VISVESVARAYA TECHNOLOGICAL UNIVERSITY

Jnana Sangama, Belgaum-590018

A PROJECT PHASE II REPORT (**15CSP85**) ON

"COMPUTE SHARING PLATFORM"

Submitted in Partial fulfillment of the Requirements for the VIII Semester of the Degree of Bachelor of Engineering in Computer Science & Engineering

By

SHAILAV SHRESTHA(1CR16CS154)

RAJENDRA GUPTA (1CR16CS127)

FIROJ SIDDIKI (1CR16CS049)

Under the Guidance of,

DANTHULURI SUDHA

Associate Professor, Dept. of CSE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CMR INSTITUTE OF TECHNOLOGY

#132, AECS LAYOUT, IT PARK ROAD, KUNDALAHALLI, BANGALORE-560037

CMR INSTITUTE OF TECHNOLOGY

#132, AECS LAYOUT, IT PARK ROAD, KUNDALAHALLI,BANGALORE-560037

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

Certified that the project work entitled **"Compute Sharing Platform"** carried out by Mr.Shailav Shrestha,USN:1CR16CS154, Mr.Rajendra Gupta, USN: 1CR16CS127, Mr.Firoj Siddiki, USN: 1CR16CS049 , bonafide students of CMR Institute of Technology, in partial fulfillment for the award of **Bachelor of Engineering** in Computer Science and Engineering of the Visveswaraiah Technological University, Belgaum during the year 2019-2020. It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the Report deposited in the departmental library.

The project report has been approved as it satisfies the academic requirements in respect of Project work prescribed for the said Degree.

.

Signature of Guide	**Signature of HOD**
(Danthuluri Sudha)	**Dr. Prem Kumar Ramesh**
(Associate Professor)	**Professor & HoD**
Dept. of CSE, CMRIT	**Dept. of CSE, CMRIT**

DECLARATION

We, the students of 8th semester of Computer Science and Engineering, CMR Institute of Technology, Bangalore declare that the work entitled "COMPUTE SHARING PLATFORM" has been successfully completed under the guidance of Associate Professor Danthuluri Sudha, Computer Science and Engineering Department, CMR Institute of technology, Bangalore. This dissertation work is submitted in partial fulfillment of the requirements for the award of Degree of Bachelor of Engineering in Computer Science and Engineering during the academic year 2019 - 2020. Further the matter embodied in the project report has not been submitted previously by anybody for the award of any degree or diploma to any university.

Place:

Date:

Team members:

SHAILAV SHRESTHA (1CR16CS154) __________________

RAJENDRA GUPTA (1CR16CS127) __________________

FIROJ SIDDIKI (1CR16CS049) __________________

ABSTRACT

The rapid development in commodity computers and network technology introduces new possibilities to how we solve computing problems. With the advent of high performance utility computers and the internet, highly distributed and parallel workloads can be achieved at a very nominal cost. The project intends to provide a platform where such workloads could be distributed in a decentralized architecture across nodes in a network using the concepts of shared Computing and grid computing. The project facilitates sharing of compute and network resource over the internet, which, in a way democratizes computing resources while keeping costs at very minimum.

ACKNOWLEDGEMENT

I take this opportunity to express my sincere gratitude and respect to **CMR Institute of Technology, Bengaluru** for providing me a platform to pursue my studies and carry out my final year project.

I have a great pleasure in expressing my deep sense of gratitude to **Dr. Sanjay Jain,** Principal, CMRIT, Bangalore, for his constant encouragement.

I would like to thank **Dr. Prem Kumar Ramesh,** HOD, Department of Computer Science and Engineering, CMRIT, Bangalore, who has been a constant support and encouragement throughout the course of this project.

I consider it a privilege and honor to express my sincere gratitude to my guide **(Danthuluri Sudha), Associate Professor,** Department of Computer Science and Engineering, for the valuable guidance throughout the tenure of this review.

I also extend my thanks to all the faculty of Computer Science and Engineering who directly or indirectly encouraged me.

Finally, I would like to thank my parents and friends for all their moral support they have given me during the completion of this work.

TABLE OF CONTENTS

	Page No.
Certificate	ii
Declaration	iii
Abstract	iv
Acknowledgement	v
Table of contents	vi
List of Figures	viii
1 INTRODUCTION 1.1 Relevance of the Project 1.1.1 Scientific Computing 1.1.2 Optimal Resource Usage 1.2 Scope of the Project 1.2.1 Exploiting Underutilized Resources 1.2.2 Parallel CPU Capacity 1.2.3 Virtual Resources and Virtual Organization for Collaboration 1.2.4 Access to Additional Resources 1.2.5 Resources Balancing	1
2 OBJECTIVES & METHODOLOGY 2.1 Objective 2.2 Methodology 2.2.1 Agile Methodology	9
3 LITERATURE SURVEY 3.1 Overview 3.2 Related Works	16
4 REQUIREMENTS SPECIFICATION 4.1 Functional Requirements 4.2 Non-Functional Requirements	20

4.3 Hardware Requirements **4.4 Language Requirements** **4.4.1 Python** **4.4.2 C Programming** **4.5 Packages Required** **4.5.1 Docker**	
5 PROBLEM FORMULATION **5.1 Core Problem Statement** **5.2 Socket Programming** **5.3 Multi Client-Server Architecture** **5.4 Fully Decentralized P2P Architecture** **5.5 Additional Problem Statements**	**25**
6 STATUS AND ROADMAP **6.1 Network Phase** **6.2 Application Phase** **6.3 Abstraction Phase** **6.4 UX Phase**	**37**
7 CONCLUSION AND FUTURE SCOPE	**40**
REFERENCES	**41**

LIST OF FIGURES

	Page No.
Fig 1.1 Convex optimization in numerical linear algebra	2
Fig 1.2 Quantification of Uncertainty	3
Fig 1.3 Exploiting Underutilized Resources	4
Fig 1.4 Access to Additional Resources	7
Fig 2.1 Computing Management and Sharing	9
Fig 2.2 Different Layers of Protocols	10
Fig 2.3 Agile Methodology	12
Fig: 2.4 General Architecture	13
Fig 2.5 Agent Architecture	14
Fig 2.6 Task server Architecture	15
Fig 5.1 Structure of Servers and Kernels	25
Fig 5.2 Steps in Socket Connection	26
Fig 5.3 Client Server Connection	30
Fig 5.4 Fully Decentralized P2P Architecture	35
Fig 6.1 Different Network Phases	37
Fig 6.2 Management of Various Protocols	38
Fig 6.3 Example Of Interface	38
Fig 6.4 Front-end layer illustrating UI	39

CHAPTER 1

INTRODUCTION

Computer development has had an exponential growth in the last decade. While modern commodity computer systems can solve a vast majority of the tasks, there are limitations to the systems. The fundamental problem emerges from the fact that a single high performance computer or a computer cluster cannot solve large quantities of data and problems independently. Hence, in the domain of high performance and data intensive computing, modern computers cannot cope up independently.

To solve highly compute intensive problems, using computers independently is not appropriate. However, this limitations can be overcome by changing the approach to these kind of problems. And the answer is to employ a multitude of machines in a highly parallel architecture to overcome the limitations of a single machine. The idea is to integrate a large number of independent computing systems in a highly distributed environment using high speed interconnection network to connect distributed, heterogeneous and high performance computers or computer cluster.

Our goal is to provide a decentralized platform which facilitates such an architecture. The core intent of the platform is to share computer resources with other people or machine using a high level interface without having to deal with implementation of the underlying infrastructure.

1.1 Relevance of the Project

This project has the potential to change the way developers interact with other systems in a distributed cloud environment. Today, the domain of cloud is dominated by few big players. They have large pools of compute, network and storage resources which they provide as service to the users through the internet. The users have to pay some amount for using the service. What our platform does is provide an alternative for a similar use case. We provide a platform where instead of using pools of resources of some cloud providers, we give them the ability to use the resources of machines which are already available to them and which they can access without much charges unlike cloud. While the platform does not obviate the need for these cloud providers, for certain uses, the platform becomes very attractive such as :

1. 1.1 Scientific Computing

The domain of Science such as Chemistry, Physics and Biology deals with massive amount of data. This is because they deals with billions of data points with very large dimensions. Due to these large volumes of data, the amount of calculations is huge. However a lot of similar combinations are involved. These conditions make it ideal for highly parallel computing.

While online cloud providers do provide the necessary infrastructure to leverage these highly parallel requirements, they are also quite costly. But what if we had access to a pool of machines that were available for no cost? Our platform gives the ability to leverage all of those machines and scale the computations horizontally with the number of machines. This is especially ideal if the machines were already available and under utilized.

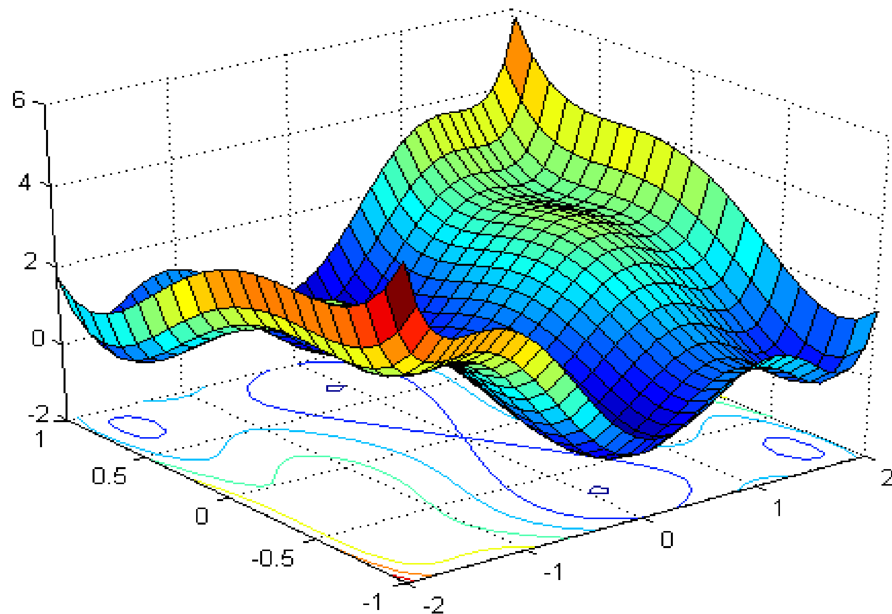


Fig: 1.1 Convex optimization in numerical linear algebra

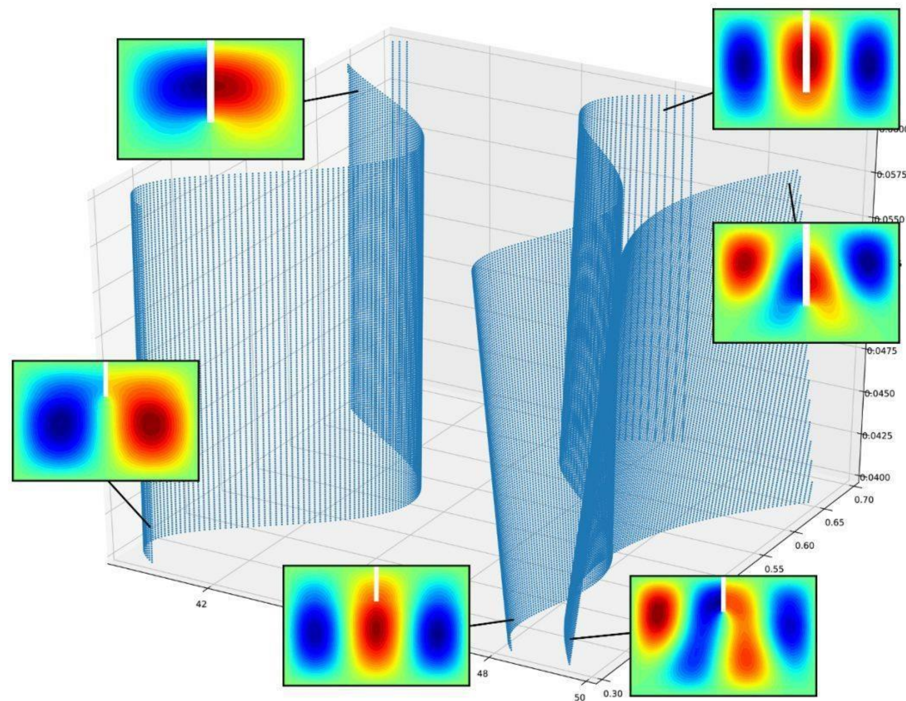


Fig: 1.2 Quantification of Uncertainty

The above Figures(Fig 1.1 and Fig 1.2) shows some NP hard polynomial time computations which can be computed in parallel for faster simulations

1.1.2 Optimal Resource Usage

Many business already have a lot of systems locally available to them or available to them in some other branch. It does not make sense for a business to pay costly fees to the cloud service providers and have the already available systems under utilized, especially for compute intensive tasks. With our platform such resources could be utilized as long as they are connected to the internet. This not only saves the business money, but also provides security to them.

1.2 Scope of the Project

1.2.1 Exploiting Underutilized Resources

Most applicable use of our platform is to run an existing application on a different machine. The machine on which the application is normally run might be unusually busy due to an unusual peak in activity. The job in question could be run on an idle machine elsewhere on the grid. There are at least two prerequisites for this

scenario. First, the application must be executable remotely and without undue overhead. Second, the remote machine must meet any special hardware, software, or resource requirements imposed by the application. In most organizations, there are large amounts of underutilized computing resources. Most desktop machines are busy less than 5 percent of the time. In some organizations, even the server machines can often be relatively idle. We provide a framework for exploiting these underutilized resources and thus has the possibility of substantially increasing the efficiency of resource usage. Another function is to better balance resource utilization. An organization may have occasional unexpected peaks of activity that demand more resources. If the applications are grid-enabled, they can be moved to underutilized machines during such peaks. In fact, some grid implementations can migrate partially completed jobs. In general, a grid can provide a consistent way to balance the loads on a wider federation of resources. This applies to CPU, storage, and many other kinds of resources that may be available on a grid. Management can use a grid to better view the usage patterns in the larger organization, permitting better planning when upgrading systems, increasing capacity, or retiring computing resources no longer needed.

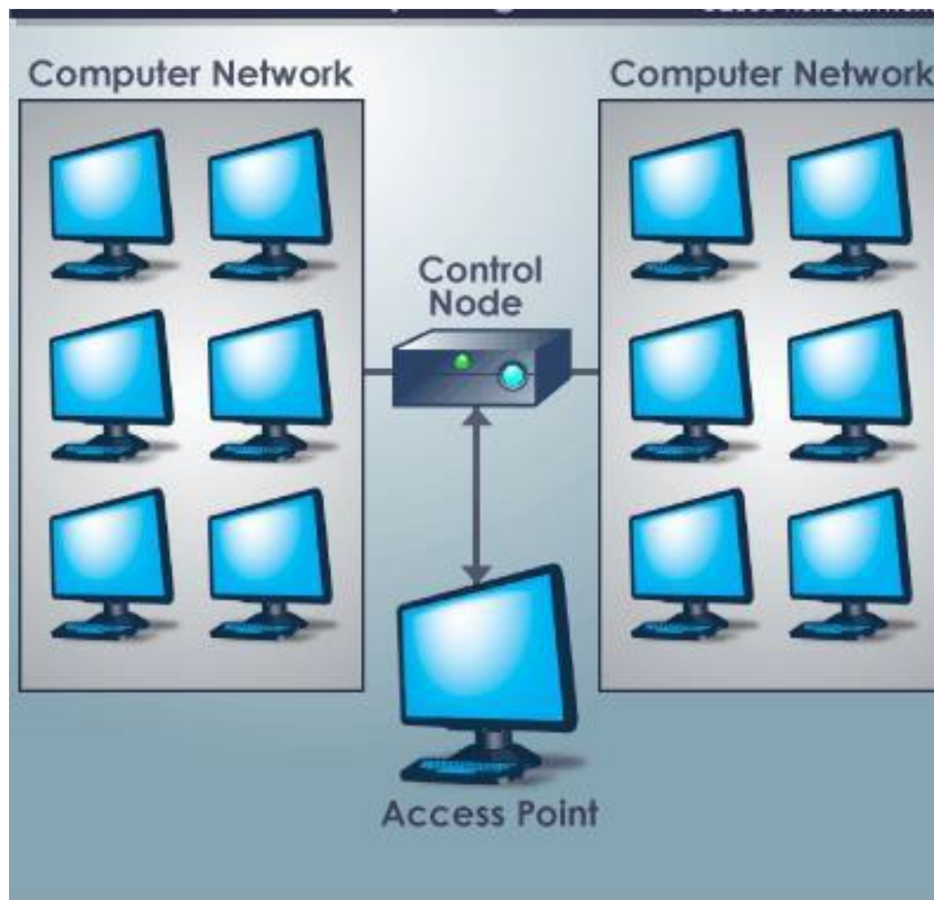


Fig: 1.3 Exploiting Underutilized Resources

1.2.2 Parallel CPU Capacity

The potential for massive parallel CPU capacity is one of the most attractive features of grid computing. In addition to pure scientific needs, such computing power is driving a new evolution in industries such as the bio-medical field, financial modelling, oil exploration, motion picture animation, and many others. The common attribute among such uses is that the applications have been written to use algorithms that can be partitioned into independently running parts. A CPU intensive grid application can be thought of as many smaller "sub-jobs," each executing on a different machine in the grid. To the extent that these sub-jobs do not need to communicate with each other, the more "scalable" the application becomes. A perfectly scalable application will, for example, finish 10 times faster if it uses 10 times the number of processors. Barriers often exist to perfect scalability. The first barrier depends on the algorithms used for splitting the application among many CPUs. If the algorithm can only be split into a limited number of independently running parts, then that forms a scalability barrier. The second barrier appears if the parts are not completely independent; this can cause contention, which can limit scalability. For example, if all of the sub-jobs need to read and write from one common file or database, the access limits of that file or database will become the limiting factor in the application's scalability. Other sources of inter-job contention in a parallel grid application include message communications latencies among the jobs, network communication capacities, synchronization protocols, input-output bandwidth to devices and storage devices, and latencies interfering with real-time requirements.

1.2.3 Virtual Resource and Virtual Organization for Collaboration

Another aim for the platform is to enable and simplify collaboration among a wider audience. In the past, distributed computing promised this collaboration and achieved it to some extent. Grid computing, what our framework is based on, takes these capabilities to an even wider audience, while offering important standards that enable very heterogeneous systems to work together to form the image of a large virtual computing system offering a variety of virtual resources. The users of the grid can be organized dynamically into a number of virtual organizations, each with different policy requirements. These virtual organizations can share their resources collectively as a larger grid. Sharing starts with data in the form of files or databases. A "data

grid" can expand data capabilities in several ways. First, files or databases can seamlessly span many systems and thus have larger capacities than on any single system. Such spanning can improve data transfer rates through the use of striping techniques. Data can be duplicated throughout the grid to serve as a backup and can be hosted on or near the machines most likely to need the data, in conjunction with advanced scheduling techniques. Sharing is not limited to files, but also includes many other resources, such as equipment, software, services, licenses, and others. These resources are "virtualized" to give them a more uniform interoperability among heterogeneous grid participants.

1.2.4 Access to Additional Resources

An addition to CPU and storage resources, a grid can provide access to increased quantities of other resources and to special equipment, software, licenses, and other services. The additional resources can be provided in additional numbers and/or capacity. For example, if a user needs to increase his total bandwidth to the Internet to implement a data mining search engine, the work can be split among grid machines that have independent connections to the Internet. In this way, the total searching capability is multiplied, since each machine has a separate connection to the Internet. If the machines had shared the connection to the Internet, there would not have been an effective increase in bandwidth. Some machines may have expensive licensed software installed that the user requires. His jobs can be sent to such machines more fully exploiting the software licenses. Some machines on the grid may have special devices. Most of us have used remote printers, perhaps with advanced colour capabilities or faster speeds.In this way, the total searching capability is multiplied, since each machine has a separate connection to the Internet. If the machines had shared the connection to the Internet, there would not have been an effective increase in bandwidth. Some machines may have expensive licensed software installed that the user requires. His jobs can be sent to such machines more fully exploiting the software licenses.. For example, if a user needs to increase his total bandwidth to the Internet to implement a data mining search engine, the work can be split among grid machines that have independent connections to the Internet.

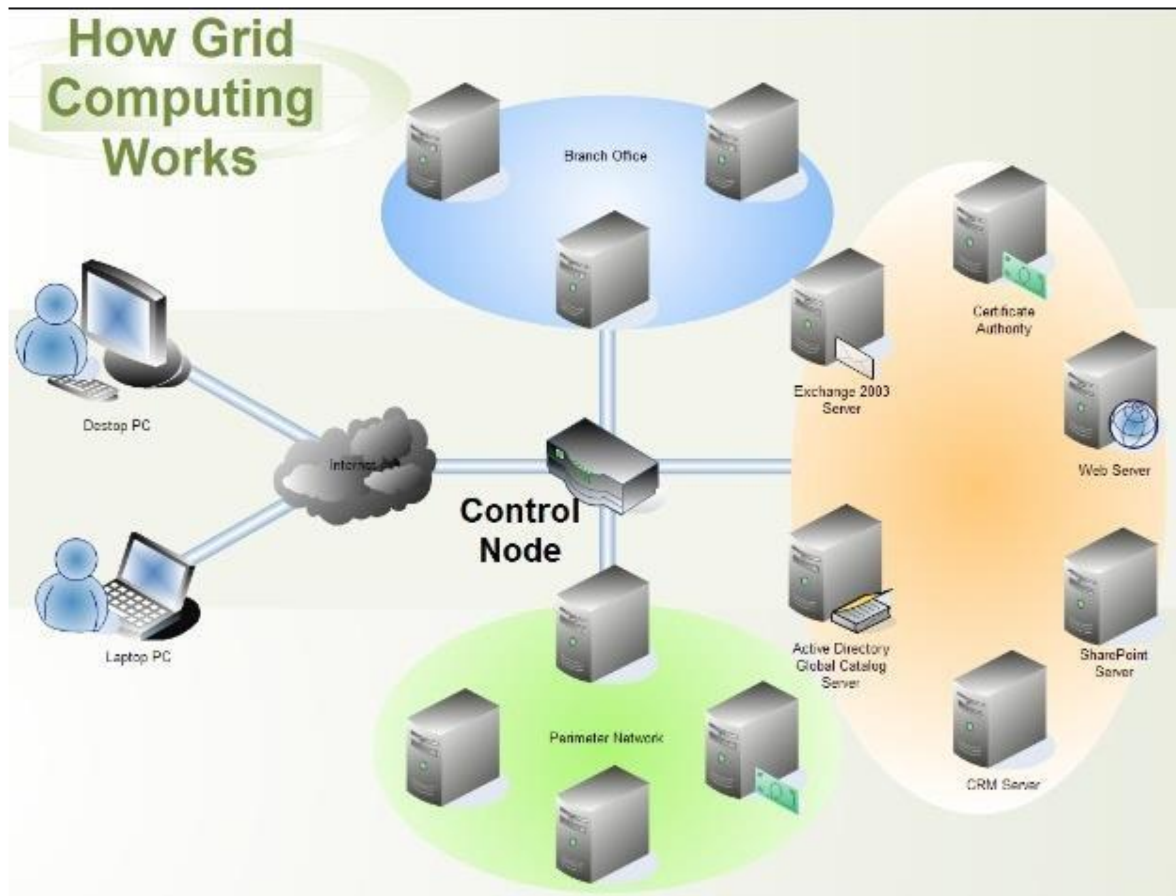


Fig: 1.4 Access to Additional Resources

1.2.5 Resource Balancing

A grid federates a large number of resources contributed by individual machines into a greater total virtual resource. For applications that are grid-enabled, the grid can offer a resource balancing effect by scheduling grid jobs on machines with low utilization. This feature can prove invaluable for handling occasional peak loads of activity in parts of a larger organization. This can happen in two ways:

An unexpected peak can be routed to relatively idle machines in the grid.

If the grid is already fully utilized, the lowest priority work being performed on the grid can be temporarily suspended or even canceled and performed again later to make room for the higher priority work.

Without a grid infrastructure, such balancing decisions are difficult to prioritize and execute. Occasionally, a project may suddenly rise in importance with a specific deadline. A grid cannot perform a miracle and achieve a deadline when it is already too close. However, if the size of the job is known, if it is a kind of job that can be sufficiently split into sub-jobs, and if enough resources are available after pre-

empting lower priority work, a grid can bring a very large amount of processing power to solve the problem. In such situations, a grid can, with some planning, succeed in meeting a surprise deadline.

CHAPTER 2

OBJECTIVES AND METHODOLOGY

2.1 Objectives

The objective of the project is to create a functional application interface that can be use to leverage other system’s computer resources or share their own. This paradigm aims to shift focus from centralized resource distributors to open peer to peer resource sharing, making better use of the world’s resources to solve important problems in the real world.

2.2 Methodology

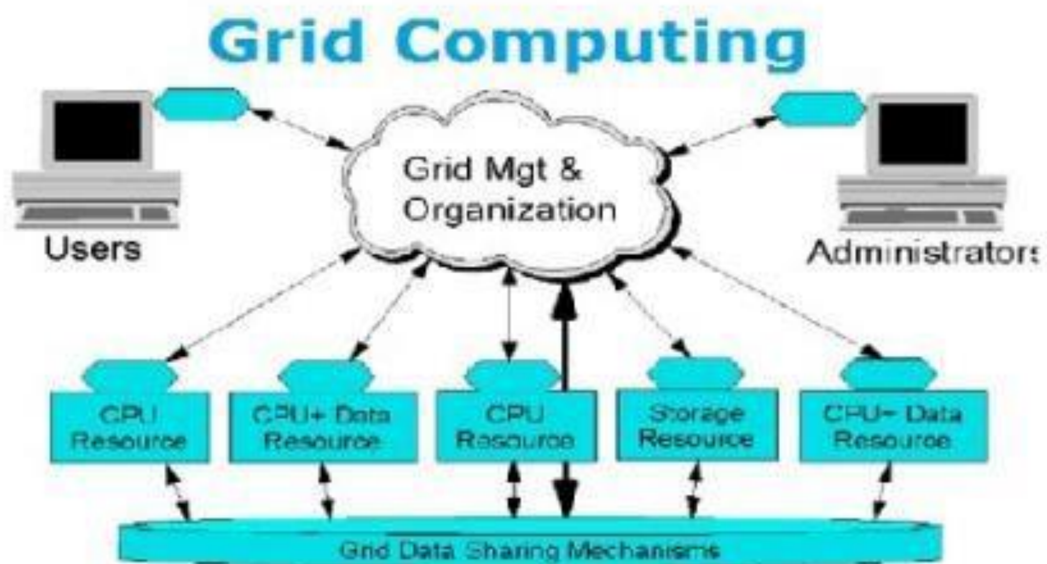


Fig: 2.1 Computing Management and Sharing

Grids came in the mid-90s to perform large-scale computation problems using a chain of resource-sharing commodity mechanism that distribute the computation power reasonably using the help of supercomputers and huge firm clusters at that time.The superior motive was that these high act computing resources were posh and hard to get access to, so during the ii initial phase it was to use organize resources that could comprise compute, storage and network related resources from several distributed organizations, and such resources are generally effective. Grids concentrated on

uniting left-over resources with their hardware, local resource management, operating systems and security infrastructure. In order to sustain the formation of so called "Virtual Organization"- a logical set-up within which disperse resources can be spot and distribute as if they were from the same organization. The primary concerns of the Grid infrastructure are the security and interoperability as materials may come across different administrative domains, which carries both global and local resource usage policies, having different platforms and hardware and software configurations, and vary in capacity and availability.. Grids come-up with the protocols and services at five different layers as recalled in the resource (whether physical or logical) as a outcome of distributive operations at admiring levels.

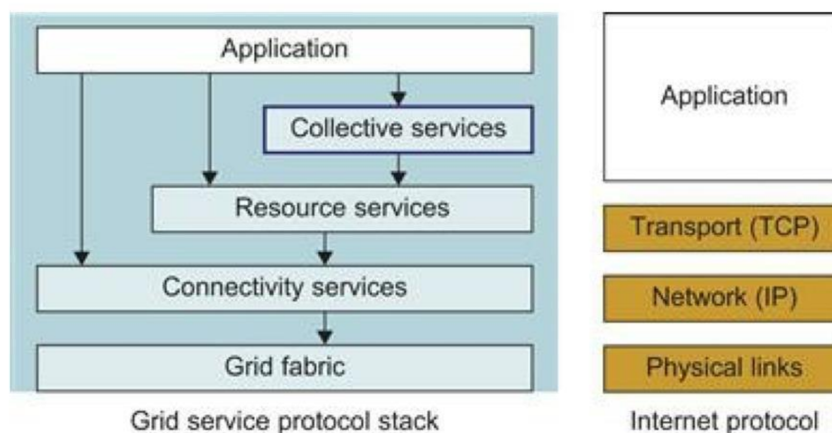


Fig: 2.2 Different Layers of Protocols

- Fabric layer (Interfaces to local control): The Grid Fabric Layer delivers the facility to which shared access is arbitrate by Grid protocols. Such as computational resources, storage systems, catalogues, network resources, and sensors.

- Connectivity Layer(Easy and Secured Communications):The Connectivity layer is responsible for core communication and authentication protocols as per the secure and easy network transactions which is Grid-specific network. Through this communication protocols the exchange of Data between Fabric Layers resource is enabled. Protocols like Authentication is Build on the communication services which provides cryptographically secure mechanisms by which it verifies the identity of different resources and users. Implementation of Grid Security Infrastructure into the network make it effectively secure and easy as it includes the technique like authorization, uniform authentication and message

protection mechanism in multi-institutional setting. It allows single sign-on delegation, mechanism such as identity mapping.

- Resources layer (Sharing Single Resources):The protocols for secure and easy initiation, monitoring, accounting, negotiation, control and payment of sharing operations on individual resources is build by Resource layer. Fabric layer function called as the resource layer implementation of these protocols to access and control local resources. The GRAM (Grid Resource Access and Management) protocol is used for allocation of computational resources and for monitoring and control of computation on those resources, and GridFTP for data access and high-speed data transfer.

- Collective layer (Coordinating Multiple Resources):Collective layer contains protocols and services (and APIs and SDKs) that are not committed with any one specific resource but rather are global in nature and capture interactions across collections of resources.

- Application layer: The final most layer in our Grid Architecture comprises with the user applications built on top of the above protocols and APIs and operate in VO environments.

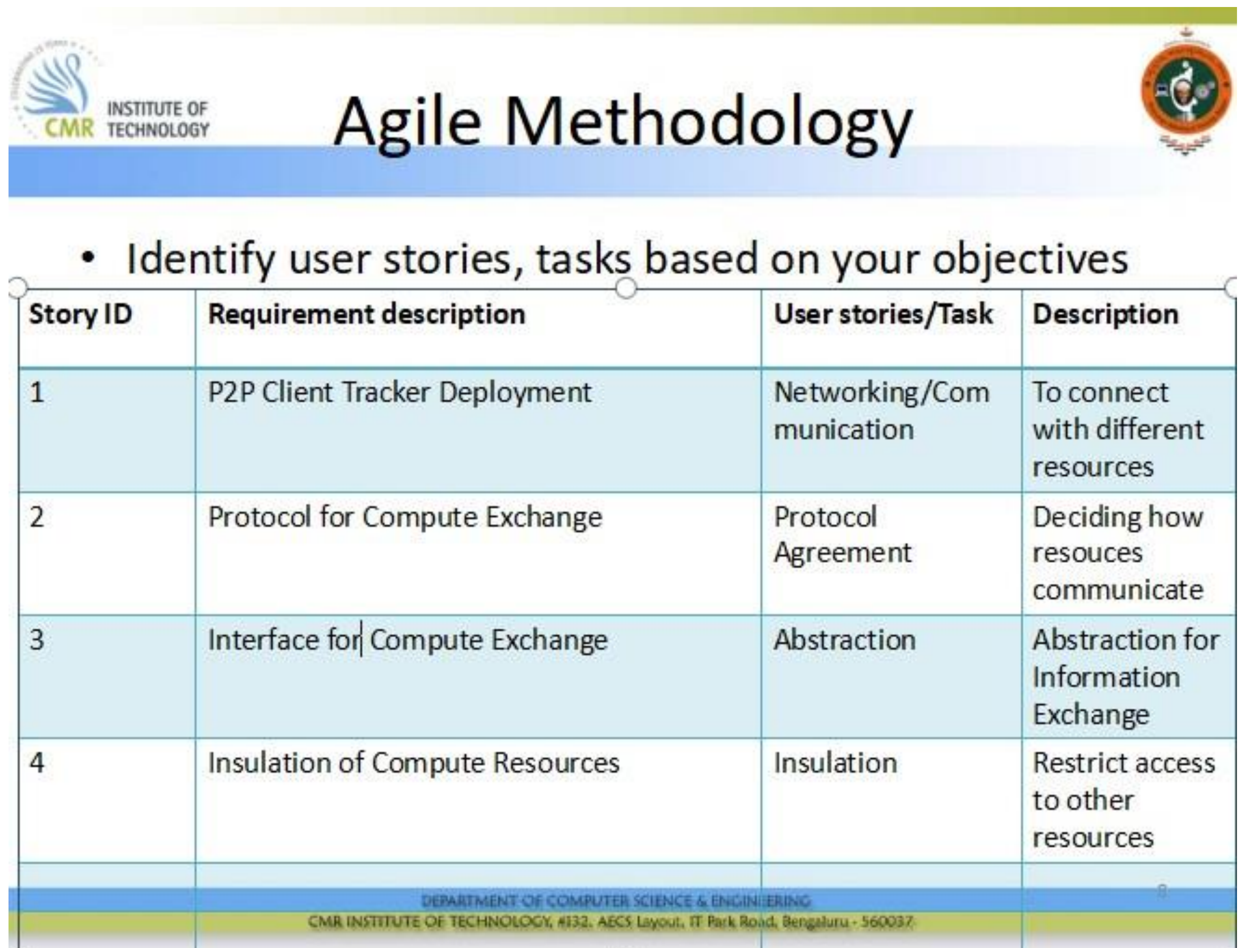
At every layer, well-enterpret protocols are there that lend access to the usefull services alike resource management, data access, resource discovery, etc.

2.2.1 Agile Methodology

We use agile methodology to implement our system. It is a type of project management process, mainly used for software development, where demands and solutions evolve through the collaborative effort of self-organizing and cross functional terms. Thus, we perform the process in steps as described below.

- P2P Client tracker deployment for various Networking and communication protocols. So that the users will be able to access different resources.
- Protocol for Compute Exchange, the users need to agree the proposed protocol that is required for deciding how the selected resources will be communicating and networking .
- Interface for Compute Exchange with the help of providing abstraction that will be further used as the abstraction for information exchange.

- Insulation of the compute resources so that the client connected to another compute resource cannot access other resources or any extra resouces then that of what is permitted, this will be done following some security protocols.



Story ID	Requirement description	User stories/Task	Description
1	P2P Client Tracker Deployment	Networking/Com munication	To connect with different resources
2	Protocol for Compute Exchange	Protocol Agreement	Deciding how resouces communicate
3	Interface for Compute Exchange	Abstraction	Abstraction for Information Exchange
4	Insulation of Compute Resources	Insulation	Restrict access to other resources

Fig: 2.3 Agile Methodology

2.3 Architectures

2.3.1 General Architectures

Components of General Architecture

WeCompute Agent : The most important component of the Framework It iis installed on the client system and deals with communication with other clients as well as the docker daemon on the client system.

Tracker/ TaskServer : The Tracker keeps track of the clients who are available for sharing compute resources as well as provides clients to send/ receive task requests. This is particularly useful when the clients are behind a NAT and cannot be reached

directly by other client without Hole Punching.

Docker Client : The client system is expected to have docker client installed which deals with the containerization of the tasks process. This Container technology is critical because out platform must be independent of technology stack used which is only achievable with virtualized environments

Registry : We need an efficient mechanism for exchanging build information across clients and processes. Thus we use a container registry for this application whichmaybeprivateorpublic.

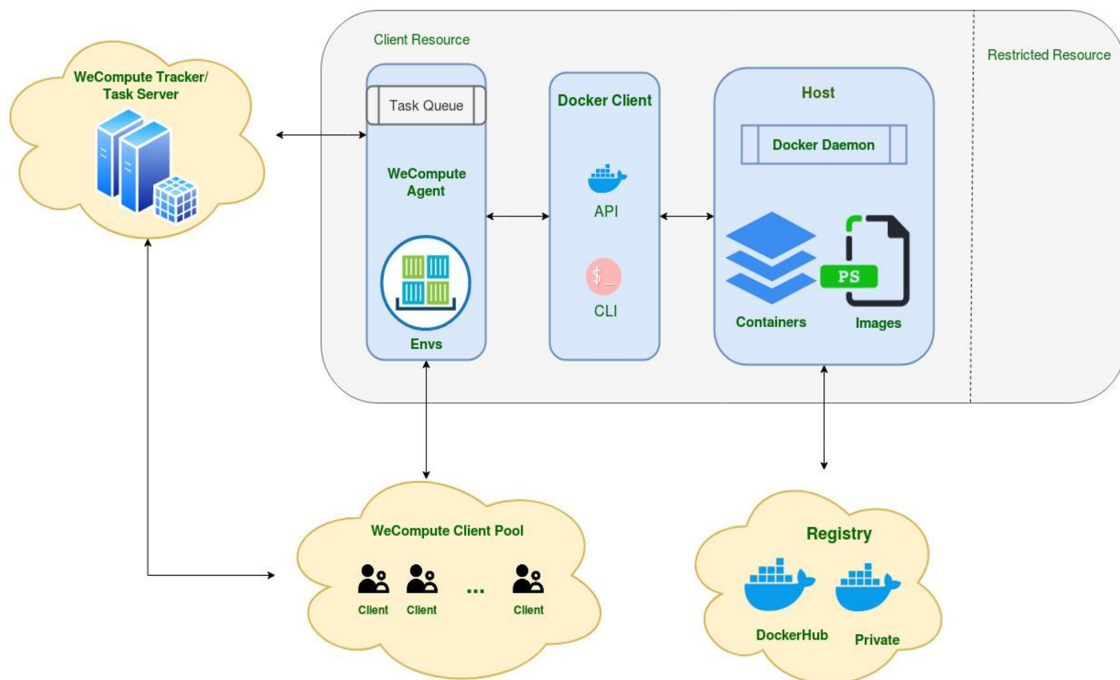


Fig 2.4 General Architectures

2.3.2 Agent Architecture

- The Agent is The central part of the entire compute sharing framework.
- Generally, any stakeholder interested in sharing their compute will simply install the Agent in their system.
- After that the entire task of synchronizing compute requests, processing Compute Requests, Responding with appropriate response to clients who requested for compute is handled by the agent.
- After installing the agent, the agent simply needs to be initiated by the interested stakeholder so that others can request compute.

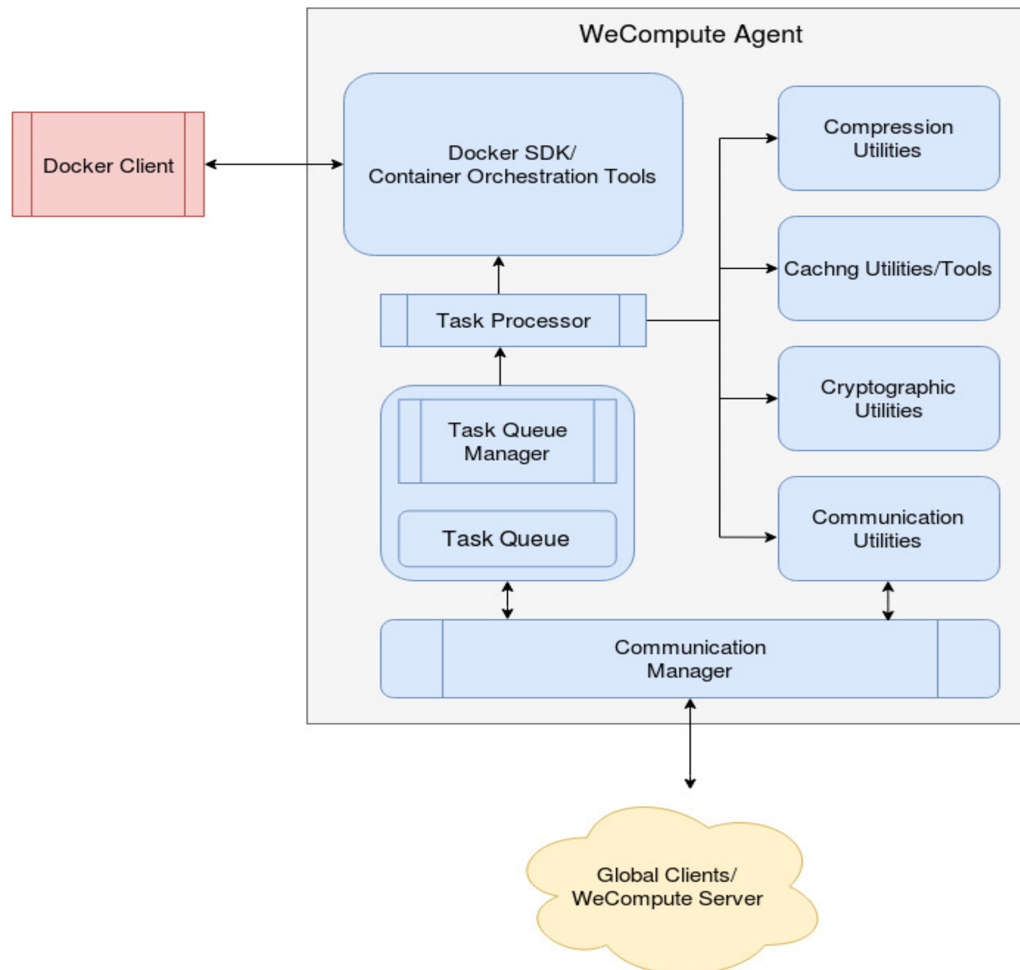


Fig 2.5 Agent Architecture

2.3.3 TaskServer Architecture

- **Rest Server** : The Tracker is a completely rest endpoint with a redis server withholding state information. Hence it can be independently scaled.
- **Redis Server** : All of the state information is held in a redis in-memory store. The reason for this is highly efficient access and query mechanisms. Further, it allows the state storage capability to be independently scaled as compared to scaling the entire server
- **Service Brokers** : While the WeCompute Framework allows compute to be shared effectively, the client still needs to host their application files so that the agents can access it independently. Service Brokers are independent services which provide us with these facilities.

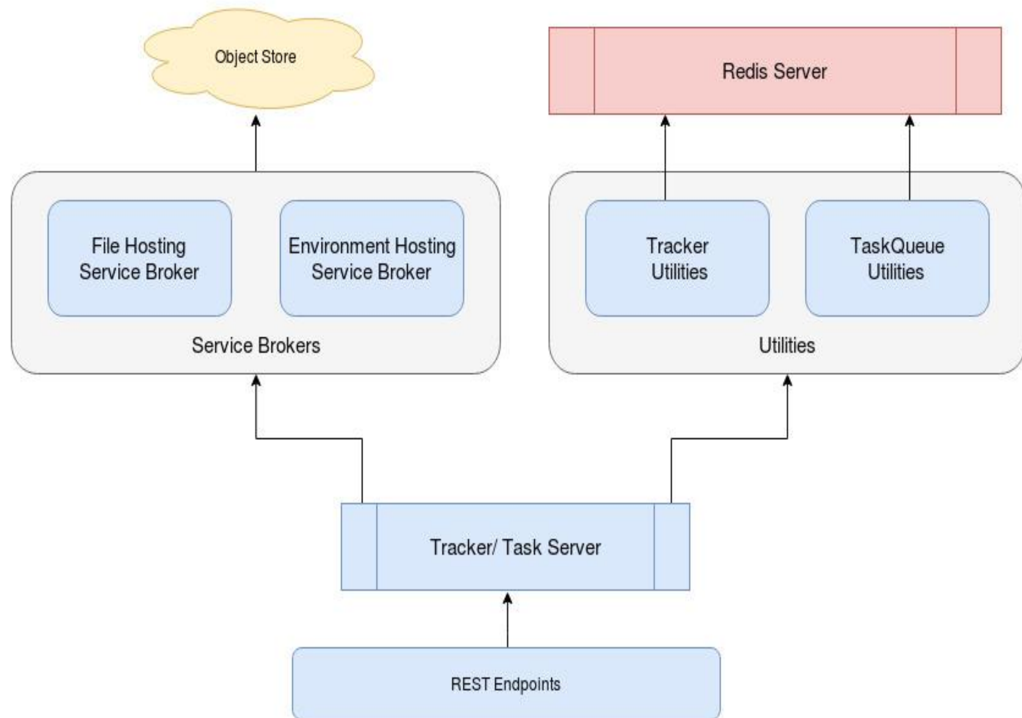


Fig 2.6 TaskServer Architecture

CHAPTER 3

LITERATURE SURVEY

3.1 Overview

Both grid and cloud are used to organize large scale calculations and data processing on remote computers. Grid which became a basic computing infrastructure for the Large Hadron Collider experiments provides unified technical solutions for sharing and merging distributed heterogeneous computing resources within big collaboration groups. Cloud became popular among data centers and computing service providers because of flexibility, manageability and efficient hardware utilization. Both approaches share common ideology “computing as a service”, so one can expect additional benefits from their integration. The paper describes our approach to the integration. We propose to use cloud within grid sites for acceleration of application deployment and easy support of multiple virtual organizations by grid sites. The cloud in grid approach has been implemented and tested in Ukrainian National Grid, a part of European Grid Infrastructure.

3.2 Related Works

Many list scheduling heuristics have been developed for scheduling workflows in Grids. However, these list heuristics are proposed for centralized Grid environment, whereas the proposed distributed list heuristic is applicable to decentralized Grids. K. Liu et al.proposed a Min-Min average algorithm for scheduling workflows in decentralized Gridenvironment, SwinDeW-G. However, the peer-to-peer(P2P) communication in SwinDeW-G is implemented by JXTA protocol, which uses a broadcast technique.In this work, we use a DHT based P2P system for handling resource discovery and scheduling coordination.

[i]. TECHNOLOGIES FOR DEVICE SHARING

This section presents prior efforts related to device sharing in a society in which devices are shared actively. In most of these device-sharing services, the authorized level of resource usage can be controlled. However, it is difficult to determine the appropriate authorized level of resource usage for each user according to device owners’ demand. FON is one of the most widely used communities of global WiFi sharing [4], [24]. FON provides a platform for members of the community to share

their spare bandwidth with other members. Those who join the FON membership are known as Foneros. A Fonero buys a local FON wireless router and shares their spare bandwidth with other Foneros. In return, a Fonero has free access to the FON's WiFi network, which consists of over 20 million hotspots worldwide, and enjoys wireless Internet connection. A cloudlet is a small-scale cloud datacenter that is located on the edge of the Internet and offers resources for mobile cloud computing [5]. Mobile devices have only limited computational resources, such as power, memory, storage, and energy, compared to static devices. To help these resourcepoor mobile devices save computational resources, a cloudlet server is connected to the mobile devices through various short-range radio communication technologies. A cloudlet offers mobile cloud computing, which offloads computational tasks of mobile devices with low latency. They proposed a service-oriented mobile cloud for sharing heterogeneous resources such as CPUs, bandwidth, and content.

They suggested that service-oriented heterogeneous resource sharing achieves low latency and high energy efficiency in a mobile cloud environment. Sensor sharing in WSNs is also a common example of device sharing. Microsoft developed an infrastructure for shared sensing called SenseWeb. By sharing sensors that were originally used for a specific application and placing those sensors into a single development system, SenseWeb enables production of new types of media and sensing applications over existing data networks. Airborne sensors are also shared. Since airborne sensors are typically idle for much of their flight time, efficient sensing can be achieved by sharing airborne sensors and allowing other information consumers to opportunistically use them during their otherwise idle time. Sensors are also shared to exchange energy. A system called eShare enables networked sensor systems to robustly extend their lifetime by exchanging energy with shared sensors. Some systems that share peripheral input/output (IO) devices through a network have been proposed. A peripheral bus extension called universal serial bus/internet protocol uses a virtual peripheral bus driver that enables users to share various devices over an IP network. A USB cross-platform extension has also been developed to share peripherals in a heterogeneous environment via a transmission control protocol/internet protocol network. A system called CameraCast provides a logical device application programming interface (API) that enables an application to gain system-level access to a remote video-sensor device. Composable IO is a resource-

sharing technology that enables IO peripherals to be shared among cloud computing members.

[ii].APPLICATIONS USING ONLINE SOCIAL RELATIONSHIPS

This section discusses prior work related to applications that use online social relationships. Various metrics can be has been extensive research on exploiting online social relationships to control networks. However, to the best of our knowledge, our work is the first on resource management for device sharing that enables device owners to control the authorized level of shared-resource usage according to their online social-relationships with device users.An example of routing in a delay-tolerant mobile adhoc network (MANET) involves performing community detection based on a dynamic online social relationship with frequent changes introduced by users joining or withdrawing from one or more groups or communities by friends connecting with each other or by new people making friends with each other. Wangetal. proposed a framework of traffic offloading assisted by social networking services (SNSs) via opportunistic sharing in mobile social networks. Their framework pushes the content object to a properly selected group of seed users, who will opportunistically meet and share the content with others, depending on their spreading impact on the SNS and their mobility impact. Through extensive trace-driven simulations, they demonstrated that their framework can drastically reduce mobile traffic load in cellular networks, while all users' access delay requirements can be satisfied. Kyleetal. suggested that online relationships in social networks are often based on real-world relationships and can therefore be used to infer a level of trust between users. On this hypothesis, they proposed to leverage those online relationships to form a dynamic ''Social Cloud''; thereby, enabling users to share heterogeneous resources. They actually implemented a social storage cloud application using the Facebook API, in which online storage is shared by people having online relationships on Facebook.Not only relationships between people but also relationships between content and people can be taken into consideration when distributing content in a network. Based on metrics produced from relationships between people and content, routers and content on the network can be managed physically to achieve load balancing, low-retrieval latency, and privacy while distributing content. Community detection from online social relationships can be used for creating a community-associated virtual network. Physical network resources

are assigned to each community associated network using a network virtualization technique. In a community-associated network, people can exchange privacy-sensitive data with only a small risk of data being disclosed to people who they are not socially connected to.

CHAPTER 4

REQUIREMENTS SPECIFICATION

4.1 Functional Requirements

I. The application should be able to share it's processor and memory with other systems across the network

II. The application should facilitate secure communication between nodes of the system.

III. The application should be provide a robust API to distribute compute among workers across the network.

IV. The application should provide the ability to control the systems that it shares it's resources with.

V. The application should be able to share usage statistics with other nodes on the network.

4.2 Non Functional Requirements

Non-functional requirements are the requirements which are not directly concerned with the specific function delivered by the system. They specify the criteria that can be used to judge the operation of a system rather than specific behaviors. They may relate to emergent system properties such as also reliability, response time and store occupancy. Non-functional requirements arise through the user's needs, because of budget constraints, organizational policies, the need for interoperability with other software and hardware systems or because of external factors such as:-

• Product Requirements

• Organizational Requirements

• User Requirements

• Basic Operational Requirements

i. The application should be secure.

ii. The application should not be able to access other systems personal data

iii. The application should be virtualized.
iv. The application should should be efficient.
v. The application should not put much burden on the normal usage of the application.

4.3 Hardware Requirements

The most common set of requirements defined by any operating system or software

application is the physical computer resources, also known as hardware, A hardware

requirements list is often accompanied by a hardware compatibility list (HCL), especially

in case of operating systems.

CPU : Pentium processor at 90 MHz or higher

Memory : 2 GB RAM

Hard drive : 50 GB available in the hard disk

Network Card : NIC card capable of high throughput

Graphics hardware : DirectX 3.0 or higher

4.4 Language requirements :

4.4.1 Python :

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.

Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Python is often described as a "batteries included" language due to its comprehensive standard library.

Python was conceived in the late 1980s as a successor to the ABC language. a garbage collection system capable of collecting reference cycles. Python 3.0, released in 2008, was a major revision of the language that is not completely backward-compatible, and much Python 2 code does not run unmodified on Python 3.

The Python 2 language was officially discontinued in 2020 (first planned for 2015), and "Python 2.7.18 is the last Python 2.7 release and therefore the last Python 2 release." No more security patches or other improvements will be released for it. With Python 2's end-of-life, only Python 3.5.x and later are supported.

Python was conceived in the late 1980s by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to the ABC language (itself inspired by SETL), capable of exception handling and interfacing with the Amoeba operating system. Its implementation began in December 1989. Van Rossum shouldered sole responsibility for the project, as the lead developer, until 12 July 2018, when he announced his "permanent vacation" from his responsibilities as Python's Benevolent Dictator For Life, a title the Python community bestowed upon him to reflect his long-term commitment as the project's chief decision-maker. He now shares his leadership as a member of a five-person steering council.In January 2019, active Python core developers elected Brett Cannon, Nick Coghlan, Barry Warsaw, Carol Willing and Van Rossum to a five-member "Steering Council" to lead the project.

A common neologism in the Python community is pythonic, which can have a wide range of meanings related to program style. To say that code is pythonic is to say that it uses Python idioms well, that it is natural or shows fluency in the language, that it conforms with Python's minimalist philosophy and emphasis on readability. In contrast, code that is difficult to understand or reads like a rough transcription from another programming language is called unpythonic.

Python is commonly used in artificial intelligence projects and machine learning projects with the help of libraries like TensorFlow, Keras, Pytorch and Scikit-learn. As a scripting language with modular architecture, simple syntax and rich text processing tools, Python is often used for natural language processing.

4.4.2 C Programming:

C (/siː/, as in the letter c) is a general-purpose, procedural computer programming language supporting structured programming, lexical variable scope, and recursion, while a static type system prevents unintended operations. By design, C provides constructs that map efficiently to typical machine instructions and has found lasting use in applications previously coded in assembly language. Such applications include operating systems and various application software for computers, from supercomputers to embedded systems.

C was originally developed at Bell Labs by Dennis Ritchie between 1972 and 1973 to make utilities running on Unix. Later, it was applied to re-implementing the kernel of the Unix operating system. During the 1980s, C gradually gained popularity. It has become one of the most widely used programming languages, with C compilers from various vendors available for the majority of existing computer architectures and operating systems. C has been standardized by the ANSI since 1989 (see ANSI C) and by the International Organization for Standardization.

C is an imperative procedural language. It was designed to be compiled using a relatively straightforward compiler to provide low-level access to memory and language constructs that map efficiently to machine instructions, all with minimal runtime support. Despite its low-level capabilities, the language was designed to encourage cross-platform programming. A standards-compliant C program written with portability in mind can be compiled for a wide variety of computer platforms and operating systems with few changes to its source code. The language is available on various platforms, from embedded microcontrollers to supercomputers.

4.5 Packages Required :

4.5.1 Docker :

Docker can package an application and its dependencies in a virtual container that can run on any Linux server. This helps provide flexibility and portability enabling theapplication to be run in various locations, whether on-premises, in a public cloud, or in a private cloud. Docker uses the resource isolation features of the Linux kernel (such as cgroups and kernel namespaces) and a union-capable file system (such as OverlayFS) to allow containers to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines. Because Docker containers are lightweight, a single server or virtual machine can run several containers simultaneously. A 2018 analysis found that a typical Docker use case involves running eight containers per host, but that a quarter of analyzed organizations run 18 or more per host.

CHAPTER 5

PROBLEM FORMULATION

5.1 Core Problem Statements

I. Share computational Resource across the network nodes though well defined protocols to avoid any kind of ambiguity in process sharing

II. A highly compatible system application that can be run across heterogeneous systems and devices. The application should be platform independent without interfering with the resource sharing module.

III. A secure channel for communication among the nodes with industry standard cryptographic protocols.

IV. Provide an API that abstracts the internal details of the application implementation.

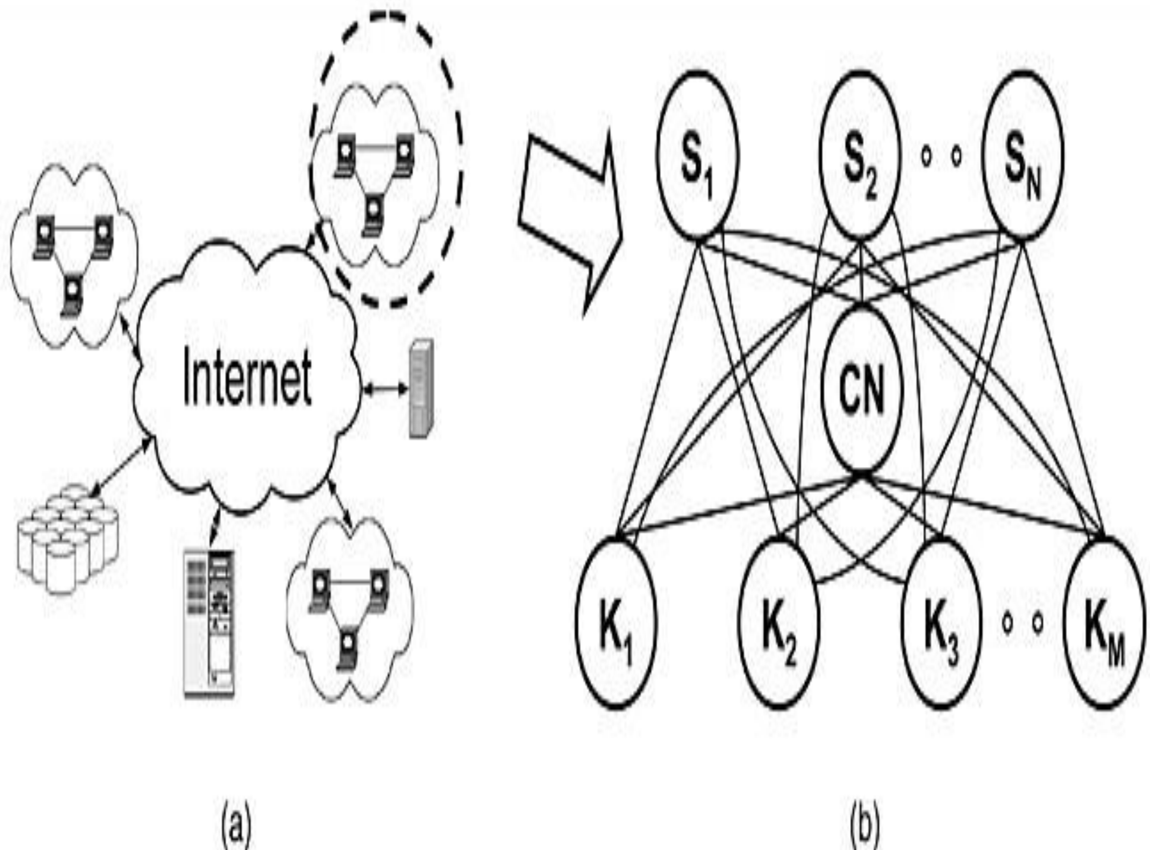


Fig: 5.1 Structure of Servers and Kernels

5.2 Socket Programming

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while other socket reaches out to the other to form a connection. Server forms the listener socket while client reaches out to the server.

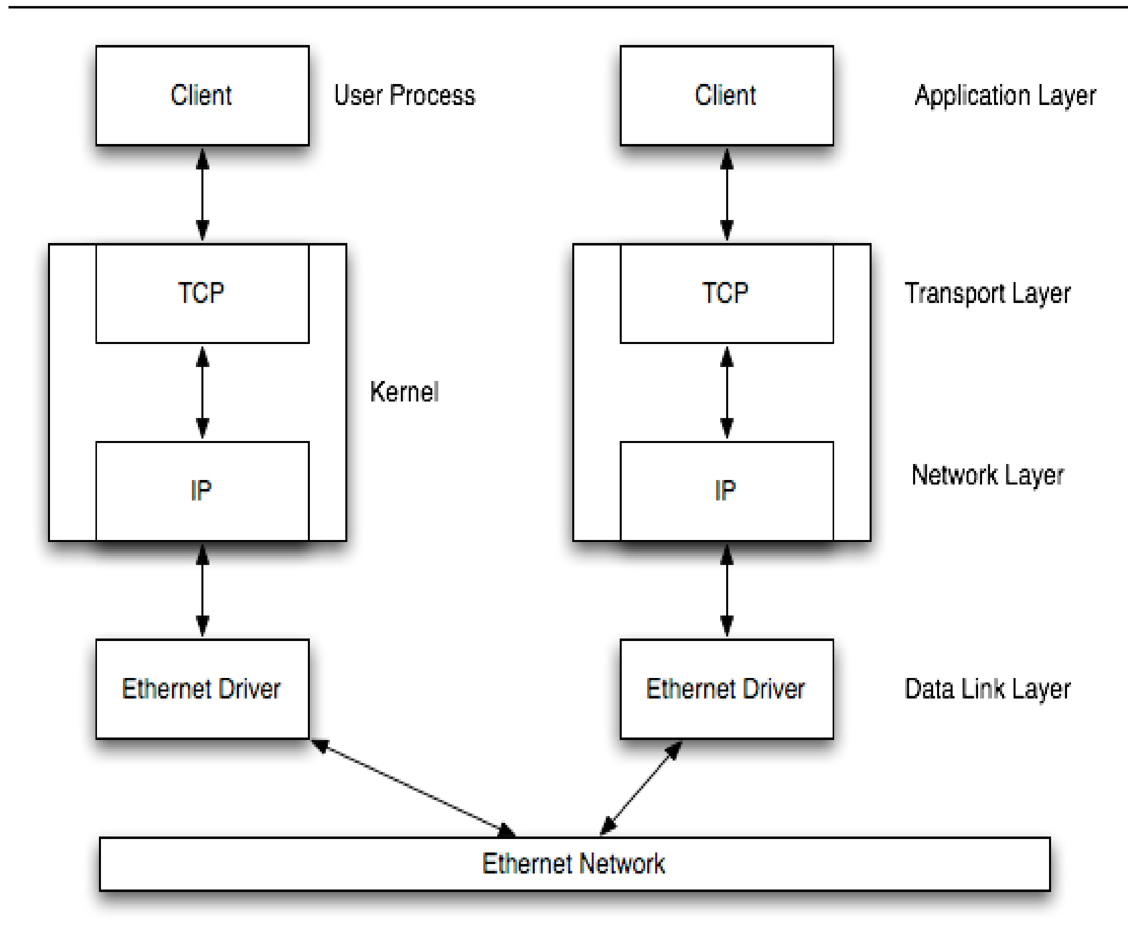


Fig: 5.2 Steps in Socket Connection

```
import redis
import time
import json

redis_endpoint = 'localhost:6379'
host, port = redis_endpoint.split(':')
r = redis.Redis(host=host, port=port, db=0, socket_timeout=10)

class Tracker :
    """
        keeps track of the number of clients that are active
        and deletes those that are not
    """

    active_clients_set = 'active_clients'       # sorted set holding the active_c
lients
    timeout = (10+5)*60*60                      # in seconds (before the client i
s removed from the active clients set)

    def get_active_clients() :
        """
            get all the clients that are currently active

            we do this by selecting clients whose scores(timestamp) is not
                well pass the timestamp cutoff period
        """
        Tracker.clear_inactive_clients()
        alive_clients =  r.zrange(Tracker.active_clients_set, 0, -1)
        return [client.decode('utf-8') for client in alive_clients]


    def clear_inactive_clients() :
        """
            get rid of all clients who have timed out
            scores are timestamps so we simply remove elements
                with scores well pass the timestamp cutoff
        """
        inactive_timeout = int(time.time()) - Tracker.timeout
        r.zremrangebyscore(Tracker.active_clients_set, 0, inactive_timeout)


    def add_active_clients(client_ids) :
        """
            add client_id to available active clients
```

```
            we use the timestamp as the scores so that we can easily remove them
later

            client_ids : a list of client id
        """
        current_time = int(time.time())
        mapping = {client_id.encode('utf-
8') : current_time for client_id in client_ids}
        r.zadd(Tracker.active_clients_set, mapping)


    def is_active(client_id) :
        """
            returns True if the client is active, else false
        """
        Tracker.clear_inactive_clients()
        client_id = client_id.encode('utf-8')
        return not r.zrank(Tracker.active_clients_set, client_id) is None


    def clear_all_clients() :
        """
            get rid of all the clients currently active (refresh)
        """
        current_time = int(time.time())
        r.zremrangebyscore(Tracker.active_clients_set,0, current_time)


class InstructionManager :
    pass


class TaskQueue :
    """
        Every agent has a queue attached to it

        This is class for that queue management in redis
        The master clients are responsible for populating the instructions

        When the slave completes the instruction, it puts the result of the instr
uction
            in the stream of the requesting master

        The client can choose whether to execute the instruction or not
        The server only manages the instruction queue
    """
```

```
def push(queue, instruction) :
        """
            queue -> the id of the client
            instruction : a dict of instructions
        """
        try :
            return r.rpush(queue, json.dumps(instruction).encode('utf-8'))
        except : return -1


    def pop(queue) :
        """
            queue -> the id of the client from which to pop
        """
        try :
            return json.loads(r.lpop(queue).decode('utf-8'))
        except :
            return None



if __name__ == '__main__' :
    #Tracker.add_active_clients(['23ASP', '57'])
    #Tracker.clear_inactive_clients()
    print(Tracker.get_active_clients())
    #print(Tracker.is_active('57'))
```

5.3 Multi Client-Server Architecture

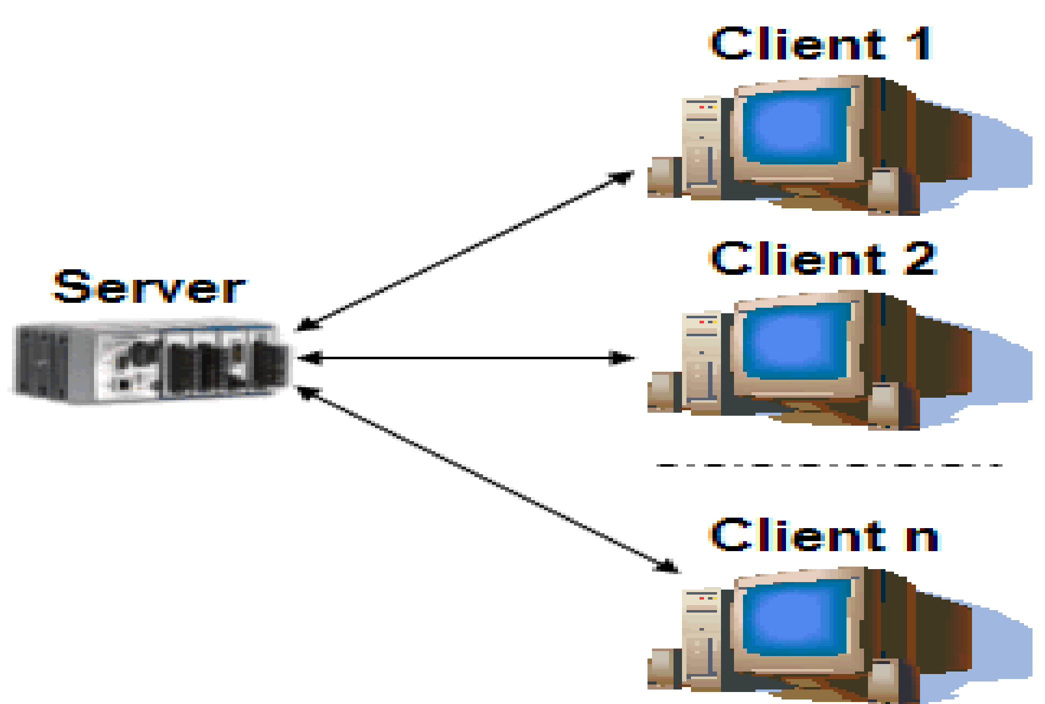
Client/server systems provide access to a central application from one or more remote clients. For example, a server application may perform some measurement or automation function (such as test cell control) and client applications may provide operators with a user interface for monitoring the state or progress of that function.

In multi-client applications, clients may connect and disconnect at random times. For example, during HIL batch tests that run for extended periods of time

various users may connect to the system several times a day to check on the progress and status of test that are of interest to them.

In order to support this scenario, the server software should be able to dynamically accept and service any number of incoming connections. The server should also keep track of client requests and be able to service each client in an individual way. For example, if the server acquires multiple channels of data, clients should be able to request a channel subset that is managed on a per-connection basis.

The goal of this document is to describe a client server design pattern that can run indefinitely, continuously monitoring for new connections and servicing them accordingly.



```
import docker
import sys

client = docker.from_env()

if __name__ == "__main__" :

    #self_id = 'DV_0011'
    #host = 'http://localhost:4500'
    host = 'http://15.206.168.1'
    try :
        self_id = sys.argv[1]
    except :
        print("usage : python3 run.py <client_id> <password>")
        exit()

    params = {
            'image' : 'shailav/images:cclient',
            'volumes' : {
                    '/var/run/docker.sock' : {
                        'bind' : '/var/run/docker.sock',
                        'mode' : 'rw'
                        }
                    },
            'environment' : {
                    'HOST' : host,
                    'SELF_ID' : self_id
                    },
            'extra_hosts' : {
                    'localhost' : '172.17.0.1'
                    },
            'detach' : True
        }

    # remove active container if any
    active_containers = client.containers.list()
    for c in active_containers :
        if 'client' in str(c.image) :
            if self_id in str(c.exec_run('printenv SELF_ID').output).strip('\n'):

                c.kill()
                print("killed : {}".format(c))

    # run container
    container = client.containers.run(**params)
```

Fig: 5.3 Client Server Connection

5.4 Fully Decentralized P2P Architecture

The architecture is peer-to-peer architecture (Figure 4.4). The same device acts as a client and as a server in this arrangement, with significant elements of each of the four functions of the app present on it. Because each device serves simultaneously as a client and a server, the consolidated device is often referred to as a servlet. In its pure form, there is no separate server or centralized point of control. This means that every client also simultaneously acts as a server. Therefore all devices connected to peer-to-peer architecture can simultaneously initiate requestsand fulfill requests from each other.

The key advantage of this approach is immense scalability: The addition of every new client simultaneously adds server capacity to the network. Scaling the capacity of any other architecture usually requires additional capacity on the server side, the need for which is eliminated by the use of peer-to-peer microarchitectures. Skype is an example of such architecture; it allows tens of millions of users to simultaneously use the service and can readily and automatically scale to meet rising demand. The incremental cost of adding another use is therefore pennies, and adding more users improves app performance unlike all other app microarchitectures where adding more users degrades performance.

However, this microarchitecture has two caveats. First, there is little or no control that the app developer has over the users of such apps. This limits the utility of this arrangement to only a few types of applications where central coordination and control are not needed and need for scalability is extremely high. Second, this architecture rarely exists in its purely decentralized form.

```python
import os
import sys
import re
from pprint import pprint
import json
from flask import Flask, url_for, jsonify, request, redirect, abort, send_file
import copy

from utils import Tracker, TaskQueue

app = Flask(__name__)


# configuration parameters
ENV_DIRECTORY = '{app_path}/environments/'.format(app_path=os.getcwd())
if not os.path.exists(ENV_DIRECTORY) :
    os.makedirs(ENV_DIRECTORY)

INSTR_DIRECTORY = '{app_path}/instructions/'.format(app_path=os.getcwd())
if not os.path.exists(INSTR_DIRECTORY) :
    os.makedirs(INSTR_DIRECTORY)


@app.route('/')
def index() :
    return 'WeCompute Interface'


"""
for dev purposes, clear the peer list
"""
@app.route('/clear', methods=['POST'])
def clear() :
    # empty the active peer list
    Tracker.clear_all_clients()
    msg = {'status' : 'success'}
    return jsonify(msg)



"""
endpoint for checking active peers
"""
@app.route('/peers', methods=['GET', 'POST'])
def peers() :
    # check the available peers visible
```

```
# improvement : create mechanism so that only peers who approve can be seen

    if request.method == 'GET' :
        try :
            # get requests parameters
            req = {k:v for k,v in dict(request.args).items()}
            if all([isinstance(v, list) for k, v in req.items()]) :
                req = {k:v[0] for k,v in dict(request.args).items()}

            # validate request parameters(skipped)
            # authentication/ authorization

            # return active peers visible to requesting client
            active_peers = Tracker.get_active_clients()
            try : active_peers.remove(req['id'])      # removing requesting clien
t from the list
            except : pass

            msg = {
                'status' : 'success',
                'count' : len(active_peers),
                'peers' : active_peers
                }
        except Exception as e :
            msg = {
                'status' : 'failure',
                'error' : '{}:{}'.format(e.__class__.__name__, str(e))
                }
        return jsonify(msg)

    if request.method == 'POST'  :
        try :
            # get json parameters
            req = request.get_json()
            if req is None :
                raise Exception('No json found')

            # validate request parameters(skilled)
            # authentication/ authorization

            # add client to active peers
            # later, you should allow clients to choose who they are visible to
            Tracker.add_active_clients([req['id']])
            msg = { 'status' : 'success'}
        except Exception as e :
```

The addition of every new client simultaneously adds server capacity to the network. Scaling the capacity of any other architecture usually requires additional capacity on the server side, the need for which is eliminated by the use of peer-to-peer microarchitectures. Skype is an example of such architecture; it allows tens of millions of users to simultaneously use the service and can readily and automatically scale to meet rising demand. The incremental cost of adding another use is therefore pennies, and adding more users improves app performance unlike all other app microarchitectures where adding more users degrades performance. However, this microarchitecture has two caveats. First, there is little or no control that the app developer has over the users of such apps. This limits the utility of this arrangement to only a few types of applications where central coordination and control are not needed and need for scalability is extremely high. Second, this architecture rarely exists in its purely decentralized form.

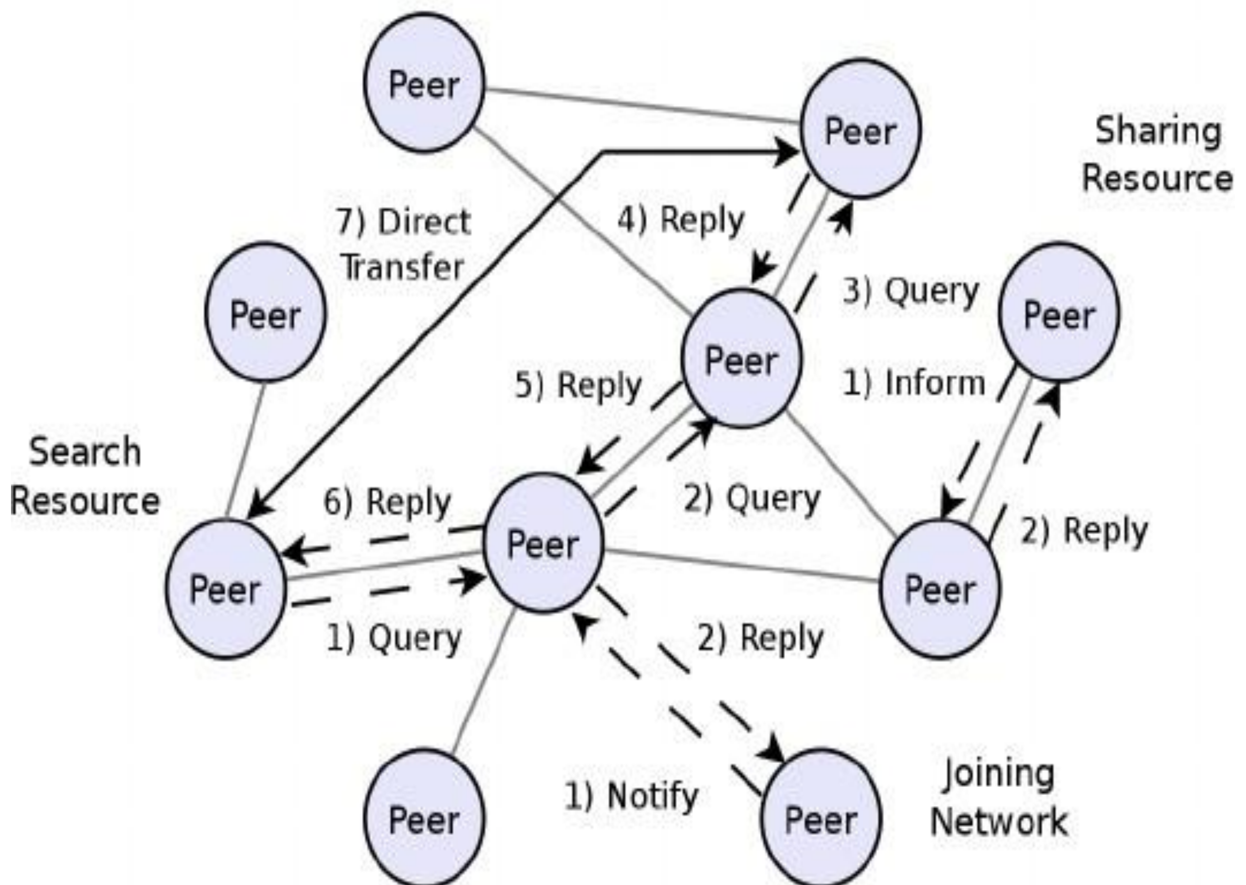


Fig: 5.4 Fully Decentralized P2P Architecture

5.5 Additional Problem Statements

i. A social network layer where user can configure who they share their resources with.

ii. Allow different level of controls as per the use specification.

iii. Report usage metrics and balance work loads across the nodes of the network

CHAPTER 6

STATUS AND ROADMAP

The summary of the work carried, the current status with the challenges and constraints and roadmap for Phase II.

The work can be divided into three phases :

6.1 Network Phase :

The challenge here is to create a robust way of exchanging compute data provided by the application layer without compromising the data to third party.

The IP/layer and Transport layer modules have to be augmented to be compatible with application layer.

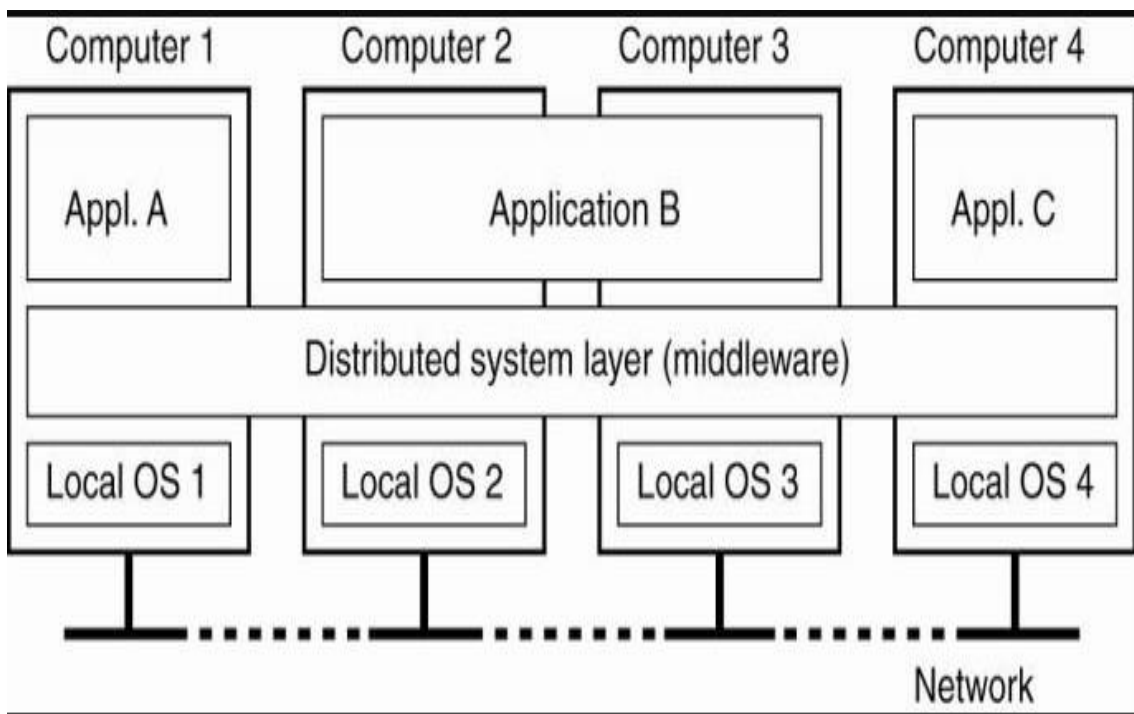


Fig: 6.1 Different Network Phases

6.2 Application Phase :

The challenge here is to determine and implement the protocols to exchange the compute information and how the distribution of workload should be done.

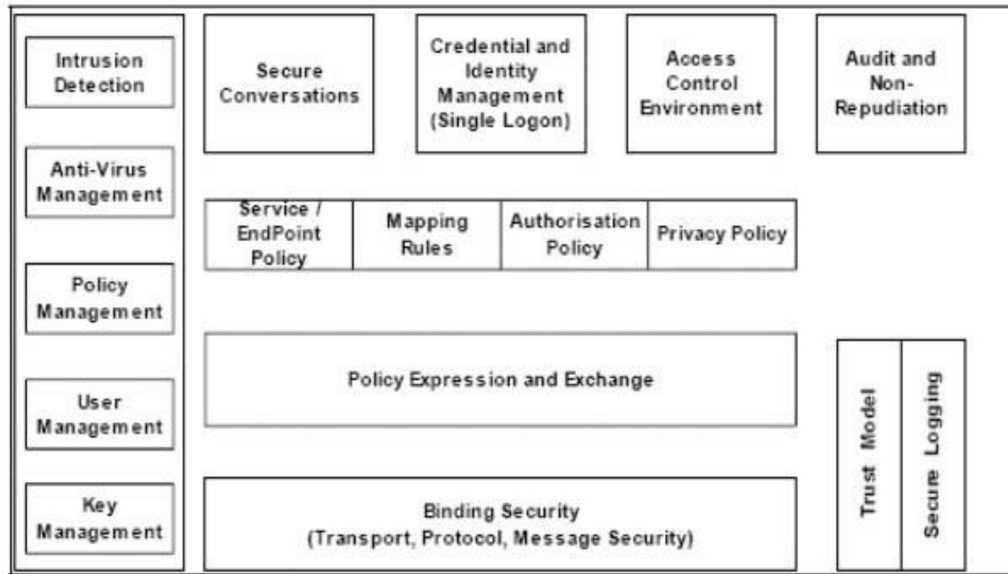


Fig: 6.2 Management of Various Protocols

6.3 Abstraction Phase :

The challenge here is to provide an interface that developers can use without compromising of the control provided by the application layer.

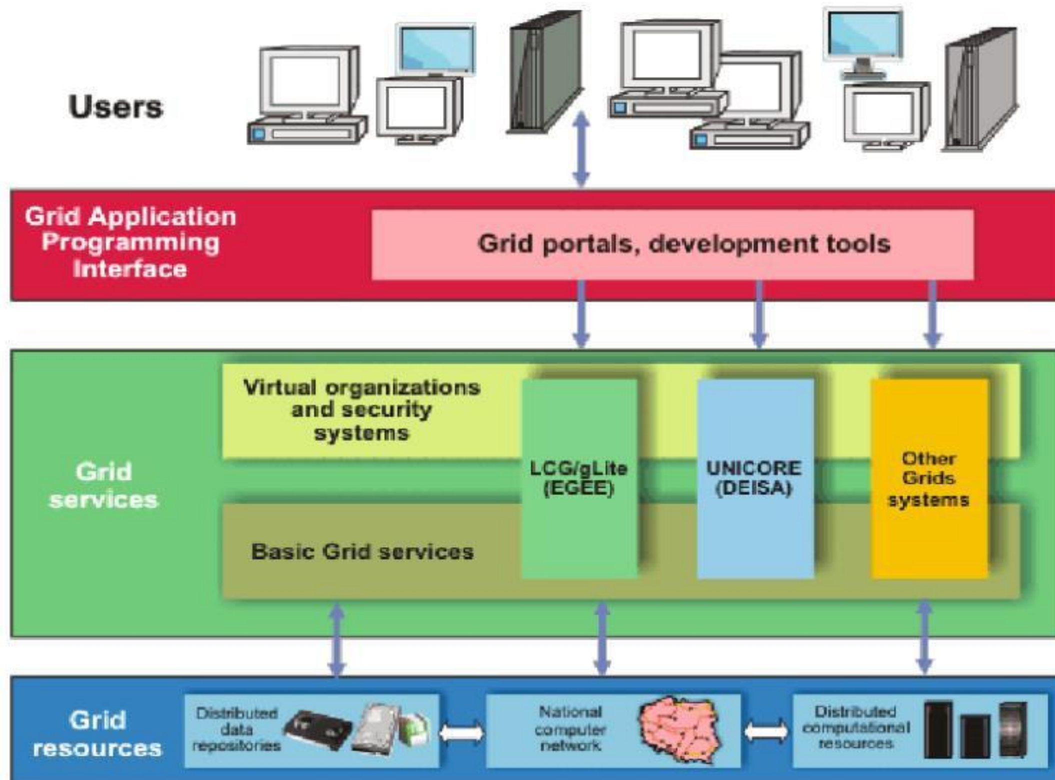


Fig: 6.3 Example Of Interface

6.4 UX Phase :

The challenge here is to provide a front-end layer of control for the application layer services to the developers with suitable UI and other controls.

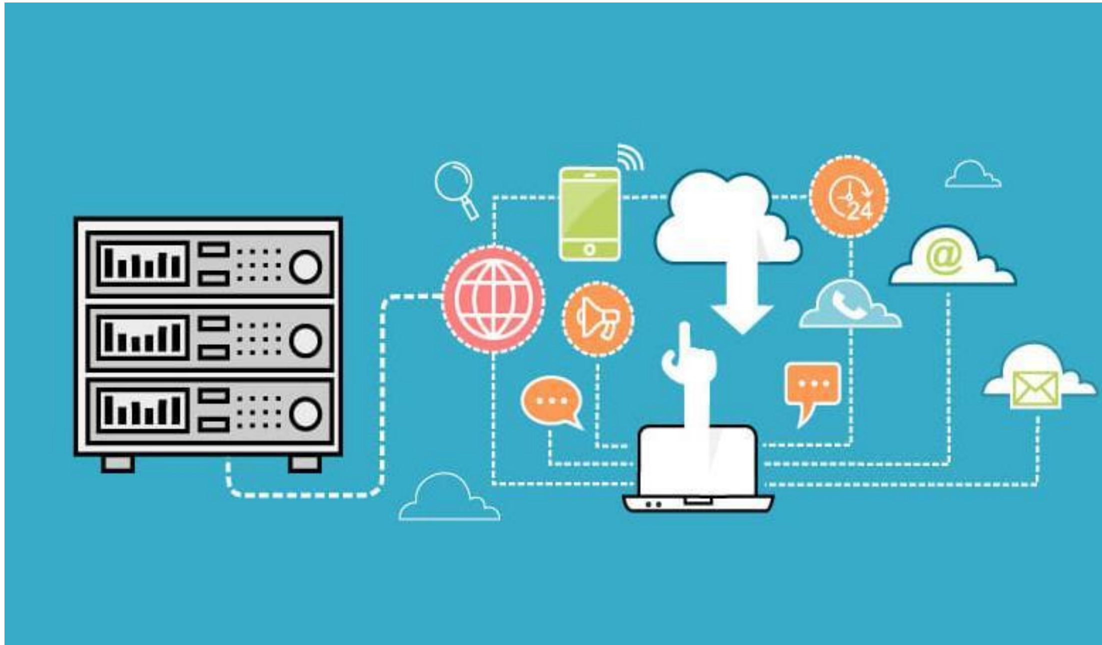


Fig: 6.4 Front-end layer illustrating UI

CHAPTER 7

CONCLUSION AND FUTURE SCOPE

In this project report, we have presented a distributed list heuristic, DHEFT for decentralized scheduling of work-flow applications in global Grids. Using simulation, we have measured the performance of proposed DHEFTbased scheduling technique. Results show that it is scalable with respect to increased workload on the system. In future, we intend to investigate performance of this proposed technique against other list heuristics in decentralized workflow scheduling environment.This report first demonstrates the general structure of the smart grid integrating to cloud computing, and illustrates the security issues for the smart grid communication network, and for the cloud computing paradigm respectively. And then we can protect the smart grid users' privacy information from adversaries using data chunk technology. At the same time, we propose a chunk information list system via which the data inserting and data querying can be implemented.

REFERENCES

[1] Thies, H.-H., Schneider, M., Zdrallek, M., and Schmiesing, J. “Future structure of rural medium-voltage grids for sustainable energy supply.” In: Integration of Renewables into the Distribution Grid, CIRED 2012 Workshop, pp.1–4, 29–30 May 2016.

[2] Lehnhoff, S., Blank, M., Gerwinn, S., and Krause, O. “Support Vector Machines for an efficient Representation of Voltage Band Constraints.” Proceedings of the IEEE International Conference on Innovative Smart Grid Technologies Europe 2011, IEEE Press: Manchester, UK, 2017.

[3] K. Maheshwari, M. Lim, L. Wang, K. Birman, R. Van Renesse, "Toward a reliable, secure and fault tolerant smart grid state estimation in the cloud ", Proceedings of 2013 IEEE PES Innovative Smart Grid Technologies, 2017, pp. 1-6.

[4] J. Yu, R. Buyya, and K. Ramamohanarao. Work- flow Scheduling Algorithms for Grid Computing, Metaheuristics for Scheduling in Distributed Computing Environments. F. Xhafa and A. Abraham (eds), Springer,Germany, 2008.