

18-842

Group 1

Distributed Compute Market



Alok Shankar

Aditya Agarwal

Pratik Indap

Kushal Dalmia

Carnegie Mellon University

18-842 Group 1

Abstract

Distributed Compute Market is a system that enables users to effortlessly and safely lease their machines in a global market of processing capacity. The consumers of these resources are user-defined compute intensive jobs that require multiple machines for an efficient and timely computation. The market provides an opportunity for some users to utilize their idle resources for monetary gains and for others to get access to a large number of nodes for running heavy computations at minimal cost and effort. The system isolates the user from various complex distributed systems issues such as Distributed Job Management, Dynamic Task Placement, and Fault Tolerance etc, to provide a safe and easy-to-use computing service.

Contents

1. Introduction	4
1.1. Purpose of this document.....	4
1.2. Scope of the document.....	4
1.3. Background	4
1.4. Motivation.....	5
1.5. Goals	6
1.6. Application Areas	6
1.7. Use Case	7
2. Functional Requirements.....	8
2.1. Description	8
2.2. Availability.....	8
2.3. Technical Issues and Related Risks	8
3. Specific Requirements.....	9
3.1 User Interface	9
3.2 Security	10
3.3 Job Manager.....	10
3.4 Network Manager	10
3.6 Non-functional Requirements	11
4. System Design	11
4.1. System Components	11
4.1.1 Flask Web Server & User Interface	11
4.1.2 Job Manager.....	15
4.1.3 Network Manager	15
4.2 Single Node Architecture	16
4.3 Node Bootstrapping Process.....	17
4.4 Inter-Node Communication	18
5. Implementation Details	18
5.1. Bootstrap Server	18
5.1.1 Bootstrap API:	19
5.1.2 Bootstrap Redundancy:.....	20
5.2. Job Manager.....	20

5.2.1 The job	20
5.2.2 Handling and Executing the Job	21
5.3. Network Manager	22
5.3.1 Network messages	22
5.4. Web Application.....	24
5.5 Cloud Integration	24
6. Sequence Diagrams.....	25
6.1. Node Initialization	25
6.2. Submit Job.....	26
6.3. Resource Available	28
6. Scalability Analysis	29
7. Evaluation & Testing	29
7.1 500 x 500 Matrix Multiplication.....	30
7.2 N Queens Problem	30
8. Project Management	32
8.1 Team Organization	32
8.2 Task List	32
8.3 Project Schedule	33
9. References	34

1. Introduction

1.1. Purpose of this document

The document provides detailed requirements and a high level design of the Distributed Compute Market (DCM).

1.2. Scope of the document

This document provides relevant background, functional requirements and implementation details for a Distributed Compute Market (DCM). It provides a comprehensive overview of the key challenges, design decisions and high-level data flow for the project. The language-specific implementation details are beyond the scope of this document.

1.3. Background

The Internet has provided us with a global network of interconnected computers. The recent technological advances in high performance networking and computing, coupled with their availability has changed the way we do computing. The trend in high performance computing is to move away from proprietary supercomputers to those based on commodity hardware and software components. This has led to the popularity of clusters of computers, interconnected through local/system-area networks, as a platform for solving large-scale compute intensive problems. Today, the Internet/Web has become pervasive and millions of computers and users are online. Many of these systems are often under-utilized, a fact accentuated by the global geography since busy hours in one time-zone tend to be idle hours in another. Distributing computations is thus very appealing over the Internet.

The traditional client-server Internet model ("client" asks for and receives information from "server") is beginning to give some ground to peer-to-peer (P2P) networking - where all network participants are approximately equal. The primary advantage of P2P networks is that large numbers of people share the burden of providing computing resources (processor time and disk space), administration effort, creativity and legal liability. Thus, using P2P networks as a low-level networking service to build distributed computing services seems obvious and effective.

There are several issues that must be resolved for this to be feasible. Although using volunteer computers' idle CPU cycles for solving supercomputing problems appears simple, realizing a flexible and widely acceptable resource management, scheduling

and general-purpose paradigm for application programming is a complex task. This is mainly due to resources' geographic distribution, heterogeneity, distributed ownership with different policies and priorities, reliability, and availability conditions.

A market-based economic paradigm for resource management would help in addressing all of these issues in a simplified manner, since economic institution has been proven to be the best mechanism for regulating demand and supply. Furthermore, it offers incentives for volunteers to share their computational resources and encourages consumers to optimally utilize resources by balancing time-frame and access-costs. Even the profit can be shared with a market for coordinating users.

1.4. Motivation

A number of projects such as SETI@Home and distributed.net have successfully exploited the distributed computing services paradigm for solving specific application areas. However, these implementations limit users to very specific applications and provide no incentive to users for making their resources available. In most cases the volunteer's resources are running applications which provide neither intellectual nor financial gains to the owner. Also, since these systems lack a business model, the demand generally exceeds the supply for computing resources.

At the other end of the spectrum are large organizations providing resources to customers in a model heavily biased towards these organizations. Although such environments allow custom applications to run, the customers are generally dependent on the organization's quality of service guarantees and SLAs to fulfill their resource needs. This leads to a model where large organizations with computing resources act as a service provider and tend to dominate the computing resource market. This leaves little or no way for a small provider with single or few resources to benefit from the concept.

The existence of such extreme paradigms for computing services clearly indicates the need for a system which has the advantages of both the worlds. With the increase in computing capabilities of commodity workstations, a computing service which allows consumers to run custom applications and provides small players a plug-n-play kind of architecture to utilize their idle resources, is an urgent need. The economics of a free computing market model would ensure that the service is self-sustaining and improves the overall utility of mostly underutilized computing resources.

To summarize, the following rationale is a strong motivation to design and develop a Distributed Compute Market:

- The proliferation of largely unused computing resources with their greatly increased CPU speed and availability of fast, universal network connections.
- High performance computers (formerly called supercomputers) are very expensive to buy and maintain.
- Many computing tasks relegated to these (especially massively parallel) computers could be performed by a “divide and conquer” strategy using many more, although slower, processors.
- The lack of incentives for volunteers to lend their computational resources for running these computing tasks.
- The absence of computing grids which allow customers to purchase these resources at minimal cost and effort.

1.5. Goals

- Allow customers to easily plug their underutilized resources to a distributed compute market with the incentive to earn money from them.
- Allow users to request multiple nodes in the distributed market to run their custom compute intensive jobs at a cost proportional to the number of requested resources.
- Perform seamless job management by job partitioning and handling common distributed systems issues such as node failures etc.

1.6. Application Areas

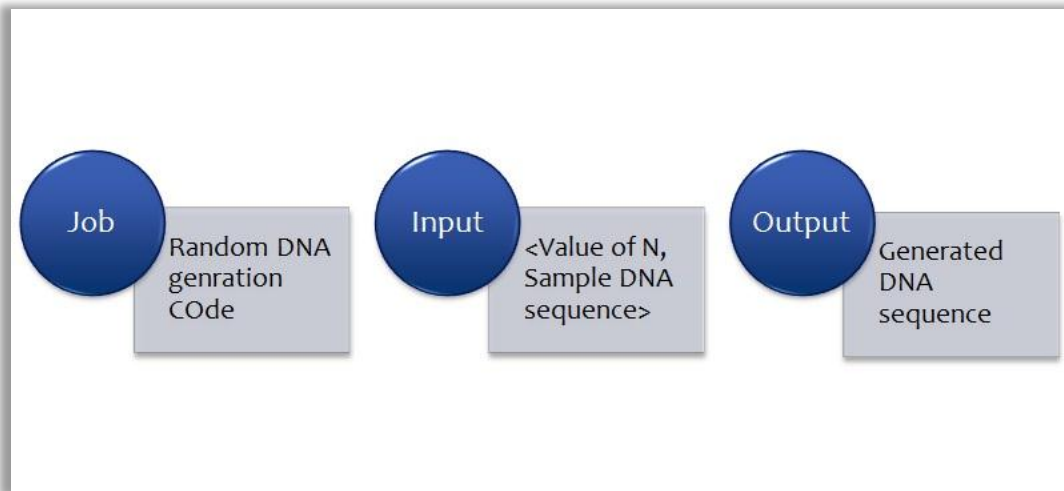
Distributed Compute Market can be used to run many different types of applications, which can be categorized into four main classes:

- distributed supercomputing
- high-throughput
- on-demand
- collaborative

One of the simplest concepts for the utilization of Distributed Compute Market is to be able to run an application somewhere else when your own machine is too busy or otherwise does not have the required resources. Often, mathematical calculations are commutative, associative, or linear in some way. The simple example of adding a list of numbers illustrates this. By altering some potentially unimportant rules in the computations involved in a calculation, it is possible to break the ordering requirement and thus make it possible to execute more of the application in parallel. DCM computing would offer a way to solve Grand Challenge problems such as protein folding, financial modeling, earthquake simulation, and climate/weather modeling. In general, any application which is compute intensive and can be trivially parallelized is a good contender for Distributed Compute Market.

1.7. Use Case

A sample use case could be a random DNA generator. The job consists of a program that takes an input sample DNA sequence and a number N. The output is a random DNA sequence generated based on the input values. A larger value of N means more work to be done by the generator. The following diagram provides a snapshot of the job as a whole.



The user will submit the job, containing the DNA sequence generator, file/s with <N, sample DNA sequences> value pairs. The code must be idempotent, and should be able to run by itself given the input file and output file. The application on the user node will accept the input code and data and number of nodes required for this computation and pass it on to the lower job management layer. The job manager would then divide the input into various sets to be sent to other nodes. The job manager sends a copy of code, and a subset of input values to each node, where it would be executed in a safe and sandboxed environment. The input pair would look like following:

```
<1000,'GGCCGGGCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGG'>
```

Each of this input would generate an output looking like :

```
'GGCCGGGCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGC
GGGCGGATCACCTGAGGTCAGGAGTTCGAGACCAGCCTGGCCAACATGGTGAAA
CCCCGTCTCTACTAAAAATACAAAAATTAGCCGGGCGTGGTGGCGCGCGCCTGTA
ATCCCAGCTACTCGGGAGGCTGAGGCAAGGAGAATCGCTTGAACCCGGGAGGCGG
AGGTTGCAGTGAGCCGAGATCGCGCCACTGCACTCCAGCCTGGGCGACAGAGCG
AGACTCCGTCTCAAAAAGGCCGGGCGCGGTGGCTCACGCCTGTAATCCCAGCACT
TTGGGAGGCCGAGGCGGGCGGATCACCTGAGGTCAGGAGTTCGAGACCAGCCTG
GCCAACATGGTGAAACCCCGTCTCTACTAAAAATACAAAAATTAGCCGGGCGTGGT
```



```
GGCGCGCGCCTGTAATCCCAGCTACTCGGGAGGCTGAGGCAGGAGAATCGCTTG
AACCCGGGAGGCGGAGGTTGCAGTGAGCCGAGATCGCGCCACTGCACTCCA'
```

This output would then be sent back to the original node's job manager. The user would be able to see the results once all the nodes have returned the results.

2. Functional Requirements

2.1. Description

Distributed Compute Market would allow users to supply and consume computing resources in a leased manner. It aims to provide a self-managed distributed system capable of submitting, executing and managing user defined jobs with minimal cost and effort.

2.2. Availability

The Distributed Compute Market guarantees availability of computing resources through fault tolerance and node management. It ensures that the user jobs are run to completion even in the presence of node and link failures, through automated node failure detection and dynamic scheduling.

2.3. Technical Issues and Related Risks

- **Security:** Security is an important component in the distributed computing environment. If you are user running jobs on a remote system, you care that the remote system is secure to ensure that others do not gain access to your data. If you are a resource provider that allows jobs to be executed on your systems, you must be confident that those jobs cannot corrupt, interfere with, or access other private data on your system.
- **Job Management:** The DCM acts as an abstract interface to the heterogeneous resources available in the system. The system needs to provide the facilities to allocate a job to a particular resource, the means to track the status of the job while it is running and its completion information, and the capability to cancel a job or otherwise manage it.
- **Distributed Resource Management:** DCM needs to maintain knowledge about resource availability, capacity, and current utilization. Within DCM, both CPU and data resources will fluctuate, depending on their availability to process and share

data. As resources become free within the market, they can update their status and the clients would require this dynamic information to make informed decisions on resource assignments.

- **Node Bootstrapping:** Since the system involves nodes dynamically getting added and removed from the distributed network, there needs to be a mechanism to bootstrap nodes such that they become part of the network. The bootstrap service needs to ensure total network connectivity even in the presence of node and link failures by providing multiple connections into the network for a single node. This connectivity needs to be performed in a load balanced manner to ensure minimal and fair network load on each node of the system.
- **Mutually Exclusive Resource Allocation:** Since multiple job requests might contend for the same available resources in the system, the DCM should arbitrate these requests and assign resources in a mutually exclusive manner. Another challenge is to allocate resources with minimal exchange of messages between the nodes, thus reducing the load over the network.
- **Fault Tolerance and Failover protection:** There can be many issues related to fault tolerance that need to be considered. In case of failures the nodes need to address the lost job state and make up for it through dynamic rescheduling the part of computation that failed. Due to the distributed nature of system, such issues need to be addressed in a graceful manner.

3. Specific Requirements

3.1 User Interface

Most users today understand the concept of a Web portal, where their browser provides a single interface to access a wide variety of information sources. The DCM portal provides the interface for a user to launch applications that will utilize the resources and services provided by the market. It would allow users to specify node requirements and define compute jobs. From the perspective of the resource provider, the portal would be the interface to make the resource available/unavailable at any instant. Resource providers and consumers can also view their account balances and the amount they have earned by running each job. Another addition that we made was to provision for users to track progress of their jobs in real-time. Both consumers as well as providers

can now view how far the job execution has proceeded. Also if the job execution fails, the user is notified and the job execution is terminated.

3.2 Security

A major requirement for distributed computing is security. At the base of any distributed untrusted code execution environment, there must be mechanisms to provide security which ensure safe and isolated execution of user code. Sandboxing untrusted code presents an efficient and fast way of providing such facilities without restricting user applications in any manner. We have defined our own sandboxing environment where impose limits on the child processes. The process limits are set before the child process starts execution. The limits include the maximum number of child processes that a process can create, maximum size of the heap and maximum number of file descriptors that can be opened by a process at any instant.

3.3 Job Manager

Once connected, the user will be launching compute intensive applications. Based on the application, and possibly on other parameters provided by the user, the next step is to identify the available and appropriate resources to utilize within the network. This task is carried out by a job manager. A distributed algorithm ensures that the Job Manager on each node has a consistent global view of available nodes. After the Job Manager finds the available nodes to run the Job on, a Job Manager partitions the job into n chunks as specified by the user. The Job Manager then runs a dynamic scheduling algorithm to schedule the chunks on different machines in the distributed system. Once the job is scheduled, the sub jobs are shipped to the chosen nodes where they are run in a safe and secure environment. The Job manager would also maintain the state of the executing sub jobs, and in the event of any failure would reallocate that specific sub job to a different node. At the end of job execution, the Job Manager would assimilate the results from the nodes running the sub jobs and return it back to the user. The word Broker and Job Manager are used interchangeably.

3.4 Network Manager

The Network Manager ensures the node's connectivity to the rest of the distributed system. Through timely heartbeats, the network manager indicates the node's liveliness and detects neighbor nodes failures. These failures would trigger update messages throughout the network, resulting in a consistent network topology graph of the current network. The network manager would also be responsible for routing sub jobs and results to/from remote nodes by estimating the shortest available path to a particular destination node.

3.6 Non-functional Requirements

- **Performance**

The measure of performance for jobs executing in this environment would be the turn-around time for submitted jobs. The turn-around time would be dependent on the following factors:

- Communication Delays
- Data Access Delays
- Resource Contention
- Reliability

The DCM would strive to achieve a lesser total turn-around time for a job as compared to the time required to execute the complete job on any single node of the system.

- **Scalability**

The system should be able to handle a large number of nodes in the system with multiple user jobs running in parallel. The large scale would impact the amount of information maintained and network traffic handled at any node. The use of smart dynamic networks which avoid a complete mesh topology among nodes would allow the system to be scalable and efficient.

- **Portability**

The initial implementation of Distributed Compute Market would be restricted to Linux and Unix environments. A Windows based Distributed Compute Market would be difficult to build since the system call library support to develop low-level Network Manager is insufficient. Extending the market to include Windows based hosts can be a future extension to the project.

4. System Design

4.1. System Components

4.1.1 Flask Web Server & User Interface

Each node in the system would run a local Flask-based web server to render dynamic web pages to the user. These dynamic pages would act as the user interface and would allow the user to query and use the system. All user-triggered activities such as making the resource available, Submitting Jobs, Checking Job Status etc. can be performed through the use of this interface. The web browser has been chosen as the means to interact with the user due to its ease of use and general availability on all systems. The Flask Web Server would communicate with the underlying components of the node through well-defined TCP ports and interfaces.

The web-based interface would comprise of the following views:

- **Home Page**

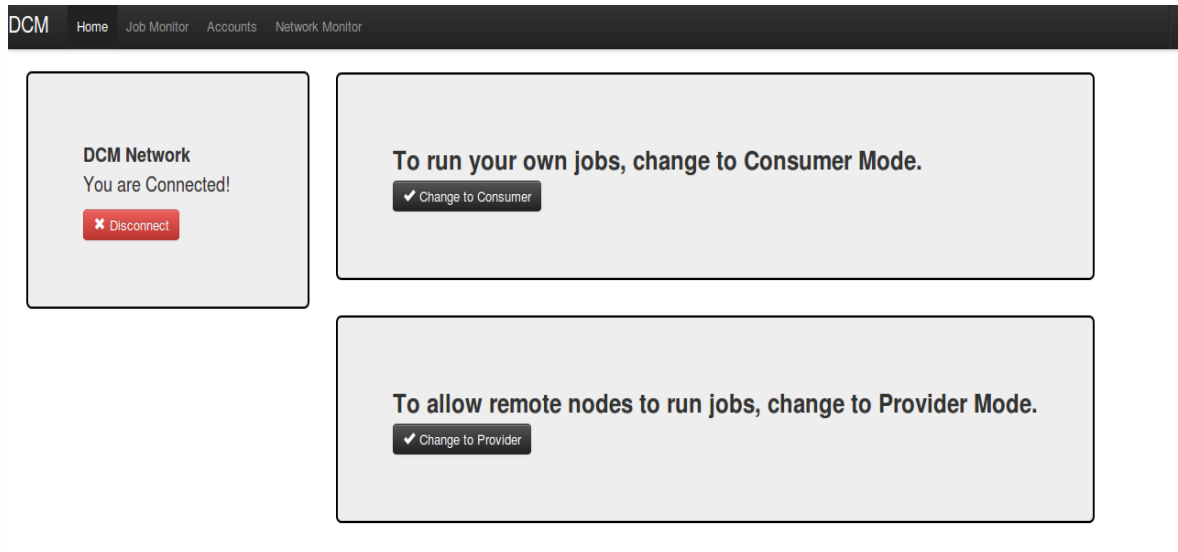
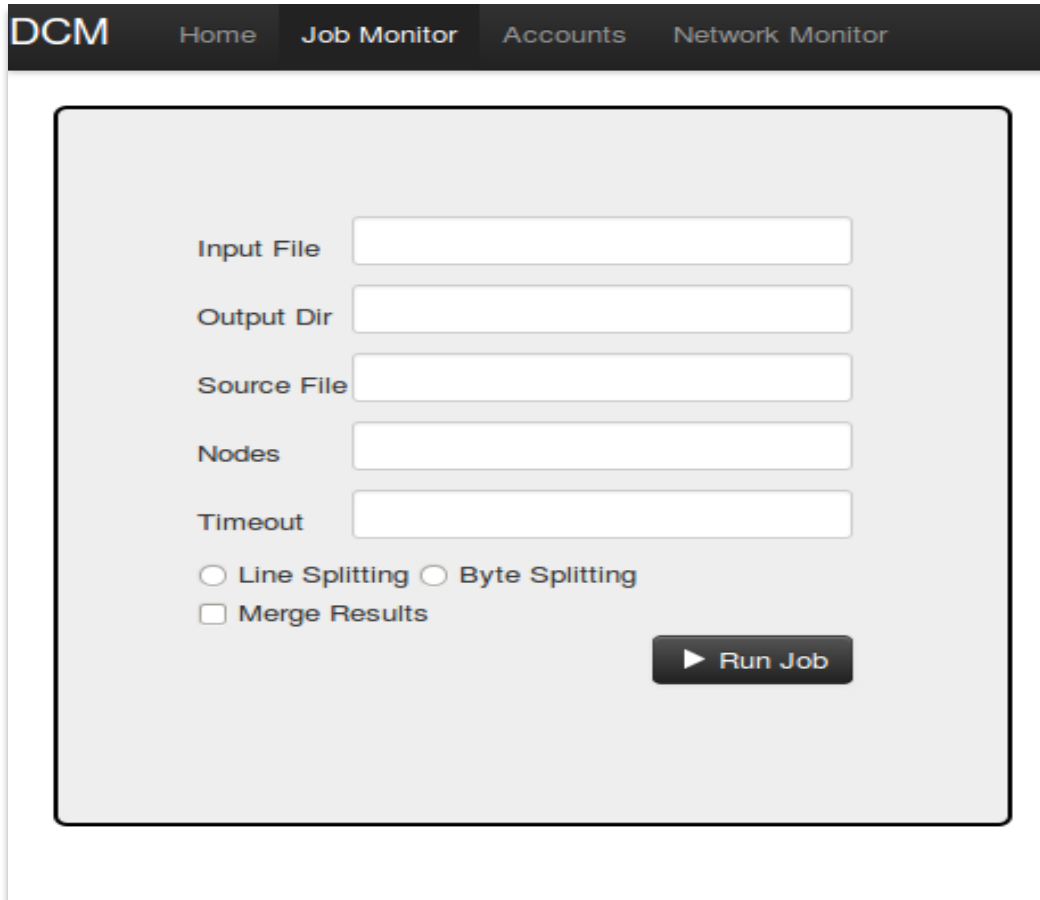


Figure 1: Home page of DCM

The home page allows the user to connect his/her node to the Distributed Compute Market. On clicking on the “Connect” button, the Web application contacts a bootstrap server running at the url “www.dcm.csproject” to obtain the three <IP-address:Port> for other nodes in the network. This information is used by the lower-level distributed component to connect to various other nodes in the network and maintain the complete graph of network topology. Once the node is connected to the network, user can choose to be Provider or a Consumer.

- **Job Manager**




The screenshot shows the 'Job Monitor' tab in the DCM web application. The interface includes a navigation bar with 'Home', 'Job Monitor', 'Accounts', and 'Network Monitor'. The main content area contains a form for submitting a job. The form has five text input fields: 'Input File', 'Output Dir', 'Source File', 'Nodes', and 'Timeout'. Below these fields are two radio button options: 'Line Splitting' and 'Byte Splitting', and a checkbox labeled 'Merge Results'. A 'Run Job' button with a play icon is positioned at the bottom right of the form.

Figure 2: The Job management page

The Job Manager page allows the user to submit jobs to the DCM. A job is specified by the 3-tuple <Source Code File, I/P File, O/P File>. Along with the job details, the user must specify the number of nodes required to run this job and the maximum time that the job should run. On submitting the job, the web application communicates all the input information to the local Job Manager which would then schedule and manage the user job. If the user is a Provider, the page displays the status of the current job using a tracker. If the user is a Consumer, then after submitting the job the page displays the current status of the job being executed.

Job Attribute	Value
Input File	/home/kushal/nqueens-ip.txt
Output File	/home/kushal
Source File	/home/kushal/nqueens.py
Nodes	1
Timeout	45

Current Job Status



NEW_JOB_REQUEST

SPLITTING_JOB

Figure 3: Dynamic display for a job

- **Accounts**

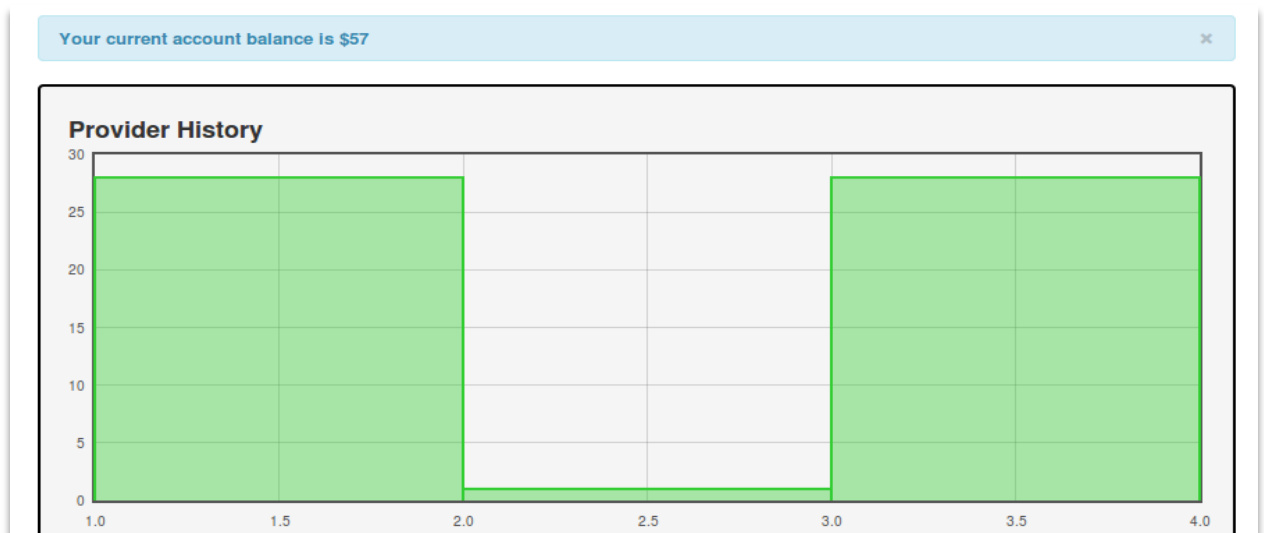


Figure 4: The account management Interface

The Accounts page allows the user to see the current account information. The page displays two graphs, one each for Provider and Consumer. For a provider, the graph displays the amount earned by running the jobs. For a Consumer, the graph shows the total cost of running the job in DCM.

- **Network Monitor**

This page has been added for debugging purposes. It shows the current network status of DCM. The page displays the Neighbor nodes, Available nodes and the messages sent and received by the node.

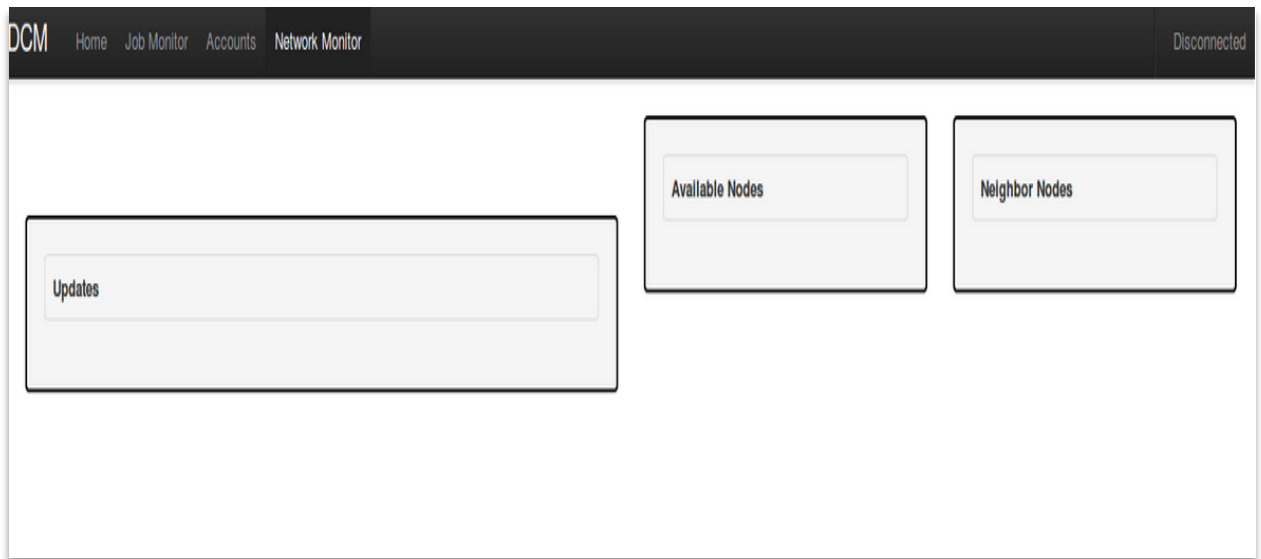


Figure 5: The network Manager

4.1.2 Job Manager

The Job Manager acts as a mediator between the user application and the DCM resources. This component would be responsible for

- Allocation of resources - It would allocate the resources in a mutually exclusive manner. The Job Manager would utilize the services provided by the Network Manager to send updates/probes for available resources.
- Job scheduling - The job manager would divide the job into n components as defined by the user and then ship the sub jobs to available nodes. The Job Manager would continuously monitor the status of the sub-jobs and ensure job completion even in the presence of node failures.
- Job execution – The job manager would be responsible for executing the received sub job in a safe and secure environment.
- Result collection - The job manager would also be responsible for gathering of results after a job has finished execution.

4.1.3 Network Manager

The network manager is the node's lowest layer component which connects the node to the distributed network. It listens for updates from other network managers on a fixed TCP port and maintains a global view of the network topology. It provides services to the Job Manager for shipping jobs and collecting results from various nodes in the

system. The network manager performs a periodic heart-beating mechanism to ensure network connectivity and discovering routes to remote nodes in the network.

4.2 Single Node Architecture

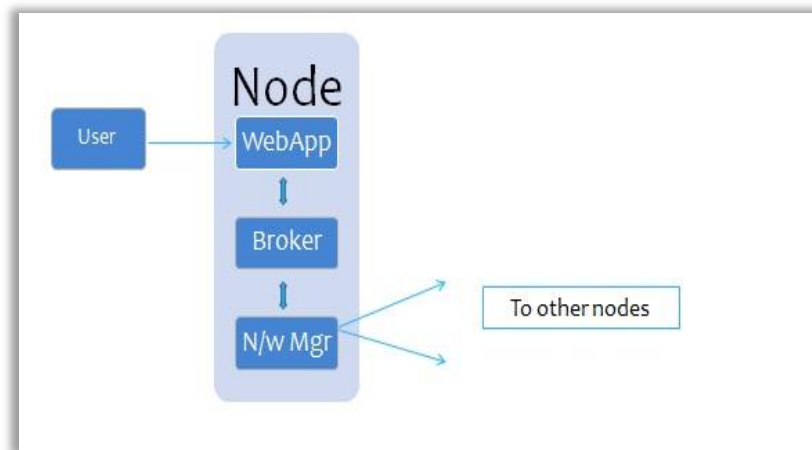


Figure 6: The architecture of a Node

A single node consists of the above mentioned three components working at different layers of the system. Since the web interface needs updated information and triggers new updates in the system, it communicates directly with the Job Manager. The Job Manager queries the Network Manager for the current state and requests services such as sending messages, receiving updates etc. from it. The communication mechanisms between various layers of the node are:

- The web application and the Broker communicate through TCP connections
- The Broker and the Network Manager request each other's services through well-defined interfaces and call-back routines.

4.3 Node Bootstrapping Process

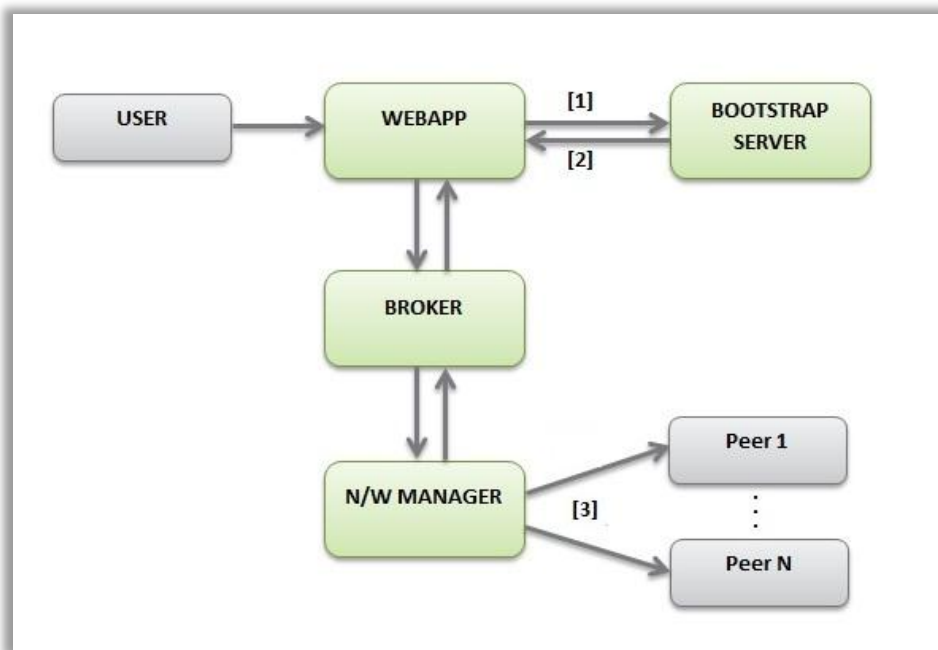


Figure 7: Bootstrapping Process

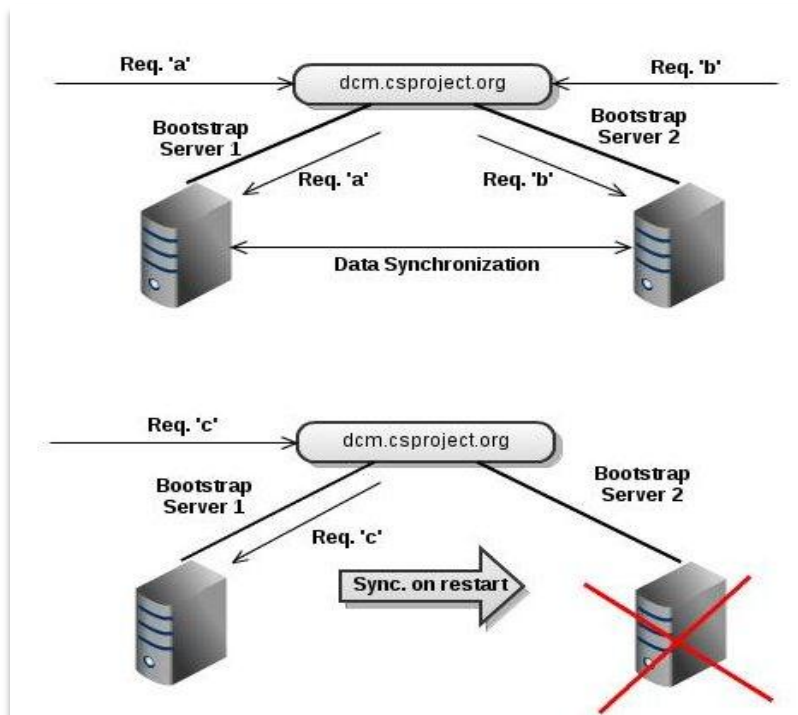


Figure 8: Bootstrap availability with DNS

- (1) On the button click, the web application contacts a bootstrap server at a predefined URL at `www.dcm.csproject`
- (2) The server returns a subset of other peer nodes in the system
- (3) This information is then passed on to the local Network Manager which initializes TCP connections to these nodes. Once connected, the network manager ensures that the node is connected to its peers through heartbeats and update messages.

4.4 Inter-Node Communication

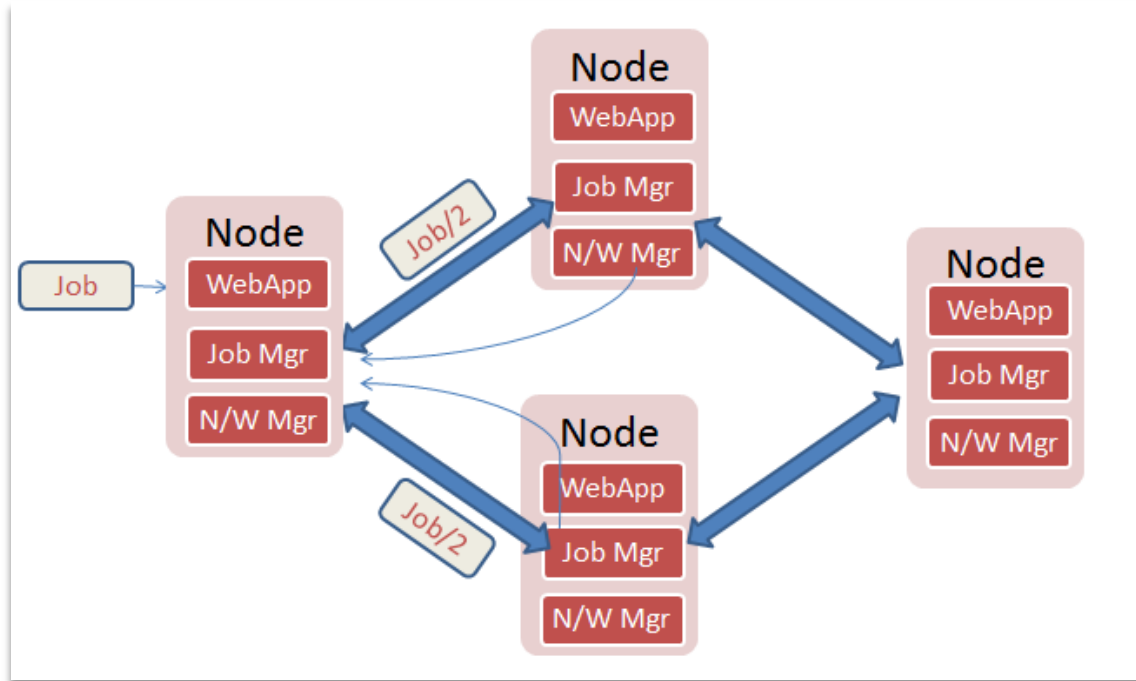


Figure 9: Inter-Node Communication

The nodes in the distributed system exchange a number of different types of messages. All messages are sent/received by the Network Manager. The messages either update the global state of the network or are passed on to the Job Manager for managing jobs and system resources.

5. Implementation Details

5.1. Bootstrap Server

The bootstrap server is responsible for the system bring up. At initialization, the bootstrap creates a database to maintain entries respective to each node. This database will be synchronized between the redundant bootstrap instances, which we will describe later. The schema of this database is shown below:

```
1 drop table if exists Nodes;
2 create table Nodes(
3     node_id string not null ,
4     ref_count integer not null,
5     primary key (node_id)
6 );
```

Figure 10: The database Schema for Bootstrap Server

The `node_id` is a unique identifier for each node in the table. We use the string `IP:Port`, to make sure that each node has only one entry in the database. The `ref_count` is a counter that keeps track of how many connections this particular node holds at a particular moment. Each time a request is made to the bootstrap server, this `ref_count` is used, either to update, or to get least loaded nodes to be sent back as peers. Any node entering the system will first contact the bootstrap server via an HTTP request. The requests are in the form of different URLs which then are implemented as different functions at the server. The API provided by bootstrap are described in next subsection.

5.1.1 Bootstrap API:

- **Register (/register/<ip_add>/<port_no>/):** This is implemented by the function `register(ip_add, port_no)` at the server side. Each time a node wants to enter the system, it will use this function to register itself in the system, and the server will return a list containing `MAX_PEERS`(system configuration parameter) nodes which will be neighbors of the node. Here, we return the nodes which have the least `ref_count` in the database, making them the nodes that are least connected. All the `ref_count` values of nodes in the list are also incremented by one, to indicate one more connection, including that of the new node that has registered. This way we help prevent network partitioning.
- **Unregister (/unregister/<remote_ip>/<remote_port>/<ip_add>/<port_no>/):** Implemented by the function `unregister(remote_ip, remote_port, ip_add, port_no)`, this function is used by a node or its peer to inform the bootstrap about unavailability of another node. The `remote_ip` and `remote_port` is the ip address and port number of the process invoking the function, and the next two arguments indicate the ip address and port information of the node that has died. Upon receiving this notification, the server will decrement the `ref_count` corresponding to `remote_ip` and `remote_port` and delete the entry corresponding to the node which has exited the system.

- **Disconnect (/disconnect/<ip_add>/<port_no>/):** This URL is used by a node to delete itself, implemented by `disconnect(ip_add, port_no)`. This just results in deletion of the entry from the database.

5.1.2 Bootstrap Redundancy:

Since the bootstrap server is a single point of failure, we implement a redundancy protocol to make sure that the system is still functional in the event of a bootstrap failure. We maintain a redundant bootstrap server, which is an exact replica of the bootstrap. All our URLs mentioned before are aware of the redundancy in bootstrap servers, and all requests are forwarded to the redundant server to ensure perfect consistency between all bootstrap servers. The function `initBootstrap()` makes sure that the database state is always same on both servers in case a server comes up after failure. Once a server comes up, it pulls the database from the functional server. The function `getLatestDatabase(backup, port)` is used to pull the database from the other server. To allow nodes to locate the bootstrap server, we use a unique URL which maps to all bootstrap servers in a round-robin fashion. The address used is `dcm.csproject.org`.

5.2. Job Manager

5.2.1 The job

We define a job in form of a class called `Job`, which consists of following members:

```
class Job:
{
    String ipFile;
    String srcFile;
    String opFile;
    Integer numNodes;
    String owner;
    ProcessObject process;
    Boolean isTerminated;
    Boolean mergeResults;
    Boolean splitByLine;
    Integer timeout;
    Float cost;
}
```

The `ipFile` field is the input file containing the input data for a particular job, `srcFile` contains the source code, `opFile` denotes the output file which needs to be created and the result of a job execution will be written. `numNodes` is the number of nodes the customer wants the job to be run on, `mergeResults` indicates whether the results have to be merged finally, `splitByLine` indicates the split granularity, and `timeout` is the maximum time a job is allowed to run for.

5.2.2 Handling and Executing the Job

- Scheduling the Job:** When the job arrives at a node, it has to be scheduled to the other nodes in the system to be run. The function `jobmgr.scheduleJob()` handles this scheduling. The first step in scheduling is splitting. It calls `jobmgr.splitjob()` to perform the splitting of input data set into multiple chunks indexed by the relative offset. After splitting, the node scheduling the task sends a `CPU_REQ` message to all the nodes in the list of available nodes. This will result in these nodes sending information about their current CPU usage data. Depending on this, the jobs are scheduled to N least loaded nodes, where N is specified by the job creator and is part of the job object. We send messages to only first 2N nodes for obtaining CPU request, to avoid flooding the network with `CPU_REQ` and response messages. After deciding the least loaded N nodes, the scheduler sends a reserve notification via the network manager `reserveNode()` function. This makes sure that the node is reserved in a mutually exclusive manner and does not conflict with other nodes attempting to reserve the node concurrently.

Once the nodes have been reserved, the job manager requests the network manager to schedule jobs on each of these reserved nodes. The network manager spawns a separate thread for each chunk in the job and schedules the job by sending the job data and code to the reserved node. The job manager also starts a job timeout timer to make sure that the job is completed in the max duration specified by the consumer.

- Executing the Job:** On receiving the job from a remote node for execution, the function `jobManager.executeJob(jobmgr)` handles the executing of the job. Here we create a sandboxed environment for the job and execute it. The sandboxing is done with the `psutil` python library. The algorithm to execute a job is as follows:
 - Extract the job from `jobmgr`
 - Open `job.ipFile` in read mode, and `job.opFile` in write mode
 - Start the timer
 - Read the `config.cfg` file, extract the system parameters for a sandbox and set them for current sandbox (function `setProcessLimits()`).
 - Create the sandbox and run the job.
 - Wait for process to finish, or timeout to occur.
 - In case of either of these events, calculate the cost , close the files
 - Invoke `jobmgr.completeJob()` to indicate job completion

In case of a failure, or exception in job execution, we kill the job, indicate the failure of execution and close the files.

- **Splitting and merging the job:**

The `jobmgr.splitJob()` divides the input for the job at the source node into `n` chunks where each chunk is a different input chunk for a node to start executing. The `jobmgr.mergeJob()` function handles the merging of results into the output file once the results of execution come back to the source node. The merging is basically a concatenation operation of various output files generated and sent across the network by various nodes in the system.

5.3. Network Manager

The network manager implements the various low-level functions required in a distributed system.

- **Start Manager:** The function `startManager()` handles the initialization of network manager. The network manager creates a multi-threaded server bound to a fixed port to accept remote node requests and connections. It also initiates connections with various neighbor nodes to exchange heartbeats. The network manager uses a separate timer for each neighbor to detect failures and missing heartbeats.
- **Destroy Manager:** The `destroyManager()` implements functionality of taking the node off the network. It closes all neighbor connections, shuts down the server bound to the port and destroys the instance of the network manager.
- **Message Handler:** Handle various network messages and take required actions, the network messages are described in the next section.
- **Job Handler:** The job handling routines of the network manager perform two basic tasks – Getting jobs from remote nodes for executing locally in case of providers and scheduling jobs on various nodes in case of consumers. Both these implementations handle message timeouts and node failures through careful exception handling.

5.3.1 Network messages

Following is the description of various message types this module processes:

- **NEIGHBOR_INIT**
Notify a remote node to add itself as a neighbor, initiate a new connection and initialize appropriate timers to exchange heartbeat messages
- **HEARTBEAT**

The usual keep alive message exchange between nodes to ensure availability of a node. Upon receipt, each node restarts the aliveTimer for the node that sent the message.

- **UPDATE**

The update message is used to synchronize the information about available nodes in the system. This periodic message is sent by each node to its neighbors to make sure all nodes in the system have a consistent view of all the available nodes in the network. This message is really crucial in cases where a node fails and comes back up and needs to populate its list of available nodes in the system.

- **RES_AVL**

When a node makes itself available for processing remote jobs, it sends this message to its neighbors to notify them of its availability. The node receiving this message updates its free list with the node that sent this message and forwards the message to ensure all alive nodes in the network are aware of this update.

- **RES_UNAVL**

This message is the notification of unavailability of a node. When a alive timer associated with a node's neighbor expires, the node increments a fail count for this neighbor. On reaching a maximum threshold, the node marks the neighbor as failed and floods the RES_UNAVL message to denote node failure. All nodes receiving this message mark the node as failed and remove it from the free list. The receiver also checks whether it was running a job for a remote node or has scheduled a job chunk on the failed node and takes appropriate action.

- **RESERVE_REQ**

It is a request to reserve the node for running a job. It is sent by a node scheduling a job, to reserve this node. Upon receipt this node sends either an ACK or a NACK message indicating availability or unavailability of this node.

- **RELEASE_REQ**

Request to release this node. If a job is running, the node receiving the message kills the job and releases itself. Otherwise, it just releases itself.

- **JOB_CODE**

This message is sent when a node sends a job to be run on another node. The node receiving the message will notify the job manager of the job arrival, run a basic ACK-based protocol to accept the job information and start job execution.

- **JOB_COMPLETE**

It is a notification of a job being completed. This message means the node receiving this has to handle the job response.

- **CPU_REQUEST**

Request to get current CPU usage. The receiving node forms a message containing details of its current CPU usage. The recipient node also indicates whether it is available to run jobs by marking the response with an ACK/NACK.

5.4. Web Application

The Web app has been implemented using the Python Flask infrastructure which allows the user to interact with the system using a Web-based interface. The Web application translates user interactions into the following low level routines:

- **connect()**: This function implements the action of connecting the node to the system. The Web app parses the configurations from config.cfg file and invokes the job manager to initiate the node. The job manager in turn instantiates the network manager to perform the connection.
- **disconnect()**: The disconnect() action routine invokes the job manager and in turn the network manager to destroy the current instance and disconnect from the network. The Web application performs checks before invoking all routines to make sure that the node is connected to the network.
- **addjob()**: The addjob routine adds a new job as a consumer. On adding a job, the job manager performs all the steps related to job execution.
- **getNWStatus()**: This routine continuously pulls information from the network manager pertaining to any network status updates and displays this information in real-time.

The Web Application initializes two threads for continuously monitoring the job status and the network status. The Web application sets up synchronized queues with the job manager and the network manager. The job manager publishes job execution status updates to the queue while the network manager publishes network status updates to its queue. The Web application polls these queues, receives status updates and displays this information in real time to the user.

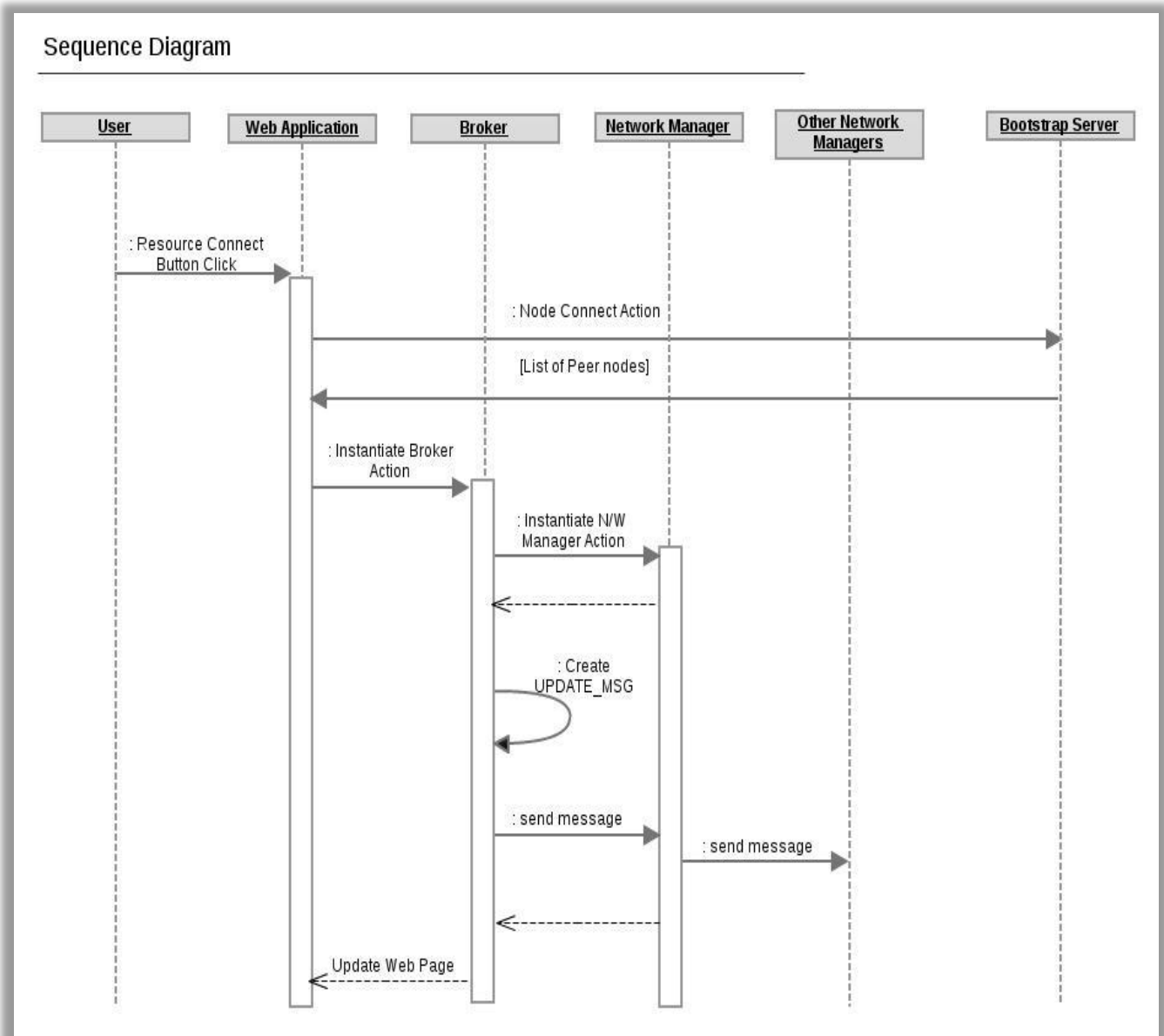
5.5 Cloud Integration

We have used the boto library to integrate Amazon S3 to our system. To reduce the load and overhead of transferring input files back and forth the nodes, users have the option of storing the input data on the cloud, and pulling the data as required from various nodes. The bucket identifiers and keys for performing the data pull need to be specified as part of the job itself.

6. Sequence Diagrams

6.1. Node Initialization

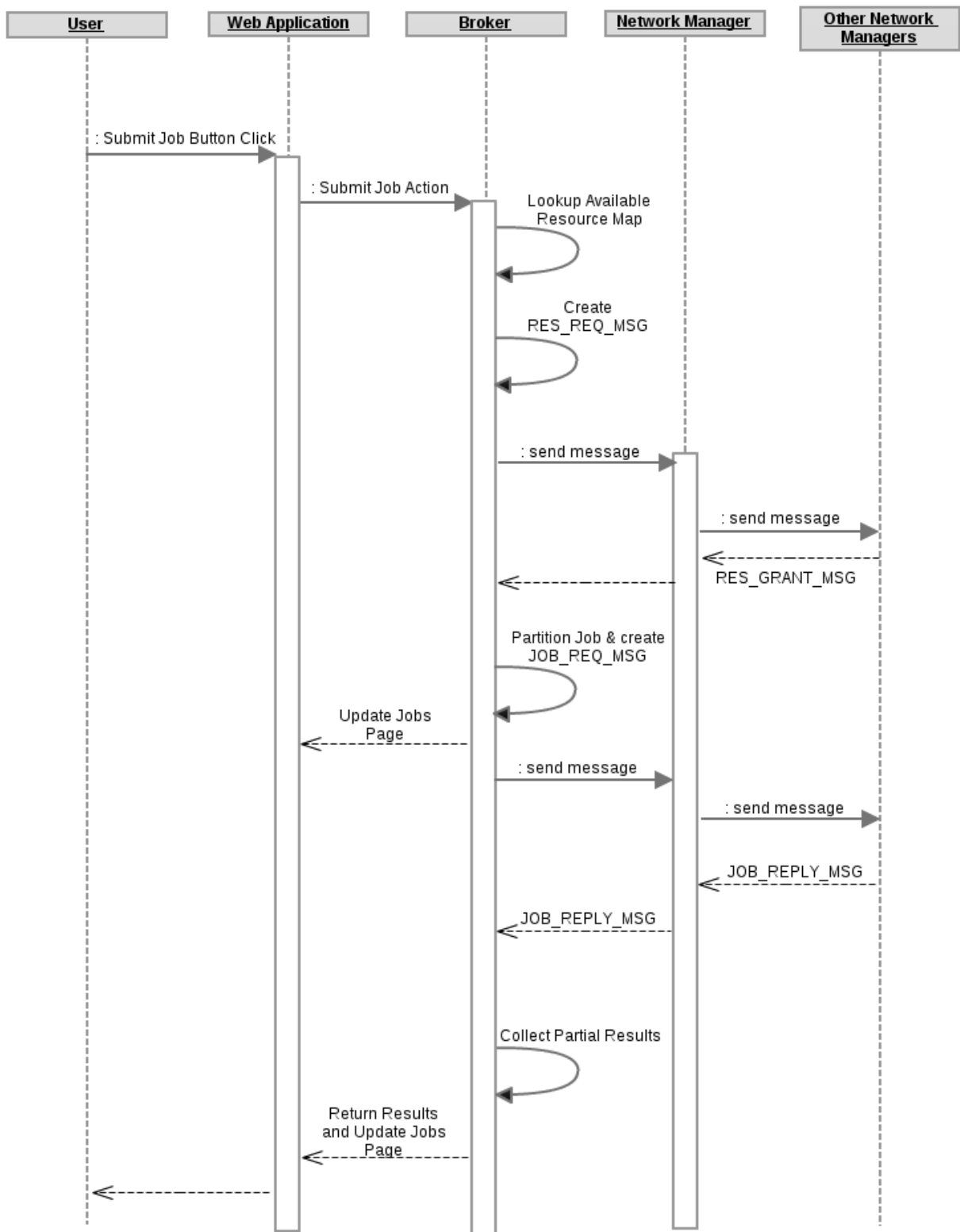
During the initialization phase, the node first initiates a connection with the boot-strap server and registers itself. The boot-strap server returns a list of peer nodes to the application. The web application then instantiates the Broker, which in turn initializes the Network Manager. The list of peer node is passed on from the Web Application to the Network Manager, which then initiates TCP connection with the peer nodes.



6.2. Submit Job

A user submits a new job into the system using the Web Application UI. The job details are passed on to the Broker from the web application and the Broker then does a search for any available resources in the distributed system. If the resources are available, the Broker tries contacting them and reserves them for this particular job. Once the resources are reserved, the Broker partitions the job and then ships the sub-jobs to the reserved resources. The Broker on the initiating node would also gather results generated from the sub jobs and on completion return the final result back to the user.

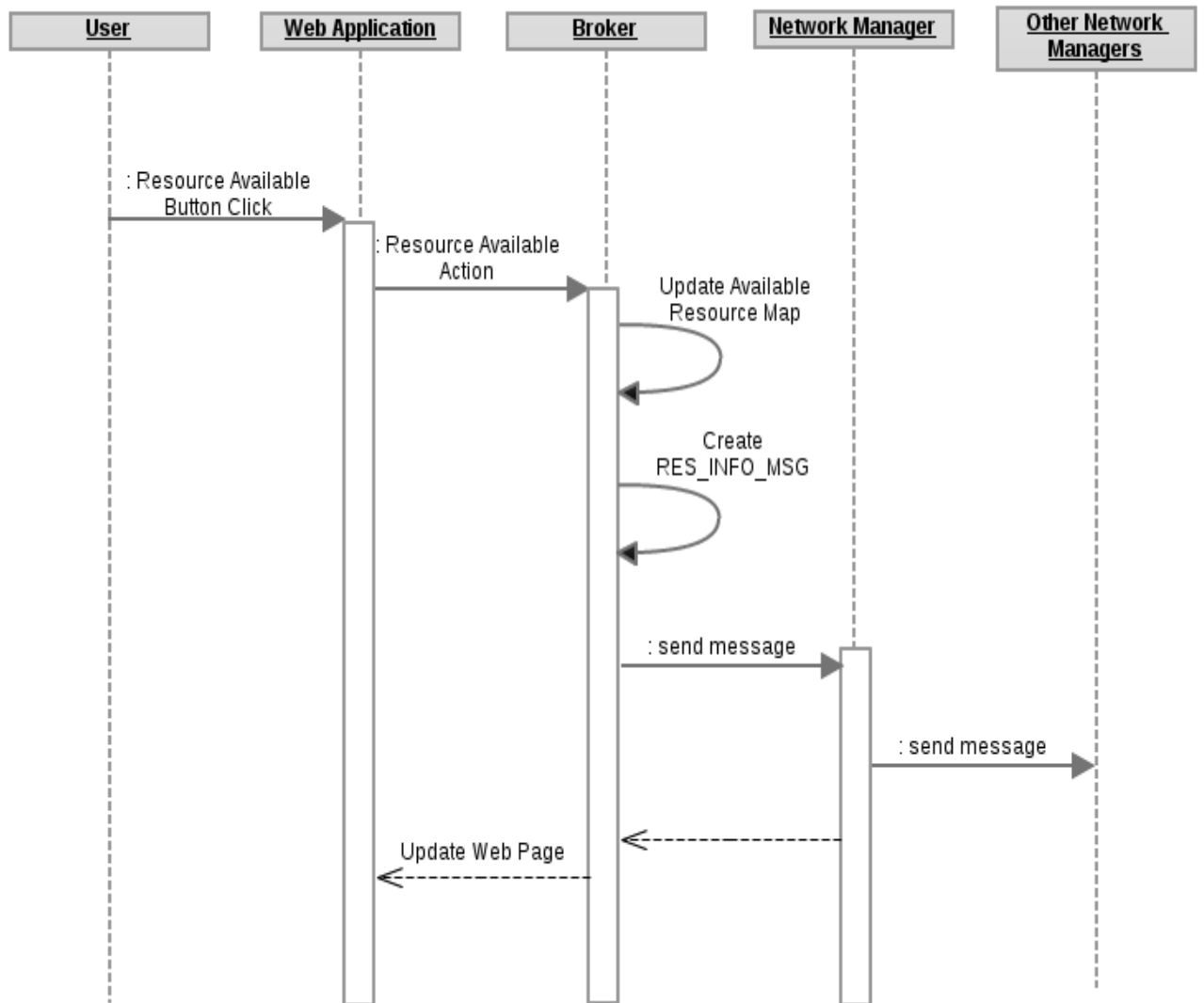
Sequence Diagram



6.3. Resource Available

A user can make his resource available to the market by clicking on the 'Submit' button on the 'Resource' page in the web application. The web application informs the Broker about the newly available resource and after updating its resource map the Broker sends a RES_INFO_MSG message to the other peer nodes. This RES_INFO_MSG is percolated to all the nodes on the distributed system by the Network Manager.

Sequence Diagram



6. Scalability Analysis

The system is designed to be highly scalable allowing a large number of nodes to connect as consumers or providers. The scalability analysis can be divided into the space and time complexity for using the services of this distributed system.

Space Complexity Analysis:

Each node in the system would maintain the following information:

- An approximation of the global network topology to perform routing/forwarding to remote nodes in the system. Since the node would use this information for any sort of communication (for messages such as Reserve Node, Send Job etc.), the forwarding table would be of the order of $O(n)$, where n is the total number of nodes in the system.
- Information about neighbors such as communication address/port etc. to exchange heartbeat messages and detect node failures. Since each node maintains a fixed number of neighbors through its lifetime m , the neighbor information structure would be of the order of $O(m)$

Thus, the total amount of state maintained per node would be $O(n) + O(m)$, which is linear with respect to the number of nodes in the system.

Time/Effort Complexity Analysis:

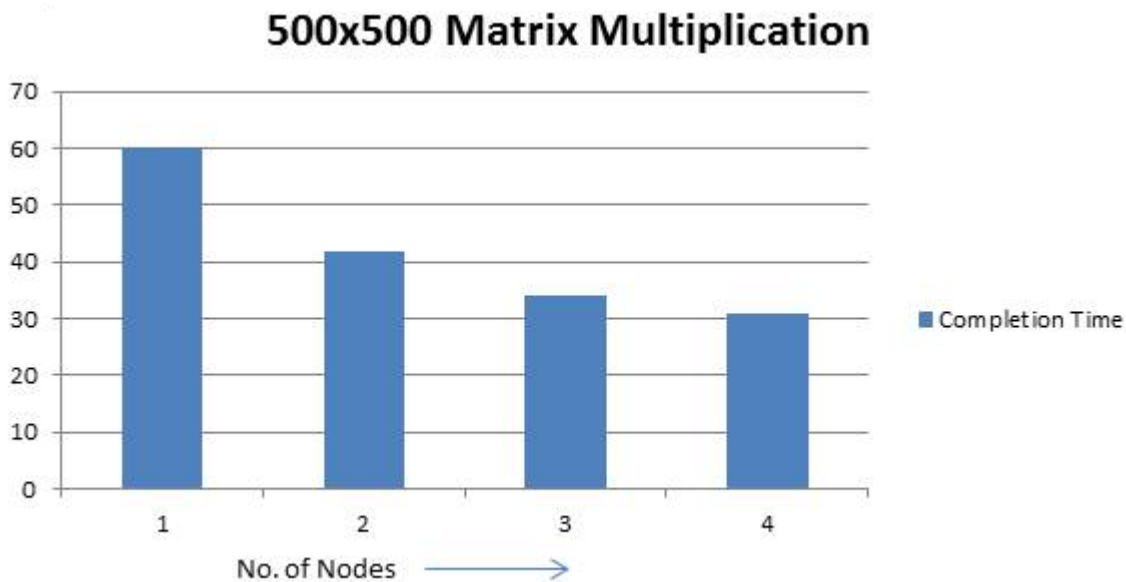
Various system operations would require messaging and distributing information from one node of the system to another. Since each node maintains m peers, the total number of hops to reach any node from some node would be $O(\log n / \log m)$. This analysis is done on the basis of the fact that at each hop, the node is connected to m peers and can choose the most optimal path to the destination by forwarding the message along one of m peers. Thus, all message passing and communication in the network would have logarithmic complexity leading to an efficient and scalable model.

7. Evaluation & Testing

We also evaluated and tested our project using different “consumer” jobs in order to verify the scalability as well as performance of our system. The 2 jobs we ran extensive evaluations on are

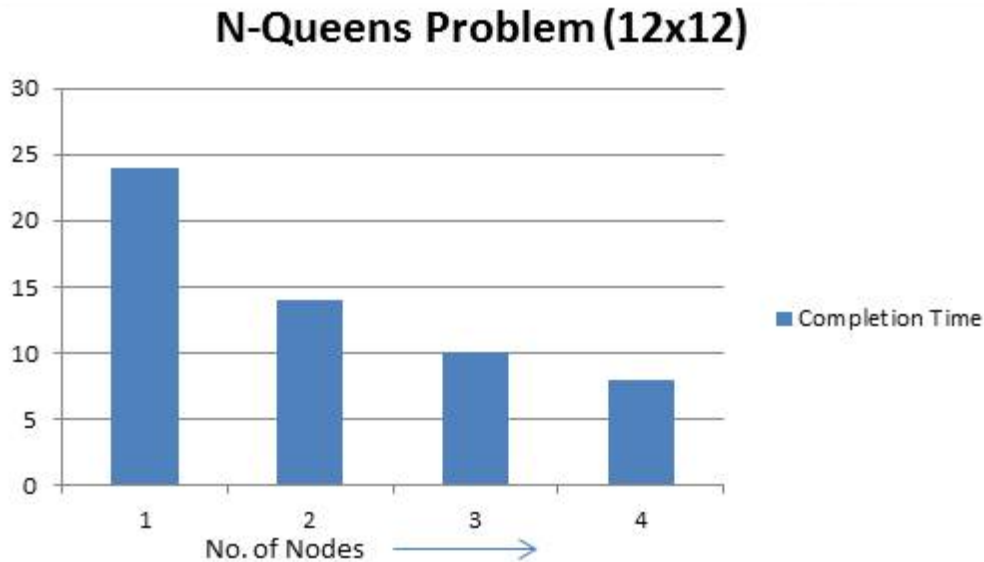
- 500 x 500 Matrix multiplication
- N Queens Algorithm

7.1 500 x 500 Matrix Multiplication



This program basically multiplies two 500 x 500 matrices. The above graph represents how the completion time of the program varies with the number of nodes in the system. On a single node the entire operation takes 60 seconds to complete. As the number of nodes on which the job is scheduled increases the completion time goes on reducing. So it takes 41 seconds on a system with 2 nodes, 35 seconds on a system with 3 nodes and 31 seconds on a node with 4 nodes. While the completion time reduces as the number of nodes increases, the increase is not linear. We believe this is due to mainly 2 factors. The first is the heterogeneity of nodes in the system and secondly as the number of nodes increase the network latency to transport the data and code start to play a significant role in overall system performance.

7.2 N Queens Problem



The above illustrates the graph produced for the N queens problem where $N = 12$. On a system with a single node computing all possible correct combinations require 25 seconds, on a system with 2 nodes the job requires 14 seconds, with 3 nodes it requires 10 seconds and on a system with 4 nodes it requires 7 seconds. This problem scales much better as compared to the 500 x 500 matrix multiplication because the N queens problem doesn't necessitate the transfer of large amounts of data. In the case of N queens no data needs to be send from the consumer to the producer whereas for matrix multiplication hundreds of rows of the matrix need to be sent over the network.

8. Project Management

8.1 Team Organization

Coordinating the schedule of all the team members involved in a group project is one of the most challenging tasks. As we are students having varied commitments for a day, the best possible way for us to communicate is through emails and chats. To facilitate parallel development, the project was divided into modules and each module was assigned a leader. The respective module leaders would be responsible for on time completion of their modules from start to end. We also decided to have a weekly group meeting where everyone gave their status updates and discussed any impending issues.

8.2 Task List

Task	Assigned
Infrastructure	All members
Configure python environment and required packages on development machines	
Setup Git repository for the project	
Web Application	Alok, Pratik
Flask Web Server initialization to render basic pages	
User Interface Design and Implementation with stubs for functionality	
Interfaces for sending/receiving information from lower layers (Broker etc.) through TCP Connections	
Bootstrap Server	Aditya, Kuhsal
Server to manage new nodes connecting to the network	
Basic load balancing algorithm for assigning peers to new nodes and maintain node info	

Web App-Bootstrap Integration	Alok, Pratik
Define and implement protocol for communication between node's web application and bootstrap server	
Broker/Job Manager	Kushal, Aditya
Design/Implement protocol to manage global available resource information	
Implement protocol to reserve resources in a mutually exclusive manner	
Implement Job Management routines to split jobs, transport files and manage job executions	
Network Manager	Kushal, Aditya
Implement protocol to send heartbeats and maintain connectivity with peers	
Design message formats and message handling for routing messages	
Implement algorithms to find best neighbor and optimal path to destination nodes	
Component Integration and Testing	Alok, Pratik

8.3 Project Schedule

Description	Timeline
Requirement Engineering	
Problem and project scope definition	Week 1
Design Phase	Week 2 - 4
High Level Architecture of the system	

Sandboxing Techniques	
Reading up on current implementation	
Preparing Design Doc	
Development Phase	Week 5 – 10
Implementing Web Application	
Implementing Bootstrap server	
Implementing Broker	
Implementing Network Manager	
System Integration and Testing	Week 11 – 14

9. References

1. [Lalis, Karipidis] An open market based architecture for Distributed Computing.
2. [Buyya, Vazhkudai] Compute power market: Towards a Market-Oriented Grid
3. Redbooks Paper: Introduction to Grid Computing