2009

Decentralized Resource Availability Prediction in Peer-to-Peer Desktop Grids

Karthick Ramachandran

DECENTRALIZED RESOURCE AVAILABILITY PREDICTION IN PEER-TO-PEER DESKTOP GRIDS

(Spine Title: Resource Availability Prediction in P2P Desktop Grids)
(Thesis Format: Monograph)

by

Karthick Ramachandran

Graduate Program in Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science

School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

Abstract

Grid computing is a form of distributed computing which is used by an organization to handle its long-running computational tasks. Volunteer computing (desktop grid) is a type of grid computing that uses idle CPU cycles donated voluntarily by users, to run its tasks. In a desktop grid model, the resources are not dedicated. The job (computational task) is submitted for execution in the resource only when the resource is idle. There is no guarantee that the job which has started to execute in a resource will complete its execution without any disruption from user activity (such as keyboard click or mouse move). This problem becomes more challenging in a Peer-to-Peer (P2P) model of desktop grids where there is no central server which takes the decision on whether to allocate a job to a resource.

In this thesis we propose and implement a P2P desktop grid framework which does resource availability prediction. We try to improve the predictability of the system, by submitting the jobs on machines which have a higher probability of being available at a given time. We benchmark our framework and provide an analysis of our results.

Keywords: Desktop Grid, Volunteer Computing, Cloud Computing, P2P Computing

For my parents

Acknowledgements

I am deeply indebted to my supervisors, Dr. Hanan Lutfiyya and Dr. Mark Perry for giving me a chance to pursue this work. This work would not have been possible without their continued support, patient guidance and encouragement.

I would like to thank the computer science department systems group, Art Mulder, Scott Feeney, David Wiseman and David Martin for letting me install and test the framework in the computer science labs. Special thanks to Art Mulder for his patience and support during the data collection phase of the research.

I would also like to thank my colleagues at DiGS (Distributed and Grid Systems) research group for fruitful discussions and their help and support throughout the year. Special thanks to Abhishek Singh of IIT Delhi, India who developed the initial version of the monitoring module.

To my sister, all my friends here and back home in India for their constant motivation, strength and entertainment.

Finally, this thesis is for my parents, who have always put my happiness before theirs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is an irony in life that the most wasted resources are the ones which are most important to us. Computing power is not an exception.

A grid system can be defined as a large-scale distributed hardware and software infrastructure composed of heterogeneous networked resources which are coordinated to provide transparent, dependable, pervasive and consistent computing support to a wide range of applications [9]. Grid systems came into existence as an answer to the scientific community which is in need of huge processing power for research. Large scale grids like Garuda [4], D-Grid [3], National Grid Service [5] are built to provide the required CPU cycles. However the installation and maintenance of a dedicated grid architecture is expensive.

Large amounts of processing power end up idle in desktop machines around the globe. This saw the emergence of volunteer computing (desktop grid), where users voluntarily donate idle CPU cycles for finding extra terrestrial life or to find a cure to a disease [8]. Users, who are interested in joining a volunteer computing project [10] download and install a client component in their computer from the project's server. Whenever the desktop becomes idle, the installed client gets a work-unit (job) from a centralized server, executes it and returns the result back to the server. Volunteer computing projects have been combined to create more than 1000 TeraFlops of CPU cycles to form the most powerful supercomputer in the world [10]. Well-known systems include BOINC [8], XtremWeb [20], Entropia [14] and Condor [25]. Chapter 2 briefly describes these systems with a focus on resource selection.

1.1 Problem Statement

A traditional grid system comprises of dedicated resources. A job submitted to a resource for execution is guaranteed to complete its execution in that resource, unless there are any hardware or software failures. However, in a desktop grid model, the resources are not dedicated. The job is submitted for execution in the resource only when the resource is idle. There is no guarantee that the job which has started to execute in a resource will complete its execution without any disruption from user activity. This problem becomes more challenging in a Peer-to-Peer (P2P) model of desktop grids where there is no central server which takes the decision on whether to allocate a job to a resource (resource selection).

P2P system removes the responsibility of resource discovery from a single dedicated server and shares it with multiple server network. Therefore unlike the client-server model, where there is a need to upgrade the server proportionally to number of clients in the network, a P2P model needs no change in the network topology when the number of clients in the system increases.

The focus of the thesis is on resource availability prediction in a P2P network. When there is no central server for resource selection, each peer needs to predict its availability. In a university lab like scenario, when the resources are not used by one particular person (non-dedicated desktops), the usage pattern of a group of resources has to be taken into account. This issue of resource availability prediction has not been directly addressed by other work, particularly in a P2P network. We examine this problem in an effort to reduce the turnaround time for a job, thereby more effectively using the resources available.

1.2 Outline of Proposed Solution

This thesis proposes to build a stable peer to peer cloud infrastructure in which the desktop users can rent their CPU cycles for research or other industrial number crunching jobs. Two issues to be addressed include *scalability* and *predictability*.

The number of participating nodes, can grow exponentially over time and make the system unstable. A P2P network with a decentralized resource matching mechanism is built to make the system less vulnerable to an increase in size. This is implemented using JXTA's resource matching mechanism [26].

A desktop grid consists of resources which are not available all the time. A system's predictability suffers when the infrastructure is built over non-dedicated

resources. The goal of the work is to improve the predictability of the grid system by scheduling jobs on those machines which have a higher probability of being available for the amount of time the job is expected to execute. This is achieved by learning the machine's availability patterns over a duration and using that to predict its future availability. Each peer learns its own behavior and predicts its future availability based on it past behavior. In the case of non-dedicated desktop resources such as machines in a university lab, each machine in the group will study the usage pattern of the every other machine within that group. This data will be used to calculate the total group availability, which will be taken into account along with the individual machine's availability while predicting.

1.3 Contribution

This thesis presents a framework for resource availability prediction for a machine or a group of machines in desktop grids on a P2P network. We provide a working prototype of a desktop grid framework, that uses a decentralized resource matching mechanism. Our system extends work already done in this area, by building an infrastructure to do user availability prediction in a P2P environment. It provides a method of sharing between the desktop grid providers and the users willing to donate CPU cycles. It will further serve as a solid framework for researchers to continue work in this area.

1.4 Thesis Outline

In Chapter 2 we present an introduction to volunteer computing and cloud computing. We also describe resource selection in desktop grids and look briefly at some existing frameworks for resource selection. In Chapter 3 we present the functional and the resource requirements of our system. We also describe some of the challenges faced in meeting the requirements stated in this chapter. Chapter 4 presents the overall architecture of our system with the description of individual components. Chapter 5 presents an overview of our system implementation. The implementation leads to Chapter 6 in which we present our verification and validation methods and experimental results. In Chapter 7 we present our conclusions and discuss future work to be done in this area.

Chapter 2

Background

This chapter provides an introduction to relevant terms and concepts associated with volunteer computing and cloud computing. Terminology is presented in Section 2.1. Section 2.2 and Section 2.3 introduces Volunteer and Cloud Computing. Section 2.4 gives a detailed overview of resource selection in desktop grids.

2.1 Terminology

This section briefly describes the terminology used in the thesis.

1. Distributed System: A distributed system is a collection of independent computers that appears to its users as a single coherent system [37].

2. Server: A server process provides services to other programs in the same or other computers.

3. Client: A client process accesses services from a server through a network. The terms *resources*, *machines*, *computer*, *peers* and *clients* are used interchangeably.

4. Client-Server System: Client-Server describes the relationship between two processes in which one process, the client, makes a service request from another process, the server.

5. Peer to Peer (P2P) Networks: Peer-to-peer is a communications model in which each party has the same capabilities and either party can initiate a communication session [30].

6. Idle CPU cycles: The duration at which the CPU is not used by a process.

2.2 Volunteer Computing

Volunteer computing (Desktop grids) is a type of distributed computing in which computer owners donate their resources (CPU cycles and storage) for one or more projects. The Great Internet Mersenne Prime Search (GIMPS) [42], started in 1996, is one of the first volunteer computing projects. It was aimed at searching for Mersenne prime numbers. Other well known volunteer computing architectures are BOINC, Xtremweb, Entropia and Condor.

In BOINC [8] there is a central server which is responsible for the control of the job execution. The desktop user who wants to take part in the voluntary computing, downloads and installs a client from the central server. It updates its availability frequently with the server. The server when scheduling the job, checks the availability of the client based on the client updates received by it and schedules the job only on the clients which are available at a given point of time.

Xtremweb [20] monitors the availability of a set of desktop computers in an effort to effectively select a resource. It uses a combination of pull model and cycle-sharing scheme for resource selection. In the pull model, unlike the push model, workstations request work from the central agent (dispatcher). The cycle-sharing scheme is characterized by sporadic user interruptions, that prevent computations from completing, even without any network or computer failure.

Entropia [14] is an enterprise desktop grid system with a layered architecture. It has a single server connected with a number of desktop clients. The Entropia server is responsible for physical node management, resource scheduling and job management.

Condor [25] is a system that has evolved over the years from 1988. The major components of Condor are agent, resource and matchmaker. The agent is responsible for submission of a job and the resource executes the job. The matchmaker receives advertising messages from the agent and resources, which is used for matching the agent to a resource.

The issues with these architectures are two fold: These systems are built on a client-server model. When the number of clients increases, the load on the server increases and the server eventually becomes the bottleneck in the system. As the clients are not dedicated, during a particular work unit execution, they can become unavailable randomly, due to user interruption or various other factors. These systems do not take into account this sporadic resource availability during resource selection.

2.3 Cloud Computing

While all this work has been going on in the academic world, industry has come up with its very own version of grid computing to provide processor cycles and data storage capability. It is called Cloud computing [39]. Cloud computing is a style of computing where IT-related capabilities are provided as a service, allowing users to access technology-enabled services from the Internet without knowledge of, expertise with, or control over the technology infrastructure that supports them. Weiss [39] describes how cloud computing takes several shapes: Data Center, Distributed Computing, Utility Grid and Software as a Service. Cloud Computing was mainly targeted at companies which cannot afford to own data centers. Amazon EC2 [1] is the first of its kind.

To accommodate the volunteer grid computing model into the cloud computing paradigm, the voluntary grid computing platform has to become mature and stable.

2.4 Resource Selection in Desktop Grids

Selection of the appropriate resources among the available resources, is a significant task of grid architecture as it determines whether a job is going to be executed in a client machine without interruption or not.

Resource selection can defined as *the problem of selecting a particular resource "x", as the candidate for job execution, among "n" matching resources.*

The selection of the resources in various architectures can be classified broadly as follows:

1. *First come first serve*: The resource which matches the resource selection query first is selected for execution. This is one of the most primitive approaches to resource selection. This model can lead to starvation of clients. The reliability of the clients is not taken into account in the resource selection process. Section 2.4.1 describes some of the systems that use first come first serve model for resource selection.

2. *Reliability ratings*: This approach quantifies the reliability of a client based on the client's past performance [12, 23]. The ratio of the number of jobs assigned to a client to the number of jobs successfully executed is calculated and is associated with each client. This determines the reliability of the client. Clients are ranked based on the reliability calculated. Reliability Ratings model is explained with examples in Section 2.4.2.

3. *Availability prediction*: Another approach to resource selection is to predict a resource's availability using mathematical models [11]. Using the availability of the client on a similar time during past, the future availability can be predicted with certain confidence [34]. Section 2.4.3 presents two systems that use availability prediction as the resource selection strategy.

2.4.1 First come first serve

This section describes the first come first serve model of resource selection used by Xtremweb [20], Entropia [14] and Condor [33]. These architectures have very unique job delivery and resource matching mechanisms. However, during resource selection, these systems assign jobs based on a first come first serve model, without taking into account the client's reliability and future availability.

Remote Sensor based Resource Selection - Xtremweb

Xtremweb [20] has a goal of monitoring the availability of a set of desktop PCs in an effort to effectively select a resource. It uses a combination of the pull model and cycle-sharing scheme for resource selection. In the pull model, unlike the push model, workstations request work from the central agent (*dispatcher*). The cycle-sharing scheme is characterized by sporadic user interruptions, that prevent the computations from completing, even without any network or computer failure.

When a participating workstation is not interactively used, as detected by the screen saver utility, the workstation downloads the work-unit (job) from the dispatcher and start executing it. As soon as the user comes back to the workstation, the screen saver vanishes and the ongoing job is terminated. The job is started again from the beginning on a new workstation.

Figure 2.1 presents the client architecture of the Xtremweb application. When a user is working (when the system is not idle) a low-priority background process monitors the computer activity. When the computer becomes available, a new process is launched. This process communicates with the dispatcher and gets the work unit from the dispatcher. Concurrently a monitor thread is spawned which watches the activity of the work unit. After sending the work unit, the dispatcher continuously monitors the workstation. When the workstation has not communicated the status of work unit execution for a sufficiently long time the computation is considered lost and rescheduled with another worker. The workstation ceases to communicate about the status of the work unit execution if it is interrupted by user activity or when there

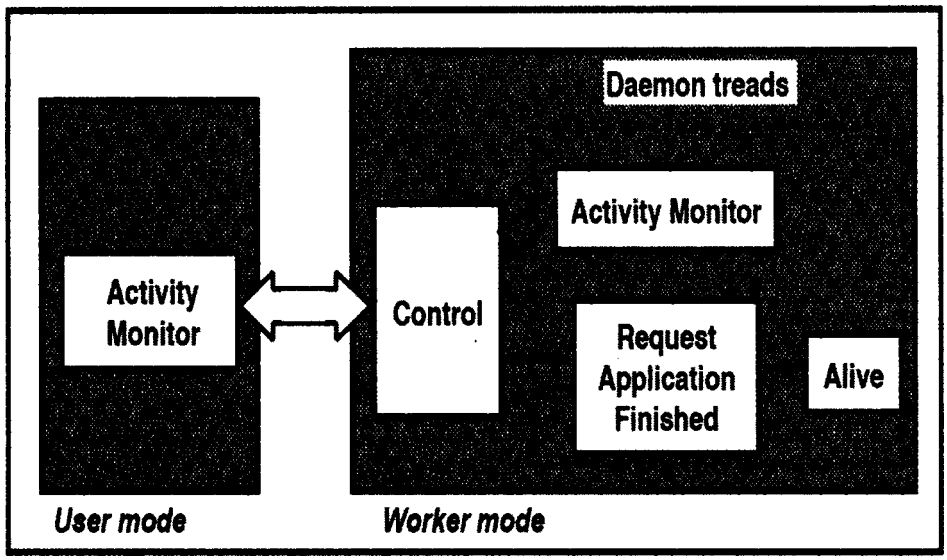

Figure 2.1: The Workstation in Xtremweb [20]

is a failure (hardware or software).

To summarize, Xtremweb web uses a pull model for its resource selection mechanism. Whenever a resource becomes idle, it contacts the dispatcher for work and any available computation that has to be performed at that point of time is dispatched to the client.

The resource selection is primitive and it does not guarantee that job execution is completed on the client machine. In this model, the client which communicates with the dispatcher *first*, will get the job, not necessarily the one which is most reliable.

Resource Selection in Entropia

The Entropia system [14] is made up of three major layers: Physical node management, resource scheduling and job management. The *Physical node management* layer is responsible for managing diverse desktop PCs and other low-level reliability issues. It provides resource management, application control and security capabilities to the architecture. *Resource scheduling* does the work of a matchmaker, where, it accepts the units of computation from the job management system and then matches these units to appropriate client resources and schedules the units for execution. The *Job management layer* of Entropia takes the responsibility of splitting a job into multiple tasks (subjob). It also manages the status of execution of each generated subjob

and then aggregates the results from the subjobs into a single output.

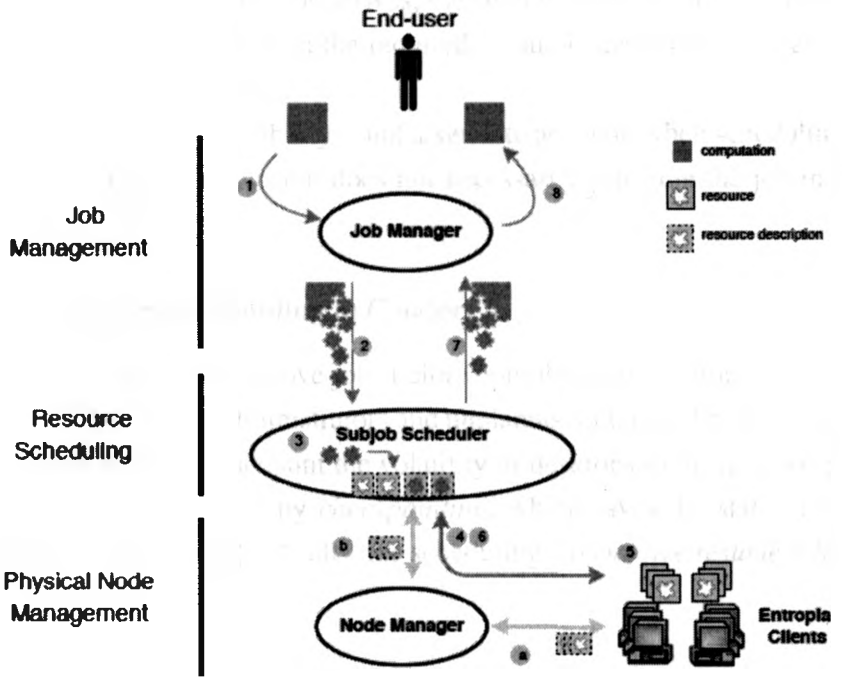

Figure 2.2: Resource selection in Entropia [14]

Figure 2.2 depicts the application execution on Entropia system. These are the steps involved in job execution in Entropia.

1. An end-user submits the job to the job manger (1 in Figure 2.2).

2. The job is broken down into subjobs before scheduling. The Job manager submits the job to the subjob scheduler by breaking up the job into subjobs (2 in Figure 2.2).

3. The subjob scheduler is updated periodically about the status of each client through the node manager, which get the client availability data from the clients (a and b in Figure 2.2). This is used by the job scheduler to match the sub jobs with the resources (3 in Figure 2.2).

4. The subjob scheduler then executes the subjob in the resources (4, 5 and 6 in Figure 2.2) and aggregates the results back (7 in Figure 2.2) and hands it back to end-user (8 in 2.2).

In the Entropia system, the subjob scheduler is responsible for resource selection. It is updated in frequent intervals about the status of idle machines along with its resource capabilities. When a job arrives, the subjob scheduler finds a match among the current active machines with the required resource capabilities and executes the job in the matching machine.

The client machine's reliability is not taken into account, when scheduling. Therefore, like Xtremweb, this system does not necessarily schedule the job in the most reliable client.

Preemptive Resume Scheduling in Condor

Preemption is the need to remove jobs before completing execution in order to meet the needs of client users, administrators and unplanned outages. The Condor scheduling system is designed to account the volatility of desktops to make preemption less expensive. This is achieved by *checkpointing*, which saves the state of the job in frequent intervals. Condor[33] calls this scheduling *Preemptive resume scheduling.*

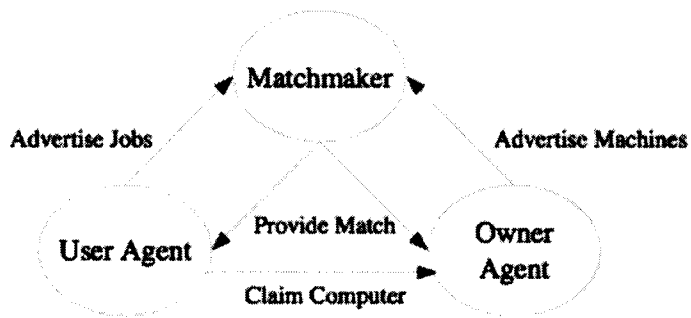

Figure 2.3: Resource selection in Condor [33]

Condor's scheduling mechanism consists of three major participants: *User Agent*, *Matchmaker* and *Owner Agent.*

The *User Agent* submits the job to the resource pool. The *user agent* maintains a persistent queue of jobs and history of past jobs. If the jobs fail, the user agent is responsible for retrying the job.

The *Owner Agent* is installed on each client of the pool. Policies determine how and when the computer has to be classified as idle. It is responsible for executing the jobs submitted to the client.

The *Match Maker* finds the match between user agent and owner agent.

Matchmaking When a user submits a job, it is stored in a user agent's persistent queue. The user agent then sends advertisements to the matchmaker every 5 minutes, about the job that need resources. At the same time, whenever a client becomes idle, its *owner agent* sends advertisements, which contain information on the client's availability, to the matchmaker at the same rate as that of user agent (5 minutes). The matchmaker makes a match between the user agent and owner agent based on the most recent advertisements. The old advertisements which are not updated for more than 5 minutes are discarded in the matchmaker.

Once the match is made, the user agent contacts the owner agent directly and sends the job for execution. The job execution is checkpointed at important intervals to resume execution just in case the execution gets interrupted. The job agent is responsible for orchestrating the execution of the job in the owner agent.

The resource selection in Condor is based on the most recent advertisements from the owner agent. Reliability of the client is not taken into account during resource selection in this architecture.

2.4.2 Reliability Ratings

This section describes two architectures that assign reliability ratings to the clients. Kondo et.al [23] study resource selection by *resource prioritization*, *resource exclusion* and *task duplication*. RIDGE system [12] proposes a generic architecture for assigning and employing reliability ratings while resource selection process.

Resource Selection for Short Lived Applications

Kondo et.al [23] focuses on the effective desktop grid execution for short lived executions, i.e., the execution which takes less than 15 minutes to complete in the client machine. The authors propose three general techniques for resource selection: Resource prioritization, resource exclusion and task duplication. Based on these techniques the performance of resource selection is studied.

In this work, the authors consider the problem of scheduling an application that consists of T independent and identical tasks in a desktop grid. The tasks are allocated to each host in the grid by a central server. The server has a queue (ready queue), which maintains the list of all the current active desktops in the grid.

The authors found (Figure 2.4) that when the number of tasks (T) is larger than the number of hosts (N), a first-come first-server (FCFS) strategy is close to being optimal. As the focus is on short-lived applications, where T is of the same order of

magnitude as N, the FCFS approach is suboptimal. In figure 2.4 results are obtained for T=100, 200, 400 and N=190, where each task executes within 15 minutes. When T=100, 90% of the tasks are executed within 39 minutes, but the remaining 10% takes another 40 minutes to execute. This takes a total of 79 minutes to execute, which is almost identical to the time needed when T=400. This plateau towards the end in case of T=100, is attributed to dependency of some tasks on certain other tasks which are executed in slower machines and failure of tasks towards the end. However, as T becomes larger when compared to N, the plateau becomes less significant, thereby justifying the use of FCFS strategy.

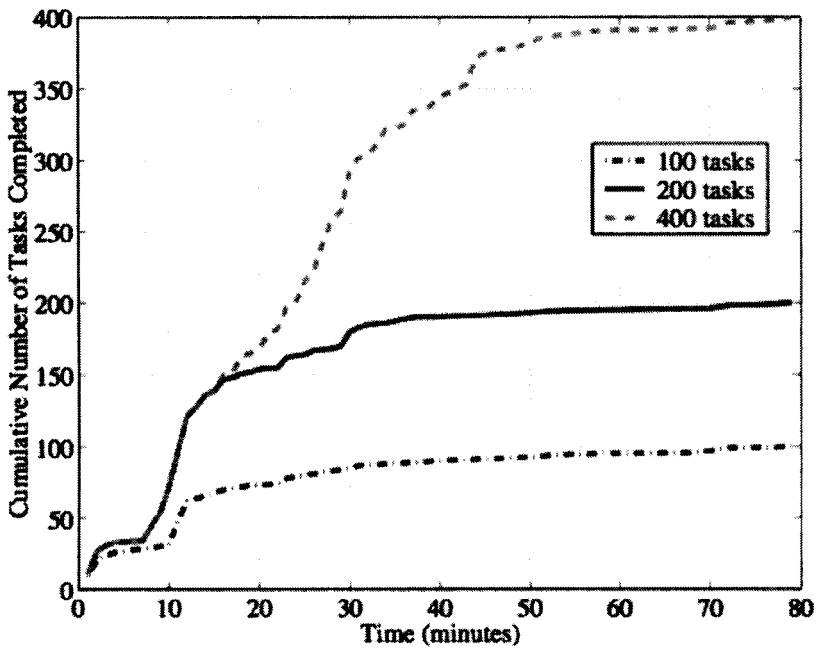

Figure 2.4: Task Completion Vs Time [23]

Therefore for short-lived applications, Kondo et.al [23] discusses three general resource selection approaches to increase the overall efficiency of the system.

Resource Prioritization This method sorts hosts in the ready queue based on some criteria (e.g., by clock rate or by the number of cycles delivered in the past) to assign tasks to the most appropriate hosts first. Three methods were used and their results were studied:

FCFS: The resources are served with jobs in a first come first serve basis.

PRI-CR: The resources are sorted based on the clock rates of CPU and assigned jobs accordingly.

PRI-CR Wait: The system waits for 10 minutes before sorting the hosts and assigning the tasks. The waiting is done for the resource information to be aggregated in the ready queue. 10 minutes gives time for more number of resource information to be aggregated in the ready queue.

PRI-History: The history of the system's past performance is used to predict its future performance. This is calculated using the previous week's usage statistics on how much operations can be performed in between host failures.

It was seen that PRI-CR outperformed FCFS consistently. PRI-CR-WAIT performed poorly with small duration tasks (5 minute). It improved with longer tasks, however, PRI-CR-WAIT never surpassed PRI-CR in its performance. PRI-History worked better for scenarios when the task duration is long. FCFS gave the least performance.

Resource Exclusion Three resource exclusion algorithms were evaluated:

Resource Exclusion using Fixed Threshold: In this method, some hosts are excluded and never used to run application tasks. These hosts are excluded based on a simple criterion, such as hosts with clock rates below some threshold or CPU availability.

Resource Exclusion using "makespan" prediction: This is a more sophisticated resource exclusion strategy as compared to the previous one. It consists of removing hosts that would not complete a task, if assigned to them, before the expected application completion time (makespan time). In other words, it may be possible to obtain an estimate of when the application could reasonably complete, and not use any host that would push the application execution beyond this estimate. The scheduler makes predictions and excludes the resource which cannot complete the task within the required makespan time. The total power of the grid relatively remains constant.

EXCL-Pred: This method makes the makespan prediction and adaptively changes the prediciton as application execution progresses.

Task Replication Task failures near the end of the application, and unpredictably slow hosts can cause major delays in application execution. This problem is remedied by means of replicating tasks on multiple hosts, either to reduce the probability of task failure or to schedule the application on a faster host.

Some methods include the following:

EXCL-PRED-DUP: Extend EXCL-Pred to use task replication when the number of hosts is greater than number of tasks.

EXCL-PRED-TO: Uses a time out for each task to measure whether replication should occur.

The results of the experiments show that EXCL-PRED-TO fares better when compared with its peers.

Kondo et.al [23] investigated methods for excluding hosts by using a fixed threshold, or an adaptive threshold based on the prediction of application makespan. The work also noted that although using a fixed threshold to exclude certain hosts is beneficial for desktop grids with a left-heavy distribution of clock rates, the adaptive makespan heuristic performs as well or better for other configurations, such as multi-cluster or homogeneous desktop grids.

Reliability Aware Scheduling in RIDGE

Krishnaveni et.al [12] introduces a concept called "Reliability Aware Scheduling" in their system RIDGE. The RIDGE system is build on the top of the BOINC[8] architecture to convert the static replication policy of BOINC into a more dynamic policy based on the reliability of the grid.

The node reliability is based on the number of timely and correct task executions performed in the past relative to the total number of tasks allocated to it.

The parameters used in scheduling are the following:

1. Target Success-Rate: A number between zero and one representing the required success rate.
2. Exec-Threshold: The maximum time that a task execution is allowed to take.
3. Scheduling-Threshold: The number of workers for the scheduler.
4. MinClients: The minimum number of workers a workunit is assigned to.
5. MaxClients: The maximum number of workers a workunit can be assigned to.

Architecture The RIDGE architecture (Figure 2.5) includes these components:

1. *Scheduler:* The scheduler is responsible for forming a group of worker nodes based on the node's reliability ratings. It also assigns a work unit to each group.

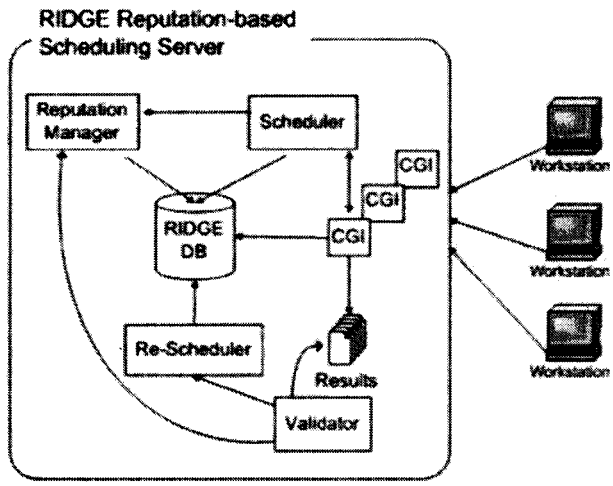

Figure 2.5: RIDGE scheduling framework [12]

Before scheduling the work unit, the scheduler waits for a specified number of workers to arrive. The specified number of workers, *SchedThrld*, is a system parameter, that can be tuned to get an optimal grouping. Once the scheduler has enough number of workers, it runs a reputation based algorithm to form the redundancy groups (groups with members of same reputation) and assigns tasks to worker nodes.

2. *Reputation Manager:* The reputation manager maintains the reliability ratings of the worker nodes. The scheduler uses these reliability ratings while making its decisions on resource selection. The reputation manger is frequently updated with the worker node's reliability after each work unit executed by the worker node is validated against the same work unit execution of the other nodes. The validation is done to check whether the worker node has correctly executed the task.

3. *Validator:* This component takes care of the validation of results which arrives from different work units for the same job. The outcome is passed to the reputation manager, which updates each work unit's reliability based on whether the worker node has correctly executed the task.

4. *Re-Scheduler:* When the validation fails, the re-scheduler decides the number of instances to be created for the failed work unit, which can be based on factors

such as the number of matching results obtained in the validation process, the reliability ratings of participating nodes, etc.

Reliability-based Scheduling The reliability ratings of a client are learned over time based on the results returned to the server. A workers reliability $r_i(t)$, at time t, is defined as follows:

$$r_i(t) = \frac{(n_i(t) + 1)}{(N_i(t) + 2)} \tag{2.1}$$

Where $n_i(t)$ is the number of valid responses generated, and $N_i(t)$ is the total number of tasks attempted by a worker i at time t.

Results were obtained in highly reliable, highly unreliable and moderately reliable environments. It showed that given a desired success rate, there is an optimal fixed replication factor, which further depends on the underlying reliability distribution.

2.4.3 Availability Prediction

This section presents two different approaches to user availability prediction: using mathematical models and probability models. John et.al [11] describes mathematical models to predict the machine availability in desktop. Ling Shang et.al [34] uses Trust Model [35] to predict the behavior of a user based on the learning done in the past.

Mathematical Prediction of Desktop Availability

John et.al [11] try to quantify and measure the volatility factors. The authors describe the parmeters to be taken into consideration for predicting machine availability in desktop grids and the ability to estimate a specified quantile for the distribution of availability, and a confidence level associated with each estimate. The work describes one parametric model fitting technique [29] and two non-parametric prediction techniques [32].

This work tries to answer the question: "*From a set of availability measurements taken from a resource, and given a desired percentile p and confidence level c, what is the largest availability duration t for which we can say with confidence c that p percent of availability time measurements are greater than or equal to?*".

The individual machine availability traces are split into training sets, (which is used to estimate the quantile lower bound) and an experimental set which is used to

verify the accuracy of the estimate. As the training set precedes the experimental set in each machine trace, the results shown in the paper detail how well each estimation method predicts machine availability for that machine during the time period covered by the experimental set.

The authors describe a mathematical model using a Weibull distribution, resample method and binomial method to predict the availability of a machine [29] [32]. They find that the availability duration can be predicted with quantifiable confidence bounds and that these bounds can be used as conservative bounds on lifetime predictions. They conclude by stating that a non-parametric method based on a binomial approach generates the most accurate estimates and can be used to estimate the lower bound quantile estimates with better efficiency.

A Trust Model based on User's Behavior

Ling Shang et.al [34] uses Dempster-Shafer's (DS) theory to predict which node is the most reliable at a given time period. Dempster-Shafer theory [18] is a method of reasoning with uncertain information. In this model, a degree of trust worthiness for a node is quantified based on two metrics.

1. How efficient that node has been? This is measured as the ratio between successful completions and total submissions of a given task

2. What is the probability that a node will be available at a give time `t`? This is calculated by studying the node's availability patterns over a period of time.

The Transferable Belief Model (TBM), an interpretation of Dempster-Shafer theory, is used to deal with the uncertainty on whether a node is active or not.

The general flow of the calculations, classification and quantification is as follows (Figure 2.6):

1. Two different types of data are collected. One on whether the task is completed correctly. The other data is a virtual timetable on the active and idle timings of the computer (Part A in Figure 2.6).

2. Based on these two values, two evidences are calculated: *Evidence based on allocated task* quantifies the reliability of the node in completing execution of the allocated task. *Evidence based on test* quantifies the availability of a client at a given time of a day in a week (Part B in Figure 2.6).

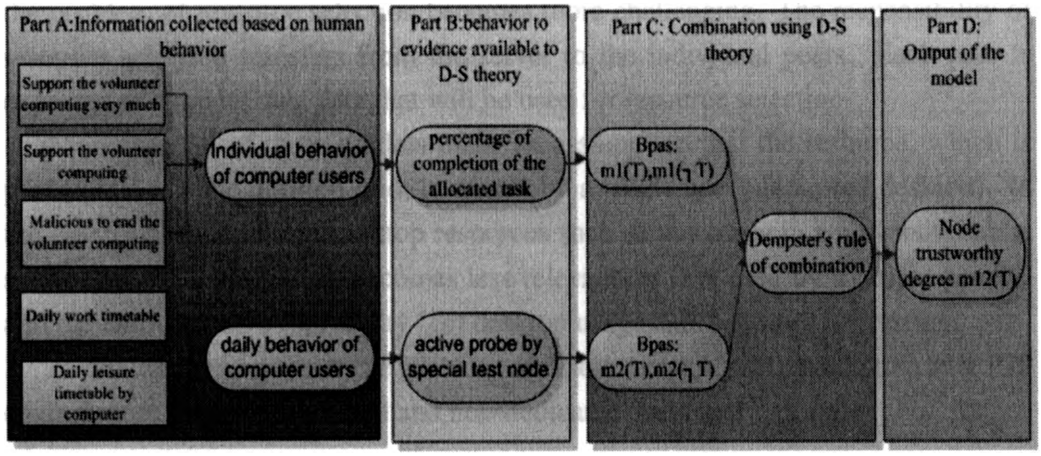

Figure 2.6: Topology of node trust model [34]

3. Using Dempsters rule of combination both these evidences are combined to form a node trustworthy value $m(t)$ (Part C and Part D in Figure 2.6)

4. Based on the calculated evidences, a four-tuple is formed: $< I, W, H, m(t) >$ where I represents the identifier of the client node, W represents the day of the week, H represents the time-interval of the day and $m(t)$ represents the degree of node trustworthy. For dedicated nodes in the grid $m(t)$ is set to be 1.

The results show that in this system that tasks have minimum or no migrations. The work emphasizes the behavioral pattern of the user which is an important but difficult to quantize aspect and has presented a method for analyzing the same. Even though, the results of this system has been very positive, it has to be noted that the experiments are conducted in an environment where users had a relatively consistent behavioral pattern.

2.5 Current Work

Resource selection has been addressed in different ways in these frameworks. Some of the frameworks use no resource selection policy, while other frameworks focus on the reliability of the clients and others select based on a prediction of availability using statistical distribution models. None of the work surveyed addresses resource availability prediction in a P2P environment. In a P2P environment, where there is no centralized server for making the decision on whether to allocate a job to a resource,

the problem of resource selection becomes more challenging. The responsibility of resource selection transfers from the server to the individual peers. Each peer is expected to store its own data that will be used for resource selection.

Moreover, all of these models make an assumption that the resource, which is participating in the desktop grid, is owned by a single user (dedicated desktop). In the case of non-dedicated desktop resources such as machines in a university lab, a single desktop's usage data becomes less relevant (as it is used by multiple people) and the entire group's (university lab) desktop usage data becomes interesting.

In this work, we look at the problem of resource availability prediction in a P2P environment for both dedicated and non-dedicated desktops.

Chapter 3

System Requirements

This chapter presents the requirements of the system. Section 3.1 and Section 3.2 describes the resource requirements and functional requirements, respectively. The challenges faced in meeting these requirements are discussed in Section 3.3. Section 3.4 presents the focus of this thesis.

3.1 Resource Requirements

This section describes the resource requirements of the software components. There are two software components involved in this system. These are the following: server and client software. The server software requirements include:

1. As in a P2P network, no resource specific data should be stored in the repository. The resource matching should be decentralized.

2. All the individual modules of this software: Job repository, Scheduling engine, Monitoring engine and the Result repository should be independent modules that can be installed in different machines if required, for scalability or security reasons.

The client software, which will be in installed in each desktop machine, should adhere to the following resource requirements.

1. As the resource discovery in a P2P network is decentralized, there will be a need for broadcasting messages in the network. The underlying P2P framework should take care of not flooding the network with broadcast messages. Broadcast messages should be restricted to the minimum.

2. The client software should not be heavyweight. When monitoring the user activity, It should run in a low priority mode without disrupting the current user experience. The software should consume a minimum amount of available memory. The CPU cycles used by the monitoring module should be negligible.

3. The client software should not monitor any other part of the system activity other than the resource monitoring such as processor usage and memory usage.

4. New jobs should be scheduled in the machines only when the resource is currently not used by the user.

3.2 Functional Requirements

This section presents the functional requirements of the system from three different perspectives. The End User Scenario of the system is explained in Section 3.2.1. Section 3.2.2 and Section 3.2.3 gives the perspective of Desktop Grid Service Client and Desktop Grid Service provider respectively.

3.2.1 Desktop User Perspective

Joe is a desktop user (Figure 3.1). He wants to take part in the desktop grid computing infrastructure. He signups with the desktop grid service provider and downloads a client to his desktop to contribute to the grid. The client learns the behavior of Joe and based on his behavior at a given time, predicts the availability of Joe's desktop. The CPU cycles contributed by Joe's desktop is recorded. These cycles can later be used for paying Joe based on the business model used.

3.2.2 Desktop Grid Service Client Perspective

ABC, an organization, is in need of a set of machines (Figure 3.2), that can take care of their CPU cycles and storage needs. ABC signs up with the desktop grid service provider and a Service Level Agreement (SLA) is formed between ABC and the desktop grid service provider on the availability of the grid and the way ABC is going to be billed for the infrastructure provided. ABC is not aware of the participating clients in the setup.

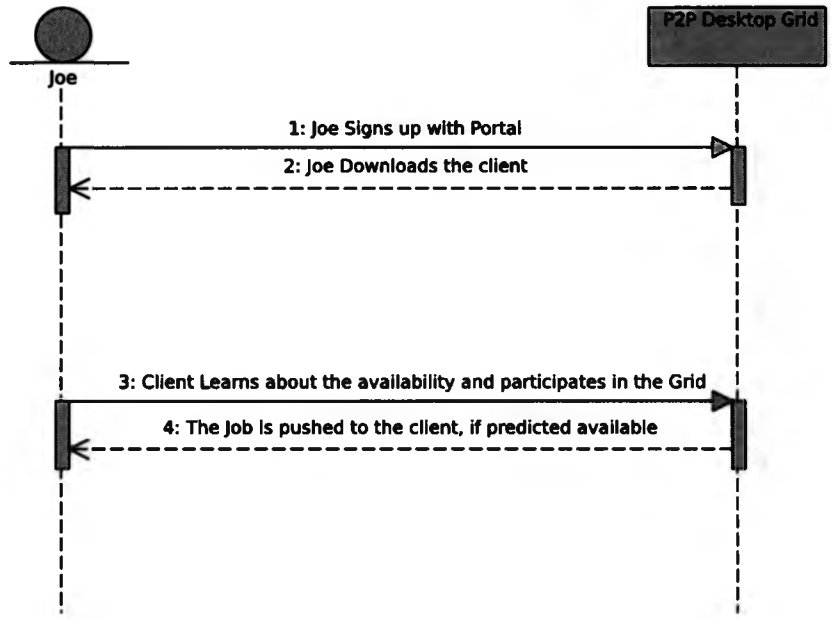

Figure 3.1: Desktop User

3.2.3 Desktop Grid Service Provider Perspective

The desktop grid service provider interacts with organizations like ABC and users like Joe to provide a business model for both the interested parties to work together. A desktop grid service provider hosts a web portal from which users can signup and download the client software needed for a peer to join in the peer to peer cloud. At the same time, organizations can signup and form a SLA with the provider. On forming the SLA the organization is provided with a security key. This security key will be used while submitting the jobs to the peer-to-peer cloud.

3.3 Challenges

Even though the technology is almost there for achieving the vision, the following are the major hurdles on its path to reality.

As the system is composed of desktop nodes which are volatile, *predictability* remains a major concern. Selecting the most reliable node out of all the available nodes makes *resource selection* challenging. The number of participating nodes, can grow exponentially over the time, making *scalability* another issue. *Maintenance* of a system of this scale remains an interesting problem. *Security* of the system cannot

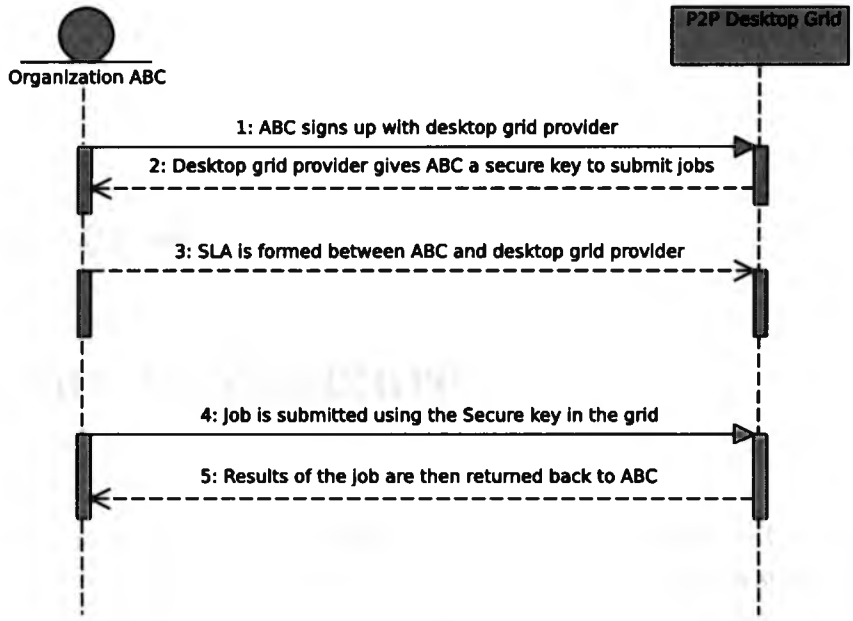

Figure 3.2: Grid Client

be ignored either. Security in this system has two dimensions to it. Security of the desktop machines which are contributing to the grid should not be compromised. The work-item running in the desktop system should not affect the state of the system. On the other hand, as the work item is running in a remote untrusted environment, making sure that the task is not corrupted by the environment, will also remain an issue to look into.

3.4 Focus

This work will concentrate on building a scalable architecture, where resource matching is decentralized, with a special focus on resource selection. Resource selection techniques such as first come first serve (random) and resource availability prediction models will be studied in the current system and an effort will be made to come up with an effective resource selection algorithm in a P2P environment.

Chapter 4

System Architecture

This chapter presents the overall VCC system architecture which is followed by a description of the individual components of the system in a top-down approach. Section 4.1 presents an overview of the complete architecture. Section 4.2 lists and briefly describes the different types of servers participating in the network. Section 4.3 describes the characteristics of individual components of P2P cloud with its characteristics.

4.1 Overview

The Voluntary Cloud Computing (VCC) system (Figure 4.1) has these logical components:

- Servers
  - Management Server
  - Job Server
  - Monitoring Server
  - Result Server
- P2P cloud
  - Voluntary Cloud Computing (VCC) Client Software
- Job (Work-Unit)

Section 4.2 describes the functionality of the servers in detail. The characteristics of P2P cloud and its components are described in Section 4.3.

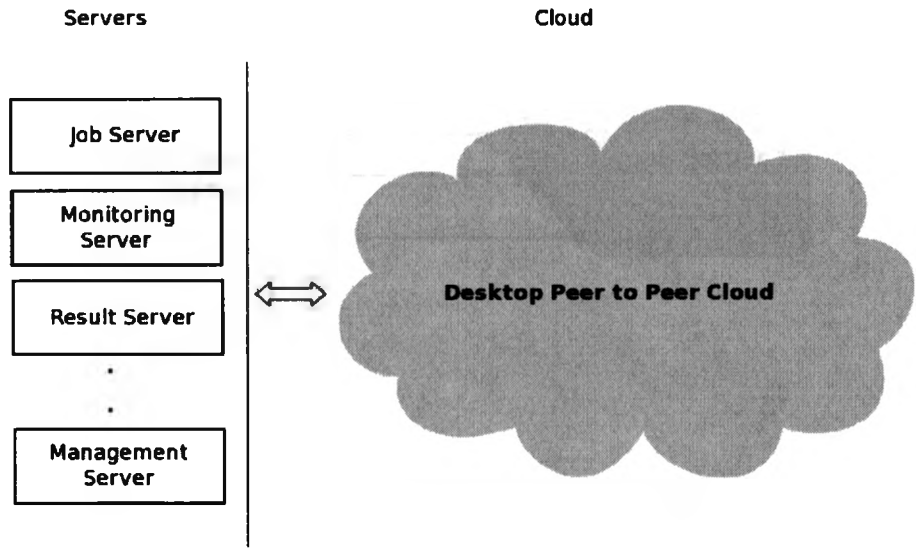

Figure 4.1: Overall view of the VCC System

The job (work unit) is in the form of a mobile agent [40]. When a resource is interrupted in the middle of job execution, the inherent migrational capabilities of the mobile agent can be used to migrate the job instance (scheduling unit of the job) to another free resource. Section 4.3.2 describes the functionality of the mobile agent in detail.

4.2 Servers

The servers are responsible for storing the definition of work unit (job server), monitoring the progress of execution (monitoring server) and storing the results of execution (result server). There can be an arbitrary number of job, monitoring and result servers. There is a centralized management server which stores the information about the job servers in the system.

4.2.1 Management Server

The management server is hosted by the desktop grid service provider. The management server (Figure 4.2) stores the information of all the participating job servers in the system. It stores the fqdn (fully qualified domain name) of job, organization name and password corresponding to each job server. The details of management

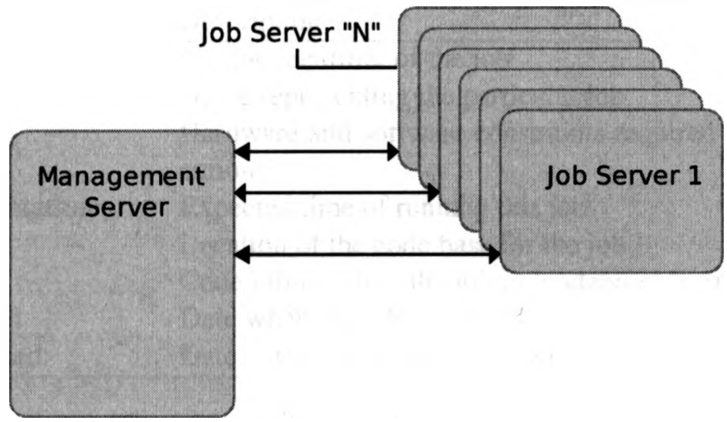

Figure 4.2: Management Server - Job Server Relationship

server are described in Section 5.2.1 (management server section in implementation chapter). Whenever an organization wants to participate in the grid, it first signs up through the management server and registers its list of participating job servers with the management server. There is one management server in the system and the relationship between management server and job servers is one-to-many and the relationship between the organization that uses the desktop resources and job servers is one-to-many. Future work will study how to allow for multiple management servers. The management server decides which job server can host jobs. Multiple management servers pose challenges with respect to trust.

4.2.2 Job Server

The job server maintains a repository of the jobs to be executed. When an organization wants to use the grid, it signs up with the management server and registers its job server(s) with the management server. An organization can have one or more job servers. A job server runs the resource selection steps (Section 5.2.2) to find a peer node and submits the job to that peer node. The information of each job stored in the job server is listed in Table 4.1.

4.2.3 Monitoring Server

An organization can have many monitoring servers. These are used to monitor the jobs submitted by the job servers owned by the organization. The relationship between job server and monitoring server is many to many. A job, when dispatched

Entity	Description
job_id	Unique identifier of the job
job_name	Name representing the particular job
constraints	Hardware and software constraints required for execution
expected_duration	Expected time of running this job
codebase	Location of the code base for the job
class_name	Code representing the job in 'codebase' location
date_created	Date when the job is created
date_modified	Date when it was last modified

Table 4.1: Job Entities

from the job server, is embedded with the monitoring server and the result server addresses. During the job's execution, the mobile agent associated with that job sends the status of execution (percentage that is completed) to the monitoring server at a frequency that is predefined in the job implementation.

The monitoring server exposes a web service to which the VCC client software sends regular updates.

```
void UpdateStatus(job_instance_id, percentage_completed)
```

`job_instance_id` is the unique identifier assigned to the instance of the scheduled job. `percentage_completed` denotes the percentage of the job that is completed. A monitoring server expects the job instance to update its percentage of completion at regular intervals.

4.2.4 Result Server

When a job is dispatched from the job server, it is provided with the address of one monitoring server and one result server. After the execution of a job, the results are sent back to the result server. The result server exposes a web service to store the result of job execution.

```
void SendResult(job_instance_id, result)
```

`job_instance_id` is the unique identifier assigned to the instance of the scheduled job. `result` indicates the application specific result data that will be stored in the database. The result of the job could be the number of prime numbers between two natural numbers or data in XML form representing the result of the test case execution. The result sending process is detailed in Section 5.4.3.

4.3 P2P Cloud

The P2P cloud is comprised of resources which act as the host for job execution. The VCC client software is installed on each of these resources. The cloud as a whole is responsible for resource discovery, machine availability prediction and work unit execution. Section 4.3.1 describes the components of VCC client software, which provides the functionality for the cloud.

4.3.1 VCC Client Software

This component is at the heart of the infrastructure. It is installed in every peer that is taking part in the cloud. The instances of VCC client software form a self-organizing network in which the resource discovery is fully distributed across the network.

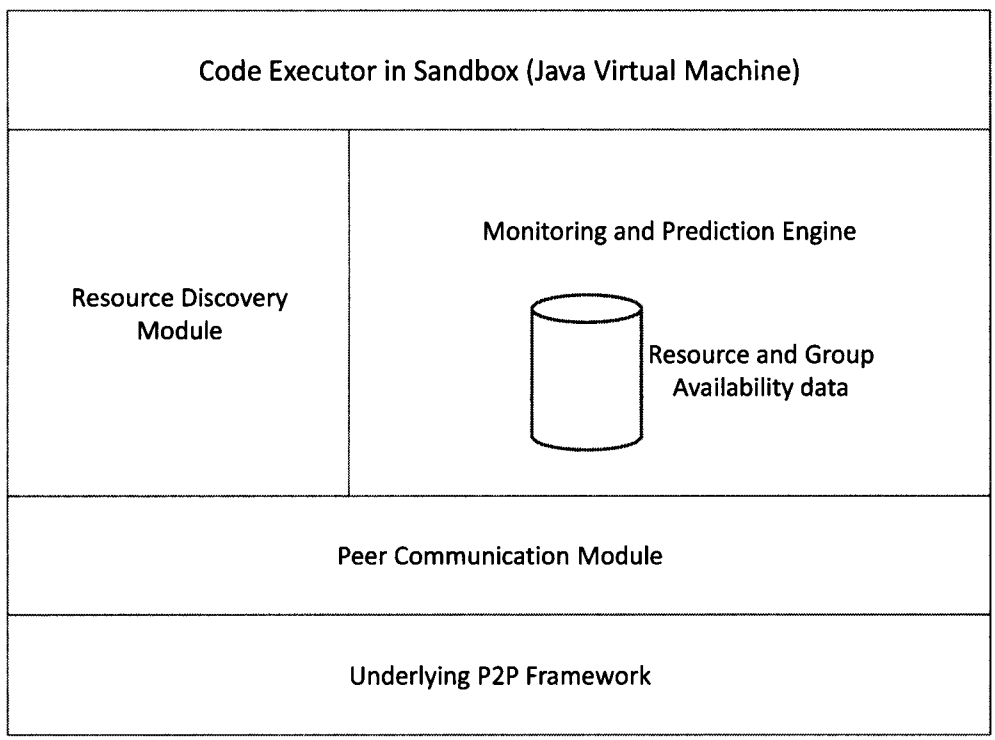

Figure 4.3: Peer Architecture

The VCC client software (Figure 4.3) consists of the following modules:

1. Communication Layer
2. Resource Discovery
3. Monitoring Engine
4. Prediction Engine
5. Code Executor

Communication Layer: The communication layer provides the API for the top layers (Resource Discovery, Code Executor, Monitoring Engine) in the VCC client software to send/receive messages with other instances of the client software (peers) and the servers. It is also responsible for the *bootstrapping* of a client. Bootstrapping is the mechanism through which a peer joins the P2P network [16]. This involves finding a member in a network, before joining the network. This layer's implementation uses JXTA framework which is described in detail in Chapter 5.

Resource Discovery: Resource discovery is done by querying the P2P network for a resource (machine) with a particular <attribute_name, attribute_value> pairs associated with it.[1] The query is broadcasted within the P2P network. Software on the matching resources respond to the peer from which the query originated. From the replies received, a specified number of resources are selected. This number is indicated by system parameter: `resourceLimit`. Each resource is sent a `AreYouIdleFor 'n' minutes` message. The list of selected peers is input to the resource selection algorithm which is executed to select the appropriate peer for job execution. The implementation of resource discovery and resource selection algorithm is described in Section 5.3.3. We will interchangeably refer to resource and the software on the resource involved in the communication.

Monitoring Engine: The monitoring engine module records the system parameters such as memory usage, CPU usage percentage and user activity to a datastore (lightweight flat-file database) within the peer, at a predefined frequency. The parameters to monitor for and the monitoring frequency is specified in an XML file. (Listing 4.1)

[1] By design we support multiple attribute pairs, however, we have implemented the design for only one pair (OS Name, Value).

Listing 4.1: Monitor Xml

```
<monitors frequency="20000"> <!-- in milliseconds -->
        <monitor name="cpu"/>
        <monitor name="memory"/>
</monitors>
```

Along with the parameters specified in the XML file (Listing 4.1), the `idle time` of the system is also recorded. The `idle time` represents the number of seconds since the last keyboard or mouse activity is sensed in the desktop. These parameters, `idle time`, CPU usage percentage and memory usage are stored in a local lightweight database and are used by the prediction engine to predict its availability.

The Monitoring Engine is also used when the node is executing a job. When the machine is interrupted by user activity, based on the migration policy of the job, it triggers an event in the Code Executor, which starts the migration process.

Prediction Engine: The data from the local datastore, populated by the monitoring engine, is used as the dataset for learning about the system's behavior. As prediction is a major focus of this thesis, the prediction engine is covered in detail in Section 4.3.3

Code Executor: The code executor module is responsible for the execution of the job in the local machine. The code executor's responsibilities as pointed out in Section 3.3 include:

1. It should provide a sandbox for the code to execute and should protect the system from malicious code.

2. The host system should not corrupt the results of the executing job either intentionally or unintentionally.

This is achieved by executing the job in a virtual machine [36]. The virtual machine can either be system virtual machine or process virtual machine [36]. We use a process virtual machine instead of a system virtual machines, as installation and running system virtual machines require a lot of memory and processing power.

4.3.2 Job (Work Unit)

A mobile agent [40] is a software module which moves from node to node autonomously and is executed at each node it moves to. Components of the work unit

are encapsulated in a mobile agent to form a job. The mobile agent is responsible for the following:

- Migration
- Communication with Monitoring server
- Reporting of results to Result server

Migration: When a node is interrupted by a user activity during job execution, the monitoring engine signals to the Code Executor that the system has been interrupted. This causes the mobile agent to start the migration process. The steps involved in the migration process are the following:

1. The mobile agent's execution is paused.
2. The mobile agent uses the resource discovery layer's API to find a free node for job execution.
3. Once a free node is identified, the mobile agent migrates itself to that node and resumes its execution from there.

Communication with Monitoring server: The mobile agent when dispatched from the job server is embedded with monitoring server and result server address. At a predefined interval, the mobile agent sends its status of execution to the monitoring server, during job execution. It consumes the web service explained in Section 4.2.3 to signal the monitoring server of its execution status.

Reporting of Results to Result server: Once the job has completed its execution, the mobile agent reports the result of the job to the result server. This is sent by using the web service described in Section 4.2.4.

Job Cleanup: The code executor subsystem takes care of collecting the job from the node's main memory. However if the job created files in the host machine during its execution, it is the job's responsibility (through its mobile agent) to delete these files on termination.

4.3.3 Prediction Engine - Dedicated and Non-dedicated Desktops

One of the goals of the volunteer computing architecture is to optimize the usage of desktop resources. These resources can be either individually owned (desktop machines at office) or used by a group of people (university lab machines). In the case of machines that are individually owned, each machine's usage pattern serves as the dataset to predict its availability in the future. However, in a setup like a university lab, where the machines are not assigned to a single person, the total lab's usage pattern becomes an important criteria for prediction along with the individual machine's usage pattern. Therefore during the prediction of such machine's availability, quantification of the group (lab) usage, becomes significant.

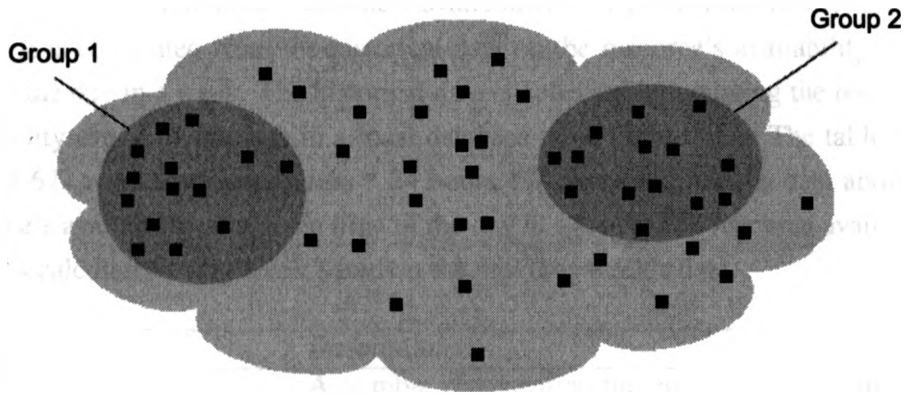

Figure 4.4: Dedicated vs Non-dedicated Desktops

The prediction engine is designed to consider the lab's behavior into account for prediction. In the rest of the thesis, a university lab is referred to as `group`. An organization can consist of multiple groups (The university can have multiple labs). Each group can have a pattern of resource availability, as can each resource in a group.

The peer-to-peer (p2p) system is used to classify the network into groups (g_1, g_2 ... g_n) where each group represents group of resources with a common usage pattern (Figure 4.4). The resource request query is sent to the network from a resource, R_{master}.[2] All the matching resources will respond to this query. The matching resources can be from different groups. The identity of the matched resources as well

[2]When there is a job to be executed, the resource request query is sent to the network from the job server. The resource request query is also also sent when a machine executing a job is interrupted and is searching for a free resource to migrate the job.

as the following parameter values are sent to R_{master}.

- Resource Availability Factor (raf)
- Group Availability Factor (ga)
- Total Number of Resources in a group (tot_{group})

The rest of the section describes these parameters in more detail.

Resource Availability Factor: The resource availability factor (raf) denotes the probability of an individual machine's availability at a given time of the day in a week. It is calculated from the historical data on the resource's availability at that time of the day in a week. The historical data is collected by updating the resource's availability every 15 minutes in a local database table (Table 4.2). The table has a total of 672 rows (4 quarter hours * 24 hours * 7 days) and has the data about the resource's availability at a given time of the day in a week. The resource availability factor is calculated every week based on the previous week's data.

Entity	Description
weekday	A number representing the current week of the day
monitor_time	Time for which the data is collected
num_of_samples	Number of samples collected till date for this time of the day in the week
num_of_available	Number of samples for which the resource was available (not busy due to user activity)
last_updated_duration	Last modified time for this row of the table

Table 4.2: Resource Availability

Resource Availability Factor (raf) at a given time of the day in a week (t) is given by the following:

$$raf(t) = \frac{N_{available}(t)}{N_{total}(t)} \tag{4.1}$$

where $N_{available}$ is number of samples at which the resource was available (not busy due to user activity) in the particular time of the day in a week and N_{total} is the total number of samples considered for that time of the day in a week (`num_of_samples` from Table 4.2).

Group Availability: The group availability quantifies the total group availability at a given time of the day in a week (t). This is represented by the following:

$$ga(t) = current_{ga}(t) - avg_b(t) \tag{4.2}$$

where $current_{ga}(t)$ (Equation 4.3) denotes the number of resources that are currently idle (not busy due to user activity) and $avg_b(t)$ (Equation 4.4) represents the average number of resources that are busy at the given time of the day in a week, t. Like resource availability factor (raf), the $avg_b(t)$ is calculated weekly. The $avg_b(t)$ of a group is collected by updating the group's availability data every 15 minutes and storing this information in a local datastore (Table 4.3). The table has a total of 672 rows (4 quarter hours * 24 hours * 7 days) and has the data about the group's availability on a specific time of the day in a week.

A resource can be busy when the user is working on the machine (user activity). The resource can also be busy because it is currently executing a job. When the state of a resource changes (from idle to busy or busy to idle) the state change is broadcasted within the group. Thus each resource in the group at a given point of time will have the data on the number of machines which are currently busy and the ones which are currently idle. This data is used to calculate the current group availability ($current_{ga}$), by every resource, during resource selection.

$$current_{ga} = tot - (N_{ua} + N_{je}) \tag{4.3}$$

where tot denotes the total number of resources in the group, N_{ua} represents the number of resources that are busy because of user activity and N_{je} denotes the number of resources that are busy due to job execution.

As discussed above, each state change and the reason for the state change of a resource within a group is recorded for every group member. A weekly user behavior table of the group is recorded in the format specified by Table 4.3.

Every 15 minutes, the current N_{ua} (number of resources that are busy due to user activity) is determined and is used to calculate avg_b. Thus avg_b is based on the average calculated in the previous time period and is the component that gives the value for previous week's availability.

$$avg_b = \frac{N_{ua} + (old\_avg_b * num\_of\_samples)}{num\_of\_samples + 1} \tag{4.4}$$

where old_avg_b represents the average value of number of busy node until the previous week.

Entity	Description
weekday	A number representing the current week of the day
monitor_time	Time for which the data is collected
num_of_samples	Number of samples collected till date for this time of the day in the week
last_updated_duration	Last modified time for this row
average_ua	Average number of resources that are busy because of user activity (avg_{ua})
total_machines	Total number of machines which have broadcasted its status

Table 4.3: Group Availability

Probability of Resource Availability: The three values explained above (raf, ga and $tot\_machines$) are sent to R_{master}. R_{master} applies the following formula to calculate the probability of the resource availability.

$$P_{resource\_availability} = (raf(t) * ga(t)) \tag{4.5}$$

where $raf(t)$ represents resource availability factor (Equation 4.1) and $ga(t)$ gives the group availability value (Equation 4.2).

In the case of dedicated desktops, $ga(t)$ is considered to be 1. There the formula in case of dedicated desktops become,

$$P_{resource\_availability} = raf(t) \tag{4.6}$$

Chapter 5

Implementation

This chapter presents the implementation details of the Voluntary Cloud Computing (VCC) system. Section 5.1 provides a brief introduction to JXTA and Aglets. Section 5.2 presents the implementation of management, job, monitoring and result server and Section 5.3 describes the implementation of the client. The lifetime of a job is discussed in Section 5.5.

5.1 Background

The VCC system is built over the peer-to-peer (P2P) framework of JXTA. It uses the decentralized resource discovery mechanisms of JXTA to find the peers. JXTA along with its components is described in Section 5.1.1. Mobile agents contribute to the migratory behavior of the jobs in the VCC system. The Aglets framework [24] is used in the VCC system to provide the mobile agent functionality. The Aglets framework is described in Section 5.1.2.

5.1.1 JXTA

JXTA was developed by Sun Microsystems to support P2P application development [21]. It is a set of open, generalized, P2P protocols that allow any networked device to communicate and collaborate [21]. These protocols are language independent and multiple implementations exist for different languages or environments such as Java, C and the .Net framework. The name *JXTA* is derived from the word juxtapose, meaning to place two entities side by side in proximity. Sun recognized that P2P solutions would always exist alongside the client/server solutions and hence the name [41].

JXTA Architecture and Components

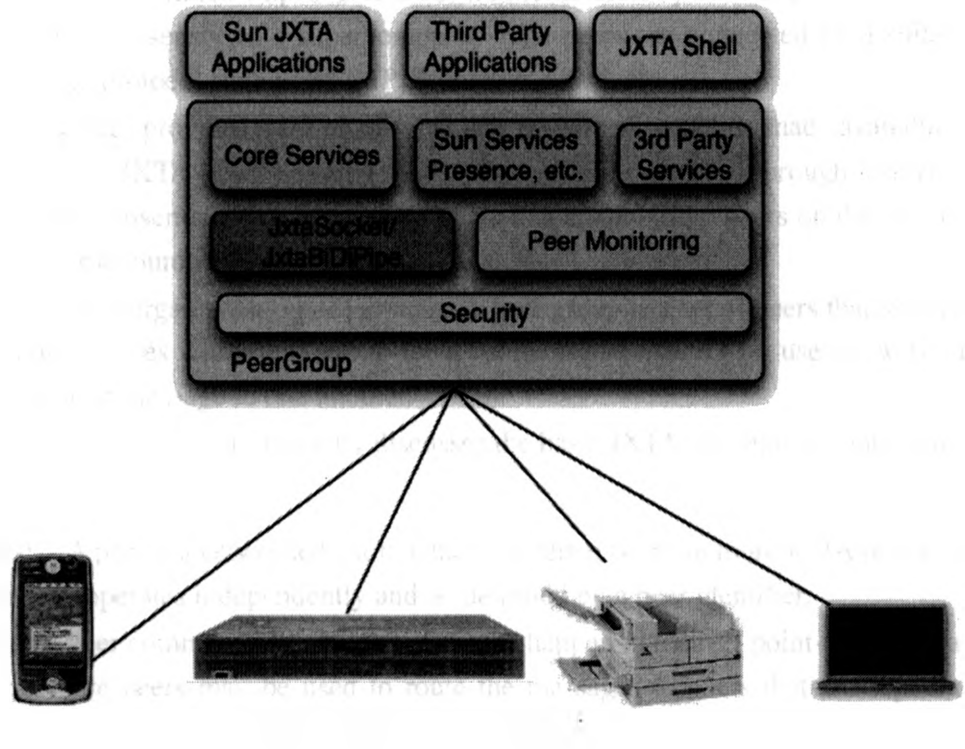

Figure 5.1: JXTA Architecture [27]

A JXTA architecture [27] is composed of three major layers:

1. **JXTA Core:** The JXTA core consists of building blocks to enable key communication for P2P applications. This includes components that enable discovery, communication, creation of peer and peer groups.

2. **Services Layer:** Network services in JXTA are the common and desirable components in a P2P network. These services provide additional functionality that may not be absolutely necessary for a P2P environment to operate. Examples of network services include searching of resources and indexing of advertisements, protocol translation, authentication and Public Key Infrastructure services.

3. **Applications Layer:** The user programs reside in the application layer. Examples of applications include P2P instant messaging, document and resource sharing, distributed auction systems and many others.

JXTA Concepts

The JXTA network is composed of set of a interconnected *peers*. A peer can be any device, from a sensor to a supercomputer. The peers are connected by a suitable networking protocol such as TCP/IP, Bluetooth, GSM, etc.

Every peer provides a set of *services* and *resources*[1] which is made available to other peers. JXTA peers advertise their services and resources through *advertisements*. Advertisements are XML documents that enable other peers on the network to discover resource and services.

Peers self-organize into *peer groups*. A peer group is a set of peers that leverage common services within that group for a common purpose. JXTA uses *sockets* and *pipes* to send message to one another.

The rest of this section briefly discusses the basic JXTA concepts in more detail.

Peers: A peer is a networked entity which implements one or more JXTA protocols. Each peer operates independently and is identified by a peer identifier.

Inter-peer communication does not always happen in a direct point-to-point way. Intermediate peers may be used to route the messages to peers that are separated due to physical network boundaries (e.g., intermediary peers can be used to route messages across the barriers such as firewall, NAT [19] and proxies).

JXTA peers are categorized as follows:

- *Minimal-edge Peer*: These peers implement only the required core JXTA services and can rely on proxy peers (described below) for additional services. Typical minimal-edge peers include sensors, home automation devices, etc.

- *Full-edge Peer*: These peers implement both the core and standard JXTA services and can participate in all of the JXTA protocols. These peers form the majority of peers on JXTA network and includes desktop machines, servers, etc.

- *Super-Peer*: These peers implement and provision resources to support the deployment and operation of a JXTA network. There are three key JXTA super peer functions.

[1] Services as described earlier provide additional functionality to P2P environment. Resources are the entities (file, CPU cycles, memory) that a peer shares in the P2P network. A JXTA peer can provide both services and resources.

- **Relay**: This function is used to route message across a private domain crossing the barrier of firewall or NAT [19]. Only peers which are unable to directly make/receive connections will require a relay.
- **Rendezvous**: A peer when configured as a Rendezvous maintains global advertisement indexes (an indexed list of the advertisements in the network) and assists edge peers (minimal and full-edge) with advertisement searches. It also handles message broadcasting.
- **Proxy**: A proxy peer is used by minimal-edge peer to get access to all network functionalities. The proxy peer translates and summarizes requests, responds to queries and provides support functionality to minimal-edge peers.

Peer Group: A peer group is a collection of peers that have agreed upon a common set of goals. Peers of similar interests form this self-organizing group. Each group is identified by a peer group identifier. A group can either allow any peer to join or it can be restricted to a peer which possesses the required credential to gain membership. A peer can belong to more than one peer group simultaneously. Peer groups are generally established to create a confined set within the global collection of peers. This confined set is formed to restrict the messages broadcasted only within that particular set. `NetPeerGroup` is the default peer group of JXTA to which every peer joins.

Network Services: Peers cooperate to publish, discover and invoke network services. There are two major network services. *Peer Services*, which operate at the peer level and *Peer Group Services*, where multiple peers collaborate and form a service. JXTA defines a set of core peer group services. These are the following:

- *Endpoint Service* - The endpoint service is used to send/receive messages in a group between peers.
- *Resolver Service* - The resolver service is responsible for sending query requests to other peers. The query request can be to find advertisements, determine the status of a service or the availability of a particular peer.

These core peer group services are implemented by every peer that joins the peer group. The standard peer group services are the following: *Discovery Service*, *Membership Service*, *Access Service*, *Pipe Service* and *Monitoring Service*.

Pipes: A pipe is an abstracted message transfer mechanism used for asynchronous, unidirectional communication. JXTA peers use pipes to send messages to one another. Pipes offer two modes of communication: *point-to-point* for one to one communication and *propagate* for one to many (broadcast) communication. *Secure Unicast Pipes* provide a secure and reliable communication channel over the pipes. The pipes are low level programming abstractions for JXTA communication. However, for reliable message exchange between the peers, *JxtaSockets* and *JxtaBiDiPipe* abstractions are used.

Advertisements: All JXTA resources such as peers, peer groups, pipes and services are represented as advertisements. Advertisements are XML documents, which the JXTA protocols use to describe and publish the existence of a peer's resource. A peer discovers resources by searching for a corresponding advertisement by first searching its local cache and then remotely. An advertisement is associated with an expiry time after which it becomes invalid. The advertisement has be to republished to extend its lifetime. The different advertisements in a JXTA system are *Peer Advertisement*, *Peer Group Advertisement*, *Pipe Advertisement*, *Rendezvous Advertisement*, *Peer Info Advertisement* etc. Listing 5.1 shows a pipe advertisement.

Listing 5.1: Pipe Advertisement

```
<?xml version="1.0"?>
<!DOCTYPE jxta:PipeAdvertisement>
<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
        <Id> urn:jxta:uuid-59616261646162614E5040... </Id>
        <Type> JxtaUnicast </Type>
        <Name> TestPipe </Name>
</jxta:PipeAdvertisement>
```

ID: Peer, peer groups, pipes and other JXTA resources are uniquely identified using ID. Uniform Resource Names (URN) [28] are used to identify a resource. URNs are a form of Uniform Resource Identifier that are intended to serve as persistent, location-independent, resource identifiers. Identifiers are used in JXTA to give its resources a common naming scheme independent of the underlying protocol.

5.1.2 Aglets

A mobile agent [40] is a software module which moves from node to node autonomously and gets executed at each node it moves to. Aglets [24] is a mobile agent

framework which enables users to create and execute mobile agents. The origin of the name is from agents and applets. IBM developed Aglets in an effort to bring in the clean design of applet programming model to the mobile agent world. Over time, Aglets has been moved to the open source community. The basic elements of Aglets and its life cycle is described in the rest of this section.

Basic Elements

The aglet model defines a set of abstractions and behavior to leverage the mobile-agent technology in a wide-area network like Internet. These abstractions [24] are the following:

Aglet: An aglet is a mobile Java object that visits aglet-enabled hosts (hosts with the aglet software installed) in a computer network. It is autonomous, since it runs in its own thread of execution after arriving at a host system and reactive because of its ability to respond to incoming messages.

Proxy: A proxy is a representative of an aglet. It is an interface which serves as a shield for the main aglet object. The purpose of this interface is to provide a mechanism to control and limit direct access to aglets.

Context: A context is an aglet's workplace. It is a container for the aglet, that provides a means for maintaining and managing running aglets in a uniform execution environment where the host system is secured against malicious aglets.

Message: A message is an object exchanged between aglets. It allows for synchronous as well as asynchronous message passing between aglets.

Identifier: An identifier is bound to each aglet. This identifier is globally unique and immutable throughout the lifetime of the aglet.

Operations of an Aglet

An aglet can be created in two ways: either it can be newly instantiated or it can be cloned from an already existing aglet. An Aglet can dispatch itself from one host to another and a host can deactivate a currently running aglet. The fundamental aglet operations (Figure 5.2) are as follows:

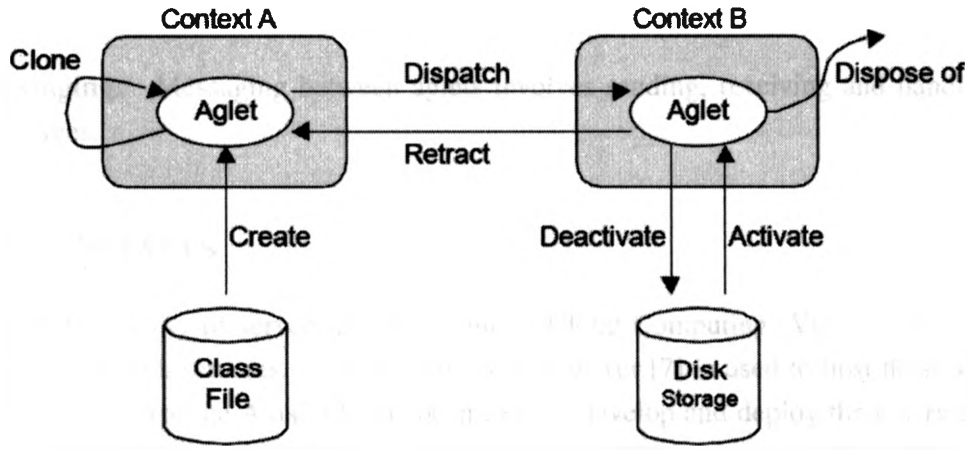

Figure 1 Aglet Life-Cycle Model

Figure 5.2: Fundamental operations of Aglet [24]

Creation: A new aglet is created in this operation. It is assigned an identifier, inserted into a context object and initialized. As soon as it is initialized the aglet starts executing.

Cloning: The cloning of an aglet results in the creation of an identical copy of the original aglet with a different identifier. Once an aglet is cloned its execution is started from the beginning.

Dispatching: Dispatching refers to the migration of an aglet from a source context to a destination context. It is referred as "pushing" an aglet to a new context. Dispatching is initiated by the context that is currently running the aglet.

Retraction: Retraction migrates an aglet from its current context to the context from which the retraction was requested. During retraction the executing aglet is "pulled" from its current context by the new context.

Activation and Deactivation: The deactivation of an aglet temporarily halts its execution and store the aglet in a secondary storage. Activation of an aglet will restore it in a context.

Disposal: The disposal of an aglet halts the current execution and removes it from its current context.

Messaging: Messaging between aglets involves sending, receiving and handling messages.

5.2 Servers

The functionality of servers in the Voluntary Cloud Computing (VCC) system are exposed as web services. Apache Tomcat web server [7] is used to host these web services and Apache Axis2 [2] has been used to develop and deploy these services. Since web services are built upon widely accepted Internet technologies and supported by most vendors, it is the natural choice for expressing the APIs of the server.

Section 5.2.1 presents the implementation details of the management server. The services hosted in the job server and its functions are discussed in Section 5.2.1. Section 5.2.3 and Section 5.2.4 describes the implementation of the monitoring server and the result server respectively.

5.2.1 Management Server

The management server as described in Section 4.2.1, stores the information of all the participating job servers in the system. Table 5.1 lists the entities stored in the management server for each job server.

Entity	Description
id	Unique identifier of the jobserver
jobservername	The fully qualified domain name (fqdn) of jobserver
organizationname	The organization to which the job server belongs to
password	The password which will be used for job scheduling

Table 5.1: Management Server Entities

The services hosted in the management server include the following:

InsertJobServer

Signature:

```
boolean insertJobServer(
```

```
   String server,
   String organization,
   String password)
```

This service takes as input the server name, organization and password, inserts the data into the management server table (Table 5.1) and returns `true/false` based on whether the operation was successful. It is consumed by the web portal which administers the management server. (The web portal was discussed in Section 3.2.3)

Authenticate

Signature:

```
boolean authenticate(
   String jobServer,
   String password)
```

This service is consumed by peers when determining if to accept a job from other peers. Every time a job is scheduled in a peer, the job server name and the password is passed on to the peer as the credentials along with the work unit. The peer before executing the job, calls the `authenticate` service to validate the credentials. This validation is done to make sure that anonymous peers cannot schedule jobs in the system.

5.2.2 Job Server

The job server maintains information on the jobs to be executed. It is the repository of the jobs that is scheduled in the VCC system. Table 5.2 lists the data stored in the job server.

InsertJob

Signature:

```
int insertJob(
   String url,
   String job_name,
   String constraints,
   String className)
```

Entity	Description
job_id	Unique identifier of the job
job_name	Name representing the particular job
constraints	Hardware and software constraints required for executing
expected_duration	Expected time of running this job
codebase	Location of the code base for the job
class_name	Code representing the job in 'codebase' location
date_created	Date when the job is created
date_modified	Date when it was last modified

Table 5.2: Job Entities

This service is used to submit the job information to the job server. The `url` parameter denotes the codebase where code can be accessed using http. The `contraints` parameter represents the hardware and software configuration under which the job is expected to run. The `className` parameter is the class containing the job in that location specified by the `url` parameter. This service returns the job identifier of the newly submitted job. The `insertjob` service is used to store the definition and the implementation details of the job.

ScheduleJob

Signature:
`int scheduleJob(int jobId)`

This service creates the instance of a job (defined as the schedule unit of the job) that is submitted by the `insertjob` service and schedules that job instance. This service is consumed to create an instance of the job that can be scheduled in a peer. When the `scheduleJob` service is called with the `jobId`, an entry representing the job is inserted in a table where each row's fields are described in Table 5.3 with the `isScheduled`'s value as `false`. The scheduling engine (described in the next section) schedules entries in this table.

Scheduling Engine

The Scheduling Engine runs as a separate process in the server. It polls the jobInstance table (Step 2 in Algorithm 1) to check for entries that have the `jobInstanceId` attribute with value not equal to -1.

Entity	Description
id	Unique identifier of job instance
job_id	Job denoted by this job instance
job_name	Name representing the particular job
isScheduled	Boolean value representing whether the job instance is scheduled yet
machineinfo	Field containing the status of the scheduled job
resultInfo	Field containing the result of the job

Table 5.3: Job Instance Entities

If any entry is found with a positive value then the job is scheduled in the available peer selected by the resource discovery module in Section 5.3.3. The logic of scheduling engine is listed in Algorithm 1.

Algorithm 1 Scheduling Engine

```
1: while true do
2:    jobInstanceId = searchJobInstancesTable()
3:    if jobInstanceId == -1 then
4:       sleep for 1 second
5:       goto step 1
6:    end if
7:    status = scheduleJobInstance(jobInstanceId)
8: end while
```

Step 7 in Algorithm 1 is described in detail in Section 5.3.3.

5.2.3 Monitoring Server

The monitoring server exposes the web services to which peers can send regular updates on the job executions status. The status of all the running jobs in the system can be retrieved by querying the monitoring server at any specific time.

UpdateStatus

Signature:

```
boolean updateStatus(
   int jobInstanceId,
   String peerName,
   int percentage_completed)
```

This service is used by peers to update the status of the job instance which is currently executing. The job decides when to send the updates to the monitoring server. The service returns `true/false` based on whether the update was successful.

GetJobInstanceStatus

Signature:

```
String getJobInstanceStatus(int jobInstanceId)
```

This service is consumed by monitoring processes to monitor the job instance execution status. The service returns the status information of the job instance execution which is represented by `jobInstanceId`. The return value comprises of the peer in which the job is executing and the percentage of completion of the job.

5.2.4 Result Server

The result server uses the same table as that of monitoring server to store its data. (Table 5.3)

UpdateResult

Signature:

```
int updateResult(
   int jobInstanceId,
   String peerName,
   String result)
```

This service is used by peers to update the result of the job instance which is executing in it. The job decides when and what to send to the result server as its result. The service returns `true/false` based on whether the update was successful.

GetJobInstanceResult

Signature:

```
String getJobInstanceResult(int jobInstanceId)
```

This service is consumed by applications that monitor the job results. The service returns the string which represents the result obtained by the server on the job instance,

represented by the `jobInstanceId`. If there are no results yet, the service returns a blank value which indicates that the job is yet to receive any results.

5.3 VCC Client Software

The VCC client software is installed in every peer that is taking part in the grid. This section describes the implementation of the VCC client software.

Section 5.3.1 presents the initialization steps of the software. The components of monitoring engine is discussed in Section 5.3.2. Section 5.3.3 describes the resource discovery module of the software. Mobile agent subsystem is presented in Section 5.3.4. Section 5.3.5 introduces the communication module of the system and Section 5.3.6 describes broadcast message listener and broadcaster modules.

Two versions of the VCC client are implemented, one for Windows and one for GNU\Linux. The VCC client has been developed using Java, therefore, there is no code change in most of the modules, except for the monitoring engine. This is described in Section 5.3.2.

5.3.1 Initialization

During initialization, the VCC client software bootstraps the JXTA client and starts all the client modules (monitoring, discovery and agent subsystem) in separate threads. Algorithm 2 lists all the steps in the initialization process.

Algorithm 2 Initialization of a peer using VCC Client

1: Bootstrap the JXTA client and join the network
2: Join the JXTA peergroup 'dgpeer'
3: Form a custom advertisement containing the attributes of the newly joined peer and publish it in the peergroup, 'dgpeer'
4: **if** peer is non-dedicated **then**
5: Join its corresponding subgroup
6: **end if**
7: Start the monitoring engine in a separate thread
8: Initialize the discovery module
9: Start the JXTA message listener in peergroup, 'dgpeer', in a separate thread
10: Initialize Mobile Agent subsystem
11: **if** peer is non-dedicated **then**
12: Start the status broadcaster thread on the subgroup
13: Start the status broadcast message receiver thread
14: **end if**

Bootstrapping is the technique used when a client joins the network. This involves finding a current member in the network. The network bootstrapping of the client is done using JXTA bootstrapping techniques (Step 1 in Algorithm 2). On initialization the peer joins the default `NetPeerGroup` of JXTA. `dgpeer` is the custom group created, in which all the peers are expected to join, so that it can participate in the cloud.

The peer is seeded (given as the input) with the IP of a rendezvous peer which caches the advertisement of the group, `dgpeer`. This helps the peer to join `dgpeer` (Step 2 in Algorithm 2). The IP of the rendezvous peer is given as the input when the JXTA client initializes in the client[2]. The rendezvous peer also caches advertisements of other peers in the network.

After joining the network, the peer broadcasts its information to the JXTA peer group. This is done by constructing a Custom JXTA Advertisement (Listing 5.2) with its attributes (name, OSName, IP) and then publishing it to the `dgpeer` group (Step 3 in Algorithm 2).

[2]This is currently a fixed IP address of a peer that runs the JXTA client as a rendezvous peer. In the future the peer can be configured to get the rendezvous peer IP address from a web page that stores the list of rendezvous peers of the system or through other bootstrapping approaches discussed in Knoll et.al.[22]

Listing 5.2: Listing of CustomAdvertisement

```
<jxta:DGGridAdvertisement xml:space="default"
xmlns:jxta="http://jxta.org">
<ID>
        urn:jxta:jxta-Null
</ID>
<name>
        CustomAdvertisement
</name>
<OSName>
        Mac OS X
</OSName>
<OSVer>
        10.4.11
</OSVer>
<osarch>
        i386
</osarch>
<ip>
        129.100.179.233
</ip>
<hwarch>
        i386
</hwarch>
<hwvendor>
        "Apple\ Computer,\ Inc."
</hwvendor>
<sw/>
</jxta:DGGridAdvertisement>
```

After the `customadvertisment` is published, if the client is a non-dedicated desktop, it joins the JXTA peergroup (custom group) formed for the group of machines to which the resource belongs to. The name of the custom group is given during the installation of the client in the peer (Step 4-6 in Algorithm 2).

The monitoring engine is started in a separate thread. The implementation details of monitoring engine is discussed in Section 5.3.2 (Step 7 in Algorithm 2). The discovery interface is also initialized and it is described in Section 5.3.3 (Step 8 in Algorithm 2). For inter-peer communication, JXTA message listener is initialized and is presented in Section 5.3.5 (Step 9 in Algorithm 2). The mobile agent subsystem is also initialized and is discussed in Section 5.3.4 (Step 10 of in Algorithm 2).

Finally, if the resource is a non-dedicated desktop, the custom group broadcast listener and message broadcaster objects are initialized for the use of prediction en-

gine. This is described in detail in Section 5.3.6 (Step 11-14 in Algorithm 2).

5.3.2 Monitoring Engine

The monitoring engine is responsible for storing the system parameters (idle, CPU usage, memory usage) in a local database, which is later used by prediction engine to determine the availability of the system at a particular time. The monitoring Engine is developed in C++ and Java. Communication is through JNI (Java Native Interface). The monitorhelper module of the monitoring engine that query the system parameters is developed in C++ as a shared library. This module is platform specific and two versions of the monitorhelper module are developed, one for windows and one for GNU\Linux.

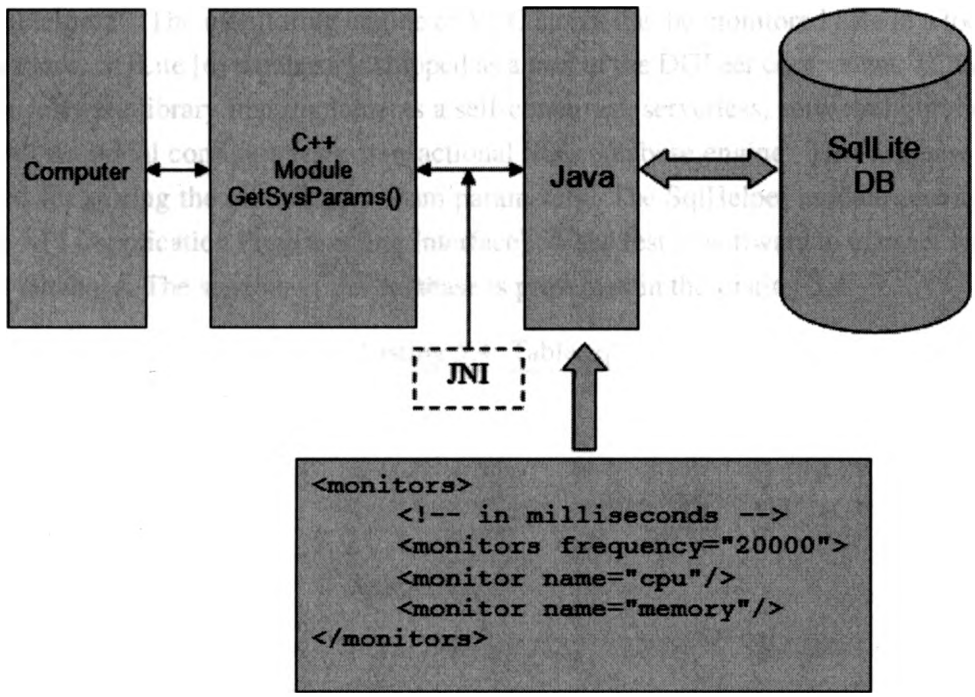

Figure 5.3: Peer Architecture

The modules of the monitoring engine (Figure 5.3) are the following:

1. XmlParser
2. SqlHelper
3. MonitorHelper

XmlParser: The monitoring engine assumes that the input is in the form of an Xml file, which contains all the parameters that is to be monitored and recorded in the local database. (Listing 5.3)

Listing 5.3: Monitor Xml

```
<monitors frequency="20000"> <!-- in milliseconds -->
        <monitor name="cpu"/>
        <monitor name="memory"/>
</monitors>
```

XmlParser processes the input xml file and then it provides the parameters to MonitorHelper class for monitoring. The Xml parsing is achieved by using XPath [15] functions in Java.

SqlHelper: The monitoring engine of VCC stores the the monitored data in a local database. SQLite [6] database is shipped as a part of the DGPeer component. SQLite is a software library that implements a self-contained, serverless, zero-configuration (with no initial configuration), transactional SQL database engine. This database is used for storing the monitored system parameters. The SqlHelper module provides the API (Application Programming Interface) for the rest of software to transact with the database. The schema of the database is presented in the Listing 5.4.

Listing 5.4: Table.sql

```
CREATE TABLE task(
        timestamp DATE,
        cpu BIGINT,
        mem BIGINT,
        idle INT
);


CREATE TABLE task_avg(
        weekday INT,
        monitor_time DATE,
        num_of_samples BIGINT,
        lastupdated DATE,
        cpu_average BIGINT,
        mem_average BIGINT,
        idle_average BIGINT
);

CREATE TRIGGER insert_task_timestamp AFTER INSERT ON task
```

```
 BEGIN
 UPDATE task SET timeStamp = DATETIME('NOW', 'localtime')
 WHERE rowid = new.rowid;

/* Increment the number of samples for the given time
               of the day in a week */
 Update task_avg
       set num_of_samples = num_of_samples + 1
       where weekday = strftime("%w", 'now')
       and monitor_time =
       strftime("%H:%M:00", 'now', 'localtime');

/* Calculate the average parameters and update the
       task_avg table for the given time
       of the day in a week */
 Update task_avg
       set lastupdated = DATETIME('NOW', 'localtime'),
       cpu_average =
               ((cpu_average * (num_of_samples-1)) + new.cpu)
               / num_of_samples,
       mem_average =
               ((mem_average * (num_of_samples-1)) + new.mem)
               / num_of_samples,
       idle_average =
               ((idle_average * (num_of_samples-1)) + new.idle)
               / num_of_samples
       where weekday = strftime("%w", 'now')
       and monitor_time
       = strftime("%H:%M:00", 'now', 'localtime');
END;
```

The `task` table and `task_avg` table are the tables that are involved in recording the monitoring information. The system parameters are inserted into the `task` table in regular intervals (`monitor_frequency` from XmlParser). The `task_avg` table has a fixed number of rows (`monitor_frequency` in minutes * 24 hours * 7 days) that stores the average system usage parameters and the average system availability data for a given time in a day of the week.

Every time an insertion is made into the `task` table, the trigger `insert_task_timestamp` is executed which calculates the average system usage and the average number of samples in which the system was idle for the given time of the day of the week. The calculated values are then updated in `task_avg`

table[3] by the trigger for the given time of the date in a week. In the Listing 5.4 the queries "`Update task_avg`" calculates the average system usage and the number of samples and updates the `task_avg` table. The `weekday` field and `monitor_time` field of `task_avg` table is used to determine which row is updated in `task_avg` table, when the rows are inserted in `task` table. The `task` table records the system parameters at a particular point of time, where as the `task_avg` table stores the average system usage (`cpu_average` and `mem_average`) and average system availability (`idle_average`) at a given point of time in the week.

MonitorHelper: The MonitorHelper component is responsible for monitoring the system parameters. As Java byte code runs in its own virtual machine (JVM), which imposes security constraints on the executing code, it is not possible to query the system parameters, such as CPU and memory usage from Java directly. Therefore, a native code module is developed for Windows and another module is developed for GNU\Linux to query the system parameters. In Windows, the Win32 API is used along with VC++ to implement this module and is compiled into a dynamic-link library (dll) file. In GNU\Linux, `glibtop` library is used with GCC for implementation and the code is compiled into a shared object (so) file. This native code is interfaced with Java using JNI (Java Native Interface). Based on the operating system in which the client is running the module, `MonitorHelper` will load the corresponding native library to gather the system parameters.

The native code interface is also made available for the other modules. This interface is queried during job execution, by the agent subsystem (Section 5.3.4), to check whether there is any user activity.

5.3.3 Resource Discovery

Resource discovery in VCC is achieved by using JXTA's advertisement discovery functions. The algorithm for resource discovery is listed in Algorithm 3. Whenever there is a new job that has to be scheduled in the grid, the job server, reads the constraints corresponding to the job and then queries the group `dgpeer` for the custom advertisements (Listing 5.2). This is achieved using JXTA's advertisement search functions `getRemoteAdvertisements` and `getLocalAdvertisements` (Step 1 and Step 3 in Algorithm 3). From a JXTA peer, a remote advertisement query is sent to discover the advertisements. After

[3]The `task_avg` table is initialized by filling it with zero values for all the time slots in the whole week, to make the update query possible.

sending the advertisement query the peer waits (Step 2 in Algorithm 3) for the local cache to be filled with the advertisements sent in reply to this query. Then the local cache is queried (Step 3 in Algorithm 3) to get the latest advertisements in the `local_advertisement_collection` object. The advertisements returned can be of different types (`SocketAdvertisement`, `PipeAdvertisement`, `ServiceAdvertisement`, `GroupAdvertisement`, `CustomAdvertisement` etc). Out of these advertisements, advertisements of type `CustomAdvertisement` are filtered out (Step 5 in Algorithm 3) and checked whether they have the same <attribute_name, attribute_value> pair as that of the constraints (Step 6 in Algorithm 3). If the constraints are satisfied the peer corresponding to the `CustomAdvertisement` is considered for job execution (Step 7 in Algorithm 3).

After the custom advertisements are filtered based on the required constraints (Step 7 in Algorithm 3), each peer is queried on whether it is available for job execution. This is done by getting the prediction parameters (Section 5.3.7) from each peer which has been identified by the custom advertisement (Step 15 in Algorithm 3). The probability of availability of each peer is calculated using the Equation 4.5 in Section 4.3.3(Prediction) and is stored in a peer array (Step 16 in Algorithm 3).

Once the peer array is formed, a reservation request is sent to each peer in the decreasing order of the peer's probability value (Step 23 in Algorithm 3). If the reservation is successful then the current peer is chosen as the discovered peer and is selected for job execution (Step 24 in Algorithm 3).

5.3.4 Mobile Agent Subsystem

The aglet mobile agent subsystem provides the necessary functionality for a job to execute and migrate. It also exposes an Application Programming Interface (API) for other modules in the system to control the behavior of the agent system.

The mobile agent is initialized by setting `DGPeerContextListener` as the context listener for the subsystem. The context listener in aglets, contains the event listeners `agletActivated`, `agletArrived`, `agletCloned`, `agletCreated` and `agletDeactivated` as methods, which are called during various stages in the lifecycle of Aglets (For example: `agletArrived` is called before the aglet is executed in the subsystem).

Listing 5.5 presents the data structure in Mobile Agent system that is used to store the details corresponding to the currently executing aglet. The peer dispatching the aglet is also expected to send the values for the data structure (Listing 5.5).

Algorithm 3 Resource Discovery in VCC Client - getPeer function

```
1: searchRemoteAdvertisementsFromDGPeerGroup();
2: Sleep for 5 seconds {Give time for jxta to query the advertisements and fill its
   local cache}
3: local_advertisement_collection = discoveredAdvertisements();
4: for every local_advertisement in local_advertisement_collection do
5:    if typeof(local_advertisement) == CustomAdvertisement then
6:       if satisfyConstraints(constraints, local_advertisement) then
7:          Insert it in the found_advertisements object
8:       end if
9:    end if
10: end for
11: peer_array = array();
12: for every advertisement in found_advertisements do
13:    connection = connectTo(advertisement.peerId)
14:    if connection was successful then
15:       Get the prediction parameters from the peer
16:       Calculate the probability value of the peer availability
17:       Insert the peer in peer_array in decreasing order of the probability value
18:       if peer_array.size > 20 then
19:          exit the for loop
20:       end if
21:    end if
22: end for
23: for every peer in peer_array do
24:    reservation = reserve(peer);
25:    if reservation is successful then
26:       return the peer
27:    end if
28: end for
```

Listing 5.5: Aglet Data

```
public String jobServer = null;
public String resultServer = null;
public String monitoringServer = null;
public String privateKey = null;
```

Once the aglet has arrived, it is authenticated to check whether the job server sending the aglet has the permission to schedule in the cloud. This is achieved by calling the management server web service, `Authenticate` (Section 5.2.1) with `jobServer` and `privateKey` as the parameters to authenticate the peer sending the job. The job's execution is started only when the authentication is successful. When the job execution is initiated, `IdleMonitor` is started, in a separate thread to the job execution. `IdleMonitor` polls the system to check whether there is any recent keyboard or mouse activity. This is done by calling `MonitorHelper.getSystemParameter` function which calls a native library through Java Native Interface (Section 5.3.2). When the system's state changes from idle to busy due to some user activity, the mobile agent is signaled to dispatch the job. The `dispatch` function tries to discover another resource for job execution by calling `getPeer` (Algorithm 3). When a candidate peer is identified, the job is dispatched to that peer by calling `dispatch` API of aglet. (Figure 5.4)

5.3.5 Inter-Peer Communication

During initialization, the peer starts to listen for any incoming JXTA socket connection requests. The messages are sent between the peers using the `JXTASockets`. Once a connection request is received, the task for that request is given to a worker thread, which services that message. The `ThreadPool` object in Java maintains a set of worker threads to support multiple connection requests.

There are two classes which are responsible for the inter-peer communication, `DGPeerListener` and `DGPeerSocketClient`.

`DGPeerListener` listens for any incoming messages on the main group, `dgpeer`. It is initialized during the startup stages of VCC software. This class implements `JXTAServerSocket` and uses a `threadpool` of fixed worker threads for handling the messages. The incoming messages are passed on to class `MessageHandler` which is executed by one of the worker threads of `threadpool`.

`DGPeerSocketClient` implements an interface for the other modules to send data to the peers participating in the grid. It defines

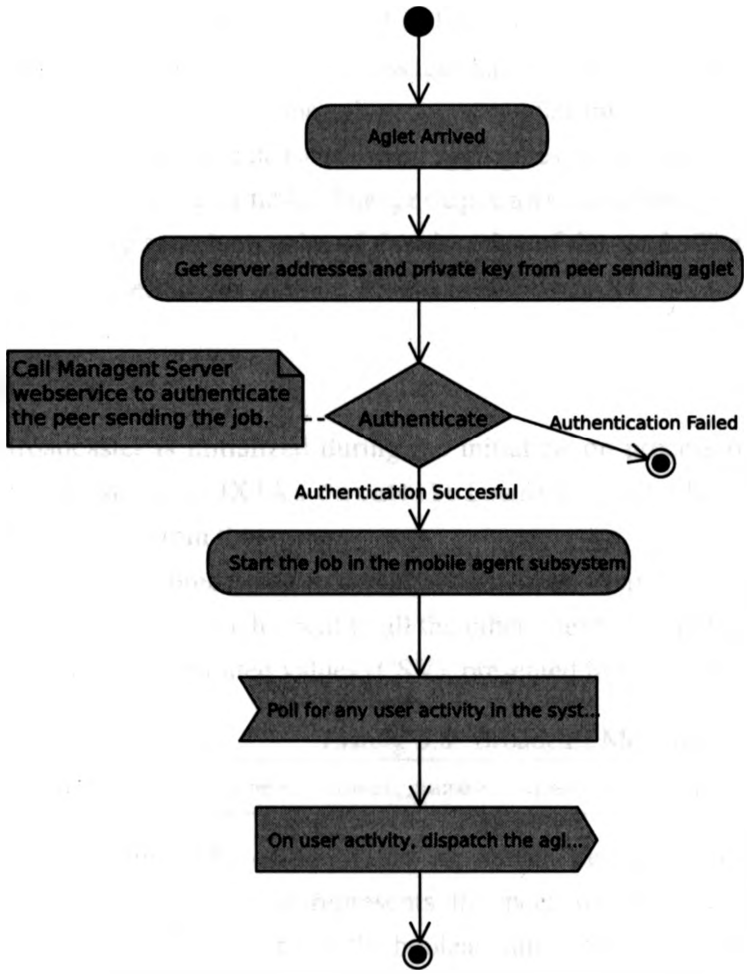

Figure 5.4: Mobile Agent

`sendMessage(message, peerID, peerGroup, waitForReply)` to send data to the peer identified by the `peerID`.

5.3.6 Broadcast Message Listener and Broadcaster

The broadcaster and broadcast message listener are enabled for non-dedicated desktop systems to share the availability status of a peer within the sub-peer group. Sub-peer group is the grouping made to exchange usage information within peers of similar usage pattern (Section 4.3.3). This grouping is specified by the user while installing the VCC client software. If the peer is a dedicated desktop, both the broadcast listener and broadcaster are disabled.

The broadcast message listener and broadcaster modules enable the calculation

of Group Availability presented in Equation 4.2 as described in Section 4.3.3. The broadcaster sends the status messages having the availability data from a peer to every other peer in the same sub-group at regular intervals (`broadcast_interval` minutes). The broadcast receiver aggregates these status messages and populates `group_task_avg` table. The `group_task_avg` table is queried to get the group availability at a given point of time in a day of the week. The implementation details are discussed in this section.

Broadcaster

Broadcaster is initialized during the initialization process of VCC client software. Broadcaster uses JXTA's `JxtaMulticastSocket` class to send messages to all the clients within that group.

Once in a predefined interval (`broadcast_interval` minutes) the following data is sent from each client to all the other clients within that subgroup. The format is of comma separated values (CSV), presented in Listing 5.6.

Listing 5.6: Broadcast Message

```
STATUS, <date_time>, <peer_name>, <peer_availability>
```

In Listing 5.6, `date_time` represents the time stamp associated with the message, `peer_name` represents the peer which is sending the message and `peer_availability` is the boolean value indicating whether the system is busy or not (true if idle, false if not idle). `MonitorHelper` class (Section 5.3.2) is queried to get the `peer_availability` value.

Algorithm 4 gives the steps involved in Broadcaster. The current system date and time is stored in `date_time` variable (Step 2 in Algorithm 4). The `peer_availability` is set to be false if the peer had some user activity since the last time it was checked (Step 3 in Algorithm 4). Using the variable `date_time` and `peer_availability` the status message is formed and is broadcasted to all the clients in the group (Step 5 in Algorithm 4). The broadcaster creates the status message (Listing 5.6) and sends it to all the peers in the group once in `broadcast_interval` seconds. Each peer receives this information using the broadcast message receiver class described in the next section.

Along with the status of the peer, the state change in peer due to job execution is also broadcasted using this module. Every time the system changes its state from busy to idle (or vice versa) because of job execution, the state change information is also broadcasted in the sub-group.

Algorithm 4 Algorithm of Broadcaster

1: **while** true **do**
2: date_time = getcurrentdatetime();
3: peer_availability = false, if peer had some user activity in the last 'broadcast_interval' minutes
4: status_message = (status, <date_time>, <peer_name>, <peer_availability>)
5: broadcast status_message to all the peers in peer group (all machines in a lab)
6: Sleep for broadcast_interval minutes
7: **end while**

Broadcast Message Receiver

The message broadcasted from the `Broadcaster` is handled by the `BroadcastReceiver` class. It is initialized during the bootstrapping process. `BroadcastReceiver` listens for new messages using `JxtaMulticastSocket`. `BroadCastReceiver` uses the `ThreadPool` object in Java to handle the incoming messages in separate threads. Ever new message received is serviced by `BroadcastMessageHandler`. `BroadcastMessageHandler` pushes the data into `GroupAggregator` class which is responsible for processing the incoming group status messages and saving it into a local database.

Algorithm 5 Algorithm of GroupAggregator

1: Input parameters: date_time, peer_name, peer_availability
2: **if** peerInfo == null **then**
3: peerInfo = new Hashtable();
4: Schedule Algorithm 6 to execute after 'n' minutes {Wait for the 'n' minutes for the messages from all the peers to arrive before processing peerInfo}
5: **end if**
6: peerInfo[peer_name] = peer_availability

Algorithm 5 presents the logic of `GroupAggregator`. The `GroupAggregator` aggregates all the incoming status messages (which is of the format presented in Listing 5.6) into `peerInfo`, a hashtable data structure with `peer_name` as the key and `peer_availability` as the value (Step 3 in Algorithm 5).

Algorithm 6 Algorithm of ProcessGroupData

```
1: Input parameters: peerInfo, date_time
2: new_count = 0, new_nua = 0
3: for every peer in peerInfo do
4:     peer_availability = peerInfo[peer_name]
5:     if peer_availability is false then
6:         new_nua = new_nua + 1
7:     end if
8:     new_count = new_count + 1
9: end for
10: Compute the new average of average_nua, total_machines with new_nua and
    new_count and update the record
11: peerInfo = null
```

The function `ProcessGroupData` (Algorithm 6) is scheduled to be executed in 'n' minutes[4] from the time at which the first message has arrived (Step 4 in Algorithm 5). This delay is introduced to wait for the messages from all the peers to arrive before processing `peerInfo`. After 'n' minutes the data aggregated in `peerInfo` data structure is processed to get the count of number of machines that are busy due to user activity (Step 10 in Algorithm 6) and then the Table `grouptaskavg` (Listing 5.7) is updated accordingly. In the Listing 5.7, weekday denotes the day of week for which the `average_nua` is stored, `monitor_time` represents the time of the day for which the data is recorded, `num_of_samples` gives the total number of samples considered for the calculation of `average_nua`, `lastupdated` denotes the last updated time for the row, `average_nua` gives the average number of machines that are busy due to user activity and finally `total_machines` gives the total number of machines that are considered for the `average_nua` calculation. Algorithm 6 lists the steps involved in `ProcessGroupData` and Listing 5.7 gives the schema of the table in which the group data is stored.

Listing 5.7: Schema of grouptaskavg table

```
CREATE TABLE group_task_avg(
        weekday INT,
        monitor_time DATE,
        num_of_samples BIGINT,
        lastupdated DATE,
        average_nua DOUBLE,
        total_machines INT
);
```

Along with this data, the state change messages broadcasted due to job execution is also handled by broadcast receiver, which is used to calculate the value of number of machines in the group that are currently busy due to job execution (which is denoted by `n_je`). When the state of the machine changes more than once, the latest status message is considered as the current state of the machine.

5.3.7 Prediction Parameters

The prediction parameters are requested by the job server or the resource which is trying to migrate the job to a peer during resource discovery (Section 5.3.3). This section describes how a peer calculates the prediction parameters.

[4] An aggregation delay, n minutes is introduced to account for unforeseen delays in receiving the broadcast messages. n minutes is always lesser than the `broadcast_interval` of Algorithm 4.

Resource Availability

The resource availability value $raf(t)$, given by the Equation 4.1 is calculated directly by querying the `task_avg` table (Listing 5.4) for `idle_average` value for the given time in the current day of the week.

Group Availability

As presented in Equation 4.2, $ga(t)$ is given by,

$$ga(t) = current_{ga}(t) - avg_b(t)$$

$$current_{ga} = tot - (N_{ua} + N_{je})$$

The $current_{ga}$ is calculated based on the information received from the current status message of broadcast message receiver module. The total number of status messages received gives the value of *tot*. The number of machines busy because of user activity, N_{ua} is calculated as `new_nua` variable in Algorithm 6 and the N_{je} is calculated from the state change broadcast messages stored in the variable `n_je` (Broadcast Message Receiver in Section 5.3.6).

The value of $avg_b(t)$ is assigned by querying the table `group_task_avg` (Listing 5.7) for `average_nua` value at the current time of the day in the week. The `weekday` and `monitor_time` fields of `group_task_average` provide the fields for querying the table against the current time of the day in the week.

5.4 Anatomy of a Job

This section investigates the components of a job. Listing 5.8 gives the source code of an example job (a prime number generator). This job returns the number of prime numbers between 0 and 1000000. Section 5.4.1 presents the details of Aglet superclass which has to be extended in a job. IAgentCommunicator interface is described in Section 5.4.2. Section 5.4.3 discusses how the state of the job is maintained during migration.

5.4.1 Aglet Superclass

Every job is expected to extend the abstract class `Aglet`. On extending the class `Aglet`, the job can override, `onCreation`, `onDisposing` and `run`.

`onCreation` is called every time the job is created. `onDisposing` is called when the object is disposed and `run` is the entry point for the aglet's own thread of execution. `run` is invoked upon a successful creation, dispatch of the aglet. The job can also implement policies of migration using `addMobilityListener` functionality provided by Aglet subsystem.

5.4.2 IAgentcommunicator

`IAgentCommunicator` is the interface of VCC client, which is used by the job to contact the monitoring and result server. The job, during its execution, sends status updates on progress towards job completion as measured by the percentage that is completed, to the monitoring server. After execution, the job sends the results to the result server. `IAgentCommunicator` provides these functionality through the following functions:

- sendStatus(int percentageCompleted)
- sendResult(String result)

`AgentCommunicator` implements the interface and it stores the monitoring and result server of the current executing job. It consumes the web services exposed by the monitoring and result servers.

Listing 5.8: Prime number Generator Job

```
package job.primenumber;

import org.dgpeer.agent.client.AgentCommunicator;
import org.dgpeer.agent.client.IAgentCommunicator;

public class PrimeNumber extends Aglet implements Serializable {
        long x, y, c = 0;
        long last_x = 2, last_c = 0;
        long ultimate_x = 1000000;

        transient IAgentCommunicator ac = null;

        public void onCreation(Object init) {
                System.out.println("Job Created");
        }

        public void run() {
```

```
            System.out.println("Staring job...");
            if(ac == null) {
                    ac = AgentCommunicator.getInstance();
            }
            System.out.println("ac all set to go!");
            for (x = last_x; x < ultimate_x; x++) {

                    double percentageCompleted =
                            ((double) x /
                            (double) ultimate_x) * 100;
                    if(percentageCompleted % 10 == 0) {
                            try {
                                    System.out.println(
                                            "Percentage: "
                             + percentageCompleted);
                                    ac.sendStatus(
                                            (int)
                                        percentageCompleted);
                            } catch (Exception e) {
                                    e.printStackTrace();
                            }
                    }

                    //
                    // Logic for primenumber generation
                    //

                    last_x = x;
                    last_c = c;
            }
            System.out.println("\nTotal: " + c);
            try {
                    ac.sendResult(
                            "Total number of primenumbers: "
                            + c);
            } catch (Exception e) {
                    e.printStackTrace();
            }
    }

    public void onDisposing() {
            System.out.println("Agent disposing...");
    }
}
```

5.4.3 State of the Job

The job's state should be maintained on migration, so that it can resume execution on the new host from the same state it has paused in the previous host. For this all the variables whose state has to be saved during migration is initialized at the class scope level. The variables initialized at the functional level lose their state. In Listing 5.8, all the important variables which define the state of the function is declared at the class level.

5.5 The Lifetime of a Job

This section describes the lifetime of a job from its creation to sending the results back to result server. Figure 5.5 gives the activity diagram of different stages in job execution.

1. The work unit (Figure 5.5) is implemented as an Aglet (mobile agent) and the class file is saved in a common codebase location.

2. The job server web service, `InsertJob` is called to insert the job details in the job repository. Along with the job_name and class_name, the constraints of the job are also inserted into the repository.

3. The web service, `ScheduleJob`, of the job server is called to schedule the job. When called, the instance of the job is created and the search for a candidate resource is started using resource discovery component. When a suitable candidate peer is found for job execution, the job schedule details, containing the job name, class name, codebase location, private key, ip addressees of job server, monitoring server, result server is sent to this peer.

4. The peer before starting the execution, calls the web service, `Authenticate`, to authenticate the job server, sending the job.

5. The mobile agent then starts executing in the host machine and the machine is monitored for any user activity.

6. Based on the monitoring policy defined inside the job, the mobile agent sends its updates to the monitoring server.

7. When a user activity is detected, the mobile agent subsystem finds another peer for job execution and then dispatches the mobile agent to the new peer.

8. When the monitoring server does not get frequent updates from the agent, it assumes the agent is destroyed and informs the job server that it should restart the execution of the job.

9. Once the job is executed in the peer, the result is sent back to the result server.

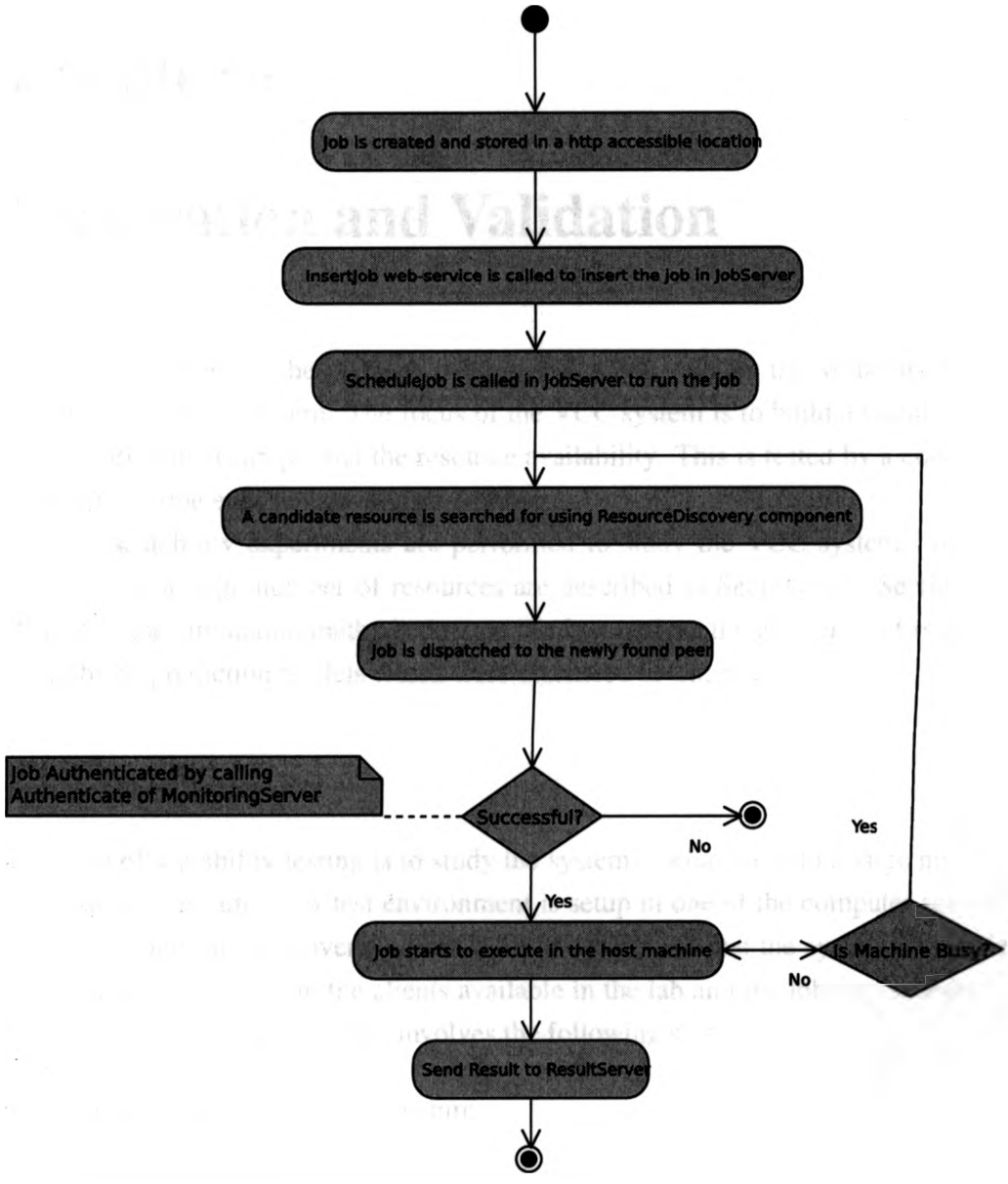

Figure 5.5: Job Life Cycle

Chapter 6

Verification and Validation

This chapter presents the methods used to verify and validate the Voluntary Cloud Computing (VCC) system. The focus of the VCC system is to build a scalable P2P framework which can predict the resource availability. This is tested by a combination of real-time experiments and simulations.

The scalability experiments are performed to study the VCC system's performance with a large number of resources are described in Section 6.1. Section 6.2 discusses the simulation methods that are used to analyze the efficiency of resource availability prediction models which were discussed in Chapter 4.

6.1 Scalability

The goal of scalability testing is to study the system's behavior with a large number of connected resources. A test environment is setup in one of the computer science department labs at the university (Lab 230) to deploy and test the system. A single peer group is formed with the clients available in the lab and the jobs are scheduled in the clients. Scalability testing involves the following steps:

1. Deployment of test infrastructure
2. Execution of Scenarios

Even though our system supports the formation of multiple groups, the scalability testing is performed using a single group. In the Voluntary Cloud Computing (VCC) system, the broadcasted resource status messages in a group are confined to that group (Section 5.3.6). It is only during the broadcast of status messages the group functionality is used by the system. Therefore, the presence of multiple groups does

not directly affect the goal of this testing. However, we have studied the system's behavior with 23 clients (1 group). The primary reason for this number is that this is currently the number of machines available to us for experimentation. Future work will attempt to look at more clients.

6.1.1 Deployment of test infrastructure

The following servers are set up during deployment (Figure 6.1):

- One instance of a management server
- Two instances of a job server
- Two instances of a monitoring server
- Two instances of a result server

Each instance of the job server, monitoring server and result server combination represent an organization in the real world. Two instances of these servers are chosen to test the functionality of the Voluntary Cloud Computing (VCC) system under multiple organizational domains.

GNU\Linux machines are selected from the cluster environment for installing the job server, monitoring server and result server. The instances of the job server, monitoring server and result server are hosted on two different machines and the management server is installed on a separate machine. Job server 1 and job Server 2 are registered with the management server.

A distributed cluster of machines running GNU\Linux (Fedora Linux 9.0) and university lab machines running Windows XP are chosen as clients for this experiment. A total of 23 machines (18 Windows XP, 5 GNU\Linux) are used for the scalability experiments. The VCC client software is installed on each of these workstations.

The management server runs on a machine using Cent OS flavor of GNU\Linux and the two machines running Fedora Linux 9.0 distribution are selected for hosting each instance of job server, monitoring server and the result server. Therefore a total of three GUN\Linux machines are used for running the servers.

The prime number generator job presented in Section 5.4 is used as the test job for executing in the machines. As the VCC system's functionality is independent of a job's implementation, the same job (prime number generator) is used in all the testing.

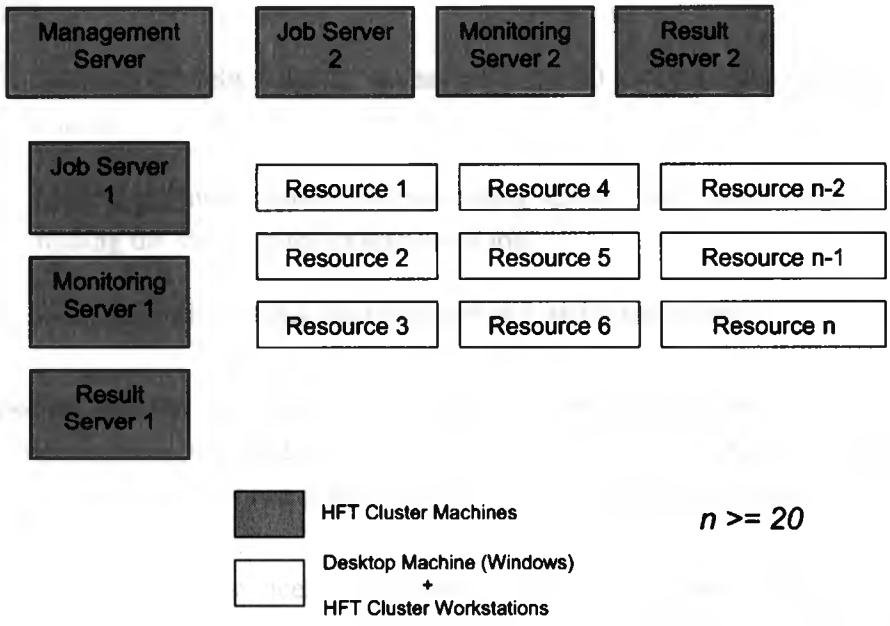

Figure 6.1: Test Infrastructure

6.1.2 Execution of Scenarios

Scenarios are run in this setup to verify the functionality of the system. The following scenarios are run in the test environment:

- Generic scalability test
- Generic Scalability test - Test Constraints
- Testing the migration

Generic Scalability Test

In this testing, a total of 40 job instances are scheduled from Job Server 1 and Job Server 2 simultaneously. The number of job instances (40) is selected to be close to two times the number of clients (23). This is done to test whether the job server waits for the execution of the job in the client before scheduling the next job. The goal of this testing is to test whether the job is scheduled in the infrastructure and it returns results. The approach take is summarized in the following steps.

Steps:

1. Schedule 20 jobs from the job server 1 and 20 jobs from job server 2 for execution.

2. During execution, query the monitoring server 1 and monitoring server 2 for finding the status of the execution of job.

3. After execution, check the result server 1 and result server 2.

Expected Result: On querying the monitoring server the list of all the job instances that are currently executed should be displayed. The results of all the executed job instances should have been populated in the result table after execution.

Results: The job instances are executed in the host machines. During the execution, the status of the execution was sent to the corresponding monitoring server (for every 10% completion of the job). After execution, the results are sent to the result server. (Listing 6.1)

Listing 6.1: JobInstance (`id monitorinfo` and `lastupdatedtime`)

```
...
| 129 | XPC-015:10,XPC-015:20,...,XPC-015:90 | 2009-02-19 12:30:00 |
| 128 | XPC-013:10,XPC-013:20,...,XPC-013:90 | 2009-02-19 12:04:59 |
| 127 | XPC-007:10,XPC-007:20,...,XPC-007:90 | 2009-02-19 11:51:04 |
| 126 | XPC-022:10,XPC-022:20,...,XPC-022:90 | 2009-02-19 11:50:59 |
| 125 | XPC-017:10,XPC-017:20,...,XPC-017:90 | 2009-02-19 11:50:05 |
| 124 | XPC-010:10,XPC-010:20,...,XPC-010:90 | 2009-02-19 11:48:34 |
...
```

Listing 6.1 gives the progress of several job instances from the monitoring server table. The `monitorinfo` represents the status of the job in the `hostname:percentage_completed` format. The third column, `lastupdatedtime` gives the time at which the row was most recently updated.

It was noticed that, if the job execution time[1] in the resource is less than the scheduling time[2] of the job, the jobs get scheduled in the same resource again and again leading to unfair scheduling policy, where the same resource is always selected for job execution. We plan to address the unfair scheduling in the future work by

[1]The total time taken for the execution of the job in the resource.

[2]Schedule time is the time taken for the job server to find the candidate resource for job instance execution. This involves the time taken for resource discovery time and resource reservation.

introducing heuristics at the scheduling level to enable the sharing of the jobs among the resources available.

Generic Scalability Test - Test Constraints

In this testing, a total of 20 jobs are scheduled from Job Server 1 and Job Server 2 simultaneously. Among the 20 jobs, 15 jobs have the constraint that the operating system on the host machine should be windows XP. This is represented by `OS=Windows XP`. The remaining 5 jobs have that constraint that `OS=Linux`. The goal of this testing is to test whether the job is scheduled in the respective operating systems represented by constraints.

Steps:

1. Schedule 10 jobs from the job server 1 and schedule 10 jobs from job server 2 for execution.

2. During execution, query the monitoring server 1 and monitoring server 2 for finding the status of the execution of job.

3. After execution, check the result server 1 and result server 2 to check whether the job has been executed.

Expected Result: On querying the monitoring server the list of all the jobs that are currently executed should be displayed. The jobs with OS constraints should be executed in the respective operating systems.

Results: All the 20 jobs are executed and the results were returned. One instance of the job execution reveals that it took 38:03 minutes for the end to end execution of the job. Out of the 20 jobs, 5 jobs which were associated with constraints `OS=Linux` were executed in the cluster machines, that were running the linux version of VCC client.

Testing Migration

This scenario tests the migrational capabilities of the job instance.

Steps:

1. Run a job from the job server 1 on resource r1.
2. When r1 is executing the job instance, disrupt the machine's state by moving the mouse or by pressing some key.

Expected Result: Upon disruption the job instance should pause its execution and migrate to a new resource r2. The job should resume its execution from the new machine r2 The monitoring server should note this migration and the result will be reflected in the monitoring server.

Results: When the host machine is disrupted by user activity, the job is migrated to a new machine, which is currently idle. The migration of the job can be seen in Listing 6.2. It shows the entries of the `jobinstance` that describes the migration of two jobs (from Section 5.2.3).

Listing 6.2: Row showing migration

```
| 52 | XPC-010:10,XPC-010:20,XPC-010:30,XPC-024:40,XPC-024:50.. |
| 24 | XPC-013:10,XPB-071:20,XPC-013:30.............XPC-014:90 |
```

The job instance with identifier 52 is interrupted when it had completed 30% of the job in `XPC-010`. It is migrated to `XPC-024` where its execution is continued. The job instance with identifier 24 went through three migrations before its completion (`XPC-013 to XPB-071 to XPB-013 to XC-014`).

The job that was scheduled calculates the count of prime numbers from 0 to 100,000. One instance of execution reveal that, it took 38:03 minutes to execute without migration and the same job took 39:13 minutes to execute with a single migration, with the difference of 1:10 minutes corresponding to the resource discovery and the job migration time from a client.

6.2 Resource Availability Prediction

The effectiveness of the resource availability prediction model is tested using simulation. The data set for simulation is created from the monitoring of the system usage from three different labs over a period of time. The behavior of two scheduling models, random scheduling model and the predictive scheduling model is studied by simulating these models over the collected data set.

In random scheduling model the resources are selected at random for job execution. The resource availability data is not considered for selection in random scheduling. In predictive scheduling, the resource's availability along with the group availability is considered for selecting a resource for job execution.

Section 6.2.1 describes the data collection part of the simulation. The simulator algorithm is presented in Section 6.2.2. The data collection for simulation is described in Section 6.2.3. The scenarios and results are discussed finally in Section 6.2.4.

6.2.1 Data collection

The resource (desktop computer) availability data is collected over a period of six weeks from three different labs of the computer science department at the university (Lab 230, Lab 235 and Lab 342). This data is used to find out at what time a particular resource was idle in a day. Each of these labs have a total of 25 machines each with each machine running the Microsoft Windows XP operating system.

A Win32 application was developed using the dot net framework for monitoring the user behavior in a resource. This application uses system hooks to listen to the events of `mouse_move` or `keyboard_press`. Once every 5 minutes it sends the data to a centralized server on whether the system was busy or not. (Figure 6.2)

The steps involved in data collection are as follows:

1. The data collector application is installed in each machine as administrator. The application runs at startup when a user logs in.

2. When a user logs in, the machine records the last known time since the keyboard or mouse activity and updates it every time a keystroke or mouse activity is sensed.

3. Once every five minutes the client contacts the centralized server and sends it the last idle time. If the machine is logged off or shutdown, then the data doesnt go to the server from the particular machine and machine is assumed to be idle at that time.

The machine information is stored based on the schema represented by Table 6.1 and the status is stored in Table 6.2.

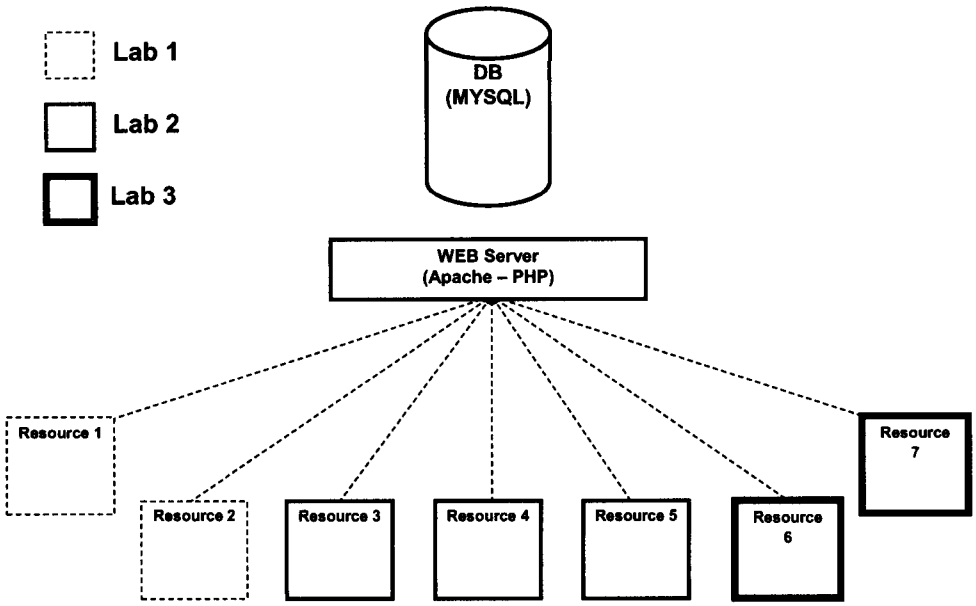

Figure 6.2: Simulation Software Setup

Entity	Description
id	Identifier for the machine
host_name	Host name of the machine
ipaddress	The ipaddress of the machine
date_added	Date when the machine was added
macaddress	The MAC address of the network interface card for security

Table 6.1: Machine Data

6.2.2 Simulator

The simulator executes the schedulers based on the data collected in the previous section.

Dataset Preparation

Table 6.2 represents information about when a particular machine was idle. For the ease of simulation, the recorded data (Table 6.1 and Table 6.2) are processed to form machine usage simulation table (Table 6.3) and group usage simulation table (Table 6.4).

Entity	Description
timestamp	The time corresponding to the data
machine_id	The machine from which the data has originated at this time
idlefor	The last time since user activity in this machine
process_usage	The average processor usage of the machine
memory_usage	The amount of memory free in the machine currently

Table 6.2: Machine Status

Table 6.3 the schema for the machine usage data to be used in the simulation.

Entity	Description
date	The date for which the data is recored
time	The time at which the data is recorded
day	The day of the week (1-7) in which the data is recorded
tot_samples	Total number of samples collected
idle_average	Number of times the system is found idle divided by tot_samples
is_currently_idle	Boolean value specifying whether the resource is currently idle or not
machine_id	The machine, for which the data is recorded

Table 6.3: Machine Usage Simulation

The `machine_id` is not in the data structure of the VCC client; however it is added at the simulation phase, as the simulation requires that data for all machines is recorded in a single table. Similarly the `date` and `day` fields are added in the simulation table for as each row in this table will represent the values of the table at that particular date. The `is_currently_idle` field is determined by processing the `idle_for` field of machine status (Table 6.2) and the previous week's `idle_average` field along with the `is_currently_idle` and `tot_samples` field is used to calculate the `idle_average` field of machine usage simulation (Table 6.3).

Along with the individual machine's behavior the group's behavior is also taken into consideration while prediction. Table 6.4 gives the table, group usage simulation, which represents the `group_task_avg` table of Section 5.3.6.

The machine usage simulation table represents the usage pattern at the machine level granularity and group usage simulation gives the usage pattern at a group level.

Entity	Description
date	The date for which the data is recorded
time	The time at which the data is recorded
day	The day of the week (1-7) in which the data is recorded
avg_nua	Average number of machines that are busy due to user activity
tot_samples	Total number of samples collected
group_id	the group (lab) for which the data is recorded

Table 6.4: Group Usage Simulation

Simulation of Prediction Models

After the tables, machine usage pattern (Table 6.3) and group usage pattern (Table 6.4) are populated, the simulations are run. The following scheduling models are run over the collected data to analyze the behavior of prediction in this environment: random scheduling and predictive scheduling.

Random Scheduling: In random scheduling, whenever a job is selected for scheduling, any resource that is currently available is selected for execution at random. This model is implemented as a simulation script using PHP. The simulation script takes `num_of_jobs`, `duration` and `start_time` as the command line arguments. The parameter `num_of_jobs` denotes the total number of jobs that has to be run during the simulation, `duration` gives the time to run a single job, `start_time` specifies the time of the day the execution of the jobs should start. Algorithm 7 gives the steps involved in random scheduling.

This algorithm takes as input an array of machine identifiers (`machine_id` in Table 6.3) that are considered for the simulation (Step 1 in Algorithm 7). The `machine_id_array` is randomized before selection (Step 3 in Algorithm 7) to make sure that the input sequence of machines doesn't influence which machine is selected for job scheduling. The `job_pointer` variable denotes the number of the jobs that are currently executed. The `machine_job_progress` is a hash table with the key denoting the `machine_id` and the value as the time elapsed in executing a job. Each entry of `machine_job_progress` is initialized with all the `machine_id` as the key and value as -1. The variable `current_time` gives the current simulated time initialized with the input parameter `start_time` (Step 4 in Algorithm 7).

After the initialization, once every five minutes in simulated time, each machine

in machine_id_array is checked to determine if a job can be scheduled on it. The availability status of the machine, for the current simulated time is queried from the machine usage simulation table (Table 6.3) (Step 7 in Algorithm 7). The machine can be either idle or busy. If the machine is idle and is not currently assigned any job (as indicated by machine_job_progress[machine_id] set to -1), then machine_job_progress is assigned to 0 (Step 10 in Algorithm 7) denoting that the machine has started executing a job. If the machine is idle and is assigned a job in machine_job_progress then machine_job_progress[machine_id] is incremented by 5 to denote that the machine has completed execution of another 5 minutes of the job it is currently executing (Step 12 and 13 in Algorithm 7). If the value of machine_job_progress[machine_id] is greater than input parameter duration, then it denotes that the job has finished executing in the machine and machine_job_progress[machine_id] is set the value as -1 (Step 20 in Algorithm 7). If the machine is not idle and is currently executing the job, the job is migrated to another machine which is currently free and not executing any job (Step 17 in 7).

Once every five minutes, job_pointer is checked to see whether all the jobs are scheduled (Step 23 in Algorithm 7). If all the jobs are scheduled and done executing (Step 24 in Algorithm 7), the simulation is terminated.

Using the Algorithm 7 various scenarios are run to study the behavior of the system. The scenarios are discussed in Section 6.2.4.

Predictive Scheduling In this scheduling, the equation (Equation 4.5) derived in Section 4.3.3 is used to predict whether the system is going to be available for a given point of time for execution. The simulation script assumes the same input parameters as that of random scheduling. The script also takes an array of machine identifiers, machine_id_array, that are to be considered for simulation as the input. Before assigning jobs to a machine, the machine_id_array is sorted by its probability of availability in descending order and then jobs are assigned to the machines. Algorithm 9 details the steps of this scheduling method.

Like random scheduling, in predictive scheduling, machine_id_array, job_pointer, machine_job_progress and current_time are initialized (Step 1, 2 and 3 in Algorithm 9). Along with these parameters, group_machine_map is initialized with each machine_id as the key and the corresponding group_id as the value.

Once every five minutes in simulated time, the functions

Algorithm 7 Random Scheduling

```
1: machine_id_array = All the machine that are considered for simulation
2: randomize(machine_id_array)
3: Initialize job_pointer, machine_job_progress
4: current_time = start_time
5: while true do
6:   for every machine_id in machine_id_array do
7:     machine_usage = query_machine_usage_table(machine_id, current_time)
8:     if machine_usage.is_currently_idle == true
       AND machine_job_progress[machine_id] == -1 then
9:       job_pointer = job_pointer + 1
10:      machine_job_progress[machine_id] = 0
11:    else
12:      if machine_usage.is_current_idle == true then
13:        machine_job_progress[machine_id] += 5;
14:      end if
15:    end if
16:    if NOT machine_usage.is_current_idle
       AND machine_job_progress[machine_id] != -1 then
17:      migrate_job_to_free_machine();
18:    end if
19:    if machine_job_progress[machine_id] >= duration then
20:      machine_job_progress[machine_id] = -1
21:    end if
22:  end for
23:  if job_pointer > num_of_jobs then
24:    if all elements in machine_job_progress is -1 then
25:      break from while loop
26:    end if
27:  end if
28:  current_time += 5
29: end while
```

load_group_avg_nua and load_group_current_nua are called (Step 6 and 7 in Algorithm 9). The function load_group_avg_nua fills the hash table group_avg_nua with group_id as the key and avg_nua (average number of machines that are busy due to user activity) as the value. The avg_nua is assigned by querying the group usage simulation table (Table 6.4). The function load_group_current_nua loads the hash table group_current_nua with key group_id and value current_nua. The current number of machines that are busy due to user activity (current_nua) in a group and is assigned by querying the machine usage simulation table (Table 6.3) for the count of all the machines in a particular group (determined by group_machine_map) that are busy.

After the loading of the hash tables, each machine is queried in the machine usage table (Table 6.3) to check its availability at the given time (Step 11 in Algorithm 9). If the machine is available and is not currently assigned any job, then its prediction_value is calculated by calling the function get_prediction_value (Step 9 in Algorithm 9). The function get_prediction_value calculates the availability factor of the resource based on Equation 4.5 and returns it as the prediction_value. This function is described at the end of this section in detail.

The calculated predicton_value is stored in the sort_machines hash table with machine_id as the key (Step 12 in Algorithm 9). If the machine is idle and is executing a job then its job execution elapsed time is incremented by 5 denoting 5 more minutes of completion of execution (Step 15 in Algorithm 9). If the machine is not idle and if it is currently executing a job, then it is migrated to a free resource by calling the function migrate_job_to_free_machine (Step 19 in Algorithm 9). If the value of machine_job_progress[machine_id] is greater than input parameter duration, then it denotes that the job has finished executing in the machine and the hash table machine_job_progress[machine_id] is set the value as -1 (Step 21 in Algorithm 9).

After every machine is checked in the machine_id_array the sort_machines is sorted based on its prediction_value in the descending order (Step 22 in Algorithm 9). The job is assigned to each machine and job_pointer is incremented denoting the scheduling of a job (Step 24 in Algorithm 9).

Once in every five minutes simulated time gap, job_pointer is checked to see whether all the jobs are scheduled (Step 31 in Algorithm 9). If all the jobs are scheduled and done executing (Step 31 in Algorithm 7), the simulation is terminated.

get_prediction_value: The function `get_prediction_value` calculates the availability factor of a peer using the Equation 4.5 of Chapter 4. It is presented in the Algorithm 8.

In the function `get_prediction_value` (Algorithm 8), $raf(t)$ is denoted by `idle_average`, N_{ua} is denoted by `current_nua`, N_{je} is denoted by `current_je`, *tot* is denoted by `num_of_machines` and $P_{resource\_availability}$ is given by `prediction_value`.

Algorithm 8 get_prediction_value

```
1: Input machine_id, idle_average, current_time
2: group_id = group_machine_map[machine_id]
3: avg_nua = group_avg_nua[group_id]
4: current_nua = group_current_nua[group_id]
5: current_je = get_current_je()
6: current_ga = num_of_machines - (current_nua + current_je)
7: prediction_value = (current_ga - avg_nua) * idle_average
8: return prediction_value;
```

The `group_id`, `avg_nua` and `current_nua` are assigned by the hash tables which are already loaded with key and values. The `current_je` denotes the number of machines that are currently busy due to job execution. This is determined by the function `get_current_je` (Step 5 of Algorithm 8) which queries the `machine_job_progress` to determine this value. Finally the prediction_value is calculated and then returned from this function.

The scenarios run using this algorithm are described in the next section. (Section 6.2.4)

6.2.3 Data Collection Results

The resource usability data collected in lab 342, lab 230 and lab 235 reveal the pattern graphed in Figure 6.3, Figure 6.4 and Figure 6.5 respectively. Each of these labs have 25 machines. Each graph shows the number of machines that are used at a given time.

It can be seen that out of the three labs, lab 230 is the busiest and lab 342 is the least busy. All the three labs are relatively not used after 12:00am in the night and they are busiest around the afternoon. The weekends, saturday and sunday, least busy periods and Monday is the most busy period.

The following can be inferred:

1. A job when scheduled in all these labs, has a better probability of running without interruption during nights than in days.

Algorithm 9 Predictive Scheduling

```
1: machine_id_array = All the machine that are considered for simulation;
2: Initialize job_pointer, machine_job_progress, group_machine_map
3: current_time = start_time
4: while true do
5:    Initialize sort_machines hash table
6:    load_group_avg_nua()
7:    load_group_current_nua()
8:    for every machine_id in machine_id_array do
9:       machine_usage =
         query_machine_usage_table(machine_id, current_time);
10:      if machine_usage.is_currently_available AND ma-
         chine_job_progress[machine_id] == -1 then
11:         prediction_value =
            get_prediction_value(machine_id, machine_usage.idle_average)
12:         sort_machines[machine_id] = prediction_value
13:      else
14:         if machine.is_currently_available then
15:            machine_job_progress[machine_id] += 5
16:         end if
17:      end if
18:      if NOT machine.is_currently_available
         AND machine_job_progress[machine_id] != -1 then
19:         migrate_job_to_free_machine()
20:      end if
21:      if machine_job_progress[machine_id] >= duration then
22:         machine_job_progress[machine_id] = -1
23:      end if
24:   end for
25:   array_sort(sort_machines)
26:   for every machine_id in sort_machines do
27:      job_pointer++
28:      machine_job_progress[machine_id] = 0
29:   end for
30:   if job_pointer > num_of_jobs then
31:      if all elements in machine_job_progress is -1 then
32:         break from while loop
33:      end if
34:   end ifcurrent_time += 5
35: end while
```

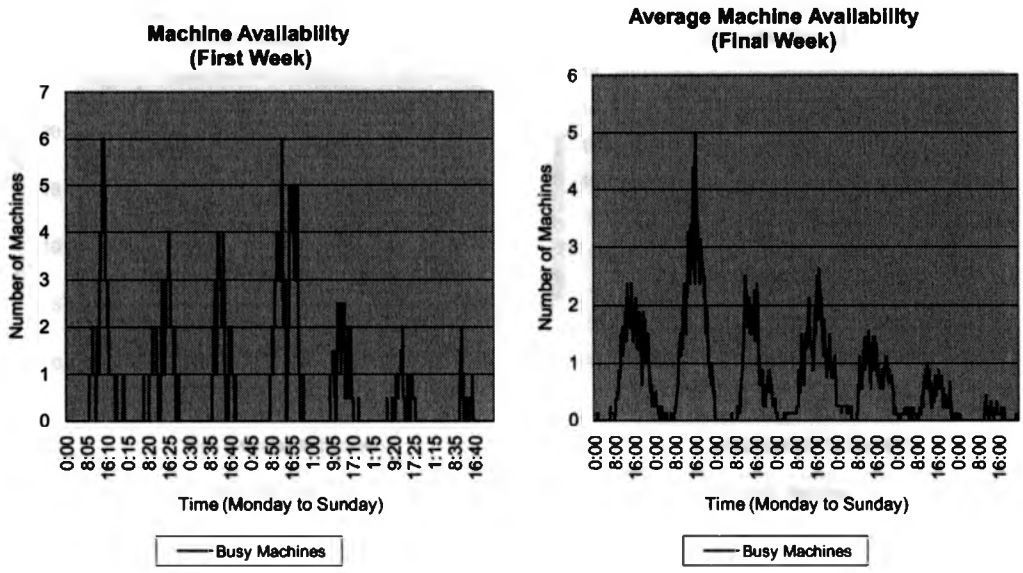

Figure 6.3: Lab 342 User Availability Pattern

2. Weekends are the ideal days to schedule jobs in the system than the weekdays.

3. At any point of time of scheduling, if lab 342 is given more preference than the other two labs, the jobs are least likely to be interrupted. At its peak usage only 7 machines are busy in lab 342 when compared to lab 230 (20 machines) and lab 235 (15 machines).

6.2.4 Scenarios

To validate these inferences, both the random and predictive scheduling simulation scripts are run with different values for their input parameters (`num_of_jobs`, `duration`, `machine_id_array` and `start_time`). For a different set of `num_of_jobs`, `duration` and `start_time` with a constant `machine_id_array` (all the machines from lab 342, lab 230 and lab 235) both the simulations are run and the results are tabulated in Table 6.5.

`num_of_jobs`: The number of jobs that are scheduled at a time in the system influences the number of migrations that the job goes through, thereby affecting the total time for job execution. It is found that, when `num_of_jobs` is less than the total number of available machines in the system then the predictive scheduling makes a huge difference in the number of migrations. When `num_of_jobs` is more than

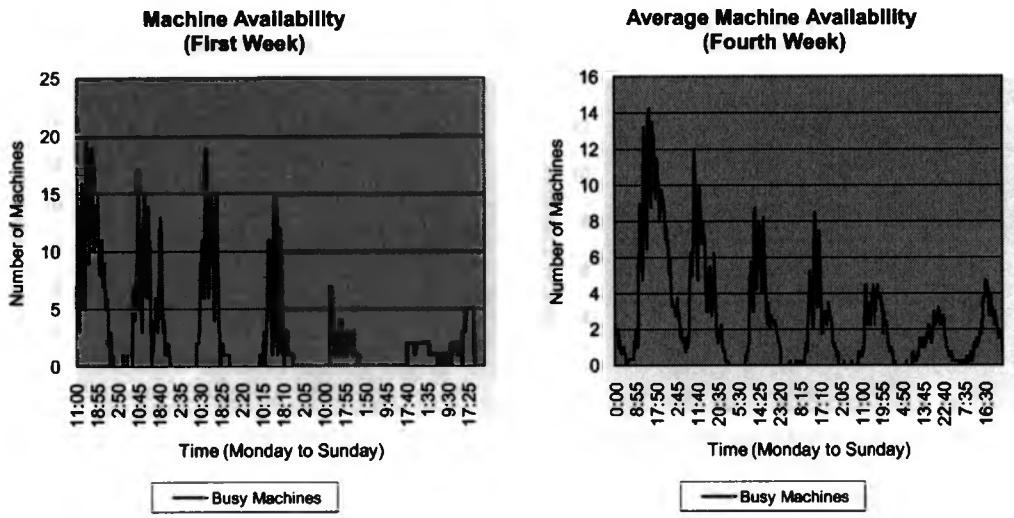

Figure 6.4: Lab 230 User Availability Pattern

the number of machines, all the free resources in the system are selected for job execution and therefore predictive resource selection makes little difference. It can be found that in Table 6.5, as the number of jobs executed in the system increases, the difference between the migration of random and predictive scheduling, decreases.

`duration:` As the job duration increases the higher the probability that the job is interrupted by user activity. Therefore predictive scheduling makes a difference when long-duration jobs are run in the system. It can be seen from Table 6.5 that, when long duration jobs are run (duration=200 and duration=300) the difference in the number of migrations using random and predictive scheduling is more prominent. The predictive scheduling results in far lesser migrations when compared with the random scheduling (16 migrations for random when compared to 4 migrations for predictive (duration = 200) and 19 migrations for random when compared to 10 migrations for predictive (duration = 300)).

`start_time:` The time at which the job is scheduled plays a crucial factor on the amount of time, the job is going to take to finish its execution. Two time slots, one peak hour (2008-12-01 12:00:00) and one relatively less busy (2008-12-11 10:00:00) are chosen to run the same number of jobs with the same duration (number_of_jobs=100 and duration =100 minutes). It is found that, in the peak hour the

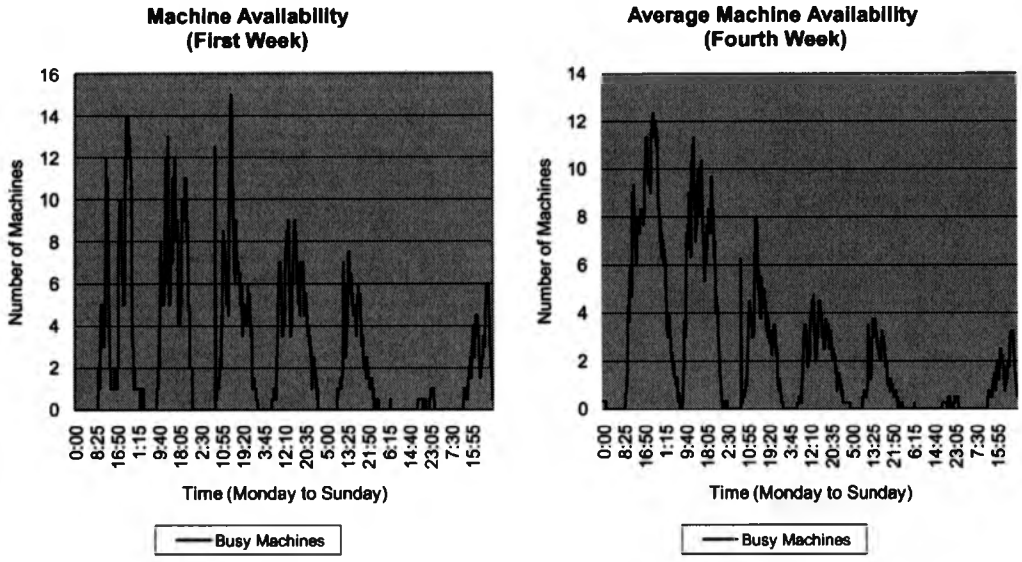

Figure 6.5: Lab 235 User Availability Pattern

number of migrations is in the order of 100. Predictive scheduling results in fewer migrations compared to the random. (Table 6.5). However, in an off-peak time, the predictive scheduling does not decrease the migration by lot.

The following characteristics are hence observed from the scenarios run in the simulation.

1. The predictive scheduling decreases the amount of migrations significantly if the `num_of_jobs` is lesser than the total number of machines in the system.

2. During peak hours, predictive scheduling is useful in selecting resources which are less prone to be interrupted by user activity and hence reduce the number of migrations.

3. Predictive scheduling also makes a difference with long-duration jobs (greater than 45 minutes).

4. During off-peak hours and when `num_of_jobs` is higher than the number of machines, the predictive scheduling does not decrease the migration by huge factor.

Input (num_of_jobs, duration, time)	Sched. type	Migrations	Elapsedtime
10, 30, 2008-12-01 12:00:00	R	4	35 minutes
	P	0	30 minutes
10, 100, 2008-12-01 12:00:00	R	10	1:50hrs
	P	2	1:45 hours
10, 200, 2008-12-11 10:00:00	R	16	3:30hrs
	P	4	3:25hrs
10, 300, 2008-12-11 10:00:00	R	19	5:10hrs
	P	10	5:10hrs
100, 30, 2008-12-01 12:00:00	R	55	2:00hrs
	P	48	1:50hrs
100, 100, 2008-12-01 12:00:00	R	427	8:25hrs
	P	304	7:10hrs
100, 100, 2008-12-11 10:00:00	R	2	3:25hrs
	P	3	3:30hrs
1000, 100, 2008-12-11 10:00:00	R	42	1 day 3:55hrs
	P	33	1 day 3:55hrs

Table 6.5: Simulation Results (R-Random, P-Predictive)

Chapter 7

Conclusion

The stated goal of this work is to develop a framework for resource availability prediction in desktop grids on a P2P network. We have designed and implemented a working prototype that has demonstrated decentralized resource availability prediction in a peer-to-peer (P2P) environment. Our experimental results shows that resource availability prediction decreases the interruption in job execution.

Section 7.1 presents the contribution of this work and Section 7.2 describes the areas of future research.

7.1 Contributions

The major contributions of this thesis are the following:

- Development of a robust desktop grid which takes into account multiple job servers, mobility of the job and desktop volatility.
- An approach to decentralized resource availability prediction of both dedicated and non-dedicated desktops.

We have proposed and implemented a unique desktop grid framework focused on resource selection in a P2P environment. Our system provides insight into problems and open issues surrounding such an environment. Our experimental results show the feasibility of such a framework and highlighted the practical importance of resource availability prediction. The results also show the possibility of migration of jobs and orchestration of multiple servers and peers in a P2P framework.

Resource availability prediction was studied in a client-server environment in other work (Section 2). Our study focused on the effectiveness of such a prediction

in a P2P environment which to the best of our knowledge has not been done. We studied the process of predicting the availability in both dedicated and non-dedicated desktops with detailed experiments run on non-dedicated desktop environment. None of the work we surveyed took the non-dedicated desktop machines into account when using prediction. Our experiments show that in a setup where the number of jobs is fewer than the total number of resources, the resource availability prediction reduces the amount of migrations (job interruption) the job experiences during its execution. This change is more prominent when the user activity is high.

7.2 Future Work

This section describes the areas of future research for this work. We acknowledge that the framework we have built will benefit from its use in a real-time environment. Work in the areas explained in this section will definitely help for the adaptation of our infrastructure in a production environment.

7.2.1 Resource Discovery

Resource discovery is currently achieved in the framework using JXTA's decentralized resource discovery methods. JXTA's basic discovery model does not provide the functionality to filter the resources based on multiple constraints. Therefore, the filtering will have to take place at the application level. Moreover, our system was tested with only 20 clients. The eventual goal is to scale it across thousands of clients.

Distributed Hash Tables [31] (DHT) can be used to tag resources with constraints as keys so that they become easily searchable in a P2P network. However implementation of DHTs with multiple keys, would be challenging.

Future research could focus on a robust resource discovery mechanism which searches for resources based on multiple constraints at the underlying P2P network layer. This will greatly increase the performance of the system by reducing the time taken for resource discovery.

7.2.2 Resource Availability Prediction

The VCC system uses probability techniques to predict the availability of the system at a given point of time, based on its past availability data. Our results show that in comparison with random scheduling, the prediction techniques we adopted, reduces the amount of migrations the job experiences during its execution, when the number

of jobs are lesser than the total number of machines. However, when the number of jobs are more than the number of machines, prediction doesn't make any significant difference. Future work should study the comparison of our model against a model which considers each machine as a dedicated resource than a non-dedicated entity.

The prediction used in the system (Section 4.3.3) can be improved to use more advanced prediction and learning models. John et.al [11] describes mathematical models to predict the machine availability. Ling Shang et.al [34] uses a Trust Model [35] to predict the behavior of a user based on the user activity learning done in the past. Implementation of mathematical models presented by John et.al [11] and the trust model [34] in the system and studying their effectiveness against our method will be interesting.

7.2.3 Job Programming Model

The VCC system uses the Aglet framework for its job execution and migration functionality. In the Aglet programming model, the job designers are expected to maintain the state of the execution of job at the class level. This is necessary for the job to resume its job execution into the migrated machine (Section 5.4.3). The job developer might have to go through a learning curve in understanding this programming model. Moreover, this model introduces design limitations when implementing complex jobs.

Future work could focus on the integration of a powerful programming model for the mobile agents which resembles a traditional programming methodology, without compromising on its migration functionality. MapReduce [17] is a programming model for processing and generating large data sets, adopted by many companies like Google, Yahoo and Amazon. Integration of such a model in the VCC system with the migrational functionality could be interesting.

7.2.4 Trust

In a desktop grid framework, trust has two dimensions to it. The desktop users should trust the software which has been installed in their machines for job execution. The work-item running in the desktop system should not compromise the security of the system. On the other hand, as the job is executed in a remote untrusted environment, the job should not be corrupted by the desktop machine where it is hosted.

Implementing trust was beyond the scope of this thesis. Any deployment of the system in the production environment should give trust issues significant importance,

as that greatly affects whether the users are going to adopt the system.

7.2.5 Fault Tolerance

Our system has multiple failure points. The job servers can fail while scheduling, the monitoring servers can crash, the result servers can be flooded with result data so that it is no more in a state to receive results and the peers can go offline while job execution. In an ideal system these scenarios should be an expected than an exception scenario.

Fault tolerance is not built into our system as it is not the focus of our research. However, our system provides the necessary framework to implement fault tolerance methodologies over it.

7.2.6 Socio-economic Models

We did not address the social aspects with our work. Our work assumed that the job servers will not take a free ride (exploit the usage by scheduling a large number of jobs) of the framework. No limits were placed on the number of jobs that a job server can schedule. There was no billing system implemented directly in the framework to detect the resources shared by a peer and number of jobs scheduled by the job server. This would not only provide incentive for users to donate more resource, but also would prevent the abuse of the system.

Buyya et.al [13] proposed various social economic models for regulating supply and demand in a computing grid environment. Future work should investigate the use of these models in our framework.

7.2.7 Resource Management

Management of a framework of this scale can be challenging. Our framework is built on the P2P backbone, therefore management in terms of topology changes and individual client management is going to be minimal in the system. However, the management server, the job server, the monitoring server and the result server should still be maintained. The client software installed in the individual peers of the system should be auto-updated frequently for bug fixes and other enhancements.

7.2.8 Leveraging Free Disk Space

In a non-dedicated environment (like university labs) the machines are not exclusively assigned to a single person. Therefore, the hard disk is not mostly used for storing private data. This leave a sizable amount of hard disk space free in the system. This space can be used for sharing in the framework. Currently, only the CPU cycles are used by the system for resource sharing. Along with the CPU cycles a massive data store can be formed by using all the free space available in the hard disks.

Vazhkudai et.al [38] proposed FreeLoader, a system that aggregates unused desktop storage space and I/O bandwidth into a shared cache space, for hosting large data sets. Future research could investigate on the feasibility of integration of a such a system in our framework.

7.3 Summary

This thesis presented a scalable framework for resource availability prediction for both dedicated and non-dedicated desktops over a peer-to-peer network. P2P system removes the responsibility of resource discovery from a single dedicated server and shares it with all the peers in the network. The prediction of resources reduces the amount of interruptions, a job experiences during its execution. It will be interesting to let the system evolve to be adaptable in a real-time environment. We look forward to furthering this research to that end.

Bibliography

[1] Amazon ec2 (http://www.amazon.com/gp/browse.html?node=201590011).

[2] Apache axis2 (http://ws.apache.org/axis2/).

[3] http://www.d-grid.de/.

[4] http://www.garudaindia.in/.

[5] http://www.grid-support.ac.uk/.

[6] Sqlite (http://www.sqlite.org/).

[7] Apache tomcat webserver (http://tomcat.apache.org/).

[8] D.P. Anderson. Boinc: A system for public-resource computing and storage. *5th IEEE/ACM International Workshop on Grid Computing*, pages 365–372, 2004.

[9] F. Berman, G. Fox, and A.J.G. Hey. *Grid Computing: Making the Global Infrastructure a Reality*. Wiley, 2003.

[10] BOINC. http://boinc.berkeley.edu/.

[11] J. Brevik, D. Nurmi, and R. Wolski. Automatic methods for predicting machine availability in desktop grid and peer-to-peer systems. *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, pages 190–199, 2004.

[12] K. Budati, J. Sonnek, A. Chandra, and J. Weissman. Ridge: combining reliability and performance in open grid platforms. *Proceedings of the 16th international symposium on High performance distributed computing*, pages 55–64, 2007.

[13] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger. Economic models for resource management and scheduling in grid computing. *Concurrency and computation: practice and experience*, 14(13-15):1507–1542, 2002.

[14] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing*, 63(5):597–610, 2003.

[15] J. Clark and S. DeRose. Xml path language (xpath) version 1.0. w3c recommendation. *World Wide Web Consortium*, 1999.

[16] C. Cramer and T. Fuhrmann. Bootstrapping Chord in Ad Hoc Networks: Not Going Anywhere for a While. In *Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW'06)*, pages 168–172.

[17] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *COMMUNICATIONS OF THE ACM*, 51(1):107, 2008.

[18] Dempster. *Annals of Mathematical Statistics*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1967.

[19] K. Egevang and P. Francis. RFC1631: The IP Network Address Translator (NAT). *RFC Editor United States*, 1994.

[20] G. Fedak, C. Germain, V. Neri, and F. Cappello. Xtremweb: A generic global computing system. *Cluster Computing and the Grid, 2001. Proceedings*, pages 582–587, 2001.

[21] L. Gong. Jxta: A network programming environment. *Internet Computing, IEEE*, 5(3):88–95, May/Jun 2001.

[22] M. Knoll, A. Wacker, G. Schiele, and T. Weis. Decentralized Bootstrapping in Pervasive Applications. In *Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 589–592. IEEE Computer Society Washington, DC, USA, 2007.

[23] Derrick Kondo, Andrew A. Chien, and Henri Casanova. Resource management for rapid application turnaround on enterprise desktop grids. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 17, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2153-3. doi: http://dx.doi.org/10.1109/SC.2004.50.

[24] D.B. Lange and M. Oshima. Mobile agents with Java: The Aglet API. *World Wide Web*, 1(3):111–121, 1998.

[25] MJ Litzkow, M. Livny, and MW Mutka. Condor-a hunter of idle workstations. *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111, 1988.

[26] Sun Microsystems. Jxta: A network programming environment.

[27] Sun Microsystems. Jxta v2. 5. x: Java programmer's guide. *White Paper*, 2007.

[28] R. Moats. URN Syntax. Technical report, RFC 2141, May 1997, 1997.

[29] D. Nurmi, J. Brevik, and R. Wolski. Modeling machine availability in enterprise and wide-area distributed computing environments. Technical report, U.C. Santa Barbara Computer Science Department, October 2003.

[30] A. Oram and A. Oram. *Peer-to-peer: harnessing the benefits of a disruptive technology*. O'Reilly, 2001.

[31] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of the 2001 SIGCOMM conference*, volume 31, pages 161–172. ACM New York, NY, USA, 2001.

[32] Sheldon M. Ross. *Introduction to Probability Models*. Academic Press, 2006.

[33] A. Roy and M. Livny. Condor and Preemptive Resume Scheduling. *INTERNATIONAL SERIES IN OPERATIONS RESEARCH AND MANAGEMENT SCIENCE*, pages 135–144, 2003.

[34] L. Shang, Z. Wang, X. Zhou, X. Huang, and Y. Cheng. Tm-dg: a trust model based on computer users' daily behavior for desktop grid platform. *Proceedings of the 2007 symposium on Component and framework technology in high-performance and scientific computing*, pages 59–66, 2007.

[35] P. Smets and R. Kennes. The transferable belief model. *Artificial Intelligence*, 66(2):191–234, 1991.

[36] J.E. Smith and R. Nair. The Architecture of Virtual Machines. *COMPUTER*, pages 32–38, 2005.

[37] Tanenbaum and Steen. *Distributed systems: principles and paradigms*. Prentice Hall Pearson Education International, 2002.

[38] S.S. Vazhkudai, X. Ma, V.W. Freeh, J.W. Strickland, N. Tammineedi, and S.L. Scott. Freeloader: scavenging desktop storage resources for scientific data. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Washington, DC, USA, 2005.

[39] A. Weiss. Computing in the clouds. *netWorker*, 11(4):16–25, 2007.

[40] J.E. White. Mobile agents. *Software agents*, pages 437–472, 1997.

[41] B.J. Wilson. *JXTA*. New Riders.

[42] G. Woltman and S. Kurowski. The Great Internet Mersenne Prime Search, 2000.