the node manager programs on all the nodes of the cluster
os.system('ssh -n -f phobia python node.py')
for h in range(2,noOfNodes+1):
   tempstr = 'ssh -n -f node' + str(h) + ' python node.py'
   os.system(tempstr)


#*****************************************************************************
# Function to put the commands in an array when the user loads a task file
def splitTasks(tasks):
    global tArray                       #array of read lines
    for i in range(0,len(tasks)):
      tArray.append(string.split(string.rstrip(tasks[i]),","))


#*****************************************************************************
# Function to keep track of the tasks that have been assigned and to return the task that is next to be assigned
after considering all the task dependencies
def getTask():
   global procsDone, tArray
   temp = []
   i = 0
   while(i< len(tArray)):               # total number of commands to be assigned
      if len(tArray[i]) == 2:           # if there are no more dependencies for that command..
         temp.append(tArray[i][0])      # store the command no. and actual command in 'temp'
         temp.append(tArray[i][1])
         del tArray[i]                  # delete command since it is going to be assigned
         return temp                    # return this as the next command to be assigned
      else:                             # else if there are dependencies..
         j = 1
         while (j < len(tArray[i])-1):          # check if each dependency command has been completed
            taskNo = tArray[i][j]
            taskN = int(taskNo) - 1
            if procsDone[taskN] == 1:           # delete dependency if it has been completed
               del tArray[i][j]
               j = j - 1
            j = j + 1
      i = i + 1
   return 0
```

```
#*******************************        In Main Code       *******************************
# commands running on each node – maximum of 2 running at the same time
procsOnNode = [ ]


#*****************************************************************************************
# Function to calculate the best node to assign a command to – using free memory and load average
def updateProd(nodeConctd, nodes, flag, runN):
   global prod                # prod holds the freemem * 1/cpu value for each node

   if flag == 1:              # main thread is calling the function to assign a task
      prod[nodeConctd] =  nodes[nodeConctd].getProduct()     # update 'prod' for that node
      return 0

   else:
      for i in range(0,len(prod)):
         maxP = max(prod)
         # required node has the maximum (freemem*1/cpu) product
         reqdNode = str (prod.index (maxP) + 1)
         y = int(reqdNode) – 1
         # if less than 2 commands are running on that node and it is running, assign to that node
         if int(procsOnNode[y]) < 2 and prod[y] != 0 and runningNodes[y] != 0:
            return reqdNode
         prod[y] = 0                  # else reset the product and get the maximum again.
      return 0


#*****************************************************************************************
#                                        MAIN CODE
HOSTS = []
nodes = []
prod = []
runningNodes = []          # holds the active and inactive nodes. (0 → inactive node, 1 → active node)
for i in range(0,len(noOfNodes)):
   HOSTS.append("10.0.0." + str(i+1))              #IP address array for nodes on cluster
   nodes.append(NodeStruct())                      #class structure for each node
   prod.append(0)                                  #freemem * 1/cpu
   procsOnNode.append(0)                           #no. of processes running on each node
   runningNodes.append(0)                          #nodes currently running


#*****************************************************************************************
# This thread takes care of the GUI connections. It waits for a GUI to connect and treats it accordingly after
connection has been established
def guiT():
   global assignTasks, runningNodes, procsDone, tArray

   PORT = 10042
   s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
   s.bind((' ',PORT))
   s.listen(1)
   taskArray = []
   stats4N = 0
   totalTasks = 0
   tasksDone = 0

   # Keep connecting to the GUI
   while(1):
      conn, addr = s.accept()
      stfFrmGui = conn.recv(1024)          # data received from the GUI
      # if user has loaded a new taskfile from the local harddrive
      if stfFrmGui == "TF":
         conn.send('send')                 # allow GUI to send task file
```

```
        line = conn.recv(1024)
        if len(procsDone)!=0:                   # boolean array for completion of commands
            del procsDone[:]                    # reset all parameters for new task file
            del taskArray[:]
            totalTasks = 0                      # no. of commands in the task file
            stats4N = 0
            tasksDone = 0
        while(line !="ENDTF"):                  # get contents of the task file, line-by-line till EOF
            taskArray.append(string.rstrip(line))
            procsDone.append(0)
            totalTasks += 1
            conn.send('send')                   # tell GUI to send the next line
            line = conn.recv(1024)
            print 'line ', line                 # for debugging
        splitTasks(taskArray)                   # to format the commands in an array
        assignTasks = 1                         # flag to start assigning tasks
        print 'got task file'                   # for debugging

    # if user sends the name of a task file that is on the cluster
    elif stfFrmGui == 'from phobia':
        conn.send('send str')                   # tell GUI to send the file name
        if len(procsDone)!=0:                   # reset all parameters
            del procsDone[:]
            del taskArray[:]
            totalTasks = 0
            stats4N = 0
            tasksDone = 0
        inFile = conn.recv(1024)                # file name
        inFile = open(inFile,'r')               # open the file and read each line.
        line = inFile.readline()
        while(line !=""):                       # process each line in the same way as above
            taskArray.append(string.rstrip(line))
            procsDone.append(0)
            totalTasks += 1
            line = inFile.readline()
        splitTasks(taskArray)
        assignTasks = 1

     # if GUI is ready to receive statistics
     elif stfFrmGui == 'send stats':
        conn.send('stats')                      # let GUI know the statistics are being sent
        if runningNodes[stats4N] == 1:          # send stats for that node if it is running
            toGui = nodes[stats4N].getStats()   # function in class to return string containing stats
            tasksDone = 0
            for i in range(0,len(procsDone)):
                if int(procsDone[i]) == 1:
                    tasksDone += 1              # tasks completed – for user info
            ind = -1
            count = 0

            # in the string to be sent to the GUI, remove the last 4 unwanted digits sent by the node
            while(1):
                # if the node is active – send the stats
                if toGui[ind] == ',':
                    toGui = toGui[:ind]
                    count += 1
                    ind = 0
                if count ==4:
                    break
                ind -= 1

            # string sent to GUI – concatenate info on the progress of commands from the task file
```

```
            toGui = toGui + '\n' + str(totalTasks) + '\n' + str(tasksDone) + '\n' + str(totalTasks - tasksDone -
len(tArray)) + '\n' + str(len(tArray))
            conn.send(str(stats4N+1) + '\n' + str(toGui))          # send the node no. for GUI to recognise

        # if the node is inactive, send code 'D#' to the GUI
        else:
            conn.send('D' + str(stats4N+1))
        stats4N = stats4N + 1
        if stats4N == len(noOfNodes):       # reset counter if GUI updated with stats for all the nodes
            stats4N = 0
    conn.close()                            # wait for next connection to send next node stats


#*****************************************************************************************
#                                       Back to Main Code

thread.start_new_thread(guiT, ())           # start the GUI thread
PORT = 10012
conct2Node = 0

# This keeps getting the stats from the nodes and updates the class
while(1):
  if conct2Node != len(noOfNodes):
    HOST = HOSTS[conct2Node]                # get IP for node
  else:                                     # reset counter to the first node
    conct2Node = 0
    HOST = HOSTS[0]

  try:
    ss = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    ss.connect((HOST,PORT))
    statstr = ""
    print 'Connected to ', HOSTS[conct2Node]    # for debugging
    runningNodes[conct2Node] = 1                # node running if can connect to it
    ss.send('getS')                             # request node to send stats
    statstr = ss.recv(4096)                     # stats received from node
    t = string.split(statstr,",")

    if int(t[-3]) == 1:                         # if child 1 of node finished process
      # set command number as completed and decrement the number of commands running on node
      procsDone[int(t[-4])-1] = 1
      procsOnNode[int(HOST[7])-1] = procsOnNode[int(HOST[7])-1] – 1

    if int(t[-1]) == 1:                         # if child 2 of node manager finished process
      procsDone[int(t[-2])-1] = 1
      procsOnNode[int(HOST[7])-1] = procsOnNode[int(HOST[7])-1] – 1

    print 'Processes Done: ', procsDone         # debugging
    print 'No of processes on each node: ', procsOnNode

    nodeConctd = int(HOST[7]) – 1               # get node no. of currently connected node
    nodes[nodeConctd].setNode(statstr)          # update class structure for that node
    runningNodes[nodeConctd] = 1                # detect if node is running or not
    ss.close()
    nodeReqd = updateProd(nodeConctd,nodes,1,runN=0)
   except socket.error:                         # connection failed due to socket error
    print 'connection failed to node: ', HOSTS[conct2Node]   # error message
    runningNodes[conct2Node] = 0                # set node as inactive

  except KeyboardInterrupt:                     # catch ctrl – C and exit
    break

  except:                                       # catch all other errors
```

```
        pass

    if assignTasks == 1:                                # if there are commands to be assigned
        nodeReqd = updateProd(0,0,2,runningNodes)       # node to assign next command to
        print 'ASSIGNING TASK TO NODE: ', nodeReqd

        # if it is a valid node, immediately connect to it and assign the command instead of polling
        if int(nodeReqd) > 0:
            ss = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
            ss.connect((HOSTS[int(nodeReqd)-1],PORT))
            task2Assign  = getTask()

            if task2Assign != 0:                        # if there is a command to assign
                ss.send('sendT')                        # indicate to node that command being sent
                data = ss.recv(1024)                    # receive confirmation
                print 'this is the task i have to assign next ', task2Assign      # for debugging
                sendSt = str(task2Assign[0]) + ',' + str(task2Assign[1])       # formatted string

                #assigned task – so update the no. of proceeses running on each node
                procsOnNode[int(nodeReqd)-1] =  procsOnNode[int(nodeReqd)-1] + 1

                ss.send(sendSt)          # send the command
            ss.close()                   # assigned command – close connection to that node

        else:
            print 'all nodes busy'       # if all the nodes are running 2 commands

    conct2Node = conct2Node + 1    # back to polling – get stats for next node
```

Appendix iii: Code for Client Program

```
#   Course:         COMPSYS 302: Software Design 2E
#   Project 2:      Task Processing Manager
#   Authors:        Thusitha Mabotuwana (9790416)
#                   Mythreyi Ragavan    (9751564)
#   Purpose:        To create a graphical user interface using Python TK for the task processing manager - allows the user to load a task file and view cluster statistics
#   Date Modified:          3 June, 2003.

# Import the required modules
from Tkinter import *
import tkMessageBox
from tkFileDialog import *
import thread, socket, string, time, sys

global root         # so that it can be accessed from all functions
root = Tk()         # create the main window widget

#**********************************************************************
#Class structure for all nodes and the cluster
class EachNode:
   global nodeNo
   # set all the parameters that define each node's display and statistics
   def __init__(self,master,num):
      self.cpuload = 0.00               # default values
      self.freemem = 0
      self.procs = ' '
      self.noOfProcs = 0
      self.usersProc = 0
      self.tasksDone = 0
      self.tasksWait = 0
      self.tasksTotal = 0
      self.tasksRun = 0
      self.newnode = master             # the root widget
      self.nodeNo = num
      self.ifOpen = 0                   # if the window for that node is already open

      # create x and y values in arrays for plotting the graph. These contain 60 points plotted along the width of the canvas - 600 pixels
      self.x_array = []
      self.y_array = []                 # the actual load average
      self.y_vals = []                  # the load average converted to number of pixels
      for o in range(0,60):
         self.x_array.append(30+(o*9))
         self.y_array.append(0)
         self.y_vals.append(0)
      self.max_cpuload = 1.0            # default maximum of the load average array

   # To check whether the instance is for a node or the cluster
   def setNodeFlag(self,flag):
      self.nodeflag = flag
   # To create the layout and set the values of all widgets for each instance
   def display(self):
      global nodeNo                     # number of nodes on the cluster

      # create a new window only if there is none open for that node/cluster
      if not (self.ifOpen):
         # create a new window only for a node, not for the cluster. For the cluster, newnode will be the root window (master)
          if (self.nodeflag):
           self.newnode = Toplevel()
```

```
            self.newnode.title('Node ' + str(self.nodeNo) + ' Statistics')
            self.newnode.geometry('+220+30')            # window placing

        self.ifOpen = 1

        self.main = Frame(self.newnode,width=800, height=400)
        self.main.pack(side=TOP)

        # Create a 'quit' button only for the node windows.
        if (self.nodeflag):
            quitf = Frame(self.main,bd=5)
            quitf.pack(side=TOP)
            quitb = Button(quitf,fg='red',text = 'QUIT',command = self.closeWin)
            quitb.pack(side=TOP)

        graphframe=Frame(self.main,width=800,height=250,relief=RIDGE,bd=10)
        graphframe.pack(side=TOP)

        # to display the process details only for the nodes and not for the cluster
        if (self.nodeflag):
            procframe = Frame(self.main,relief=RIDGE,bd=11,height=350,width=400)
            procframe.pack(side=LEFT)

            lab1 = Label(procframe, text = 'PROCESS DETAILS', fg='red', relief = RAISED, padx=100)
.grid(row=0,column=0,columnspan=2,sticky=NSEW)

            lab2 =  Label(procframe, text = 'Name', fg='dark blue', bg='gray', relief = RAISED)
.grid(row=1,column=0,sticky=NSEW)

            lab3 = Label(procframe, text = 'Status', fg='dark blue', bg ='gray', relief = RAISED)
.grid(row=1,column=1,sticky=NSEW)

            scrollb = Scrollbar(procframe,orient='vertical')
            scrollb.grid(row=2,column=2,sticky=NS)

            # Text box to show the process details. Scroll bar is attached to it
            text = Text(procframe,state=NORMAL,width=45,height=24, yscrollcommand = scrollb.set)
            text.grid(row=2,column=0,columnspan=2)
            text.insert(END,self.procs)  # insert string containing the process details
            text.config(state=DISABLED)          # so that user cannot edit it
            scrollb.config(command=text.yview)

            statsframe = Frame(self.main,bg='gray',width=350,height=350, relief=RIDGE, bd=10)
            statsframe.pack(side=RIGHT)

            temptext = 'STATISTICS SUMMARY'

        # to define frame settings for the cluster instance since there is no 'procsframe'
        else:
            statsframe = Frame(self.main,bg='gray',width=350,height=350,relief=RIDGE, bd=10)
            statsframe.pack(side=TOP)
            temptext = 'CLUSTER SUMMARY'

        # To create gaps to make the layout look clearer – make empty labels
        ygap = Label(statsframe,bg='gray',width=5).grid(row=0,column=0,rowspan=20)
        xgap = Label(statsframe,bg='gray',height=5).grid(row=0,column=1, columnspan=3)

        # Create layout and set values for the statistics displayed as numbers
        l1 = Label(statsframe, text = temptext, relief=RAISED, bd=3, bg='dark blue', fg='gray', font='arial 14
bold').grid(row=4,column=1,columnspan=2)
        lgap = Label(statsframe, bg='gray').grid(row=5, column=1, columnspan=2, sticky=NSEW)
        l2 = Label(statsframe, text = 'CPU Load', relief = RIDGE, bd=5, bg = 'light blue')
.grid(row=6,column=1,sticky=E+W+N+S)
```

```
        l3 = Label(statsframe,text = 'Free Memory (kB)',relief=RIDGE,bd=5,bg='light blue')
.grid(row=7,column=1,sticky=E+W+N+S)
        l5 = Label(statsframe,text = '# Users running processes',relief=RIDGE, bd=5, bg='light blue')
.grid(row=9,column=1,sticky= E+N+S+W)
        l6 = Label(statsframe,text = 'No. of Processes',relief=RIDGE,bd=5,bg='light blue')
.grid(row=10,column=1,sticky=E+N+S+W)
        lv2 = Label(statsframe,text = self.cpuload,relief=RIDGE,bd=5,bg='light blue')
.grid(row=6,column=2,sticky=W+N+S+E)
        lv3 = Label(statsframe,text = self.freemem,relief=RIDGE,bd=5,bg='light blue')
.grid(row=7,column=2,sticky=W+N+S+E)
        lv5 = Label(statsframe,text = self.usersProc,relief=RIDGE,bd=5,bg='light blue')
.grid(row=9,column=2,sticky=W+N+S+E)
        lv6 = Label(statsframe,text = self.noOfProcs,relief=RIDGE,bd=5,bg='light blue')
.grid(row=10,column=2,sticky=W+N+S+E)

         # Display the number of nodes in the cluster summary
        if not self.nodeflag:
            l7 = Label(statsframe,text = 'No. of Nodes',relief=RIDGE,bd=5,bg='light blue')
.grid(row=11,column=1,sticky=NSEW)
            lv7 = Label(statsframe,text = str(nodeNo),relief=RIDGE,bd=5,bg='light blue')
.grid(row=11,column=2,sticky=NSEW)

        xgap2 = Label(statsframe,bg='gray',height=5).grid(row=12,column=1, columnspan=3)
        ygap2 =Label(statsframe,bg='gray',width=5).grid(row=0,column=5,rowspan=20)

        # Call function that draws the graph
        self.createGraph(graphframe)

        #Show summary of user's taskfile – only on the main window for the cluster
        if not (self.nodeflag):
            self.taskf = Frame(self.newnode,width=700)
            self.taskf.pack(side=TOP)
            taskstr = 'Total no. of Commands: ' + str(self.tasksTotal) + '\tCommands Completed: ' +
str(self.tasksDone) + '\tCommands Waiting: ' + str(self.tasksWait) + '\tCommands Processing: ' +
str(self.tasksRun)
            taskprocs = Label(self.taskf,text=taskstr,bg='gray',fg='black',relief=SUNKEN, padx=20)
            taskprocs.pack(side=BOTTOM)

    # To close the window displaying node details – for quit button
    def closeWin(self):
        self.ifOpen = 0
        self.newnode.destroy()

    # To create the graph of CPU Load over the last 3 minutes for each instance
    def createGraph(self,gframe):
        graph = Canvas(gframe,height=200,width=675,bg='black')
        graph.pack(side=TOP)

        # Set title and axis labels
        graph.create_text(325,10, text='Plot of Load Average Over the Last 3 Minutes', fill='red', font = 'arial 12')
        graph.create_text(620,188, text='Elapsed Time(sec)', fill='white')
        graph.create_text(27,10, text='Load Avg', fill='white')

        # To convert the Load Average values into pixels, using a variable y-axis scale - depends on the maximum
value being plotted
        for r in range(0,60):
            t1 = float(self.y_array[r])
            t2 = float(self.max_cpuload) + 0.2       # add offset so that the graph does not get drawn at the very top
            t3 = (t1/t2) * 145                  # height of the graph = 145 pixels. Scale each y-value.
            self.y_vals[r] = 165 - (int(round(t3,0)))   # to turn the graph upside down

        # To create a line graph by drawing lines between consecutive points in the array
        for p in range(0,59):
```

```
            graph.create_line(self.x_array[p], self.y_vals[p], self.x_array[p+1], self.y_vals[p+1], fill='white',
width=3)

        # To create the axes
        graph.create_line(30,20,30,165,fill='blue',width=2,arrow=FIRST)
        graph.create_line(30,165,600,165,fill='blue',width=2,arrow=LAST)

        # To create the ticks and labels on the x-axis – in descending order (since graph for last 180 sec)
        x = 60
        for u in range(0,18):               # 18 ticks and labels every 10 seconds
            graph.create_line(x,160,x,170,fill='blue')
            graph.create_text(x,175,text=str((18-u)*10),fill='white')
            x = x+30

        # To create the ticks and labels on the y-axis -- variable scale depending on the maximum value
        d1 = float(self.max_cpuload) + 0.2
        d2 = d1/10              # 10 divisions to be used
        div_y = round(d2,2)
        temp_y = div_y          # no. of pixels in 1 division = max value / no. of divisions
        y = 151
        for t in range(0,10):
            graph.create_line(25,y,35,y,fill='blue')
            graph.create_text(15,y, text = str(temp_y), fill = 'white')
            temp_y = temp_y + div_y
            y = y – 14

    # ********************************  End of Class    *********************************

    #*****************************************************************************
    # Function to update the statistics of each node whenever the statistics are received from the server
    def updateVals(self,procs,users,cpu,mem,procstr,total,done,running,waiting):
        self.cpuload = cpu
        self.freemem = mem
        self.noOfProcs = procs
        self.usersProc = users
        tempprocs = string.split(procstr, ',' )
        self.procs = ' '

        # to display formatted process names and status in the text box
        y = 1
        while (y < len(tempprocs) - 1):
            self.procs = self.procs + tempprocs[y] + '\t\t    '
            if (len(tempprocs[y]) < 8):             # to align properly – number of tabs
                self.procs = self.procs + '\t    '
            if (tempprocs[y+1] == 'S'):
                tempsr = 'Sleeping'
            elif (tempprocs[y+1] == 'R'):
                tempsr = 'Running'
            elif (tempprocs[y+1] == 'Z'):
                tempsr = 'Zombie'
            self.procs = self.procs + tempsr + '\n'
            y = y + 2
        # Add the latest value of the CPU Load and remove the oldest value – update the array
        self.y_array.append(self.cpuload)
        self.max_cpuload = max(self.y_array)
        del self.y_array[0]


#*********************************************************************
#Function to display 'About' window
def aboutwin():
    about = Toplevel()
    about.title('About Our Program')
```

```
        about.geometry('+275+80')
        screen = Frame(about,width = 400, height=250, bg = 'dark blue')
        screen.pack()
        program = Label(screen,text = 'Task Processing Manager', font = 'Helvetica 24 bold', fg = 'green', bg='dark
blue')
        program.pack(side=TOP, padx=10, pady=10)
        purpose = Label(screen,text = 'Purpose: To efficiently manage the running of tasks on the Department
Cluster', fg = 'light blue', bg = 'dark blue')
        purpose.pack(side=TOP)
        author = Label(screen, text='Developed by:\nThusitha Mabotuwana\nMythreyi Ragavan', fg = 'green', bg =
'dark blue')
        author.pack(side=TOP, pady=10)
        close = Button(screen,text = 'Close this window', fg = 'red', command = about.destroy)
        close.pack(side = BOTTOM, pady = 10)


#*********************************************************************
#Function called when user closes the main window
def exit():
    global exitflag
    exitflag = 1        # flag to disable refresh while the message box is displayed
    if tkMessageBox.askyesno("Exit", "Do you want to close the program?"):
        sys.exit()
    exitflag = 0
    updateAll()         # return control to the update function


#*********************************************************************
#Function to allow user to load a task file from local harddrive using a file dialog window
def loadfile():
    global fileflag, taskfile, sflag

    fileflag = 1       # to disable refresh if the file dialog box is open
    taskfile = askopenfile(title='Select Task File to Load')
    if taskfile:
        sflag = 1       # to let the server know that the task file has been loaded from the local hard drive.
    fileflag = 0
    updateAll()      # return control to update function


#*********************************************************************
#Function to allow user to type the name of a task file that is on PHOBIA
def enterfile():
    global enter, taskfile, cflag

    cflag = 1
    taskfile = enter.get()       # get the file name that was entered by the user
    enter.delete(0,END)

#*************************************************************************************
# Recursive function that keeps refreshing the statistics - destroys and redraws the frames containing the
statistics
def updateAll():
    global cluster, root, nodes, nodeNo
    global fileflag, exitflag

    if not fileflag and not exitflag:   #Disable refresh if any dialog boxes are open. Wait for user response
        cluster.main.destroy()
        cluster.taskf.destroy()
        cluster.display()

        for y in range(0,nodeNo):
            if (nodes[y].ifOpen):           # Redraw the node windows only for those that are open
```

```
            nodes[y].main.destroy()
            nodes[y].display()

        root.after(3000,updateAll)      # Update after every 3 seconds – recursive
        root.mainloop()                 # Event loop


#*****************************************************************************
#                                   Main Code
#*****************************************************************************

HOST = '130.216.216.30'
PORT = 10097
taskfile = ' '

# Thread to continuously receive node statistics from the central manager (server)
def clientT():
    global cflag, nodeNo, nodes, cluster, fileflag, taskfile, killNodePs, sflag
    count = 0
    counter = 0

    # Keep connecting to server and receiving statistics
    while(1):
        # Connect to the server and catch any errors
        try:
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.connect((HOST, PORT))

            # if the task file has been loaded from local harddrive
            if sflag == 1:
                s.send('TF')
                count =2
                sflag = 0
            # if the task file name has been entered (i.e. file is on Phobia)
            elif cflag == 1:
                s.send('from phobia')
                s.recv(1024)
                s.send(taskfile)
                cflag = 0
            # if not sending any task file, then request node statistics
            else:
                s.send('send stats')
            data = s.recv(1024)

            # ** after sending appropriate request to server, send the actual data if server is ready **

            # send the task file line by line to the server if requested
            if data == "send":
                line = taskfile.readline()
                while(line!=""):
                    if line!="\n":
                        s.send(line)
                        data = s.recv(1024)
                    line = taskfile.readline()
                s.send('ENDTF')
                print 'Input file sent '       # for debugging

            # if not sending task file, then receive the statistics from the server
            elif data == 'stats':
                data = s.recv(1024)
                stats = ""
                counter += 1
                while (data):                   # as long as there is data in the receive buffer
```

```
                stats = stats + data
                data = s.recv(1024)

            stats = (string.split(stats,'\n'))        #
            if len(stats[0]) == 2:                    # inactive code = D?, sent by server
                print  'NODE IS INACTIVE'
            else:                                     # update stats for that node
                n = int(stats[0])                     # get the number of the node
              # stats contains: [0] - node no, [1] - processes running, [2] - no of users running processes, [3] - load avg, [4] - freememory, [5] - stats for the node (processes and status), [6] - total no of commands, [7] - commands completed, [8] - commands running, [9] - commands waiting
                nodes[n-1].updateVals(stats[1],stats[2],stats[3],stats[4],stats[5],stats[6],stats[7],stats[8], stats[9])
                cluster.tasksTotal = stats[6]
                cluster.tasksWait = stats[9]
                cluster.tasksDone = stats[7]
                cluster.tasksRun = stats[8]

            s.close()           # close connection after receiving all stats

        # Calculate the cluster summary statistics after getting all the node statistics
        if counter == nodeNo:
            tempavg = 0
            tempprocs = 0
            tempmem = 0
            for e in range(0,nodeNo):
                tempavg = tempavg + float(nodes[e].cpuload)
                tempprocs = tempprocs + int(nodes[e].noOfProcs)
                tempmem = tempmem + float(nodes[e].freemem)
            cluster.cpuload = tempavg                   # sum of all nodes
            cluster.freemem = tempmem                   # sum of all nodes
            cluster.noOfProcs = tempprocs               # sumof all nodes
            cluster.y_array.append(cluster.cpuload)     # add cpu load value to array for plotting
            cluster.max_cpuload = max(cluster.y_array)
            cluster.usersProc = nodes[0].usersProc      # no. of users on cluster = no. on node 1
            del cluster.y_array[0]

            time.sleep(2)               # wait after getting all node statistics before getting the next round
            counter = 0

    except KeyboardInterrupt:       # catch Ctrl-C
        print 'Client thread exiting'
        thread.exit()
#*******************************************************************************
# Back to Main Code
cflag = 0
sflag = 0
thread.start_new_thread(clientT,())
fileflag = 0
exitflag = 0

#*******************************************************************************
# Initialisations
cluster = EachNode(root,1)
nodeNo = 9
nodes = []                  # array of node instances (to account for any number)
for j in range(0,nodeNo):
  nodes.append(EachNode(root,j+1))

# To add menu bar
menu = Menu(root)
root.config(menu=menu)
filemenu = Menu(menu, tearoff=0)
menu.add_cascade(label = "File", menu = filemenu)
```

```
filemenu.add_command(label = "Exit", command = exit)

#Use menu to allow user to decide which node to display statistics for
statmenu = Menu(menu, tearoff=0)
menu.add_cascade(label = "Select Node", menu = statmenu)
for i in range(1,nodeNo+1):
    tempstr = 'Node ' + str(i)
    nodes[i-1].setNodeFlag(1)
    statmenu.add_command(label = tempstr, command = nodes[i-1].display)

helpmenu = Menu(menu,tearoff=0)
menu.add_cascade(label = "Help", menu = helpmenu)
helpmenu.add_command(label = "About", command = aboutwin)

# Display the cluster window
cluster.setNodeFlag(0)
cluster.display()

bframe = Frame(root, bd = 5, relief = GROOVE)
bframe.pack(side=TOP)
loadbutton = Button(bframe, text = 'Select TaskFile', fg = "blue", command = loadfile)
loadbutton.pack(side = LEFT, anchor = W, padx = 50)

enter = Entry(bframe)      # for user to enter the name of a task file
enter.pack(side=LEFT)

enterbutton = Button (bframe, text='Load Entered TaskFile', fg='blue', command =enterfile )
enterbutton.pack(side=LEFT)

quitbutton = Button(bframe, text="QUIT", fg="red", command=exit)
quitbutton.pack(side=LEFT,padx=50,anchor=E)

root.protocol("WM_DELETE_WINDOW", exit)      # catch if user clicks on the 'X' of the window.
root.title('Task Processing Manager')
root.geometry('+225+0')

updateAll()
```

Appendix iv: Code for the Class that Stores Statistics

```
#   Course:         COMPSYS 302: Software Design 2E
#   Project 2:      Task Processing Manager
#   Authors:        Thusitha Mabotuwana (9790416)
#                   Mythreyi Ragavan    (9751564)
#   Purpose:        To create a class structure on the central manager to gather statistics for each node.
#   Last Modified:  June 3, 2003.


import string, re

# Class to store processes  running on each node and their status, free memory available and CPU load
class NodeStruct:

   # Function called when instance created – to initialize variables
   def __init__(self):
      self.st = ""
      self.sts = ""


   # This function stores the processes running, their status, free memory available and CPU load for a given
node
   def setNode(self,stats):
      self.st = stats                  # statistics string received from the node
      x = string.split(stats, "\n")
      self.load = x[0]                 # 1st element after splitting 'stats' is load average for the node
      self.freemem = x[1]              # 2nd element is free memory


   # Function to return the product of (1/CPU load * free mem) to determine the best node
   def getProduct(self):
      if float(self.load) != 0:
         return float(self.freemem) / float(self.load)
      else:
         return self.freemem           # if the load average = 0


   # Function to return the 'stats' string stored for the given instance (ie. for a given node) of the class
   def getStats(self):
      return self.st
```