Balancing the Load of the Workstations of a Distributed System

6.1 Introduction
6.2 Load Balancing and Load Sharing
6.3 The Goal
6.4 The Mechanism
6.5 Concluding Remarks
CHAPTER – 6

Balancing the Load of the Workstations of a Distributed System

6.1 Introduction

The usage of distributed computing systems is increasing day by day because of the availability of prevailing computing hardware at lower cost and revolutions in the networking and communication technologies. Some of the significant benefits of distributed systems include the strength of resource sharing to serve the users with a rich collection of resources which are usually spread across through the member workstations of a network and the load balancing of network’s workstations [N. A. Joshi, 12b].

The last decade has observed the development of tightly coupled parallel multi-processor systems which have been trendy in high performance computing environments. By connecting simpler, smaller and cheaper processors and other components, a computing system may be assembled with the same processing capacity as that of one utilizing a highly complex, large and powerful processor but for a fraction of the price. There exist number of drawbacks with such parallel computing systems like a limited support or lacking of fault tolerance and limited or negligible support in hardware upgrading [M. Liu, 09].

On the other side, the advancements in microelectronic technology and communication technology has resulted in the availability of fast, inexpensive processors and cost-effective “loosely-coupled systems” known as “Distributed systems” in which software components located at networked nodes communicate and cooperate their actions to achieve the benefits like – information sharing, resource sharing, better flexibility, higher throughput and higher reliability. It is
practically possible to either insert additional workstations to or remove existing workstations from the distributed systems. It means that the limitations of the parallel computing architectures are minimized with the distributed systems [M. Liu, 09].

The overall computational capability of the interconnected workstations across the network may theoretically seem to be too much larger than the capability of the isolated uniprocessor system. It is common that, in a computing environment with a number of workstations connected within a network, some workstation-users may repetitively execute some applications on a particular workstation, or the workstation may be running processes slower than other workstations; this may gradually overload the workstation. On another side some remaining workstations may be left almost idle, where the processes may gain benefit of the computing power that goes unused. This means, there are chances that some of the workstations are heavily loaded, while others are left almost idle; which in effect causes utilization of only a fraction of the network’s theoretical computational power. Moreover, chances are there in distributed system that, there could at least one workstation be idle or lightly loaded, whereas some other workstations are overloaded; this may happen because of statistical variations in the process arrival times at the workstation and the variations in the service times of the processes [M. Livny, 82].

The discussion made above suggests that performance gains may be achieved by transferring processes from the currently heavily loaded workstations to either the idle workstations or the lightly-loaded workstations. Such a technique of computing-power sharing for the purpose of improving the performance of a distributed system by redirecting the workload among the available workstations is known as ‘Load Balancing’ or ‘Load Sharing’ [N. G. Shivaratri, 92]. Moreover, [Z. Khan, 10] and [N. G. Shivaratri, 92] suggest that sharing the
workload among the interconnected workstations may enhance the overall performance of the distributed system and raise system utilization.

### 6.2 Load Balancing & Load Sharing

There are different techniques available to attain better performance from the network of workstations. Among such techniques, the load sharing or load balancing technique may be implemented by migration of already executing processes from the highly loaded workstations to the lightly-loaded or idle workstations within the network. This chapter focuses on the problem of workload redistribution among the workstations of a network in order to improve overall performance. The chapter describes some mechanisms for load balancing to divide up the workload of the workstations belonging to the network (through the process migration mechanism) to gain enhanced performance from the overall network of workstations.

The problem of load balancing has been studied using a number of different approaches for long time. The earlier works primarily focused on static load balancing [S. Sharma, 08]. In the static load balancing mechanism, the process transfer decisions are made probabilistically without taking into consideration the current runtime state of the system, i.e. the load transfer decisions are hardcoded inside a load-balancing mechanism using some previously gained working information about the system [A. B. Saxena, 11]. Static load balancing is simple and effective when the workload can be sufficiently well characterized beforehand, but the mechanism fails to adjust to the fluctuations in the system load [K. M. Kavi, 95]. For example, some probabilistic algorithm based upon the static load balancing approach, may simply follow the round-robin approach or assign a process to a certain workstation $W$ with probability $P$; where the probability has been pre-calculated statically on the basis of some
aspects such as the average process-creation and process-completion rates for each workstation. One more example of static load balancing is cyclic splitting in which among the N number of available workstations, each workstation allocates its i\textsuperscript{th} newly created process to the (i mod N)\textsuperscript{th} workstation.

In distinction, the dynamic load balancing approach is a deterministic approach which attempts to balance the system’s load dynamically as the processes arrive to the system. As the dynamic algorithms have to maintain and analyze the system’s state information dynamically, they may cost some overhead to the overall system, such overhead may be ignored with respect to the limitations incurred with the static approach of load balancing [N. G. Shivaratri, 92]. Moreover, [P. Krueger, 88] also has suggested another classification of load balancing such as preemptive and non-preemptive load balancing.

Therefore, in this chapter, we describe dynamic load balancing algorithms to achieve better runtime performance which is gained by distributing the processes as they arrive by considering the system’s current state information. E.g. a dynamic load balancing algorithm may consider the workload of all the workstations at the time of decision making for workload distribution.

The benefit of load balancing can be achieved through either the server-managed load balancing or the client-initiated load balancing, both of them having specific advantages and drawbacks. In this chapter, both of these load balancing approaches have been supported by the presented mechanism in order to gain the benefits like reliability, fault tolerance and efficiency. The mechanism comprises of the tasks like maintaining workload information of the interconnected workstations on some dedicated workstation, determining appropriate workstations for distribution of workloads, and transferring the desired processes from one workstation to
another workstation, which have been suggested in this chapter. In order to perform process migration, the mechanism takes help of the concepts discussed in the previous chapters [N. A. Joshi, 12b].

6.3 The Goal

The proposed load balancing technique has been designed with certain goals described in this section.

Load balancing and Load Sharing

In research, the two terms load balancing and load sharing are used quite interchangeably. The concept of load balancing refers to the network of workstations where algorithms are applied to make equal the workload of each of the workstation in the network. This varies in the general concept such as workload-distribution which can be implied by the concept of load sharing. The term load sharing designates the practice of ensuring that no workstation remains idle while work is under queue at other overloaded workstations in the network, which leads to the efficient usage of the resources which are spread across the network. [M. Nuttall, 97] suggests that, in practice, the theoretical distinction is not very imperative; as most of such algorithms fall somewhere between the two terms and do more than ensuring that no host remains idle.

Therefore, we intend to efficiently use the overall processing power of the workstations in the network by the load sharing mechanism in order to decrease the workload of the overloaded workstations.

Fault Tolerance

The above described objective of transferring workload among the networked workstations may be achieved with either of the following two load balancing approaches:

(1) Server-managed load balancing and
(2) client-initiated load balancing.

**Server-managed load balancing**

The server-managed load balancing approach implies a centralized policy of load balancing over the network of workstations. In server-managed load balancing, one of the workstation in the network is assigned responsibility to carry out the decision making involved in the mechanism of load balancing. Such a dedicated workstation first identifies the overloaded workstations and the lightly-loaded workstations, and then instructs the overloaded workstation to share its load with the idle or least-loaded workstation in the network.

The server-managed load balancing approach maintains control over all the workstations of the network. Moreover, it frees up rest of the workstations in the network from performing the tasks like determining the workstations which are having the highest or the lowest load. Sever-managed mechanism requires lesser communication among the peer workstations in the network as the workstations interact with the central server and not with the other n number of workstations [A. Ali, 10]. In this chapter we have described the algorithm for server-managed load balancing policy.

However, one of the drawbacks with this approach is the unexpected unavailability of the server-managed load balancing service (e.g. the workstation running the server-managed policy is either not running or crashes down); this may prevent other workstations from taking benefits of the load balancing service.

**Client-initiated load balancing**

Whereas, the second approach of client-initiated load balancing operates on the client side, i.e. the peer-workstations. In client-initiated load balancing technique, the workstations are allowed to
take load-balancing decisions themselves. The agents residing on the client workstations are assigned such responsibility.

This second approach of load balancing especially becomes useful in the circumstances such as either the server-managed policy is not under operation or the workstation (which is) running the server-managed policy is in faulty-state.

Therefore, to meet the goal of fault tolerance, in this chapter, we propose the client-initiated load sharing mechanism also, which becomes significant especially in the absence of the server-managed load balancing technique.

**6.4 The Mechanism**

We describe both of the mechanisms to enable load distribution among the networked workstations: the server-managed load balancing mechanism (i.e. the centralized load balancing technique) and the client-initiated load balancing mechanism (i.e. the decentralized load balancing technique). In order to facilitate the server-managed mechanism, in the network of workstations, we dedicate one of the workstation as a server and treat rest of the workstations as clients. In the centralized technique, the server is assigned certain responsibilities; while in the decentralized technique, the peer workstations are assigned responsibility to make decision about workload transfer [N. A. Joshi, 12b].

In the server-managed load balancing technique, the client workstations periodically send their workload information to the server. On the server side, the server iteratively and concurrently performs various functions such as it receives the clients’ workload information, maintains the received workload information in server’s repository and accomplishes the job of load sharing. To act upon the
load sharing job, the server determines the highly loaded and lightly loaded clients in the network using the workload information available in the server repository and asks the highly loaded client to share workload with the lightly loaded client [N. A. Joshi, 12b].

An important attribute of the proposed load balancing mechanism is that it protects the lightly loaded clients from an instantaneous overburdening; which may be accidentally enforced by the load balancing mechanism as the mechanism may stereotypically choose the same lightly loaded client as a destination for workload transfer from highly loaded client(s) for several instantaneous load balancing decisions. E.g. suppose at time $T_0$ some client $C_1$ is determined as the lightly loaded client and therefore it may be chosen as a destination for workload sharing. Because of the latency in process migration, the actual workload transfer and process resumption will take place after some time $T_m$ on the client $C_l$. As a result, in the intermediate time ($T_m - T_0$), the client $C_l$ will remain lightly-loaded. And that is why during the intermediate time period, the load balancing policy may choose and choose it as a destination to receive additional and additional workload; which may lead the client towards overburdening.

To achieve, the above discussed attribute, we maintain the current status information of all clients on server. The current status can be one of the three status values: INACTIVE, ACTIVE and PASSIVE. The client, whose workload information is not available (or the available information is very older i.e. invalid) to server, is considered as an INACTIVE client, such a client is not considered for participation in load balancing. The client, whose valid workload information is available and which is not chosen as a destination of workload transfer is considered as an ACTIVE client, only ACTIVE clients may be chosen as a destination for workload transfer. Otherwise, the client whose valid workload information is available, but which is chosen as a destination of a workload transfer is considered as a PASSIVE client.
for certain time period $T_p$, meaning that the client is already chosen earlier before short time to receive additional workload and therefore it can not be again chosen as a destination for additional workload transfer i.e. the client is not available for load balancing before completion of the time duration $T_p$. (Such a concept will protect the lightly loaded client from a sudden overburdening.) After completion of the passive-time duration $T_p$, the client’s status will be reset to ACTIVE; thereby green signaling the client’s availability for the further incoming load balancing requests.

Likewise, we apply nearly similar concept to the client-initiated load balancing mechanism. In client-initiated load balancing technique, the highly loaded workstation sends requests (i.e. the request sender) for workload transfer to the peer workstations (i.e the request listeners) in the network. If the request listener workstation, accepts the workload transfer request, it immediately does not listen for more requests for some time period in order to protect itself from a sudden overburdening.

In the proposed server-managed load distribution mechanism, the server is responsible for the following activities:

1. **Manage workload information**
   The server has to maintain the workload information about all the clients of the network.

2. **Manage the current status information**
   The server has to keep track of the current status (ACTIVE, PASSIVE, INACTIVE) of clients.

3. **Decision making**
   The server has to carry out load distribution by determining the highly loaded and lightly loaded clients in the network.

4. **Workload transfer**
   The server instructs the highly loaded client to carry out process migration towards the lightly loaded workstation.
6.4.1 The Modules

We describe the suggested load distribution mechanism in the form of the following modules (figure A on pg. 158):

1. LoadPortal module
   The LoadPortal module gathers the workload information of the workstation and provides the gathered details to the LoadServer module.

2. LoadServer module
   The LoadServer module is responsible to maintain the workload and the current status information of all the clients in the network, and determine the highly loaded and lightly loaded clients.

3. LoadBalancer module
   The LoadBalancer module performs the job of workload transfer from the highly loaded workstation to the lightly loaded workstation with the help of ProcessMigrator module.

4. ClientStatusMaintainer module
   The ClientStatusMaintainer module maintains and periodically updates the current status information of the network clients.

5. RequestSender module
   The RequestSender module sends the workload transfer requests to the peer workstations (on round robin basis) and waits for the reply from the peer workstations. If the reply is affirmative then the sender performs process migration with the help of the ProcessMigrator module.

6. RequestListener module
   The RequestListener module resides on peer workstations and listens for workload transfer requests which are sent by the RequestSender module. It decides whether to accept the requests or not and replies the decision back to the request sender.

7. ProcessMigrator module
The ProcessMigrator module performs migration of a process from its originating workstation to the destination workstation.

Here, we describe the functionality and major work of the above listed modules to synthesize the load distribution mechanism [N. A. Joshi, 12b]:

1. Create and implement a software module LoadPortal which performs the following work:
   a. It continuously executes on the participating client workstations (in the load balancing activity) in the network.
   b. It determines the workload information of the workstations periodically.
   c. And periodically sends the above gathered workload details to the LoadServer module (which executes on the server).

2. On the server side, create and implement a software module LoadServer which performs the following work:
   a. It continuously takes note about the workload information (which is periodically sent by the client workstations in the network), receives the information when it arrives to the server system and maintains the information in its repository.
   b. From the workload information gathered in the step 2.a, the module determines the highly loaded client whose current status is ACTIVE; the LoadBalancer module will make efforts to reduce the workload of this highly loaded client, say it client_A.
   c. From the workload information gathered in the step 2.a, the module determines the lightly loaded client whose current status is ACTIVE; the LoadBalancer module will
make efforts to transfer workload from the highly loaded client_A to this lightly loaded client, say it client_B.

d. Immediately, this module toggles the current status of the above chosen lightly loaded client_B (2.c) into PASSIVE state from ACTIVE state.

3. On the server side, create and implement a software module ‘LoadBalancer’ which performs the following work:
   a. It runs continuously and concurrently with the LoadServer module.
   b. It determines the highly loaded and lightly loaded clients with help of the LoadServer module.
   c. It instructs the highly loaded client_A to transfer its workload to the lightly loaded client_B.

   Thus, the LoadBalancer module provides the way out to attain the server-managed load balancing.

4. Furthermore, create and implement one more module ClientStatusMaintainer on the server-side which performs the following work:
   a. It also runs continuously and concurrently with the LoadBalancer and LoadServer modules.
   b. It periodically (say after some pre-decided time period $T_p$) toggles the current status into ACTIVE state from PASSIVE state (i.e. PASSIVE $\rightarrow$ ACTIVE) of the client which was earlier (i.e. before time period $T_p$) selected for workload transfer by the LoadServer module.
   c. Moreover, it toggles the current status from ACTIVE state into INACTIVE state (i.e. ACTIVE $\rightarrow$ INACTIVE) of the client whose workload information is either not available or not latest.

   Thus, the work discussed in the above points 2.d and 4 helps the proposed server-managed load balancing mechanism to
protect the lightly loaded clients from a sudden overburdening and thereby providing the efficient load balancing.

5. Create and implement the module RequestSender on the peer workstations which performs the following work:
   a. It continuously runs and determines the client’s workload information.
   b. It periodically determines whether it has become highly overloaded or not.
   c. If it determines its workstation as highly overloaded, then it sends the workload-transfer request to the peer workstations (RequestListener module).
   d. The requests are sent to the peer workstations on the round robin basis.
   e. It waits for the reply from the peer workstations. If the reply is affirmative then it performs process migration with help of the ProcessMigrator module; otherwise the step 5.d is followed for next workstation in turn as per the round robin policy.

6. Create and implement the module RequestListener on the peer workstations which performs the following work:
   a. It continuously listens for workload transfer requests which are sent by the RequestSender module.
   b. It decides whether to accept the requests or not and replies the decision back to the request sender.
   c. If it decides to accept the workload transfer request in the above step, then for next some time duration (say $T_p$), it doesn’t accept further incoming requests (for workload transfer).

Thus, the LoadBalancer module provides the way out to attain the client-initiated load balancing. Moreover, the work discussed in the above points 5.d and 6.c helps the suggested
client-initiated load balancing mechanism to work efficiently.

7. On the workstations, implement one more module ProcessMigrator with the following features:
   a. It iteratively listens for the requests to perform process migration; the request comprises of the information like what is to be migrated i.e. the process identifier, and where the migration has to take place i.e. the information about the destination workstation.
   b. After receiving the process migration request, the ProcessMigrator module performs migration of the desired process to the destination client_B by using the process checkpointing and restore mechanisms. The process checkpointing involves capturing current state information of the desired partially-executed process, migration of the captures state information to the destination workstation, restoration of this state information on the destination workstation and resumption of the restored process on destination workstation.

   The mechanism for capturing and restoration of state information of the desired process involves gathering information of the memory regions occupied by the process which is described in the fourth chapter [N. A. Joshi, 10] and [N. A. Joshi, 11], the details of system call which is currently being executed by the process which is described in the first chapter [N. A. Joshi, 09] and the credentials possessed by the process which is described in the fifth chapter [N. A. Joshi, 12a].

Thus, the above discussed mechanism describes both of the load balancing approaches – the server managed and the client initiated mechanisms and helps to balance the workload of overloaded nodes.
6.4.2 The Implementations

Here, we go over to some of the foremost implementation data structures and functions in brief. In order to maintain the workload information of a workstation (on the client) and the information of all the workstations (on the server) we have implemented the data structures `struct Load` and `struct Load_of_Clients` respectively. The client sends its workload information to the server in the form of the variable of type `struct Load`:

```c
enum workstation_status {INACTIVE=0, ACTIVE=1, PASSIVE=2};

struct Load
{
    /* IP address of the associated workstation */
    char ip_address[16];

    /* workstation’s past load history */
    double cur_load;

    //following fields are only for server-side.
    int current_status; // e.g. PASSIVE, ACTIVE, INACTIVE
    /* the time when load info.reaches to server */
    unsigned long time_of_active_inactive_set;
    /* time when current_status is set to PASSIVE
    unsigned long time_of_passive_set;
};

struct Load_of_Clients
{
    /* represents the workload table of all clients */
    struct Load load_arr[MAX_CLIENTS];
    /* number of clients in the network */
    int total;
```
The Load_of_Clients acts as a repository (on server) that encapsulates an array of load information which is sent by the clients.

In the beginning, during the LoadServer module’s initialization phase, the repository would be empty, i.e. yet the LoadServer module has not received workload information form none of its client workstations. So, reset the data member load_of_clients.total.

```c
void init_Load_of_Clients_struct()
{
    load_of_clients.total=-1;
    /* we initialize ‘total’ with -1 (instead of 0) as ‘total’ also acts as an index to the load_arr[] data member of the load_of_clientsstructure variable.*/
}
```

The LoadServer module consists of the following two concurrently running functions on the server which are responsible to receive and manage the workload information sent to the LoadServer module by the client.

```c
//run on server
void recv_and_process_load_info(void* arg)
{
    struct Load load;
    int bytes_read;
    int client_socket_fd = *(int*)arg;

    while(1) //always run
    {
        //Receive workload info. sent by client
        bytes_read = recv(client_socket_fd,
                           (void*) &load,
                           sizeof(struct Load), 0);
```
/ * handover the processing task to other function. It is described below. */

process_client_load_info(&load); // see below
}
}

void process_client_load_info(struct Load *pload)
{
    int i;
    int found = 0;
    /* return if load table overflows */
    if(load_of_clients.total >= MAX_CLIENTS-1)
        return;
    /*
    traverse through the load table until it the workstation is found in the load table
    */
    for(i=0;i<=load_of_clients.total&& found==0;i++)
    {
        if(strcmp(load_of_clients.load_arr[i].ip_address,
            pload->ip_address) == 0)
            {found=1;
            /*
            client’s details already exist in the load table, so update the details (instead of creating a new entry in the table)
            */
            strcpy(load_of_clients.load_arr[i].avg_load,
                pload->avg_load);
            load_of_clients.load_arr[i].cur_load =
                pload->cur_load;
Chapter-6: Balancing the Load of the Workstations of a Distributed System

```c
/*if current_status of pload client is not PASSIVE, then set it to ACTIVE*/
/*set time_of_active_inactive_set of pload client to current time*/
...
return;
}
}

/*
traversal throughout the table is over, but workstation’s details are not found in it. So, create a new entry for this workstation in the load table */
if(!found)
{
/*set current_status of pload client to ACTIVE state */
/*set time_of_active_inactive_set of pload client to current time*/

    load_of_clients.load_arr[++(load_of_clients.total)]=
        *pload;
}

...}

As described in step 3, the server runs LoadBalancer module; some of its implementation steps are described below in the form of balancer_thread_fn() which runs concurrently in a separate thread of execution.

void balancer_thread_fn()
//represents server-managed load balancing
{
```
/*indicates highly loaded workstation*/
struct Load high_load;
/*indicates lightly loaded workstation*/
struct Load light_load;

//indicates sleeping time-interval for this function
int sleep_interval = 10;

while(1)  //run iteratively
{
    sleep(sleep_interval);

    /* determine lightly loaded workstation */
    if(find_least_loaded_client(&light_load)==0)
        continue;  /*no more active-clients in network*/
    /*set current_status of light_load client to PASSIVE*/
    /*set time_of_passive_set of pload client to current time*/
    ...
    /* determine highly loaded workstation */
    if(find_max_loaded_client(&high_load)==0)
        continue;
    /*
    Otherwise, we have determined the highly-loaded and lightly-loaded workstations.
    */
    if(strcmp(high_load.ip_address,
                light_load.ip_address) == 0)
    {
        continue;
        /*
        both variables refer to same ip-address,
Chapter-6: Balancing the Load of the Workstations of a Distributed System

The `send_migration_order()` function sends the details of lightly loaded workstation (i.e. `light_load`) to the highly loaded workstation (i.e. `high_load`).

```c
int send_migration_order(struct Load high_load, 
                        struct Load light_load)
{
    int socket_fd = 0, ret;
    socket_fd = connect_to_target(high_load.ip_address, 
                                   PORT_NO);
    /* here PORT_NO, represents port-number of the 
       highly-loaded workstation */
    if(socket_fd == -1) //error
        return -1;

    ret = send(socket_fd, (void*) &light_load, 
               sizeof(struct Load), 0);
    ...
}
```
As described in step 4, the server runs ClientStatusMaintainer module; some of its implementation steps are described below in the form of `clients_status_mgmt_thread_fn()` which runs concurrently in a separate thread of execution.

```c
void clients_status_mgmt_thread_fn()
{
    int i;
    /* least-loaded client is kept PASSIVE for 20sec. */
    int PASSIVE_DURATION = 20;
    /* client is considered ACTIVE for 20sec in absence of latest load-info on server; otherwise turn its status to INACTIVE */
    int ACTIVE_DURATION = 20;
    /* current thread will sleep for 10sec. */
    int current_thread_sleep = 10;

    while(1) // always run
    {
        /* after completing each iteration, this thread sleeps for some time */
        sleep(current_thread_sleep);

        // traverse through the load table
        for(i=0; i<=load_of_clients.total; i++)
        {
            if(load_of_clients.load_arr[i].current_status == PASSIVE)
            {
                if(get_current_time() -
                   load_of_clients.load_arr[i].time_of_passive_set
                   >= PASSIVE_DURATION)
```
The thread execution traverses through the load table. While traversing it first searches for the PASSIVE clients whose duration to remain passive is completed; such clients’ current status is toggled to ACTIVE. Further, the thread searches the ACTIVE clients whose available workload information is not latest; such clients’ current status is toggled to INACTIVE.

Moreover, some of the implementation details about the threads and functions running on the client-workstations have been briefed below: The main() routine runs two threads of execution.

int main()
{
    int server_socket_fd;
    if(bind_to_and_listen_on_port() == -1)
        return -1; // Error: could not bind to local port
while(1)
{
    pthread_t client_thread;
    ...
    server_socket_fd= /*connect to server in order to receive migration orders */;
    if(server_socket_fd == -1)
        return -1; // could not connect to server

    pthread_create(
        &client_thread,
        NULL,
        (void*)&thread_fn_to_recv_and_perform_servers_order,
        (void*)&server_socket_fd//function's parameter
    );
    /*Create a thread and bind the newly created thread with the below described function thread_fn_to_recv_and_perform_servers_order()*/
    ...
}
...

The following function is a thread-based function that runs on the client workstation and iteratively listens for the server’s instruction (for process migration). After receiving migration order from the server the function migrates the desired process to lightly loaded workstation.

void thread_fn_to_recv_and_perform_servers_order(void* arg)
{
    int server_socket_fd = *(int*)arg;
    struct Load light_load; //will be sent by server
```c
int victim_process_id = /*process-id of desired
    process which is to be migrated */
...
while(1)
{
...
    /*Receive the migration-order sent by server*/
    recv(server_socket_fd,
         (void*)&light_load, //lightly-loaded node
         sizeof(struct Load),0);
    /* migrate the desired i.e. victim process to
    lightly loaded workstation */
    migrate_process(victim_process_id,
                    light_load.ip_address);
}
...
}
```

### 6.5 Concluding Remarks

In the initial part of this chapter, we have described significance of the load balancing technique with the help of the process migration technique in the context of distributed computing systems. Certain issues appearing in the area of load balancing such as – static load balancing and dynamic load balancing have been discussed in this chapter.

Furthermore, the goal of fault tolerance is emphasized behind the design of the presented solutions in this chapter. The chapter suggests the dynamic load balancing solution in the form of both the server-managed and the client-initiated algorithms and presents implementation of these algorithms.
The suggested algorithms provide novel mechanisms through which the lightly loaded workstations do not suffer from sudden overburdening caused by the server’s consecutive multiple load sharing advises. Some of the important implementation functions also have been explained in this chapter.