ADDIS ABABA UNIVERSITY

COLLEGE OF NATURAL SCIENCE

Integrated Caching and Prefetching on Dynamic Replication to Reduce Access Latency for Distributed Systems

Yilkal Binalf Worku

A Thesis Submitted to the Department of Computer Science in Partial Fulfillment for the Degree of Master of Science in Computer Science

Addis Ababa, Ethiopia
July, 2021

Addis Ababa University

College of Natural and Computational Sciences

Yilkal Binalf Worku

Advisor: Mulugeta Libsie (PhD)

This is to certify that the thesis prepared by Yilkal Binalf, titled: *Integrating Caching and Prefetching on Dynamic Replication to Reduce Access Latency for Distributed System* and submitted in partial fulfillment of the requirements for the Degree of Master of Science in Computer Science complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the Examining Committee:

	Name	Signature	Date
Advisor:	Dr. Mulugeta Libsie		
Examiner:			
Examiner:			

Abstract

Distributed computing is a rapidly developing IT technology. Every system connects to other systems via the network to improve its performance. Thanks to distributed systems technology, workers from all over the world can collaborate to work for a single company, and customers of these companies can access data and receive service as if they were in the same location. However, as the number of users and organizations requesting and delivering these services grows, there is a problem with access latency. One of the major problems of distributed systems is response time latency. As a result, we developed the integrated Caching and Prefetching on Dynamic Replication (CPDR) algorithm, which reduces access latency in distributed computing environments.

Cacher, Prefetcher, and Replicator are the three main components of the developed system. There is one more unique component in the cacher called Notifier, which has the Prefetcher's status and is used to save time when the prefetcher is not active and the requested data is not available. Furthermore, the Cacher, Prefetcher, and Replicator each has a manager component that contains an algorithm for controlling the Cache storage, prefetching data, replicating data, and determining where data should be placed.

Moreover, taking various scenarios, which depict the minimum and maximum capacity of the computing environment as well as different requirements of incoming jobs, we evaluated our algorithm With caching, prefetching, dynamic replication, the integration of caching and prefetching, the integration of caching and dynamic replication, integration of prefetching and dynamic replication algorithms. It is observed that the proposed algorithm outperforms the counterparts from the perspective of response time and storage utilization.

Keywords: Distributed Computing, Caching, Prefetching, Replication, Dynamic Replication Algorithm, and CPDR algorithm

Acknowledgments

My first thanks are to the Almighty God and his mother, Saint Mary! Without His/Her blessing, I wouldn't have been writing even this acknowledgment.

Then, I would like to express my heartfelt thanks to my advisor, Dr. Mulugeta Libsie for his invaluable guidance, giving insightful comments, constructive criticisms, encouragement to the use of correct grammar, consistent notation in my writings, and for carefully reading and commenting on countless revisions of this document. I hope one day I would become a good teacher and advisor to my students as he has been to me. I have learned a lot from you; thank you very much again.

I am very thankful to my families for their invaluable support throughout this study. Also to all those people who made this thesis possible and because of whom my graduate study experience has been one that I will value forever. Especially, to Natneal Tefera, Mikiyas Bayew, Wezam Seyuom, Tesfanesh Esayas, Eyasu Taye, Leykun Mihret, and Girma Negash for their support and companionship.

Finally, last but not least, my heartfelt thanks go to colleagues, my classmates, and friends whom I have not mentioned their names above, for their unlimited encouragement during my study.

Table of Contents

List of Figures

List of Algorithms

List of Tables

List of Acronyms

BHR	Bandwidth Hierarchical Replication
CPDR	Caching and Prefetching on Dynamic Replication
CRF	Combine Recency and Frequency
DDBS	Distributed Database System
DHR	Dynamic Hierarchical Replication
DNS	Domain Name System
FBR	Frequency Based Replacement
FIFO	First In First Out
FLSDR	Fuzzy Logic System Dynamic Replication
Fuzzy_Rep	Fuzzy Replication
HTML	Hypertext Markup Language
I/O	Input or Output
IPC	Integrated Prefetching/Caching
LFU	Least Frequently Used
LRFU	Least Recently Frequently Used
LRU	Least Recently Used
MBHR	Modified Bandwidth Hierarchical Replication
MRU	Most Recently Used
NoRep	No Replication
OS	Operating System
P2P	Peer to Peer
PPRA	Prediction and Prefetching Replication Algorithm

1. Introduction

1.1 Background

There are several definitions of what distributed systems are, Coulouris defines a distributed system as “a system in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing” [1], Tanenbaum defines it as “A collection of independent computers that appear to the users of the system as a single computer” [2], and Leslie Lamport said that “A distributed system is one on which I can’t get any work done because some machine I have never heard of has crashed” [3].

In a distributed environment, many factors affect the performance of the distributed system. Access delay is one of the major issues from them. The availability of network bandwidth affects the majority of access delays. Because transferring large files requires a large amount of bandwidth and takes a long time [4]. The three most common methods used for reducing access latency for distributed systems are prefetching, caching, and replication [5].

In a replicated environment, copies of data are stored by multiple sites [6]. Increasing the number of replicas increases system performance by improving locality. However, Additional data transmission is required to keep copies up to date to maintain data consistency. Because the cost of excessive fragment replication rises, a replication technique that can control the number of replicas for each fragment is required. Note that individual fragments may have varied numbers of replicas; for example, a regularly requested fragment may be replicated at a large number of sites, whereas an infrequently requested fragment may be replicated at a smaller number of sites. Dynamic replication creates and deletes replicas based on the user's activities. It can adjust to changes in user activities to reduce overall transmission time. But excessive replication can degrade the performance of distributes system because of the additional cost involved in managing and applying update queries.

A cache is defined as temporary storage for data on a system that is reserved for the quicker servicing of future requests [7]. The data in the cache is either duplicate copies of data saved elsewhere or is stored there as a result of being frequently accessed. Caching essentially speeds up future data accesses.

Prefetching is a method for moving data objects up in the memory hierarchy or to the client system before the data objects are required. The movement of data improves the access latency if they are subsequently requested. The prefetching method has proved its ability to significantly reduce long wait times [8]. To solve the access latency problem, we propose a model that integrates the three basic methods by taking the strong side of each to reduce access latency and improve the performance of distributed systems.

1.2 Motivation

Nowadays, developing a model for improving system performance becomes a key concern in the research community. Reducing access latency is one of the most challenging tasks of a distributed system [9]. The different authors tried to use different methods and algorithms to reduce access latency and to improve system performance in a distributed system. However, it is difficult to make the distributed system as efficient as we want because each method and algorithm used has its limitations.

Authors in [10] use the integration caching and prefetching for client-side system web traffic, Caching is a method of reducing response time by storing copies of frequently used documents in a local cache. However, as a web document gets more dynamic, the value of caching diminishe. The goal of prefetching is to get beyond the limitations of passive caching by fetching documents in advance in preparation for future demand requests.

Another work in [11] uses the integration of prefetching and data replication to reduce access latency in the cloud system. Most of the authors use either one or the combination of the two from the three basic methods to improve system performance. We propose to use the integration of the three basic reducing access latency methods by taking the strong side of each which can solve the limitation of the other to improve the performance of a distributed system. These issues inspire us to investigate more to reduce access latency to improve performance, particularly for a distributed system.

1.3 Statement of the Problem

Nowadays finance, education, health, and business organizations need better performance to do their daily work effectively. In a distributed system the main challenge is the access latency since the systems are distributed in different geographical locations. Many authors develop different methods to solve access latency.

Authors in [12] described a distributed caching, prefetching, and replication method and system. As the authors stated, the invention provides an autonomous, distributed, and transparent caching scheme that takes use of the fact that document requests naturally build a routing graph as they travel via a computer network from a client to a specific document on a certain home Server. Client request messages are directed up the graph, as they would be in the absence of caching, to the home Server. Cache Servers, on the other hand, are located along the path and may intercept requests if they are functional. The cache server should be able to insert a packet filter into the router associated with it, as well as a proxy for the home Server from the user's perspective, to service requests in this manner without deviating from standard network protocols. Cache servers can help Service users by caching and deleting documents based on their current load.

Here the authors focus on distributed caching schema and replicate the cached document from one cache server to others. But, they did not use the integration of the three methods at once. We need more advanced methods. So, this proposal will provide a better solution.

Accordingly, the research attempts to answer the following research questions:

- What is the basic factor that affects or increases access latency in a distributed system?
- What are the techniques that can be used to improve the performance of a distributed system?
- What are the efficient algorithms used to reduce access latency methods in a distributed system?
- How to develop a model that can reduce access latency for a distributed system?

1.4 Objectives

General Objective

The general objective of the research is to design and develop a model to improve the performance of a system by reducing access latency using the integration of caching, prefetching, and dynamic replication for a distributed system.

Specific Objectives

To realize the general objective of the research, the following specific objectives are identified.

- To understand the different algorithms used to reduce access latency

- Understand major factors that affect access latency
- Design as a scientific model
- Develop a prototype of the system
- Evaluate the performance of the system

1.5 Methods

To achieve the general and specific objectives mentioned above, we use the following methods.

- **Literature Review**

We will review literature, including conference papers, journal articles, and books about reducing access latency of the distributed system.

- **Develop a Model**

It is a scientific model and a visual paradigm will be used to design the reduced access latency model. Various document processing, data analysis, and model designing tools will be used to compile the document. Moreover, NetBeans IDE with Java programming language will be used to develop the prototype.

- **Testing and Evaluation**

The proposed model will be evaluated based on different parameters. So, different tests will be done, to evaluate whether it improves the performance of the system by using evaluation metrics. We use evaluation metrics to evaluate the result by comparing with the results of previous work.

1.6 Scope and Limitations

This research work aimed to propose a model to reduce access latency for a distributed system. The proposed model works by integrating the three methods to increase system performance. Particularly focuses on caching, prefetching and dynamic replication in a distributed system. Besides that, this research work does not address issues related to the consistency and security of a distributed system.

1.7 Application of Results

The result of this research work could show a radical improvement in performance for a distributed system. And this research will contribute to ongoing researches on the same domain. It will be an input for different stakeholders, authors, and students. Likewise, it provides its contribution to the beneficiaries like cloud computing, edge computing, and other related fields that want to increase system performance by reducing access latency.

1.8 Organization of the Rest of the Thesis

The rest of this thesis is organized into five Chapters. Chapter Two gives an overview of characteristics, challenges, limitations, factors that affect access latency, and basic methods and algorithms used in distributed systems. Chapter Three describes related work which is about methods and algorithms used by different authors to solve different distributed system access latency problems. Chapter Four presents the proposed solution. In this Chapter, the big picture of the proposed model, as well as issues considered during its design. In addition to the above, relationships between components and algorithms used are explained in detail. Chapter Five presents topics that are concerned with describing the implementation details of the proposed model. Tools used to develop the prototype as well as their specific significance to this prototype development are described. Additionally, this Chapter presents the results gained after comparing the proposed algorithm with others. Results are illustrated using a tabular and graphical view. The last section of this thesis, Chapter Six, summarizes the major concepts raised in this research work. In addition, the contributions of the research, recommendation, and by pointing out future works we identified during our research, works that could be done to improve the performance of the proposed model are briefly described.

2. Literature Review

This Chapter deals with the review of important concepts concerning the research work. It provides detailed information about definitions, characteristics, architectures, challenges, and limitations of distributed systems. It also discusses briefly the definitions, factors, methods, and algorithms used in access latency of distributed systems.

2.1 Overview of Distributed Systems

There are several definitions of what distributed systems are: some of the authors define distributed systems as follows.

Coulouris defines a distributed system as “a system in which hardware or software components placed at networked computers interact and organize their activities solely by message passing” [1]. According to Coulouris, a system to be a distributed system can include either hardware components or software components, which are located in a networked computer that is linked via a network.

According to Tanenbaum, a distributed system is “a series of independent computers that appear to machine users as a single device” [2]. According to Tanenbaum's description, a distributed system refers to a software system rather than the hardware involved in its development. The transparency of distributed systems is the focus of this term. These computers are autonomous computers that are distributed all over the world but behave to the user as a single machine. The user has no knowledge of what is going on in the background of these systems, how data is stored, whether other systems are failing or not, and does not know the complexity of the systems.

Lamport is an author on distributed networks message ordering and clock synchronization. A distributed system, according to this author, is "one in which I can't get any work done because some computer I've never heard of has crashed" [3]. A distributed system prohibits you from doing any job when a computer that you've never heard of fails. This author considers the many problems that distributed system designers face. When a system fails, the user has no idea which device is malfunctioning; what s/he understands that s/he can't do something. Despite these difficulties, the advantages of distributed networks and implementations outweigh the disadvantages, making them worthwhile to try.

Sharan is a well-known Google author who has made significant contributions to software engineering, security research, and the application of IoT protocols, as well as other areas. As defined by Sharan, a distributed system is a system that is a series of self-governing computer systems capable of transmitting and cooperating through hardware and software interconnections. It is a collection of loosely coupled processors that appears to its users as a single systematic system. Since a failure of one system does not affect the success of another, distributed systems should be loosely coupled. If the systems are tightly connected, a failure in one system can fail the other. This causes the whole machine to fail. The distributed system's properties are so loosely coupled.

The authors in [13], defined a distributed system as a set of independent computing components that appear to users as a single coherent system. This description applies to two distinguishing characteristics of distributed systems. The first is that a distributed system is made up of a group of computational components that can all act independently of one another. A computational element, referred to as a node, may be either a hardware device or a software device. The fact that customers (whether individuals or applications) feel they are working with a single device is a second factor. This implies that the autonomous nodes can exist in some form or another.

According to Firdhous [14], a distributed system is an application that interacts with multiple spread hardware and software to organize the activities of multiple processes operating on separate autonomous computers through a communication network. A distributed system, in this view, is not a program, but rather a network of autonomous computers that presents their users with a user interface.

Finally, combining these concepts, a distributed system can be described as a group of autonomous computers that interact across a network to coordinate the activities of multiple processes operating on separate autonomous computers using message passing, such that both hardware and software components work together to perform a series of specific tasks targeted towards a common objective.

2.1.1 Characteristics of Distributed Systems

Some of the major characteristics of a distributed system are the following [1]:

a. Independent Failures: Any computer device has the potential to fail. Faults in the network result in the separation of the computers connecting to it, meaning the distributed system can fail in a new way. However, this does not imply that they cease to function; instead, programs running on them can be unable to detect whether the network has crashed or has been abnormally slow. Failure of a computer or a crash is not immediately made known to other components with which it communicates. Each component of the system can fail independently, leaving the others still running. This independent failure is one basic characteristic of the distributed system because it improves performance when one system fails the other continues its work independently. But in distributed systems, because of the complex design of distributed networks, providing high fault tolerance, high availability, and reliable access to necessary data is a difficult task [15].

b. Concurrency: Concurrent program execution is the standard in a network of computers. We can work on our computers while others work on their computers, and we can share resources (such as websites or files) as needed. In general, it is difficult to define and interpret concurrent interactions in distributed networks [16]. Since there is no universal clock, maintaining consistency is difficult. Concurrency management is a critical consideration in the architecture of distributed systems. This is because concurrency requires several transactions to run at the same time and resulting in a coherent list of manipulated data items.

2.1.2 Architecture of Distributed Systems

Distributed systems are created by layering existing networks and operating system applications on top of each other. A distributed system is made up of a collection of autonomous computers linked together by a computer network and middleware. For two computers in a network to become independent, they must have a simple master-slave relationship. Users view the system as a single, interconnected processing facility because the middleware allows machines to organize their operations and share the system's resources [2]. As a result, middleware acts as a bridge between distributed systems that are run on different hardware platforms, network technologies, operating systems, and programming languages and are spread over several physical locations. The middleware is created by established standards and protocols. The middleware service is spread through several devices. Figure 2.1 shows a simple architecture of a distributed system [2].

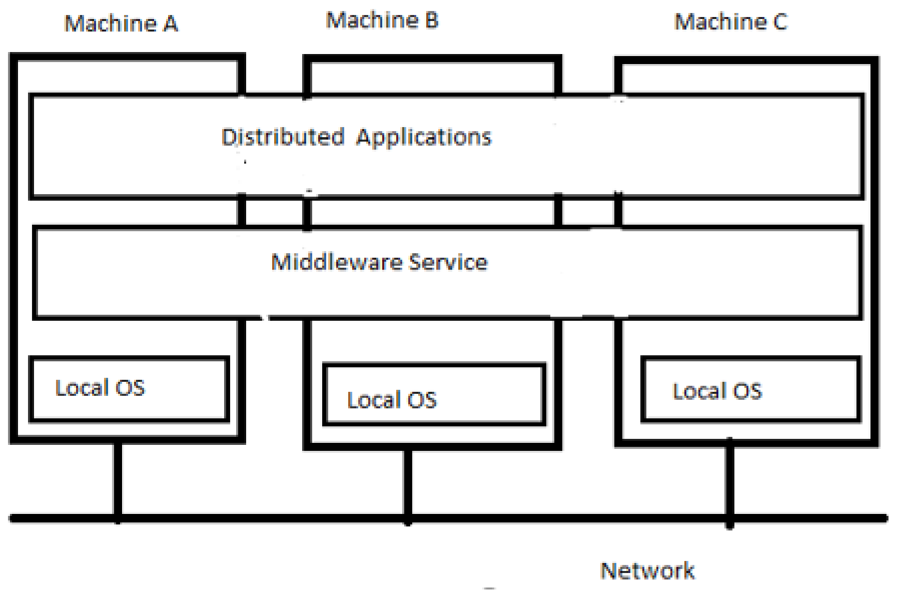


Figure 2.1: General Architecture of a Distributed System

The architecture of a distributed system can vary since the separation is dependent on various parameters. The following categories are based on the method with the type of system.

a. The Architecture of Cache in Distributed Systems

Users from all over the world are increasingly using the World Wide Web (WWW), resulting in a rise in network traffic. The primary goal of a cache is to store a copy of data close to the user so that a web user can access it without going to the webserver. When a user requests a web page, the cache is initially examined to verify if the page requested is available. The request redirects the web server and sends the answer to the client if the web page is not located in the cache. A web cache is installed between the server and the client to track client requests. It would support the request from the cache rather than sending it to the browser. This would reduce latency and also save bandwidth. Figure 2.2 shows the architecture of web caching [17].

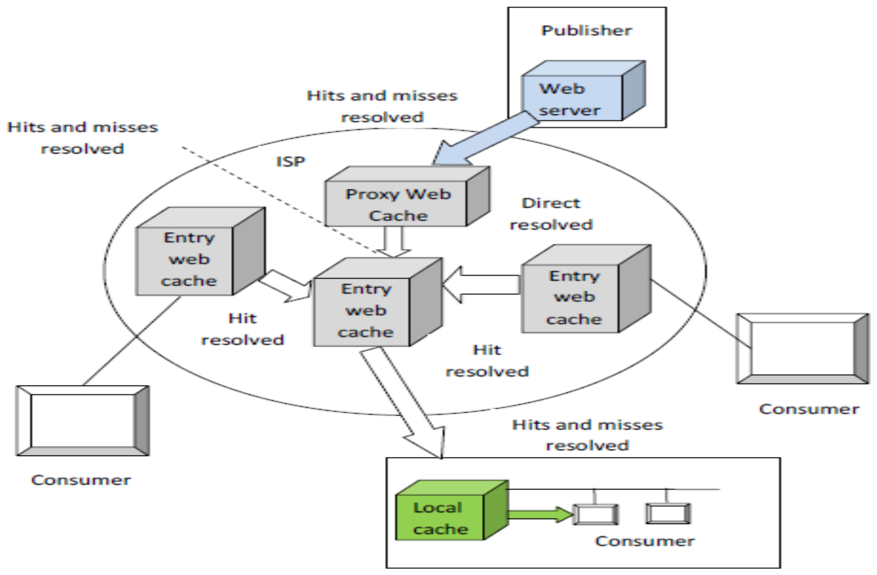


Figure 2.2: Web Cache Architecture

b. Architecture of Peer to Peer (P2P) Collaborative Cache

A P2P network is a decentralized computing network. Each node in a P2P network model will serve as both a client and a server. The P2P system accepts that network performance and operating reliability must be maintained [18]. If the user requests data that isn't in the local cache, P2P Cooperative Caching will check to see if all of its related peers have it. If the requested data is located in its peer's cache, the requesting node will retrieve it directly over a high-speed local connection, bypassing any external network such as the Internet. If the requested data is not in any peer cache, it must be fetched from the server. Data caching lower access time for such contents so they can be automatically fetched from the local or peer's cache, lowering network access costs. Figure 2.3, shows the architecture of a P2P collaborative cache [19].

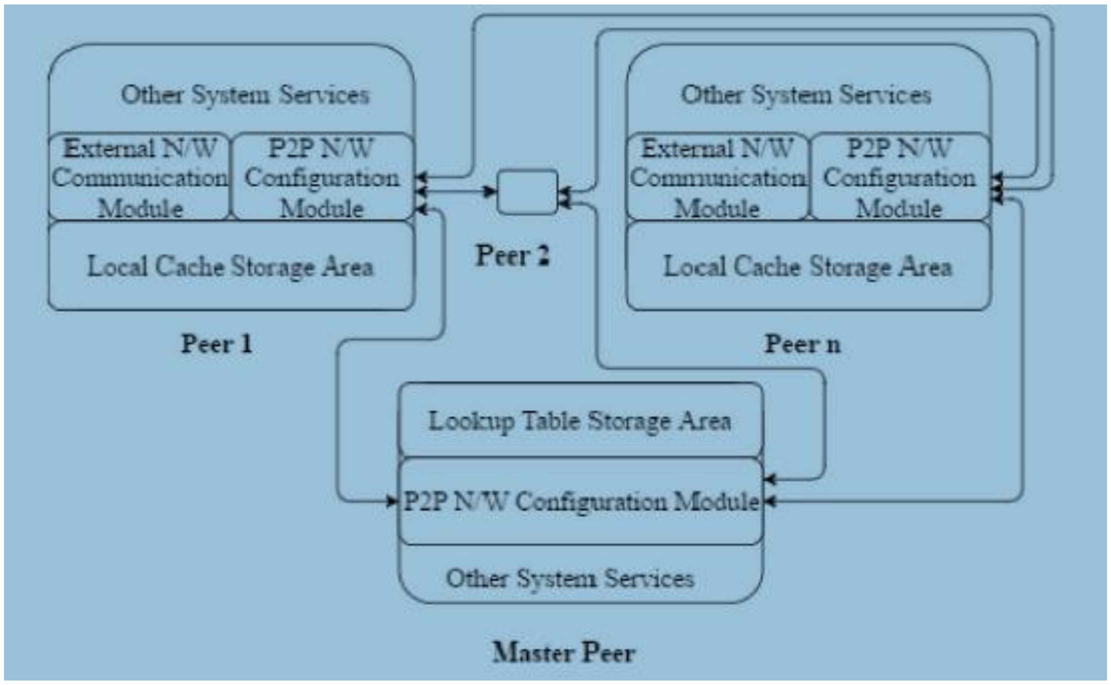

Other System Services
External N/W Communication Module
P2P N/W Configuration Module
Local Cache Storage Area
Peer 1
Peer 2
Other System Services
External N/W Communication Module
P2P N/W Configuration Module
Local Cache Storage Area
Peer n
Lookup Table Storage Area
P2P N/W Configuration Module
Other System Services
Master Peer

Figure 2.3: Architecture of P2P Cooperative Cache

❖ Communication diagram of Cache Functionality

The cache is commonly used in database servers, web servers, file servers, storage servers, *etc*. [20]. The cached data is usually stored in small storage; the access time is often faster than the original data. On the other hand, the cache capacity is insufficient to contain all of the required data. A distributed file system's cache can be found on both the client and server. It is not possible to improve system speed by using cache on both the server and the client. On the client side, increasing the cache hit ratio produces an increase in the miss ratio on the server side and vice versa [21]. The communication of cache functionality is shown in Figure 2.4 [21].

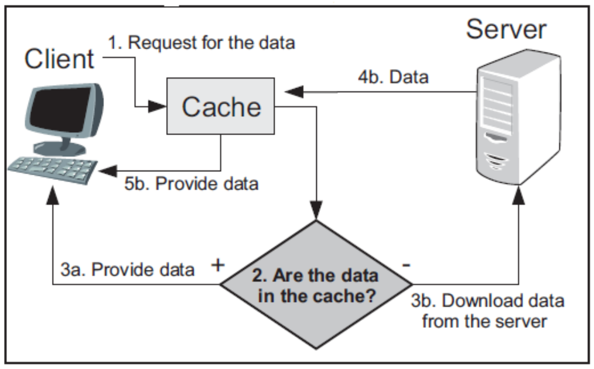


Figure 2.4: Communication diagram of Cache Functionality

c. Architecture of Prefetch for Distributed Systems

Automatic prefetching outperforms regular demand prefetching in distributed file systems by a small amount. However, because these techniques can only make an educated guess, some of the prefetched files will not be used by the client, resulting in increased network traffic without improved speed. Automatic prefetching predicts upcoming file system requests based on the previous file access information. The aim is to provide data before a request is received, essentially eliminating access latency. One method for automatic prefetching was proposed by Griffioen *et al*. [22]. Gwan *et al* [23] discussed the Appointed File Prefetching (AFP) method to reduce the latency. The user and system administrator may use the designated file-prefetching language (AFPL) to direct the system to execute requested prefetching at the necessary times. The AFPL prefetching instructions are split into two categories: (1) choosing the appropriate files and (2) deciding when prefetching should be performed. Figure 2.5 presents the demand fetching mechanism employed in many distributed file systems [23].

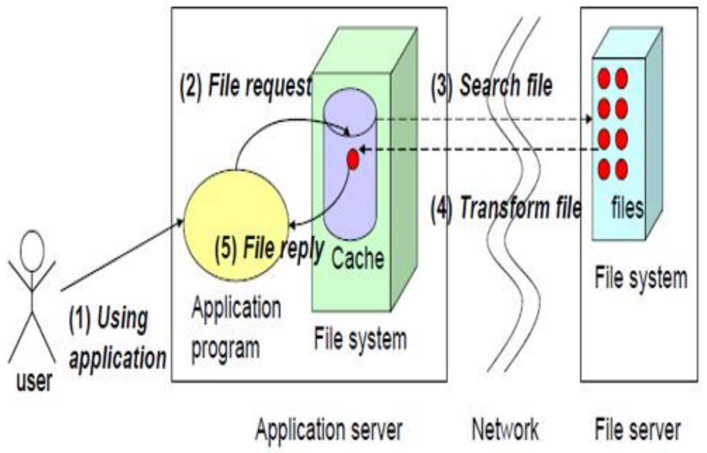


Figure 2.5: Demand Fetching in the Distributed File System

d. Architecture of Replication for Distributed Systems

Replication is a commonly used strategy in distributed systems for assuring high availability, fault tolerance, and performance [24]. Replication is the process of replicating and storing data objects across several locations in a distributed system. An object can be accessed from several locations in a replicated system, such as a local area network or geographically dispersed across a large region or even the entire world. Replication enables users with easy, local access to shared data while simultaneously protecting application availability. If one of the sites is unavailable, the user can still access it from others. There are two types of replication [25]: centralized replication and distributed replication. In a centralized replication setup, data replicas provide read-only access to data that originates from a master site. In a distributed replication scenario, users can access and update data replicas across the system. A distributed system consists of clients, a replica control mechanism, and a collection of data objects stored on a set of sites, Sl,.., Sn, joined by an interconnection network, as shown in Figure 2.6 [25].

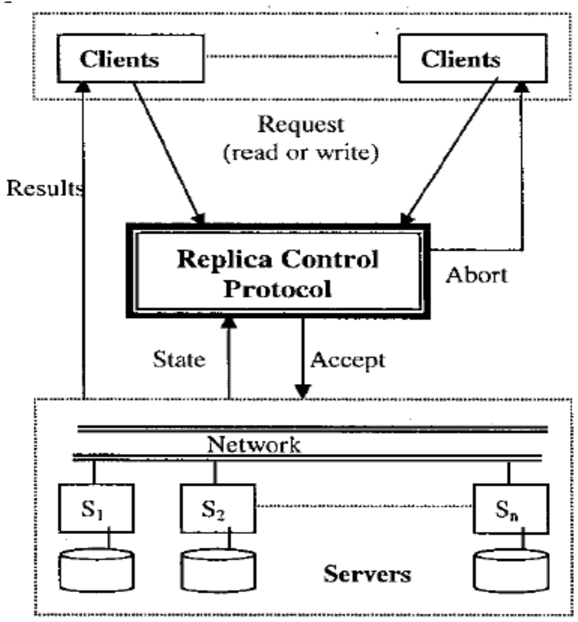


Figure 2.6: Architecture of Replicated Distributed System

2.1.3 Challenges of Distributed Systems

Any user will benefit from a distributed system since it reduces latency and saves money. Many factors should be considered when planning and using a distributed system since constructing a distributed system is not as simple as it seems. To achieve an optimal method, several obstacles must be solved. There are numerous distributed systems available today, each focused on solving a specific problem. Depending on the system's requirements, the challenges of designing a distributed system vary. However, most systems will need to deal with heterogeneity, transparency, openness, concurrency, security, scalability, and failure management in general [1]. Each issue is discussed in detail below:

a. Heterogeneity

Users may use the Internet to access services and execute programs through a wide variety of computers and networks. Hardware components, operating systems, programming languages, and functionalities are all examples of heterogeneity. In different programming languages, characters and data structures like arrays and records are displayed differently. These challenges must be handled if programs written in different languages are to interact with one another. The programs of different developers cannot communicate with one another unless they employ the same criteria. For this to happen, standards such as Internet protocols must be agreed upon and approved.

Middleware is used to solve heterogeneity [26]. It's a software layer that hides the heterogeneity of the underlying networks, hardware, operating systems, and programming languages by abstracting programming. Although the majority of middleware is introduced using Internet protocols, which mask differences in the underlying networks, all middleware must deal with OS and hardware variances.

b. Transparency

Transparency is defined as the ability to hiding the isolation of a component in a distributed system from the user and application programmer [13]. So, the program is used as a whole rather than a series of disparate parts. To put it another way, distributed system designers would cover the systems' complexities as much as possible. While no distributed infrastructure provides transparent access to all resources, there are sub-systems based on distributed architectures that provide transparency for specific resources such as disks, directories, or memory. Access transparency, location transparency, replication transparency, relocation transparency, concurrency transparency, and failure transparency [15] are all examples of transparency in a distributed system [13, 27]

c. Openness

The openness of a distributed system is determined by the degree to which new resource-sharing services can be integrated or added and made accessible to users of many clients or applications [13]. If the system's well-defined interfaces are disclosed, it is easier for developers to add new features or replace sub-systems in the future.

d. Concurrency

In a distributed environment, all utilities and applications have resources that can be exchanged by clients. As a result, there is a chance that many clients will want to use a common resource at the same time. In general, concurrent interactions in distributed networks are difficult to define and interpret [16]. An object's operations must be synchronized in such a way that its data remains consistent in a concurrent environment for it to be safe. Common methods, such as semaphores, are found in most operating systems to do this.

e. Security

Cryptography, protected networks, access control, key management (generation and delivery), authentication, and secure community management are all aspects of distributed systems security [28]. The collection of threats emerging from the attack surfaces generated through the resource structure and from the distributed system's functionalities is addressed by distributed systems protection [29]. Vulnerabilities in data flows can endanger the security of distributed system infrastructure, access management mechanisms, data transfer mechanisms, middleware resource coordination facilities, and finally distributed applications that focus on them (*e.g.*, web services, storage, databases, among others). Protection schemas are often used in distributed file systems to avoid data corruption and theft [30, 31].

f. Scalability

According to Neumann, a system is said to be scalable if it can handle the addition of users and resources without a significant loss in performance or increase in operating costs [32]. There are three techniques for scaling: hiding communication latencies, distribution, and replication. Hiding communication latencies are important to achieve geographical scalability. The basic concept is straightforward: aim to stop waiting as long as possible for answers to remote service requests. Another critical scaling method is distribution, which involves breaking down a portion into smaller parts and then dispersing those parts across the device. The Internet Domain Name System (DNS) is an outstanding example of distribution. The DNS namespace is separated into non-overlapping zones and arranged hierarchically into a tree of domains.

Scalability problems often arise in the form of performance degradation; it is generally a good practice to replicate components through a distributed system. Replication not only improves availability but also helps to reduce the load between components leading to improved performance. Getting a copy nearby will also hide much of the communication latency of geographically distributed systems.

g. Failure Handling

Computer systems sometimes fail. When hardware or software malfunctions, programs can produce incorrect results or stop before completing the intended computation. Because of the dynamic nature of distributed systems, providing high fault tolerance, high availability, and reliable access to necessary data is a difficult task [15]. Many techniques should be closely studied to prevent errors from causing service damage. A checkpoint function may be activated at a low level to store information required to restart the server or client process in the event of a crash. These checkpoints, on the other hand, have been expressly controlled by the programmer. More abstract structures, such as atomic actions, may be seen at a higher degree to free the user from the checkpoint and recovery data management [33].

2.1.4 Limitations of Distributed Systems

A distributed system has several drawbacks, including the absence of a global state and the absence of shared memory. This distinguished distributed system computing from database computing, which maintains a constant global state. The design and implementation of a distributed system are both affected by distributed system limitations. The distributed system has primarily two drawbacks, which are discussed below.

a. Absence of a global clock

There are several systems in a distributed system, and each system has its clock. Each system's clock starts running at a different rate. The accuracy with which computers in a network can synchronize their clocks has limits [1]. The clocks are initially controlled to keep them stable, but after one local clock cycle, they are no longer synchronize, and no clock has an exact time. Time is understood to a certain degree of accuracy because it is used in distributed systems for temporal event ordering, gathering up-to-date information on the state of an interconnected system, and process scheduling. Due to asynchronous messages passing, the precision with which processes in a distributed system can synchronize their clocks is restricted. Any clock in a distributed system is synchronized with a more accurate clock, but the clocks become difficult to schedule due to transmission and execution time lags. The algorithms for designing and debugging distributed systems become more complicated without a global clock. Since there is no global state in a distributed system, recognizing the global property of the system is difficult. A distributed system's global state is divided into smaller entities by a large number of computers.

b. Absence of shared memory

Distributed systems have no physically shared memory; instead, each device in the system has its physical memory [34]. Because the computers in a distributed system do not share a memory, no one system will be able to recognize the global state of the distributed system. A process in a distributed system obtains a coherent view of the system, but this view is only a partial view. Heterogeneity [35] is another issue of using distributed shared memory. Processes with different architectures can have different byte orders, word sizes, and so on.

2.2 Access Latency in Distributed Systems

Latency is the time between the start of one process transmitting a message and the start of another process receiving it [36]. In certain applications, end-user response times are critical, so getting data close to users is also essential, and the ability to send traffic to various regions makes this easier. The delay time is the most important parameter in the efficiency of a distributed system. A low delay value means that the device is performing well. The key cause of system delay is remote file accessing; thus, if we can predict which files will be accessed soon and know from where they are accessed, we can replicate them and reduce system delay. Indeed, providing knowledge about potential accesses allows us to better monitor the replication process.

Several algorithms are implemented to reduce access latency for distributed systems. However, as the number of clients grows, so does the number of requests to access information, resulting in extremely high access latency. One strategy for lowering latency is to increase the number of replications, which will improve the efficiency of applications.

In a distributed system the total amount of time or latency that a given job will take can be calculated as the sum of transfer time, queuing time, and execution time. When a user submits a request for a job, the data access latency, which is represented as T, generally divided into three components [5], queuing time of the job, which is denoted as T_{qu}, execution time which is denoted as T_{job}, and transferring time of data scheduling, which is denoted as T_{sch}. Then, we have

$$T = T_{qu} + T_{job} + T_{sch} \qquad (1)$$

The transferring time of data scheduling T_{sch} consists of two parts: transferring time of data scheduling between clusters (parts), which is denoted as $T_{clt\ sch}$, and transferring time of data scheduling between nodes (resource sites), which is denoted as $T_{nod\ sch}$, where $T_{clt\ sch} \geq T_{nod\ sch}$.

Replication strategy has a great impact on T_{sch}. Because of the poor bandwidth performance between clusters, $T_{clt\ sch}$ is overwhelming in data access latency. The unreasonable replication strategy causes the corresponding job to hang up while waiting for data scheduling from another cluster, resulting in a significant increase in data access latency.

Meanwhile, the node where the job is running is inactive, causing resource usage to suffer. As a result, a replication strategy must carefully consider the issue of data scheduling. T_{qu} is affected by replication strategy as well. In general, user requests for jobs and data are unbalanced, resulting in significant differences in computing and Input or Output (I/O) load between nodes. To avoid overloading nodes and a long queue period, the replication strategy must balance loads of each node.

In our study, latency will be reduced because the job will be accessed from cache data, prefetched data, or from the replication closest to it. Transfer and queuing times would be reduced significantly.

Factors of Access Latency in Distributed Systems

Numerous factors influence the efficiency of a distributed system's access rate. Network bandwidth, data availability, the number of replicas, the algorithm used, data transfer time, storage space, energy consumption, policy, and the number of clients are just a few of them.

a. Network Bandwidth

Network bandwidth refers to the ability of a wired or wireless network communication link to transport the most data from one point to another via a computer network or Internet connection in a given length of time [37]. The data transfer rate is defined by bandwidth, which is synonymous with capacity. When a large number of communication channels share a network, the available bandwidth must be shared. Because transferring huge files consumes a lot of bandwidth and takes a long time, network bandwidth availability is the main determinant of access delay in a distributed system [38]. Replication is the most widely used method for ensuring high data availability, minimal bandwidth utilization, greater fault tolerance, and improved overall system scalability [39]. Network delay is becoming the dominant factor in remote file system access as distributed file systems grow numerically and geographically [40]. Concerning this issue, numerous data caching, prefetching and data replication mechanisms have been proposed to hide the latency in distributed file systems caused by network communication and disk operations.

b. Data Availability

Availability refers to the likelihood that a system will be operational at a given time, *i.e.* the percentage of the overall time a device should be operational that it is currently running. It always limits access latency capacity, and if the data sets needed by a job aren't available, the job will hang until a user terminates it. In distributed data storage, replication is the only way to increase data availability [39].

c. Number of Replicas

When a distributed system has to scale in terms of the number of nodes or geographical area, data or file replication is used to improve performance. As the number of processes accessing a data item grows and there is only one server handling the operation, scaling in number is needed. In this case, the output can be improved by replicating the server and splitting the work between them. Scaling in size in terms of geographical area can also include duplication of data. Data transfer time and bandwidth utilization can be reduced by keeping a copy of the data close to the client [2]. Adding redundant replicas to the closest node reduces service time. However, excessive redundancy will lead to more resource use, such as processing time and network bandwidth.

d. Algorithm Used

In a distributed system, methods include a variety of algorithms that are used to perform various tasks. To perform a single task, there are numerous algorithms to choose from, each with different efficiencies. As a result, choosing the best algorithm from the available options necessitates careful consideration to improve the system's performance.

e. Data Transfer Time

The data transfer rate is a standard metric for determining how quickly data is moved from one place to another. When data is transferred too quickly, the cost is reduced. A hard drive, for example, may have a maximum data transfer rate of 480 Mbps, while an Internet Service Provider (ISP) may only offer a maximum data transfer rate of 1.5 Mbps. When the number of users grows, so does the cost of contact between the server and the users [19]. This is due to a shortage of available bandwidth. Cooperative caching enables peers to share data, which can increase efficiency. Proper peer collaboration can also minimize bandwidth usage, which is a valuable resource.

f. Storage Capacity

Every year, computer memory and storage capacity increase, yet they can't keep up with the demand for large data storage. The challenge faced is a decision problem that requires determining how many replicas should be made and where they should be stored. As each storage node in a distributed system has limited storage space and data volume continues to grow, data replication is becoming more difficult [41]. So, placing the replicas on the right site cuts down on bandwidth use and speeds up job execution. Additionally, rather than storing files around several locations, they may be stored in an appropriate site. As a result, storage space is saved. The response time is a critical factor that affects replica selection. Since each site has its capabilities and characteristics, selecting a suitable site from among the several that provide the necessary data is a crucial decision.

g. Energy Consumption

High-performance and scalable computing systems are needed since a large amount of data is collected and processed to control systems. High-performance and scalable systems are made up of a large number of servers and use a lot of electricity, so power consumption must be kept to a minimum [42]. It is necessary to reduce the overall amount of electricity used by computers and networks. Clients submit service requests to servers, and servers respond to clients in distributed networks, consuming a lot of power to complete the process. For high-end distributed systems, power is becoming a critical design constraint.

h. Policy

A policy is a script or definition that is implemented regularly, rather than a single instance of order. Policies are a collection of laws that control the system's action choices [43]. The goal is to be able to adjust the actions of a system without having to re-implement a regulation. These systems must be able to span multiple administrative domains, support replication, security, and redundancy, and address scalability and fault tolerance issues in a distributed environment. They should encourage various organizations to engage in the decision-making process and dynamically update policies. As a result, the policies we used may have an impact on a distributed system’s performance.

i. Number of Clients

The significance and development of distributed systems have increased dramatically. The number of clients who use a system may have an impact on its performance. The fewer the clients have the shorter the delay and the more clients have the longer the delay. Large numbers of concurrent clients are becoming more popular on servers [34]. Servers must handle clients simultaneously to optimize throughput and prevent anyone client from holding up progress due to network I/O blocking and waiting for an unspecified amount of time. So, to manage a large number of clients while reducing latency and improving system performance, a better mechanism is required, and our solution provides this.

2.3 Methods and Algorithms

Communication and I/O latency are the two latency factors that most affect the performance of a distributed system. Many strategies have been introduced by various systems to improve the aforementioned latency points in favor of the system's performance. To improve the performance of a distributed system, it is necessary to pay more attention to selecting better methods and algorithms. In distributed systems, replication is a commonly used method for ensuring high availability [38], fault tolerance, and good performance [24]. The following are some of the basic methods and algorithms that are used in distributed systems [5]:

2.3.1 Caching

A cache is a temporary storage location for files or parts of files that are filled with a copy of the downloaded content on-demand and can be deleted [7]. The aim of caching is to make it easier for a client (possibly a different client from the one that first downloaded the file) to access the same information more quickly the next time. Nodes in cooperative caches allow the use of data that has been stored for them by other nodes. It is purely coincidental that they come across information that was cached by another node.

Caching is a method for reducing response time that involves keeping copies of popular material, such as Web documents, in a local cache, a proxy server cache close to the end-user, or even on the Internet. However, the benefit of caching reduces as Web documents get more complicated [44]. Because the majority of today's Web caching systems are passive, a cached page may be outdated at the time of its request. Caching technology is used in many areas of the Internet, including CPU cache, disk cache, Web cache, and DNS cache. Documents such as Hypertext Markup Language (HTML) files, images, and other media are temporarily stored in the web cache. The system keeps a copy of the content that can be retrieved from a cached copy of the same request. Many different systems will benefit from the Web caching process.

A website may be cached and forwarded to the client by a search engine.

The overall system efficiency improves when a cache is used. Database servers, web servers, file servers, and storage servers all use caching [20]. Since the cache capacity is small, we are unable to store all of the requested content. We must use algorithms that identify old content in the cache that needs to be replaced. Cache eviction policy or caching policies are common names for these algorithms. The caching algorithm should overwrite material that will not be used soon.

The cache in the distributed file system may be on the client-side as well as on the server-side. On the client side, the cache stores content that has been downloaded by a user running a client program. On the server-side, the cache holds the data that has been submitted by most users [45]. A distributed file system provides two basic operations: data reading and writing. Because of the read performance of physical media, the read performance of a distributed file system [46] is extremely high, and read operations do not need to consider data protection.

The cache eviction policy is a feature that allows space for new files by releasing file data items in the cache when a file set reaches its soft quota. Eviction is the method of releasing blocks. The most commonly used caching policies in distributed systems are as follows [47].

a. First-In-First-Out (FIFO): ensures that the entry with the longest time in the cache is evicted first. It differs from other policies in that it ignores the order in which entries are accessed. The oldest data in the cache has been chosen to be replaced. The information in the cache is arranged in a queue. The latest information is pushed to the back of the line. When the cache fills up and new data arrives, the data at the top of the queue is replaced.

b. Least Recently Used (LRU): ensures that the entry that hasn't been read in the longest time is evicted first. This algorithm keeps the cache's most recently used entries near the top. This policy is suitable for the majority of caching scenarios. The LRU algorithm, which employs easy and quick cache updates, is a basic and commonly used caching scheme. However, due to cost and benefit considerations, LRU cannot provide flexibility in object-choice due to size, transport path from the origin, or other preferences from the content providers' or users' perspective [48].

LRU is considered to outperform FIFO in terms of performance [49]. The LRU substitution policy takes advantage of the data's temporal localization [50]. Temporal locality ensures that data units that haven't been accessed in a long time won't be used soon and can be replaced as the cache fills up. A priority queue is commonly used with LRU. The last access timestamp takes precedence. However, since the policy does not know which data blocks will be accessed; it will evict data blocks that will be accessed soon. It can also retain other unnecessary data blocks in memory in the meantime.

c. Least Frequently Used (LFU): When the cache overflowed, the LFU caching algorithm removes the LFU cache block. In LFU, we check the old page as well as the frequency of the new page; if the new page's frequency is greater than the old page's, we can't delete it; if all the pages have the same frequency, we use the last process, FIFO, to remove it. The LFU algorithm is also interesting because it necessitates a more complex replacement method. However, it has several flaws when it comes to adding new material or removing older ones [51]. Furthermore, the LFU's complexity makes it more difficult to model and analyze. There is a counter for each data unit that is incremented any time the data unit is accessed [50]. The downside of this strategy is that data units in the cache that have been accessed several times in a short time stay in the cache and cannot be replaced, even though they are never used again.

d. Most Recently Used (MRU): this algorithm works well in cases where an entry's age determines how likely it is to be accessed. In contrast to LRU, the Most Recently Used substitution policy operates in the opposite direction. The most recently accessed data units are replaced by MRU. MRU is appropriate for a file that is scanned in a looping sequential reference pattern repeatedly [52].

e. Random Eviction Policy: It is a simple replacement policy that selects data to be replaced based on a random selection [50]. This replacement strategy is simple to enforce. It is mostly used for testing and debugging.

f. Frequency Based Replacement (FBR): the advantages of both LFU and LRU policies are combined in a substitution strategy [53]. The cache is divided into three sections by FBR: a new segment, a middle segment, and an old segment. Only data units in the middle and old segments are divided into parts based on their recency of increase. When a replacement is needed, the policy selects the data unit with the lowest hit count from the old segment.

g. Least Recently Frequently Used (LRFU) is a new caching algorithm that attempts to combine Recency and Frequency (CRF) to overcome one or more LRU and LFU drawbacks [54]. The policy often evicts a block with the lowest CRF value. This value represents the probability of the unit being referenced soon. At the same time, this replacement algorithm uses both LRU and LFU replacement policies. LFRU has been tested in database systems and is suitable for use.

The LFU and LRU two substitution policies are combined in the LRFU policy, which converts from the LFU to the LRU by changing the parameter λ from 0 to 1. However, the LRFU λ (changing parameter) value is fixed, making it an ineffective substitution policy in practice.

h. **Improved LRFU**: Yang [55] proposed an **improved adaptive policy** based on the Recency and Frequency algorithm that can dynamically adjust the λ value of the LRFU to obtain the appropriate replacement strategy for the given situation.

2.3.2 Prefetching

Prefetching is a method of speeding up fetch operations when they start with a result that will be needed soon. This is usually performed until it becomes known that it will be required. However, prefetching data may not be used and will waste time. Pre-transfer prefetching (prefetching) approaches have proved their capacity to considerably minimize long wait times as a direct alternative [8]. A prefetching scheme necessitates predicting the documents that are most likely to be accessed shortly. Network delay is becoming the dominant factor in remote file system access as distributed file systems grow both numerically and geographically [40]. To hide the latency in distributed file systems caused by network communication, several data prefetching algorithms have been proposed.

There are two types of prefetching techniques currently available: predictive and informed. Informed prefetching techniques allow preloading decisions based on applications' potential access clues [55], while predictive prefetching methods forecast future I/O access trends based on historical I/O accesses of applications [22].

The predictive approach eliminates the need for applications to include clues and can handle other issues. Appleton proposed automatic prefetching, a new approach for reducing file system latency that uses a heuristic-based algorithm to analyze the experience of past access events to predict future access requests without the need for application interaction [56].

Informed prefetching uses hints from the application to decide what data should be read ahead of time, believing that file system output can be enhanced by using the application's information [57]. However, if there are no sufficient hints from the applications, these prefetching mechanisms cannot make correct decisions, and the incorrect predictions hurt system results.

Prefetching data at the exact moment it is needed is a difficult task. Prefetching data should not be done too early or too late. After analyzing disk I/O traces, several prefetching techniques have been proposed in succession to read the data on the disk in advance [58]. These are some of the basic prefetching scheme:

a. Read Ahead Prefetching Scheme: Prefetching data in next neighbor blocks on storage servers using a read-ahead prefetching scheme can change the amount of prefetched data depending on the amount of data already requested by the application. This mechanism is straightforward and a successful scheme for improving I/O efficiency [59].

b. Signature-based Prefetching Scheme: An initiative data prefetching method is presented by Liao *et al.* [60]. The proposed schema analyzes I/O tracks to predict future I/O requests so that storage servers can retrieve data easily, and then present the perfected data to the requester for future possible uses. The disk I/O access operations are first modeled, and then access patterns are classified as sequential or random. Then, they implemented two prediction methods: chaotic time series prediction and linear regression prediction, to systematically forecast future access based on access patterns.

2.3.3 Replication

Replication refers to the process of a node purposely placing information in multiple locations [7]. Replication aims to reduce the load on the server by offering mirror sites where the same content can be acquired. As a result, the client can experience a faster access time, due to the lower load on the server rather than proximity. A replica store is not typically removed to make way for more currently in-demand data objects, but objects that become outdated may be evicted.

Data replication is a very useful technique in distributed systems for improving data availability and reliability, reducing user waiting time, improving fault tolerance, and minimizing bandwidth usage by providing several replicas of data on different nodes [61, 62].

As distributed systems scale across various geographical areas, providing full data availability and fault-tolerant data access is a difficult challenge. To address this issue, data must be replicated across a large number of sites. By producing several copies of the same data. Data replication is a realistic and efficient way to improve the efficiency and reliability of distributed systems.

There are two types of data replication: static and dynamic [63]. The static data replication algorithm replicates data based on prior information and ignores users who modify their actions. As a result, they are unable to adapt to changes in the network, user behavior, or access patterns. Static replication is simple to set up, but it is rarely used because it doesn't allow data replication during job execution. The advantages of static replication methods include lower overhead and faster job programming.

The dynamic replication algorithm creates and deletes replicas based on network and user access patterns, making them more appropriate and efficient. Because dynamic replication methods do not produce replicas of all files, the required data files must be dynamically adjusted. Three key issues are addressed by the dynamic data replication algorithm: (i) what is the file that needs to be replicated? (ii) What is the best location for data replication? (iii) When does replication take place? [64]. However, there are also disadvantages: having too many copies does not always increase data accessibility because reserve disbursal occurs frequently, and it is once again difficult to put new replicas on different nodes on time. More attention must be made to selecting better replication algorithms to improve the system's efficiency. A variety of data replication algorithms are used in distributed systems. Some of the major dynamic replication algorithms are the following.

a. Least Frequently Used (LFU): When a required file is not available and storage is not enough, LFU replication continuously replicates data and deletes the least frequently accessed file if there is no enough space. In an economic algorithmic program, replication happens regularly, and if the replica placement does not have enough space, the value of each entry data is determined, and if the file has a lower frequency value, it is deleted.

b. Least Recently Used (LRU): is also used to continuously perform replication, and if there is no enough space, the files that have been used for the least amount of time in the recent past are removed. In comparison to LFU, LRU provides superior efficiency.

c. Bandwidth Hierarchical Replication (BHR): uses network-level localization to its advantage. BHR duplicates common files in a requester region site connected to the task execution site through a quicker link [65].

d. Modified Bandwidth Hierarchical Replication (MBHR): Sashi and Thanamani improved the BHR strategy and proposed the MBHR [66]. In MBHR, the grid network is organized into many regions with nearby sites. In the area header, BHR stores file access history and the number of times a file has been accessed and repeats the most frequently accessed file on the site. For the longest duration, the file can be accessed locally. As a result, as compared to BHR, the Mean Job Execution Time and Network Consumption are lower.

e. Dynamic Hierarchical Replication (DHR): Mansouri *et al.* [67] proposed the DHR algorithm. It uses the Hierarchical method to choose the best replica for the requester task. When measuring replica access cost, it takes into account the time spent waiting in storage and the time spent transferring data.

f. Modified Dynamic Hierarchical Replication Algorithm (MDHRA): Mansouri *et al.* [68] proposed another algorithm called MDHRA. In the replica replacement phase, MDHRA used effective parameters such as the last time the replica was requested, the number of accesses, and the size of replicated data. The selection of replicas is based on the transfer time.

g. Prediction and Prefetching Replication Algorithm (PPRA): PPRA is two dynamic data replication algorithms [69]. PPRA uses a statistical prediction process to predict the future location and appropriate place where the future file request is likely to occur. To limit the number of searches and improve performance, this method uses a hierarchical search. The experimental results demonstrated that PPRA outperformed well-known algorithms such as No replication, LRU, BHR, and MBHR in terms of Mean Job Time and Effective Network Usage metrics [70]. Other replication mechanisms exist, and which one is optimal and how it is used is dependent on the user's behavior.

h. Fuzzy Replication Algorithm (Fuzzy_Rep): is a new replication algorithm that is proposed by Beigrezaei *et al.* [64]. A fuzzy replication algorithm could improve the modified BHR algorithm. It selects the best replication location using a fuzzy inference scheme with three inputs and one output. The fuzzy rule is used in this inference method to determine a value for each location, showing how valuable it is. It is more precise than the updated BHR.

2.4 Summary

In this Chapter, we have reviewed the literature on the distributed system and its access latency. Moreover, the major characteristics, challenges, limitations, and access latency factors in distributed systems are presented. The various types of methods and algorithms used in distributed systems along with their objectives are also discussed.

Furthermore, concepts that could facilitate the formulation of the solution domain are acquired. Literature revised and summarized in the early sections of this has a greater contribution to the effort to know a distributed system and its access latency.

3. Related Work

For clear understanding, we group the researches done into two. The first one is single solutions that can use caching, prefetching, or replication. The second is a hybrid of two methods that use caching and prefetching, caching and replication, or prefetching and replication to increase performance. In this chapter, we will review each paper in detail by considering the objectives, methods used, achievements, and gaps. Finally, we summarize the chapter by clearly showing the gaps.

3.1 Single Method Solutions

3.1.1 Caching Method

Feng *et al.* [71] proposed an adaptive multi-level caching strategy for distributed database systems (DDBS) that dynamically changes adjusts cache resource allocation and cache size in different data nodes according to real-time access. Many resources could be allocated to hot data nodes in this way, allowing queries to run faster and eliminating the bottleneck of latency-sensitive services. The query trimming algorithm breaks the query down into two parts. Moreover, they employ the Replacement for Semantic algorithm, which is faster than the LRU algorithm. The experiment shows that in a DDBS environment, the adaptive multi-level caching strategy is very successful. However, it will need to improve the query trimming algorithm to improve query matching speed, as well as a merging strategy to address the problem of fragment generation caused by query trimming.

Anja Feldmann *et al* [72] presented web proxy caching in heterogeneous bandwidth environments. Requests to cacheable and previously accessed documents can be reduced by caching documents in browsers and proxies. Any bandwidth savings obtained from a proxy cache can be negated by the bandwidth mismatch between the clients and the proxy, as well as the proxy and the Internet. Additionally, due to partially transferred data for aborted requests, bandwidth usage may increase. The advantages of using persistent connections between clients and the proxy may outweigh the advantages of caching documents in heterogeneous networks. The solution to the above limitations is to use the proxy in a dual role as a data cache and a connection cache. In comparison to a non-caching proxy, this approach achieves an 8% reduction in average user-observed latency in a low bandwidth environment using an infinite-size data cache. However, simply caching connections improves average latency by up to 25%. Data and connection caching combined will reduce latency by up to 28%. In a high-bandwidth system, data caching reduces mean latency by 38%, connection caching by 35%, and the combination of the two by 65%. But this work does not consider hit ratios, bandwidth utilization, and user-perceived latency.

Snehal *et al*. [19] proposed a mechanism of using a cooperative caching technique which is built with the help of a Mobile P2P network. The proposed solution aims to increase the probability of cache hits. The rise in the number of people who use handheld devices has boosted multi-modal content sharing between them. If the user requested data is not found in the local cache, P2P Cooperative Caching will check if its connected peers have the requested data in their cache. If the requested data isn't in any peer cache, it must be fetched from the server. Data caching lower access time for such contents since it can be immediately fetched from the local or peers cache that reducing network access rates.

Authors in [12] described a distributed caching, prefetching, and replication method and system. As the authors stated, the invention provides an autonomous, the distributed and transparent caching scheme that takes use of the fact that document requests naturally build a routing graph as they travel via a computer network from a client to a specific document on a certain home Server. Client request messages are directed up the graph, as they would be in the absence of caching, to the home Server. Cache Servers, on the other hand, are located along the path and may intercept requests if they are functional. The cache server should be able to insert a packet filter into the router associated with it, as well as a proxy for the home Server from the user's perspective, to service requests in this manner without deviating from standard network protocols. Cache servers can help Service users by caching and deleting documents based on their current load. The authors concentrate on a distributed caching schema in which the cached record is replicated from one cache server to another. However, the authors did not combine the three approaches at the same time.

3.1.2 Prefetching Method

Dan Duchamp [73] developed a new method for prefetching Web pages into the client cache called a prefetching hyperlink. Clients submit reference data to Web servers, which collect the data in near-real-time and then distribute it to all clients. Based on knowledge of which hyperlinks are commonly popular, the information indicates how frequently hyperlink URLs embedded in pages have been previously accessed relative to the embedding page. This work enhances prefetching by lowering latency by 52.3% and reducing network bandwidth waste by 24%. However, the information exchange between clients and servers makes implementation more difficult.

Gwan *et al*. [23] proposed a novel method called appointed file prefetching, in which the user or system administrator may decide how to file prefetching should be performed. This specifies the file prefetching language (AFPL) that the user and system administrator may use to tell the system when and how to perform desired prefetching. AFPL prefetches instructions in two stages: first, it selects the appropriate files, and then it specifies when to prefetch them. The experimental results show that, in most cases, the waiting time for remote file fetching is decreased by 30% to 90%, and the hit ratio increases by 6% to 18%.

3.1.3 Replication Method

Mustafa *et al*. [39] presented an efficient replicated data access technique for large-scale distributed systems. This method is a new technique for keeping duplicated data on distributed computing systems that is low-cost and high-availability. The goal of this study is to allow remote users to safely access the same information from many different places at the same time. For sustaining replicated data across distributed systems, a new quorum-based protocol outperforms Grid-based and Read Only Write All (ROWA) protocols. In comparison to current replica control methods, the proposed approach provides high data availability, minimal bandwidth usage, increased fault tolerance, and enhanced overall system scalability.

Carman *et al.* [74] presented dynamic replication methodologies for the interaction of different optimization units at each node/site in the network, based on economic models. The primary focus is on optimizing local resources and achieving global optimization via emergent marketplace behavior. Given a finite amount of storage resources, the goal is to reduce the overall cost of file access on the Grid. It focuses on a specific area of optimization and is largely concerned with the problem of optimizing data replication in a Grid system, that is, determining when and where to produce and remove data file copies.

Some studies on the integration of job scheduling strategies with dynamic replication strategies have been proposed, in which the Integrated Replication and Scheduling Strategy

(IRS) [75] has outstanding results. The main aim of this study is to create and test an iterative replication and scheduling method. The data replication is done in an asynchronous timer-controlled procedure that considers job history and data access patterns and is essentially based on the concept of predicted data file delay and a greedy optimization technique.

Sonali *et al* [76] proposed a new dynamic replication algorithm called MBHR. This algorithm tends to present a dynamic algorithm on the data grid system replication. A data grid is a service cluster or structural design that allows users to access, change, and transfer large amounts of geographically distributed data. This algorithm needs a variety of parameters to locate a suitable website where the file will most likely be needed again in the future. When compared to other strategies like LRU, LFU, and BHR, this strategy is also appropriate for grid sites with limited storage sizes, prevents unnecessary replication, is used to minimize access cost, and takes less execution time.

The fuzzy logic method [5] can also be used for dynamic replication to reduce access latency. This includes replica selection, layout, and replacement. Fuzzy Logic System Dynamic Replication (FLSDR) always selects the best replica based on a hierarchical data scheduling transferring time, and places replicas on the best node based on the spatial-temporal proximity of data access, based on this statistical analysis. When the available storage space is insufficient, the algorithm determines and calculates the replica life value (RLV) and deletes the replica with the lowest RLV. In comparison to the other techniques, FLSDR has a shorter mean job execution time, more data scheduling amongst clusters, and a higher number of replicas, as well as a higher computing resource utilization. As a result, FLSDR can greatly reduce access latency.

3.2 Hybrid of Two Methods

3.2.1 Hybrid of Caching and Prefetching

Cao [77] examined the algorithms that individual mobile apps can use to make caching and prefetching decisions, as well as the methods that mobile operating systems can employ to facilitate application-controlled caching and prefetching. The benefits of data caching and prefetching for mobile devices include mediating the data cost differential between WiFi and cellular by prefetching (preloading) data over Wi-Fi and reducing mobile web and application latency by caching data that is crucial to user perception. Mobile device speed may be enhanced by implementing intelligent caching and prefetching of data methods in mobile applications, which can improve the user's mobile experience significantly. The proposed mechanisms for mobile operating systems assist mobile operating systems in controlling mobile apps by allowing them to query the current per-byte cost of data and be notified when that cost changes considerably. It keeps track of how an application uses its cache storage and calculates the cache storage's return on investment for each application. So, for mobile devices, caching and prefetching are attempted approaches for reducing remote data fetches and improving latency.

The Integrated Prefetching/Caching (IPC) for the multimedia server is presented in [70] to take advantage of both prefetching and interval caching while applying dynamic threshold values to overcome the difficulty of employing either caching or prefetching. Incoming streaming requests are scheduled by the IPC to maximize cache space and disk bandwidth consumption. As a result, as system resources are added, the IPC can continue to enhance performance without becoming overloaded. It is better to combine caching with prefetching than to use either caching or prefetching separately.

3.2.2 Hybrid of Caching and Replication

The authors in [78] used the combination of caching and replication for a system like responsive web service. For many years, object replication and caching have been utilized separately in distributed systems. Both strategies allow for the generation and maintenance of numerous copies of an data (a resource), but for opposite reasons: replication for availability and caching for performance. Replication or caching are used by applications that demand one or the other. However, because the two types of procedures differ sufficiently, attempting to create one using the other may result in inefficient implementations. The paper describes a basic architecture for integrating object replication with caching so that applications can benefit from both availability and performance at the same time. Because this approach allows copies to be organized into distinct consistency domains. applications can continue to use existing replication and caching mechanisms. Clients can be served by domains with varied levels of consistency, performance, and availability. This is beneficial to clients that use these consistency domains without understanding how the domain maintains its internal and external consistency.The contrasts between the two make developing an integrated approach a difficult challenge.

Wietrzyk *et al*. [79] proposed a responsive web service architecture that uses distributed replication and caching to solve the inefficiencies associated with the distributed business logic location in the simpler client-server approach. It also helps in the resolution of issues such as overloading, which creates performance problems. This strategy involves decisions on the number of nodes in each Web group as well as the distribution method for each Web group. Web pages are dynamically generated based on the current state of a company's operations, such as product prices and inventories, which are recorded in database systems. This feature necessitates the use of electronic commerce. Web servers, application servers, and database systems are deployed and integrated at the backend. Web document replication can increase the Web service's performance and reliability. Web clients can utilize server selection algorithms to choose one of the duplicated servers that are closest to them and save the web page on the client-side for later usage, reducing the Web service's response time. This work does not improve the data distribution method to enable dynamic data reorganization, and more research into access patterns and resource requirements is needed.

A dynamic data replication algorithm called prefetching and PPRA in data grids is proposed in [80]. PRRA uses prefetching for replication to prepare files before they are needed at the requesting sites. Based on file access history, it forecasts future requirements. Indeed, it extracts and prefetches related files to the requester grid. As a result, the site could have local access to the required file. Users of the data grid, who are geographically dispersed throughout the grid, require this information. As a result, one of the most fundamental issues in data grid networks is guaranteeing efficient access to scattered data. The most prevalent ways of overcoming this difficulty are data replication algorithms. To save access time, transfer costs, and bandwidth consumption, they distribute several copies of a file on the appropriate site. The authors employed a data grid simulator called OptorSim to test the proposed algorithm's performance. In comparison to No Replication (NoRep), LRU, LFU, BHR, and MBHR algorithms, simulation results demonstrate that PPRA can provide improved average job execution time and bandwidth utilization.

3.2.3 Hybrid of Prefetching and Replication

Mansouri *et al.* [11] proposed a new dynamic replication technique called Prefetching-aware Data Replication (PDR), which analyzes file access history to isolate the correlation of data files and prefetches the most common files. The dependencies between all files are first stored in the dependency matrix using this strategy. The most popular group files are then determined based on the total average of file accesses. Finally, because storage space is limited, a Fuzzy-replica replacement approach is proposed, in which the value of each replica is decided by the number of accesses, replica cost, last access time, and file availability. As a result, the PDR strategy deletes the replica with the lowest value to make space for the new file. PDR method deletes the replica with the lowest value to make way for the new group of replicas and then tries to find the most popular document with strong correlations and replicate them in the desired location.

3.3 Summary

The works, we reviewed use caching, prefetching, dynamic replication, or a combination of them to reduce access latency for a different area of the distributed system and they tried to solve many problems. As we saw in the related work single method solution are better in reducing the latency than directly access from a remote one. But still, the combination of the two methods gives a better solution than the single solution in reducing access latency for the distributed system. Taking the above issues into consideration, we are seeking to develop a model that integrates the three methods to improve the performance of a distributed system. The three methods will be integrated and used one after the other in one path of accessing documents. First, the server will replicate dynamically based on the number of requests coming from clients. For the first time, the client will access or request the nearest replica and stores it as cache data near the client system. Based on the first or previous access history, the system guesses and prefetches the related data and store near to the client system. Later on, the client first checks from the client cache if there is. Else it will check from the prefetched document. If it is not there still it will access from the nearest replica. This way the access latency may decrease and performance can be improved.

4. The Proposed Model for Integrating CPDR

This Chapter presents the design of the integration of Caching and Prefetching on Dynamic Replication (CPDR) for distributed systems. The Chapter is composed of two parts namely, Design Considerations and System Architecture. The design considerations section presents various determinants for high performance distributed systems using an integrated CPDR algorithm. In the system architecture section, components of the proposed model along with their detailed description and algorithms are presented.

4.1 Design Considerations

This design aims to propose a general model for distributed systems using the integration method of caching, prefetching, and dynamic replication and can yield an optimized latency time and improve the performance of the system. The main factors considered during the design are summarized below.

a. Performance

Performance is a critical consideration in our work since we try to decrease the latency of the system using integrated CPDR methods. When the number of users, systems and the distance between them increases rapidly, using caching, prefetching and dynamic replication is a basic option to increase performance. Our proposed system has cache, prefetch and replication parts as options to access data instead of wasting time by accessing from a remote server. If a given request accesses data from one of these three parts, it will increase the performance of the system by decreasing the transfer, queuing and accessing times from the remote server. We consider performance as the main factor when designing our system in terms of time and storage.

b. Availability

Another design consideration is availability. In distributed systems, it plays a great role to increase the performance of the system because a system has a high geographical coverage area, which has network and power failure problems. To solve this problem, the system should have the option of accessing the data from another place. This can be done using replication of the system to different places and store data either on the cacher[1], prefetcher, or replicator

[1] Note that we are using the terms casher, prefetcher and replicator as will be defined later in Section 4.2.

to increase availability since they do not need high computational power and time. Our system design considers this issue to increase the availability of the system.

c. Computational Time

In distributed systems, cost and time factors are at the center of resource delivery. In the new model, computation time could be decreased using caching, prefetching and replication since accessing data from cache and prefetch decreases transfer time and queuing time from the remote server. Also, since these parts have small data and a smaller number of users as compared to a remote server, the computational time decreases rapidly. Any job that computes without queuing decreases the computational time. If we can decrease latency, we decrease the computational time of the system that could result in a decrease in the cost of jobs. Our system considers this factor and accesses data from the cache, prefetched data or the nearest replica without much queuing.

d. Scalability

The system components are designed and managed to interact independently. It is carefully designed in a way it can yield high cohesiveness between components in each part and low coupling among major parts *(i.e.*, Cacher, Prefetcher and Replicator). This facilitates the scalable implementation of the architecture as each component can be distributed across different places to increase the scalability of the designed system as the number of users increase because the parts are loosely coupled.

e. Resource Utilization

In distributed systems, the major resources are time, storage, bandwidth and power/energy. Time can be optimized since the request is accessed from the nearest part. Storage can also be optimized since it automatically deletes data when storage becomes full. The energy consumption of underutilized resources, particularly in the case of distributed environment, accounts for a considerable amount of the actual energy use because the resources are accessed either from the cacher, prefetcher or from the nearest replica. Furthermore, a resource allocation strategy that takes into account resource utilization leads to better energy efficiency. Selecting an effective method and algorithm plays a great role to increase resource utilization and reduce energy consumption.

4.2 System Design

In this section, we present the architecture of the integrated CPDR algorithms used to access or evict data from their components. In addition, the communication method is also presented. The proposed architectural components are explained in detail.

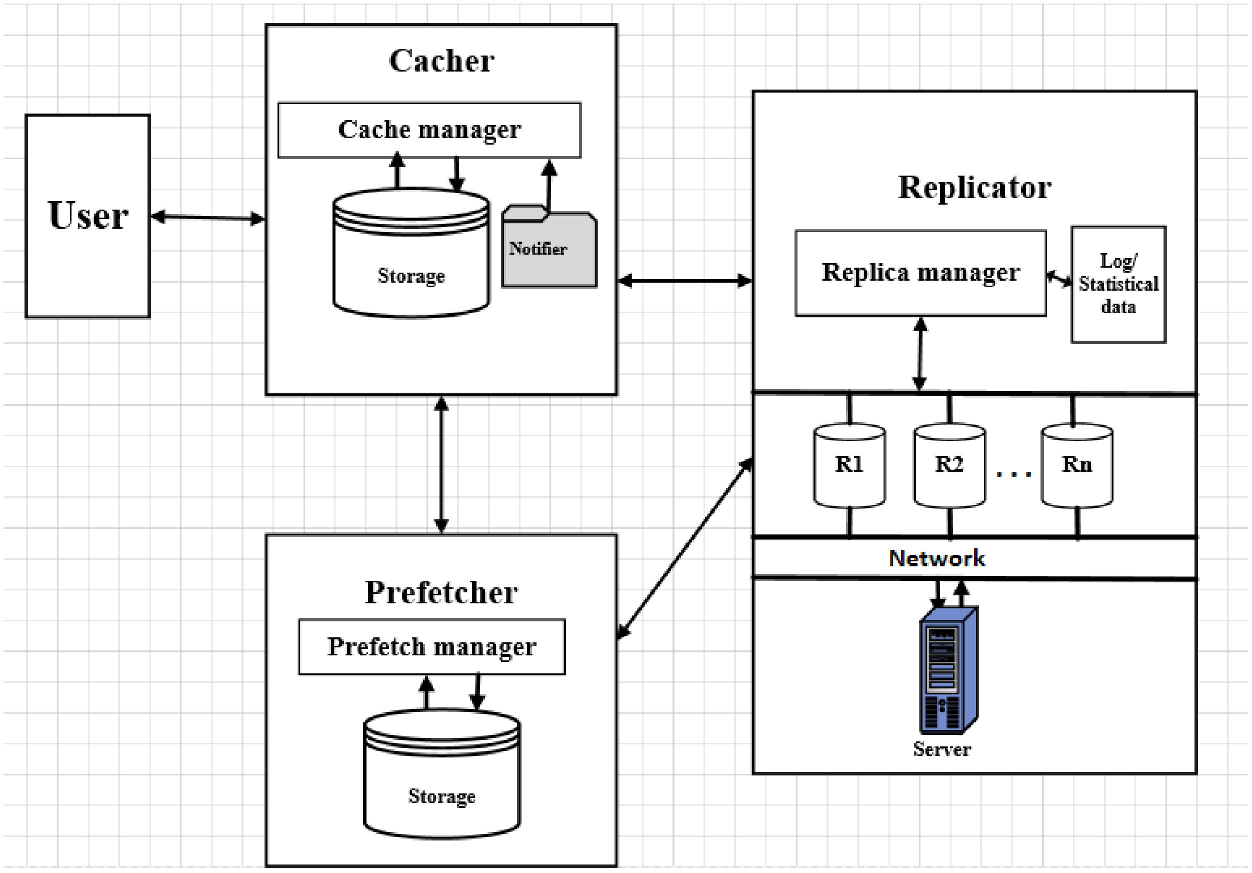


Figure 4.1: The General Architecture of the Proposed System

The general architecture of the proposed system is shown in Figure 4.1. The user requests the cacher for data and if the requested data is available in the cache storage, the cacher sends the response to the client. Otherwise, the cache manager checks the notifier file whether the prefetcher is active and if the ID of the requested data is found in the prefetcher. If the prefetcher is active and the request ID is found, the cache manager requests the prefetcher and the response is sent by the prefetcher to the cacher storage and then to the user. Also, the prefetch manager sends the data ID to the cacher and is stored at notifier when new data is fetched and stored at the prefetch storage. Else the request is sent to the nearest replica and the response is sent to the cacher, which is also stored at the cache storage and transferred to the user. The prefetch manager requests the replicator and the replica manager communicates either with the file of statistical data (log file) or replica and transfers the prefetched data from the replica or log file to the prefetcher. The major components of the proposed system and their functions are explained below.

4.2.1 Cacher

The cacher consists of a cache manager, storage and notifier. The storage component is used to store recently accessed data coming from the replicator or prefetcher that was used in the latest access. Since the store has a limited size, an algorithm is used to evict data that is not frequently required and this activity is controlled by the cache manager. The cache manager is the other component that is used to control all activities performed by the cacher. The cache manager has an algorithm that is used to control the data in the storage and decides which data should be stored and which data should be evicted. Another function of the cache manager is checking whether the prefetcher is active or not by reading the notifier's status. The third component of the cacher is the notifier which has two basic pieces of information about the status of the prefetch. The information is about the current prefetched data, created by the prefetcher and sent to the cacher which is useful for later use. The first information is the status of the prefetch whether it is active or not. The second information is the ID of the prefetched data which is important to know whether the requested data is found in the prefetcher storage or not before going to the prefetcher component. So, when the request which is coming from the user is not found in the cache, the cache manager communicates with the notifier whether the status value is active or not and whether the requested data ID is in or not. If the status is inactive or the request ID is not in the notifier, the request will be sent to the nearest replica, not to waste time by requesting the prefetcher.

In distributed systems, many cache eviction algorithms are widely used [81]. The most extensively used method is LRU, which is popular because of its O(n) time complexity and strong approximation to the behavior required by most applications. The preferred algorithm is LRFU [54], which could eliminate the disadvantages of the LRU and LFU. This approach replaces the smallest of the Combine CRF values [54] with a weight function. An improved adaptive strategy based on a recency and frequency algorithm that can dynamically change the λ value of the LRFU to select the best replacement policy [55] that is either LRU or LFU is used for our system as given in Algorithm 4.1.

In Algorithm 4.1, `H` is the heap data structure, t_c is the current time, and `LAST(b)` and $\mathrm{CRF}_{last}(b)$ are the time of the last reference to block `b` and its CRF value at that time, respectively. The algorithm first checks whether the requested block `b` is in the cache. If it is present, the algorithm recalculates its CRF value, updates the time of the last reference, and, if needed, restores the heap property of the subhead rooted by `b`. If the block is not in the cache, the missed block is fetched from the remote server and its CRF value and the time of the last reference are initialized. Then, the root block of the heap is replaced by the newly fetched block and the heap property is restored. In addition, if the replaced block is dirty, it is written back to the disk (removed). Finally, the current time t_c is incremented by one to reflect the progress of the virtual time due to the reference.

The CRF value is calculated by the weight function given in Equation 4.1 [54, 55].

$$C_{tpresent}(b) = \sum_{i=0}^{k} F(x) \qquad \text{equation 4.1}$$

where k is the reference time of block b. $F(x) = (1/p)^{\lambda x}$, x is the difference between the current time and the time of a reference in the past, $C_{tpresent}(b)$ is the CRF value of b block in the current time. λ is the time interval between the previous and current access, and p is the locality and frequency balance parameter.

Algorithm 4-1: LRFU Cache Replacement Algorithm

```
Input: heap, block b, storage
Output: List of blocks and CRF value,last b
Begin
   If b is already in the buffer cache
      $CRF_{last}(b) = f(0) + f(t_c - LAST(b)) * CRF_{last}(b)$
      $Last(b)=t_c$
      Restore (H, b)
   Else
     //fetch the missed block from the disk
     $CRF_{last}(b) = f(0)$ //this the initial time of the block.
     $Last(b)=t_c$ //Current access time of the block is assigned to
     last(b)
     victim = ReplaceRoot(H, b)
     If victim is dirty
```

```
            write-back the victim to the disk
        Endif
    $t_c$ = $t_c$+1
    Endif
End
```

The `Restore(H, b)` and `ReplaceRoot(H, b)` functions that are used in Algorithm 4.1 are given below.

```
Restore(H, b)
Begin
    If b is not a leaf node
        //let smaller be the child that has the smaller CRF value at
        the current time
        If f($t_c$ - LAST(b)) * $CRF_{last}$(b) > f($t_c$- LAST(smaller)) *
            $CRF_{last}$(smaller)
            swap(H, b, smaller)
            Restore(H, smaller)
        Endif
    Endif
End
ReplaceRoot(H, b)
Begin
    victim=H.root
    H.root=b
    Restore (H, b)
    return victim
End
```

Algorithm 4.2 is an improvement or extension of Algorithm 4.1. The LRFU algorithm (Algorithm 4.1) would be unable to dynamically adapt to the practical situation. The value of λ can be dynamically modified using an improved LRFU algorithm (Algorithm 4.2) [82]. This algorithm counts the block information with longer time access and the CRF value of the evicted block using a FIFO queue named Lout, with the queue length set to C. Lout. The recently evicted block's block metadata and CRF value are saved in the FIFO queue. If the corresponding recorders in the Lout queue are found successively when accessing the new block, and the times of the block in the queue are greater than the times of the block out of the queue, these cases indicate that the block being accessed more times is more likely to be accessed again, so lower the λ value to make the replacement strategy more purpose to the LFU algorithm. Increase the λ value to make the replacement strategy closer to the LRU algorithm if the new block is not in Lout accessed successively and the times of a new block not in Lout are larger than the Lout times, which suggests that the block accessible first is the least likely to be accessed again. Furthermore, finding the requested block in the cache means that the block that has been accessed previously is more likely to be retrieved again.

The λ value is dynamically adjusted as follows (Algorithm 4.2). In Algorithm 4.2

- `InTimes` holds the count information of the block when the block is found in `Lout` or cache.
- `SuccTimes` holds the succession information of the block when the block is found in `Lout` or hit last time.
- `OutTimes` holds the count information of the block when the block is neither found in `Lout` nor hit in the cache.

Algorithm 4-2: An Improved Adaptive LRFU Algorithm

```
Input: block b, storage
Output: λ value and List of replaced blocks

Begin
   If ((new block is recorded in Lout) or (hit in cache))
      Count times with a parameter InTimes
      If (this block is in Lout or hit last time)
         count the successional times with SuccTimes
      Else clear SuccTimes
      Endif
   Else
Count times with another parameter OutTimes
   Endif
   If ((InTimes>M_Times AND InTimes/OutTimes>M_Ratio) OR
         (SuccTimes>M_Times AND successional in the Lout))
      Modify λ to λ/M_Value
        // Clear all parameters
   Endif
   If ((OutTimes>M_Times AND OutTimes/InTimes>M_Ratio) OR
         (SuccTimes>M_Times AND successional out the Lout))
      Modify λ to λ* M_Value
      If the value is more than 1,regarded as 1// the value of λ[0-1]
         // Clear all parameters
      Endif
   Endif
End
```

4.2.2 Prefetcher

The prefetcher has two components. The first component is the prefetch manager which controls all activities. This component has an algorithm that is used to decide which data should be prefetched from the replicator by accessing the statistical data or log file component of the replicator. The second function of the prefetch manager is like the cache manager. It controls the status of the storage. If the storage is full, the prefetch manager uses an eviction algorithm to remove data. When new data is prefetched, it sends the new prefetched data ID as a notification to the cacher. The second component is the prefetch storage that is used to store data that is prefetched from the replica until it is accessed by the client.

Like caching, many prefetching algorithms are used to prefetch data from the replica. Among them, the most appropriate and widely used one is the initiative prefetching scheme which is selected because it is the best one in terms of appropriate fetching as implemented in [83]. The Initiative prefetching scheme has two basic algorithms that are used to predict and prefetch the future required data from the replica and send it to the client. These are linear regression prediction and chaotic prediction algorithms. Linear regression prediction uses a sequential access pattern and chaotic prediction uses a random access pattern for determining the I/O operation that belongs to either a sequence tendency or a random tendency. The linear regression prediction algorithm is intended to forecast future access when the current access follows a linear access pattern and the chaotic prediction algorithm intends to forecast future access when the current access follows a random access pattern. Sequential access means that when an application accesses a certain piece of data or I/O operation in a file (read or write), the next access proceeds from where the prior access ended. In a random access I/O pattern, the next data access is not the next chunk of data but some other data chunk within the file (ideally, a random location). If the I/O operation makes a sequential pattern, it will predict using a linear regression algorithm. Else, if all I/O operations in a random access tendency are in chronological order and the neighbor I/O operations might not have any dependency, it will be predicted using a chaotic algorithm. The details of the linear regression prediction algorithm and chaotic algorithm [83] are presented in Algorithm 4.3.

Algorithm 4.3 demonstrates the linear regression and chaotic prediction algorithms. We first count the total number of access events in the Access set when their target data addresses are within the range [Addrcur-Addrdef, Addrcur]. Addrcur is the current starting address, Addrdef is another predefined threshold to confine the offset range of access events. Access to information about the next read request that is predicted [Addrnew, Sizenew, Tnew].The prediction algorithm searches the provided block access history for all recent block access occurrences after a specific time point.All Access [Addrprev, Sizeprev, Tprev] with a Tprev greater than (Tcur-Tdef) should be gathered and placed in an Access set. Tdef is a predefined threshold based on the density of I/O operations and limits the time range for the previous access operations that will be used for judging whether the current access sequence is sequential or not. After that, if the total number of access events is greater than our predefined threshold value, we can ascertain that this access request is part of a linear access sequence. Therefore, the future request can be predicted by using linear regression prediction, and the details about how to forecast the address, request size and time of the future access request have been shown in the last part of the regression prediction algorithm. A positive value of maximum Lyapunov exponent is usually leveraged as an indication that the system of the sequence is chaotic, and then we can utilize chaotic time series prediction algorithms to forecast future I/O access. Otherwise, I/O access prediction and the corresponding data prefetching should not be done while the access pattern is random but not chaotic.

Algorithm 4-3: Linear Regression and Chaotic Prediction Algorithm

```
Input: Data amount, Time used, Threshold
Output: Predicted Data
Begin
    Initialization
        $Data_{amount}$ = total size of processed data during the period
        $Time_{used}$ = time consumed for completing all relevant access
            requests
        $T_{serving}$ = time required for processing the currently received
            read request
        $T_{cur}$ = the current time that a new access request comes
        $Addr_{cur}$= the starting current address at current time
        $Size_{cur}$ = the current request size at current time.
        // throughput is the rate of successful message delivery
        //over a communication channel and calculated by dividing
        //the file size by the time it takes
        // if the total number of access events is greater than the
        pre-defined threshold value, this access request is part of a
        linear access sequence
    //Counting the total number of access events and store in
        Access_set
    If (Access_set >= threshold)
        Slope = $Data_{amount}$/ $Time_{used}$
        $T_{serving}$ = $Size_{cur}$/ Data Throughput
        $Addr_{new}$ = $Addr_{cur}$+ $T_{serving}$/ Slope
        $T_{new}$ = $T_{cur}$ + $T_{serving}$
        $Size_{new}$ = $Size_{cur}$ or Average size
    Else
        //Random algorithm, there are two steps in this algorithm
            calculate the Lyapunov exponent
        // which is a norm to indicate a chaotic series
        Calculate Lyapunov exponent using equation 4.2
        If ($\lambda$>0) // high sensitivity on initial condition, If the value
        of $\lambda$ is greater than 0 then we can find the prediction data
        using equation 4.3.
        Endif
        // no sensitivity dependent on initial conditions
        If ($\lambda$<0)
        //data prefetching should not be done while the access pattern
            is random but not chaotic.
        Endif
End
```

To calculate the Lyapunov exponent, take two points and find the initial distance between them. After time t take another two points and find the distance between them then find the exponent by using the two distances.

Equation 4.2.

Two initial conditions, X_0 and Y_0

$D_0 = |X_0-Y_0|$

$D(t)=|Xt-Yt|$

//separation distance between blocks at time t

$D(t)= D_0 2^{\lambda t}$ //on average for all small t and small D_0, λ Lyapunov exponent

Equation 4.3.

$d_{M(0)} = \min_j \| X_M - X_j \| = \| X_M - X_K \|$

// After the time interval of t+1, X_M comes to X_{M+1}.

// As a result, the nearby point of X_M changes to $X_{K+1}$from X_K

$\| X_{M+1} - X_{K+1} \| = \| X_M - X_K \| e^{\lambda}$

$$\sqrt[2]{\sum_{J=1}^{m} |X_{M+1(j)} - X_{K+1(j)}|^2} = \sqrt[2]{\sum_{J=1}^{m} |X_{M+1(j)} - X_{K+1(j)}|^2 e^{\lambda}}$$

X_M is the center for each prediction process, X_k is the nearest point to X_M, and the distance between two points is $d_{M(0)}$. As for the phase point X_{M+1}, only its last component is unknown, but this component is predictable and can be predicted with a given λ.

4.2.3 Replicator

The replicator is another part of the proposed system which has the following basic components: replica manager, replica (R_1, R_2... Rn) and statistical data or log data. Replica manager controls and communicates with all the other components. The replica manager has an algorithm that is used to control the number of replicas, the data which should be replicated and where the replicas should be placed. Another function of the replica manager is communicating with statistical data to record the activities of the system. The replica is the storage of necessary data that is currently required by clients found near to the clients. The data in the replica comes from the master replica or server and is frequently changed by the needs of the clients. Statistical data or log data record is an important component of the replicator which is used to make decisions as to which data should be fetched that is required by the client soon.

In our proposed system, to decide which data should be replicated and where should it be placed, we use dynamic replication using a fuzzy logic method that is most effective [64]. In Algorithm 4.4, when the job requires data that does not exist in the local storage, replication takes place. The replicated files are stored in the best sites where the file will most probably be accessed in the future. To obtain the best replication site, each node having a storage element is given a quantity as Place of Replica Value (PRV) calculated through a fuzzy function, by calling a fuzzy function having three input parameters that are the number of accesses, the sum of the bandwidths of nodes and last access time interval. After the PRV is calculated for all the sites existing in the region where the request for the file was received, the algorithm chooses the node with the highest value of PRV as the replication site. If the selected node has enough space available, the file is stored in the selected site and the region header.

Algorithm 4-4: Fuzzy Dynamic Replication Algorithm

```
Input: Grid Site 'g', File 'f'
Output: File in a grid
Begin
   If (a grid site needs a file "f" that is not in its local site)
      For (all nodes in the region that have SE) //SE is a storage
         element
         PRV(node i) = Fuzzy Function (Number of Accesses(f),
            Bandwidth(i), Last Access Time Interval(f))
      Endfor
      Find the node that has maximum PRV and set g with its address
      If (g.availableStorageSize> f.size)
         replicate file "f" to grid site "g" and exit;
      Else If (another site in the region has duplication of file
         "f")
         Access file "f" remotely;
 Else
         // Sort files in g by using LFU;
         For (each file in sorted list)
            If (file "f" was duplicated in other sites within the
               region)
               Delete it;
            Endif
            if (g.availableStorageSize> f.size)
                  replicate file "f" to grid site "g" and exit;
            Endif
         Endfor
         // Sort files in "g" by using LFU;
         For (each file in sorted list)
            If (access frequency of new replica >access frequency
               of "f")
               Delete it;
            Endif
            if (g.availableStorageSize>f.size)
               break;
            Endif
         Endfor
         if (g.availableStorageSize> f.size)
            replicate file “f” to grid site “g” and exit;
         Endif
      Endelse
   Endif
   if (g.storageSize< f.size)
      Access file file“f” remotely, or replicate “f” to the closest
         storage element to “g”
   Endif

End
```

Proposed System Algorithm

The proposed model has three phases of activities. These are the first phase, second phase and third phase. The first phase holds activities before request sending includes replication of data and prefetching of necessary data. The above replication and the prefetching algorithm used in this phase. The second phase includes activities during the request sent from the user up to the response come back to using this procedure explained in algorithm 4.5. The third phase includes the cache activities which use cache algorithm 4.2, to control the cache storage. When the cache storage becomes full the cache algorithm is used to evict unnecessary data from the storage and then store the required data. Algorithm 4.5 shows the procedure from the request sent from the user or application up to the response reply. The proposed system algorithm includes the integration of the three algorithms. First, use replication and prefetching algorithm to fetch the necessary required data and store to the appropriate place and when storage becomes full use algorithm to free up space for most recently needed data.

Algorithm 4-5: The Proposed System Algorithm

```
Initialization
   Request/ Response data = D
   Cacher = C
   Prefetcher = P
   Remote Server = S
   Replicator = R
```

```
    Notifier = N
    Replica = r ($r_1$, $r_2$, … $r_n$)
Input: D, C, P, R, N, S, r
Output: D
Begin
    if D in C
        Return D to client
    Else If (P active && D ID in N)
        Access D from P
        store D in C
        return D to Client
    Else
        Scan the nearest R
        1: If (D found in current R)
            Access D from R
            store D in C
            return D to Client
        Else (if R next nearest)
            goto 1:
        Endif
        Else if (D found in S)
            access D from S
            store D in C
            return D to Client
        Endif
    Endif
End
```

4.3 Summary

In the general CPDR model, the most relevant components are added to increase the performance of distributed systems. All components in the new architecture are aimed at addressing the computational time or latency. The new architecture has a manager responsible to manage the activities of each computation performed in each part. Each manager has an algorithm that is used to evict or fetch data to/from the storage. The cache manager uses an adapted LRFU algorithm that is used to evict data from the cache storage when it becomes full. This algorithm first calculates the CRF and dynamic values of the LRFU to choose the appropriate replacement policy according to the practical case. The cacher has a notifier component that has basic information about the prefetcher, whether it is active or not, and the

ID of the data that is currently available in the prefetcher so this component decreases the wasted time when the prefetcher is not active and the data is not available there. In the prefetcher, there is also a prefetch manager that has an algorithm called initiative prefetching which has two algorithms working based on the history access pattern. These are linear regression prediction and chaotic prediction algorithms. Linear regression prediction uses the sequential access pattern and chaotic prediction uses the random access pattern. These two algorithms predict and fetch the most relevant data. In the replicator, we used dynamic replication using fuzzy logic to replicate the necessary data and place the data in the appropriate place which is the nearest place for the clients. As a whole, this new architecture solves the existing latency of distributed systems by decreasing the transfer time, queuing time and execution time of the request.

5. Experiments

This Chapter presents tools and methodologies that have been used to implement the Integrated CPDR algorithm for distributed system Computing Environments. The prototype is developed based on the proposed architecture presented in Chapter Four. The last topic discussed in this Chapter, experiment and evaluation, presents the performance of integrated CPDR that is evaluated by undertaking experiments and comparisons with other standard algorithms.

5.1 Development Tools

The following tools are mainly used to develop the prototype.

a. Java Platform, Standard Edition Development Kit (JDK) Version 8.2

The Java Development Kit is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), and other tools needed in Java development [84].

b. NetBeans Integrated Development Environment (IDE) Version 8.2

The NetBeans IDE is an official open-source Integrated Development Environment. It enables the creation of programs using a set of modular software components known as modules. Applications based on the NetBeans Platform, including the NetBeans integrated development environment (IDE), can be extended by third-party developers. NetBeans is cross-platform and runs on Microsoft Windows, Mac OS X, Linux, Solaris and other platforms supporting a compatible Java Virtual Machine (JVM [85].

c. GridSim 5.2

Simulation has been used extensively for modeling and evaluation of real-world systems, from business processes and factory assembly lines to computer systems design. As a result, modeling and simulation have become increasingly important over time, prompting the development of many standards as well as application-specific tools and technologies. GridSim is a toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing [86]. This toolkit contains simulation capabilities for a wide range of heterogeneous resource classes, users, applications, resource brokers, and schedulers. It may be used to simulate application schedulers in distributed computing systems such as clusters and grids for single or multiple administrative domains.

Resource brokers, which are application schedulers in a grid context, provide resource discovery, selection, and aggregation of a diverse set of distributed resources for a single user. That is, each user has his or her private resource broker, which may be targeted to meet the needs and interests of the owner. Schedulers, on the other hand, who manage resources such as clusters in a single administrative domain, have complete control over the resource allocation policy.

5.2 Scope of the Prototype

The prototype is designed to meet the basic requirements of the system architecture shown in Figure 4.1. Components such as Cacher, prefetcher and replicator are implemented. Other sub-components which are found within each main component are also implemented. However, the prototype does not visualize data center creation and manipulations using a pictorial view. Hence, the default input and output mode of NetBeans IDE, Text-Based Mode, is used.

5.3 Prototype Development

A prototype is developed to implement the proposed CPDR algorithm. Modeling the largest computing environment, the grid at least requires an efficient abstraction of grid entities. GridSim based simulations contain entities for the users, brokers, resources, information service, statistics, and network based I/O. Moreover, core packages of the simulator namely, “grid. Gridsim” and “gridsim.gridsimcore” and some of its classes are overridden to make the toolkit suitable to the proposed algorithm. Specifically, two gridsim models with different resource characteristics are designed using built-in classes such as Resource, User, and gridlet/jobs. The purpose of using various gridsim models is to evaluate the proposed algorithm in various size grid models; particularly in both small scale and large-scale grid computing environments. Table 5.1 describes the characteristics of the first grid model used in this experiment.

Table 5.1: GridSim Model I - Resource Constrained Scenario

Resource site	*Characteristics*							
	Reso_ID	*Machine*	*Core*	*OS*	*Architecture*	*BW*	*Storage (MB)*	*Time zone*
ResourceSite_0	0	Machine_1	1	Window_8.1	X_86	500	2000	7.0
		Machine_2	2	Window_8.2	X_64	500		
		Machine_3	3	Window_10	X_86	500		
ResourceSite_1	1	Machine_1	1	Window_8.1	X_86	500	2000	8.0
		Machine_2	2	Window_8.2	X_64	500		
		Machine_3	3	Window_10	X_86	500		
ResourceSite_2	2	Machine_1	1	Window_8.1	X_86	500	2000	9.0
		Machine_2	2	Window_8.2	X_64	500		
		Machine_3	3	Window_10	X_86	500		
ResourceSite_3	3	Machine_1	1	Window_8.1	X_86	500	2000	10.0
		Machine_2	2	Window_8.2	X_64	500		
		Machine_3	3	Window_10	X_86	500		
ResourceSite_4	4	Machine_1	1	Window_8.1	X_86	500	2000	11.0
		Machine_2	2	Window_8.2	X_64	500		
		Machine_3	3	Window_10	X_86	500		
ResourceSite_5	5	Machine_1	1	Window_8.1	X_86	500	2000	12.0
		Machine_2	2	Window_8.2	X_64	500		
		Machine_3	3	Window_10	X_86	500		
ResourceSite_6	6	Machine_1	1	Window_8.1	X_86	500	2000	13.0
		Machine_2	2	Window_8.2	X_64	500		
		Machine_3	3	Window_10	X_86	500		

This model is categorized as a small scale (resource constrained) whose resources are located in a different site. Each resource site is found in a different time zone which indicates the resources are distributed. Each site contains a data center that has three machines with different characteristics. Seven resource sites are distributed in different time zones. The first resource site (ResourceSite_0) is assumed to be a cacher since the time zone is nearest to the user and has a small amount of storage. The second resource site (ResourceSite_1) is assumed as the prefetcher which has a storage that is used to store the nearest required data needed by the user. The time zone of the prefetcher is also nearest to the user as compared to the other replica resource sites. The other resource sites are replicators. The replicator has five resource sites. ResourceSite_2, ResourceSite_3, ResourceSite_4, ResourceSite_5 are replicas and ResourceSite_6 is the master replica. In this scenario, the number of users, data and resource sites are small or constrained.

The other model used to test the proposed algorithm is shown in Table 5.2. This model is designed as a resourceful distributed system. In this model, we increase the number of users/jobs, the resource sites, the number of data within each resource site and the distance between each resource site is also increased. This model has unconstrained resources (large resources) that have surpassed the first model. It is designed and deployed to show the performance of the proposed algorithm in a resource unconstrained distributed environment.

Table 5.2: GridSim Model II - Resource Unconstrained Scenario

Resource site	*Characteristics*							
	Reso_ID	*Machine*	*Core*	*OS*	*Architecture*	*BW*	*Storage (MB)*	*Time zone*
ResourceSite_0	0	Machine_1	1	Window_8.1	X_86	500	5000	7.0
		Machine_2	2	Window_8.2	X_64	500		
		Machine_3	3	Window_10	X_86	500		
ResourceSite_1	1	Machine_1	1	Window_8.1	X_86	500	5000	8.0
		Machine_2	2	Window_8.2	X_64	500		
		Machine_3	3	Window_10	X_86	500		
ResourceSite_2	2	Machine_1	1	Window_8.1	X_86	500	5000	9.0
		Machine_2	2	Window_8.2	X_64	500		
		Machine_3	3	Window_10	X_86	500		
ResourceSite_3	3	Machine_1	1	Window_8.1	X_86	500	5000	10.0
		Machine_2	2	Window_8.2	X_64	500		
		Machine_3	3	Window_10	X_86	500		
ResourceSite_4	4	Machine_1	1	Window_8.1	X_86	500	5000	11.0
		Machine_2	2	Window_8.2	X_64	500		
		Machine_3	3	Window_10	X_86	500		
ResourceSite_5	5	Machine_1	1	Window_8.1	X_86	500	5000	12.0
		Machine_2	2	Window_8.2	X_64	500		
		Machine_3	3	Window_10	X_86	500		
ResourceSite_6	6	Machine_1	1	Window_8.1	X_86	500	5000	13.0
		Machine_2	2	Window_8.2	X_64	500		
		Machine_3	3	Window_10	X_86	500		
ResourceSite_7	7	Machine_1	1	Window_8.1	X_64	500	5000	14
		Machine_2	2	Window_8.2	X_86	500		
		Machine_3	3	Window_10	X_64	500		
ResourceSite_8	8	Machine_1	1	Window_8.1	X_64	500	10000	15
		Machine_2	2	Window_8.2	X_86	500		

Resource site	*Characteristics*							
	Reso_ID	*Machine*	*Core*	*OS*	*Architecture*	*BW*	*Storage (MB)*	*Time zone*
		Machine_3	3	Window_10	X_64	500		
ResourceSite_9	9	Machine_1	1	Window_8.1	X_64	500	2000	16
		Machine_2	2	Window_8.2	X_86	500		
		Machine_3	3	Window_10	X_64	500		
ResourceSite_10	10	Machine_1	1	Window_8.1	X_64	500	2000	17
		Machine_2	2	Window_8.2	X_86	500		
		Machine_3	3	Window_10	X_64	500		

As shown in Table 5.2, the resources in this model are located in eleven different resource sites. Like Table 5.1, ResourceSite_0 and ResourceSite_1 are assumed as cacher and prefetcher, respectively. Resource site_10 is assumed as a master replica and the rest are normal replicas. Hence, this model is used to show the performance of the proposed algorithm for geographically distributed resources. Resources in the above model are created using a source code attached in Annex that shows the sample code used to create one resource site with its characteristics**.**

a. Cache Manager

This is an abstract class that describes the basic functionality of a Cache Manager in a Cacher. This class is responsible for all data manipulation on storage in the cacher.

b. Notifier

This class has two attributes. The first one is the Boolean type which shows the status of the prefetcher, whether it is currently active or not. The second is the ID of the prefetcher storage data that is currently stored. This class is called before the prefetcher resource site to check its status, which is very important to decrease the latency or wastage of time of the prefetcher when it is not active or the data is not found in the storage.

c. Prefetch Manager

This is an abstract class that describes the basic functionality of a Prefetch Manager. This class is responsible for all data manipulation on a prefetcher.

d. Replication Manager

This is an abstract class that describes the basic functionality of a Replica Manager in a Data Grid. This class is responsible for all data manipulation on a Data Grid Resource.

e. Statistical data/log

This class records statistical data reported by other entities. It stores data objects along with their arrival time and the ID of the machine allocated to them. It acts as a placeholder for maintaining the amount of resource share allocated at various times for simulating time-shared scheduling using internal events. This recorded data is used for the prefetcher to analyze and forecast what the next recorded data is.

5.4 Experimental Results

The experiment is held on the two grid models shown in Tables 5.1 and 5.2. The two major Grid models are used to run various types of gridlets/jobs to show the performance of the proposed algorithm in both resources constrained and resource unconstrained grid environments. The performance of the proposed algorithm is presented in comparison with caching algorithm, prefetching algorithm, dynamic replication algorithm, integrated caching and prefetching algorithm, integrated caching and dynamic replication algorithm and integrated prefetching and dynamic replication algorithm. These algorithms are selected considering their popularity. These algorithms are usually considered as standard for comparing newly proposed models.

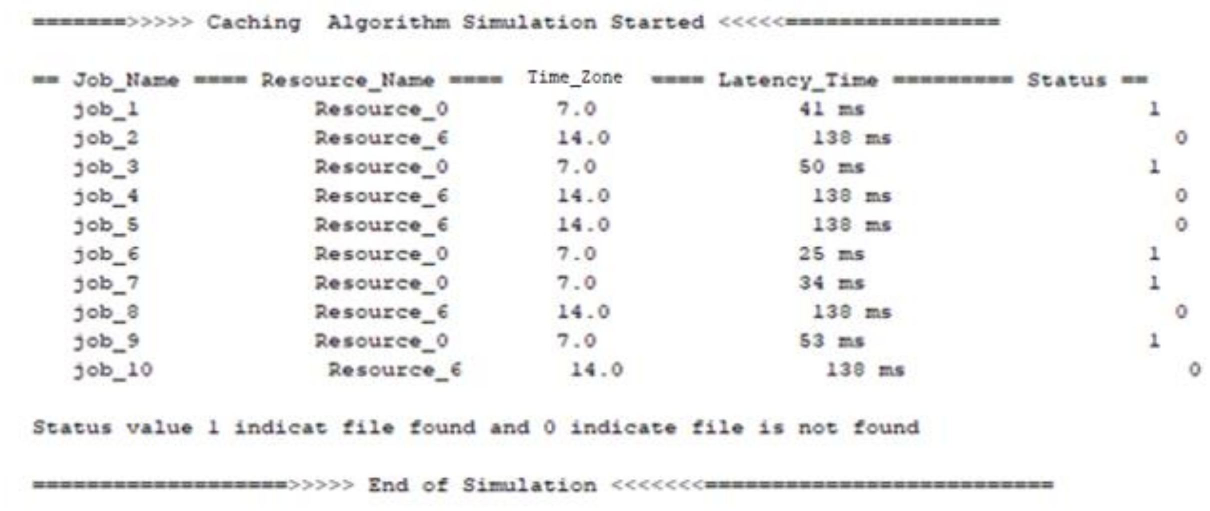
The proposed algorithm tries to decrease the latency of the distributed system using Equation 2.1. The proposed algorithm is compared with the mentioned algorithms by calculating the total average latency time of each job. The following sections briefly describe the results gained after intensive experiments for each algorithm.

5.4.1 Running a Few Gridlets/Jobs

a. Caching Algorithm

Caching is one of the methods used to reduce latency in distributed systems. The relevant and previously accessed data are stored for later use. If the jobs are requested again it will access from the cacher. The request will not waste time by going to the remote server. There are many caching algorithms to control the cacher storage. From them, the Adaptive LRFU eviction algorithm is used when the cacher storage becomes full. So, comparing the result of this caching algorithm alone without integrating the prefetching and replication algorithms with the CPDR algorithm helps to know the performance of the proposed algorithm. The user jobs can be accessed either from the cacher or from the remote server, in our case from the master replica. When we use caching, the result can be considered in two basic cases: the best case and the worst case. The worst case is when a user sends jobs for the first time and when all or most of these jobs are not found in the cacher storage. This results in high total latency. On the other hand, the best case happens when most of the jobs are accessed from the cache storage which can decrease the overall latency of the jobs.

As shown in Figure 5.1, 10 jobs are requested by the user. From them, 5 jobs are executed from cacher storage and the latency of each job is too small, and the rest are accessed from the master replica but still not found and the latency is too high. The average latency for the caching algorithm is 89.3 milliseconds.



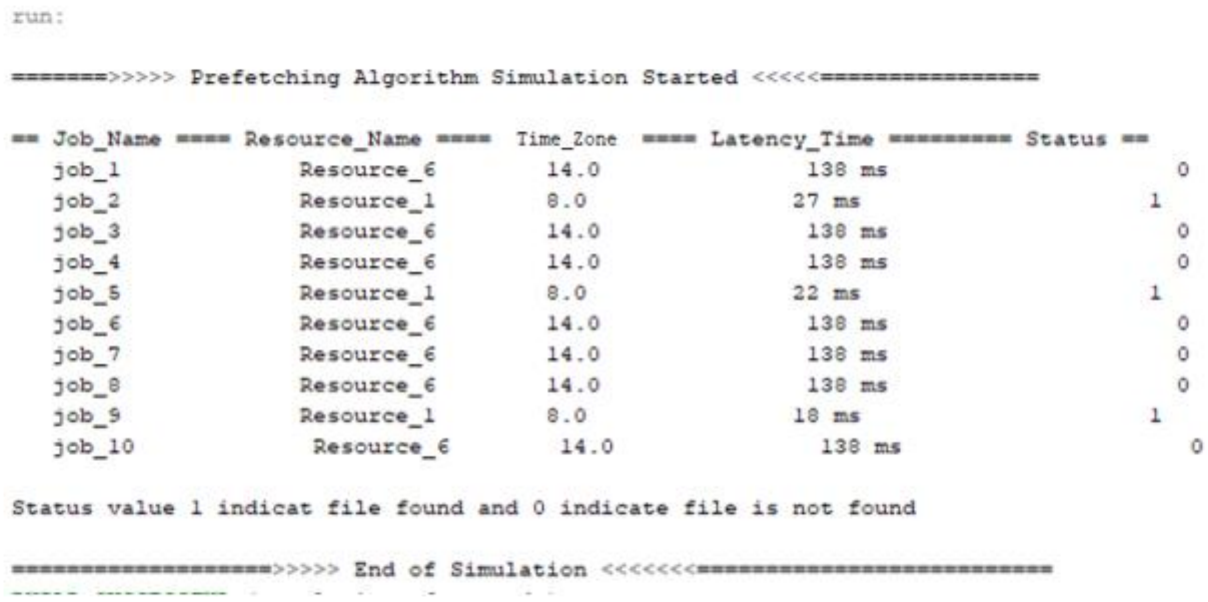
```
=======>>>>> Caching  Algorithm Simulation Started <<<<<=================

== Job_Name ==== Resource_Name ====  Time_Zone  ==== Latency_Time ========= Status ==
   job_1            Resource_0        7.0              41 ms                 1
   job_2            Resource_6        14.0              138 ms                  0
   job_3            Resource_0        7.0              50 ms                 1
   job_4            Resource_6        14.0              138 ms                  0
   job_5            Resource_6        14.0              138 ms                  0
   job_6            Resource_0        7.0              25 ms                 1
   job_7            Resource_0        7.0              34 ms                 1
   job_8            Resource_6        14.0              138 ms                  0
   job_9            Resource_0        7.0              53 ms                 1
   job_10            Resource_6        14.0              138 ms                  0

Status value 1 indicat file found and 0 indicate file is not found

==================>>>>> End of Simulation <<<<<<=========================
```

Figure 5.1: Result of Caching Algorithm - Scenario I

b. Prefetching Algorithm

Prefetching is also another important method that is used to reduce latency for distributed systems. There are many prefetching algorithms and the Initiative prefetching algorithm (Linear regression and chaotic prediction algorithm) is used to predict which data could be needed by the user soon. In this scenario, the user jobs can be accessed either from the prefetcher or from the remote server, in our case from the master replica. When we use prefetching as shown from the result it can be also considered in two basic cases: the best case and the worst case. The worst case is when the jobs are not found in the prefetcher storage. This leads to high latency. On the other hand, the best case happens when most of the jobs are found from the prefetch storage which can decrease the overall latency of the jobs.

As shown in Figure 5.2, 10 jobs are requested by the user. From them, only 3 jobs are executed from prefetcher storage and the latency of each job is too small, and the rest are accessed from the master replica but still not found and the latency is too high. The average latency for the prefetching algorithm is 103.3 milliseconds.



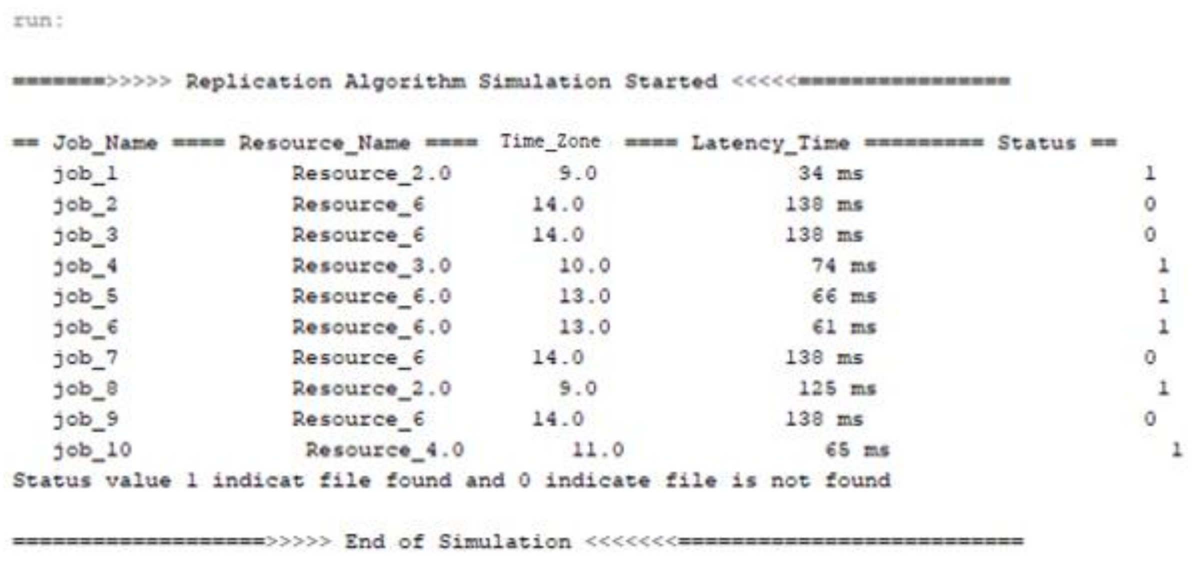
```
run:

=======>>>>> Prefetching Algorithm Simulation Started <<<<<================

== Job_Name ==== Resource_Name ====  Time_Zone  ==== Latency_Time ========= Status ==
   job_1           Resource_6        14.0              138 ms                    0
   job_2           Resource_1        8.0              27 ms                   1
   job_3           Resource_6        14.0              138 ms                    0
   job_4           Resource_6        14.0              138 ms                    0
   job_5           Resource_1        8.0              22 ms                   1
   job_6           Resource_6        14.0              138 ms                    0
   job_7           Resource_6        14.0              138 ms                    0
   job_8           Resource_6        14.0              138 ms                    0
   job_9           Resource_1        8.0              18 ms                   1
   job_10           Resource_6        14.0              138 ms                    0

Status value 1 indicat file found and 0 indicate file is not found

==================>>>>> End of Simulation <<<<<<==========================
```

Figure 5.2: Result of Prefetching Algorithm - Scenario I

c. Dynamic Replication Algorithm

Dynamic Replication is a method used to reduce the latency of distributed systems. There are many algorithms within the dynamic method. From them, the better one is Fuzzy Dynamic Replication Algorithm which is used to replicate the most important data in the appropriate replica which is nearest to the user. The user jobs can be accessed either from the nearest replica, from another replica, or the remote server, in our case from the master replica. Like the above two algorithms, dynamic replication results can be considered in two cases: the best case and the worst case. The worst case is when a user sends the jobs and when most of these jobs are not found in the replicas. This leads to high latency. On the other hand, the best case happens when most of the jobs are accessed from the nearest replica which can decrease the overall latency of the jobs.

As shown in Figure 5.3, 10 jobs are requested by the user. From them, 6 jobs are executed from replicas and the latency is too small for each job, and the rest are accessed from the master replica but still not found and the latency is too high. The average latency for the dynamic replication algorithm is 97.7 milliseconds.



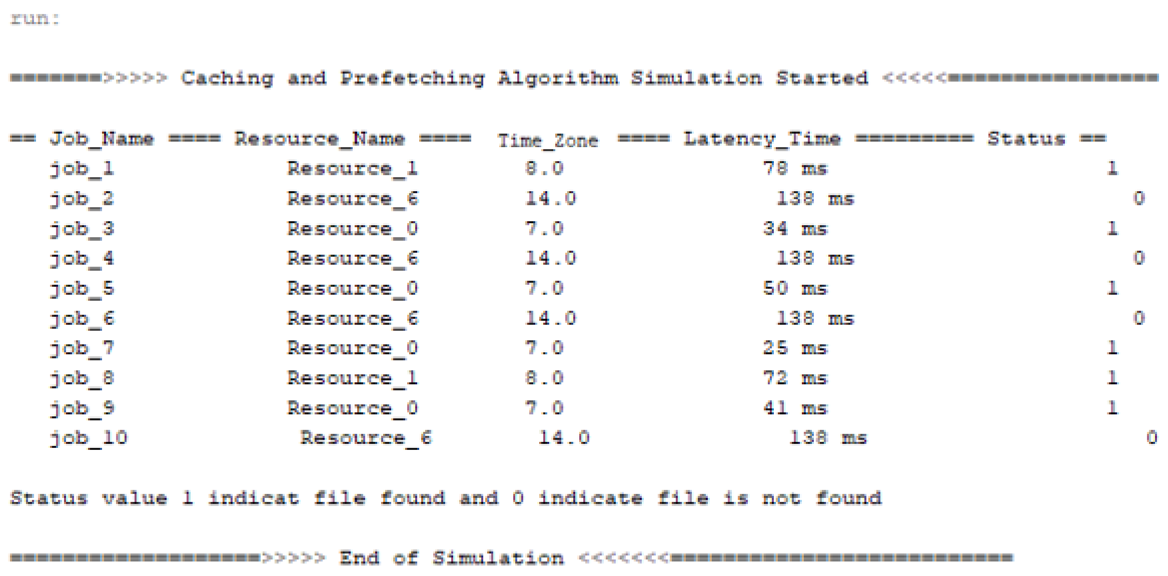
```
run:

=======>>>>> Replication Algorithm Simulation Started <<<<<================

== Job_Name ==== Resource_Name ====  Time_Zone  ==== Latency_Time ========= Status ==
   job_1            Resource_2.0        9.0              34 ms                    1
   job_2            Resource_6         14.0             138 ms                    0
   job_3            Resource_6         14.0             138 ms                    0
   job_4            Resource_3.0        10.0              74 ms                    1
   job_5            Resource_6.0        13.0              66 ms                    1
   job_6            Resource_6.0        13.0              61 ms                    1
   job_7            Resource_6         14.0             138 ms                    0
   job_8            Resource_2.0        9.0             125 ms                    1
   job_9            Resource_6         14.0             138 ms                    0
   job_10            Resource_4.0        11.0              65 ms                    1
Status value 1 indicat file found and 0 indicate file is not found

==================>>>>> End of Simulation <<<<<<=========================
```

Figure 5.3: Result of Dynamic Replication Algorithm - Scenario I

d. Integrated Caching and Prefetching Algorithm

The integration of caching and prefetching is also used to reduce latency. This integrated method gives a better result as compared to the above single solution algorithms (caching, prefetching and dynamic replication algorithms). The user job can be accessed from the cacher if found else it can be accessed from the prefetcher before going to the remote server. So comparing the result of this with the CPDR algorithm helps to know the performance of the proposed algorithm. The user jobs can be accessed either from the cacher, prefetcher or from the remote server, in our case from the master replica. In the integration of the two algorithms, the result can be considered in two basic cases: the best case and the worst case. The worst case is when the requested jobs are not accessed either in cacher or prefetcher storages. This leads to high total latency. On the other hand, the best case happens when the jobs are accessed either from the cacher storage or from the prefetcher storage which can decrease the overall latency of the jobs.

As shown in Figure 5.4, 10 jobs are requested by the user. From them, 6 jobs are executed from cacher storage and prefetcher storage and the latency is too small for each job, and the rest are accessed from the master replica but still not found and the latency is too high. The average latency for the integrated caching and the prefetching algorithm is 85.2 milliseconds.



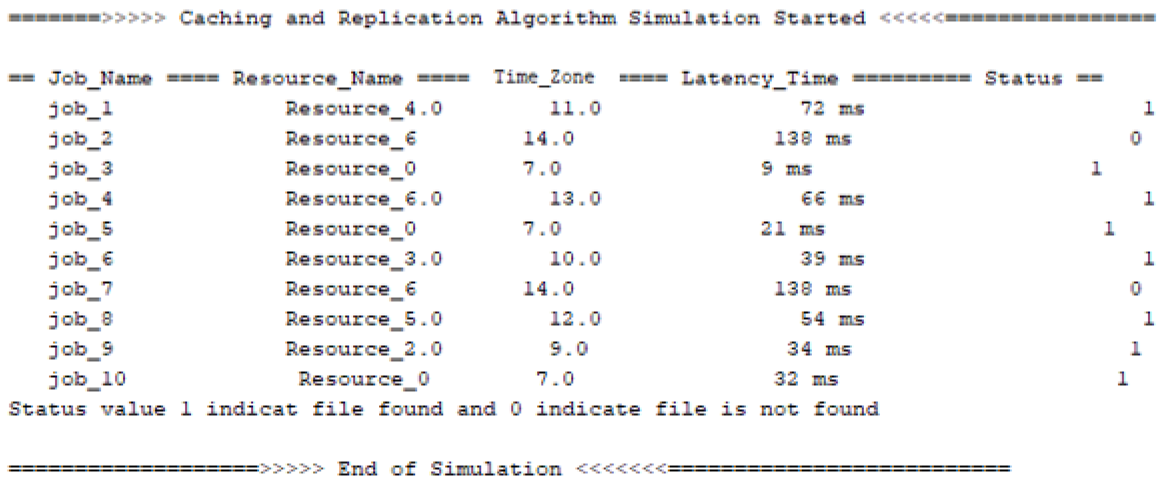
```
run:

=======>>>>> Caching and Prefetching Algorithm Simulation Started <<<<<=================

== Job_Name ==== Resource_Name ====  Time_Zone ==== Latency_Time ========= Status ==
   job_1            Resource_1        8.0              78 ms                    1
   job_2            Resource_6        14.0              138 ms                    0
   job_3            Resource_0        7.0              34 ms                    1
   job_4            Resource_6        14.0              138 ms                    0
   job_5            Resource_0        7.0              50 ms                    1
   job_6            Resource_6        14.0              138 ms                    0
   job_7            Resource_0        7.0              25 ms                    1
   job_8            Resource_1        8.0              72 ms                    1
   job_9            Resource_0        7.0              41 ms                    1
   job_10            Resource_6        14.0              138 ms                    0

Status value 1 indicat file found and 0 indicate file is not found

===================>>>>> End of Simulation <<<<<<==========================
```

Figure 5.4: Result of Integrated Caching and Prefetching Algorithm - Scenario I

e. Integrated Caching and Dynamic Replication Algorithm

The integration of caching and replication is also used to reduce latency. This integrated method also gives a better result as compared to the single solution algorithms (caching, prefetching and dynamic replication). The user job can be accessed from the cacher if found else it can be accessed from the nearest replica before going to the remote one. So comparing the result of this with the CPDR algorithm helps to know the performance of the proposed algorithm. The user jobs can be accessed either from the cacher, the nearest replica or from the remote server, in our case from the master replica. In the integration of the two algorithms, the result can be considered in two basic cases: the best case and the worst case. The worst case is when the requested jobs are not accessed either in the cacher or nearest replica. This leads to high latency for each job. On the other hand, the best case happens when the jobs are accessed either from the cacher or from the nearest replica which can decrease the overall latency of the jobs.

As shown in Figure 5.5, 10 jobs are requested by the user. From them, 8 jobs are executed from cacher storage and replica, and the latency is too small for each job, and the rest are accessed from the master replica but still not found and the latency is too high. The average latency for the integrated caching and dynamic replication algorithm is 60.3 milliseconds.



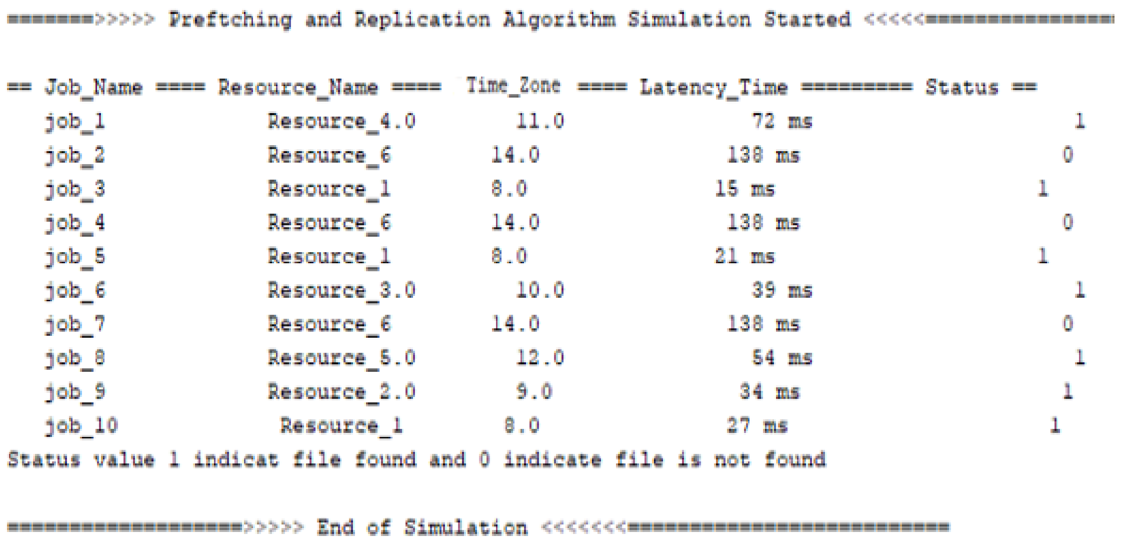
```
=======>>>>> Caching and Replication Algorithm Simulation Started <<<<<================

== Job_Name ==== Resource_Name ====  Time_Zone  ==== Latency_Time ========= Status ==
   job_1            Resource_4.0        11.0              72 ms                  1
   job_2            Resource_6        14.0              138 ms                 0
   job_3            Resource_0        7.0              9 ms                1
   job_4            Resource_6.0        13.0              66 ms                  1
   job_5            Resource_0        7.0              21 ms                1
   job_6            Resource_3.0        10.0              39 ms                  1
   job_7            Resource_6        14.0              138 ms                 0
   job_8            Resource_5.0        12.0              54 ms                  1
   job_9            Resource_2.0        9.0              34 ms                 1
   job_10            Resource_0        7.0              32 ms                1
Status value 1 indicat file found and 0 indicate file is not found

==================>>>>> End of Simulation <<<<<<=========================
```

Figure 5.5: Result of Integrated Caching and Dynamic Replication Algorithm - Scenario I

f. Integrated Prefetching and Dynamic Replication Algorithm

The integration of prefetching and replication is also used to reduce latency. Like the above integration, this integrated method also gives a better result as compared to the single solution algorithms (caching, prefetching and dynamic replication). The user job can be accessed from prefetcher storage if found else it can be accessed from the nearest replica before going to the remote one. The user jobs can be accessed either from the prefetcher, nearest replica or from the remote server, in our case from the master replica. When we use the integration of the two algorithms, the result can be considered in two basic cases: the best case and the worst case. The worst case is when the jobs are not accessed either in prefetcher storage or nearest replica. This leads to high latency. On the other hand, the best case happens when the jobs are accessed either from the prefetcher or from the nearest replica storage which can decrease the overall latency of the jobs.

As shown in Figure 5.6, 10 jobs are requested by the user. From them, 7 jobs are executed from prefetcher storage and replicas and the latency is too small, and the rest are accessed from the master replica but still not found and the latency is too high. The average latency for the integrated prefetching and dynamic replication algorithm is 67.6 milliseconds.



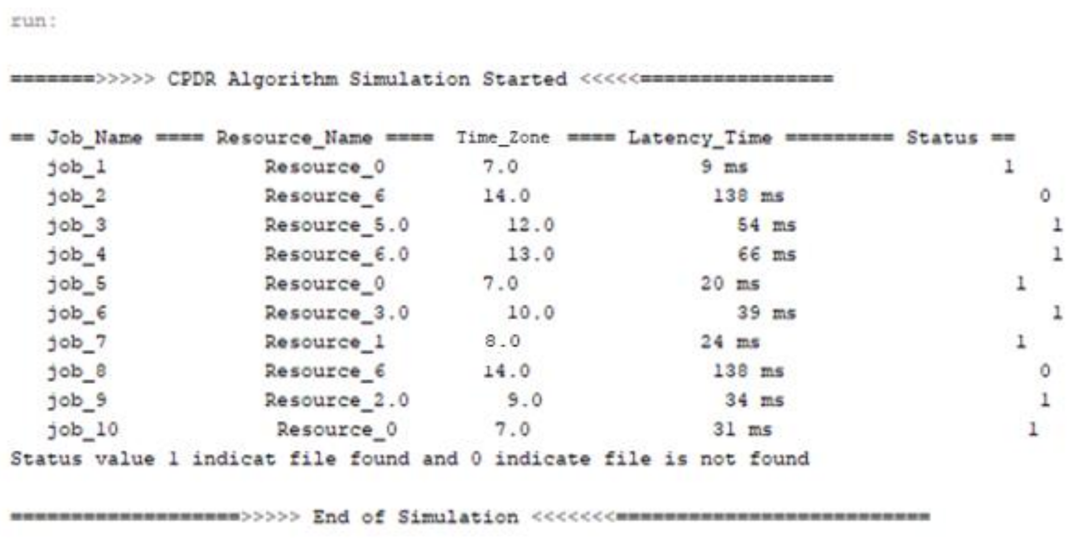
```
=======>>>>> Preftching and Replication Algorithm Simulation Started <<<<<================

== Job_Name ==== Resource_Name ====  Time_Zone ==== Latency_Time ========= Status ==
   job_1            Resource_4.0       11.0             72 ms                  1
   job_2            Resource_6       14.0             138 ms                 0
   job_3            Resource_1       8.0            15 ms                1
   job_4            Resource_6       14.0             138 ms                 0
   job_5            Resource_1       8.0            21 ms                1
   job_6            Resource_3.0       10.0             39 ms                  1
   job_7            Resource_6       14.0             138 ms                 0
   job_8            Resource_5.0       12.0             54 ms                  1
   job_9            Resource_2.0       9.0             34 ms                 1
   job_10            Resource_1       8.0             27 ms                1
Status value 1 indicat file found and 0 indicate file is not found

==================>>>>> End of Simulation <<<<<<=========================
```

Figure 5.6: Result of Integrated Prefetching and Dynamic Replication Algorithm - Scenario I

g. CPDR

CPDR is a resource constrained model where user jobs can be accessed from the cacher storage, prefetcher storage, nearest replica or the remote server, in our case from the master replica. When we use the CPDR algorithm the result can be considered also in two basic cases: the best case and the worst case. The worst case is when the jobs are not accessed either from cacher storage, prefetcher storage or nearest replica. This leads to high latency. On the other hand, the best case happens when the jobs are accessed from the cacher storage, prefetcher storage or the nearest replica. The best case is when the job accesses the cacher which can decrease the overall latency of the jobs.

As shown in Figure 5.7, 10 jobs are requested by the user. From them, 8 jobs are executed from cacher storage, prefetcher storage and replicas and the latency is too small, and the rest are accessed from the master replica but still not found and the latency is too high. The average latency for the CPDR algorithm is 55.3 milliseconds.



```
run:

=======>>>>> CPDR Algorithm Simulation Started <<<<<================

== Job_Name ==== Resource_Name ====  Time_Zone  ==== Latency_Time ========= Status ==
   job_1            Resource_0       7.0              9 ms                    1
   job_2            Resource_6       14.0              138 ms                    0
   job_3            Resource_5.0       12.0              54 ms                    1
   job_4            Resource_6.0       13.0              66 ms                    1
   job_5            Resource_0       7.0              20 ms                    1
   job_6            Resource_3.0       10.0              39 ms                    1
   job_7            Resource_1       8.0              24 ms                    1
   job_8            Resource_6       14.0              138 ms                    0
   job_9            Resource_2.0       9.0              34 ms                    1
   job_10            Resource_0       7.0              31 ms                    1
Status value 1 indicat file found and 0 indicate file is not found

==================>>>>> End of Simulation <<<<<<=========================
```

Figure 5.7: Result of CPDR Algorithm - Scenario I

h. Summary for Constrained Resources

The above comparisons within each algorithm are done in two ways. The first one is the latency within the algorithm that is between the executed jobs from cacher, prefetcher and replica, and jobs executed from the remote server. The second is between each resource site. Jobs that are executed from the cacher have smaller latency than jobs executed from prefetcher and replicas and jobs executed in prefetcher have smaller latency than replicas. This comparison is done by taking the average latency of the above algorithms. A shown in Figure 5.8, the average latencies (in milliseconds) of each algorithm are 89.3, 103.3, 97.7, 85.2, 60.3, 67.7 and 55.3 for caching, prefetching, dynamic replication, integrated caching and prefetching, integrated caching and dynamic replication, integrated prefetching and dynamic replication, and CPDR, respectively. The average latency of jobs in the prefetching and dynamic replication is high. This is because the of number of users/jobs, the number of data within each resource site, the distribution of resource sites and the efficiency of each algorithm have a great impact on the latency. CPDR algorithm has the lowest total average latency since most of the jobs have the possibilities to be executed from the cacher, prefetcher and nearest replicas by considering the above parameters in constrained resources.

Note: in section 5.4.1, we use the words too small and too high to explain the relative latency of each job in a given algorithm. As shown in the Status column of each Figure, too small latency jobs have the status value of 1 and too high latency jobs (rest jobs) have the status value of 0. In this section, the magnitude of too small is less than 138 ms and the magnitude value of too high (rest jobs) is equal to 138 ms.

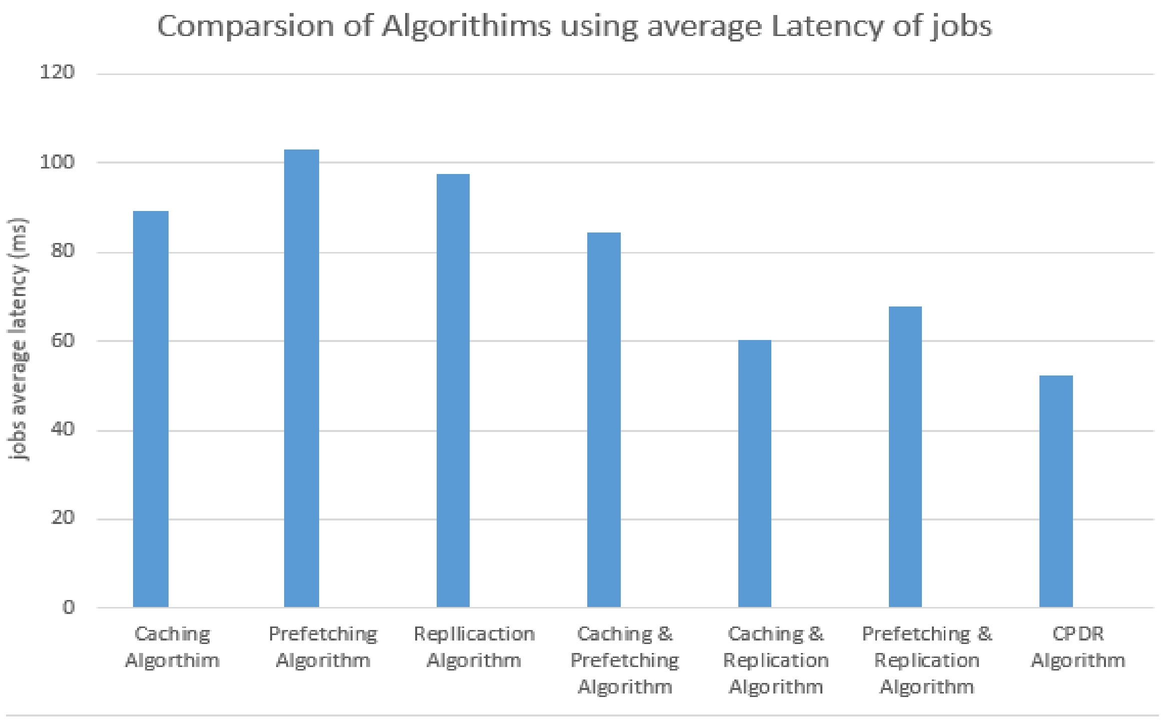


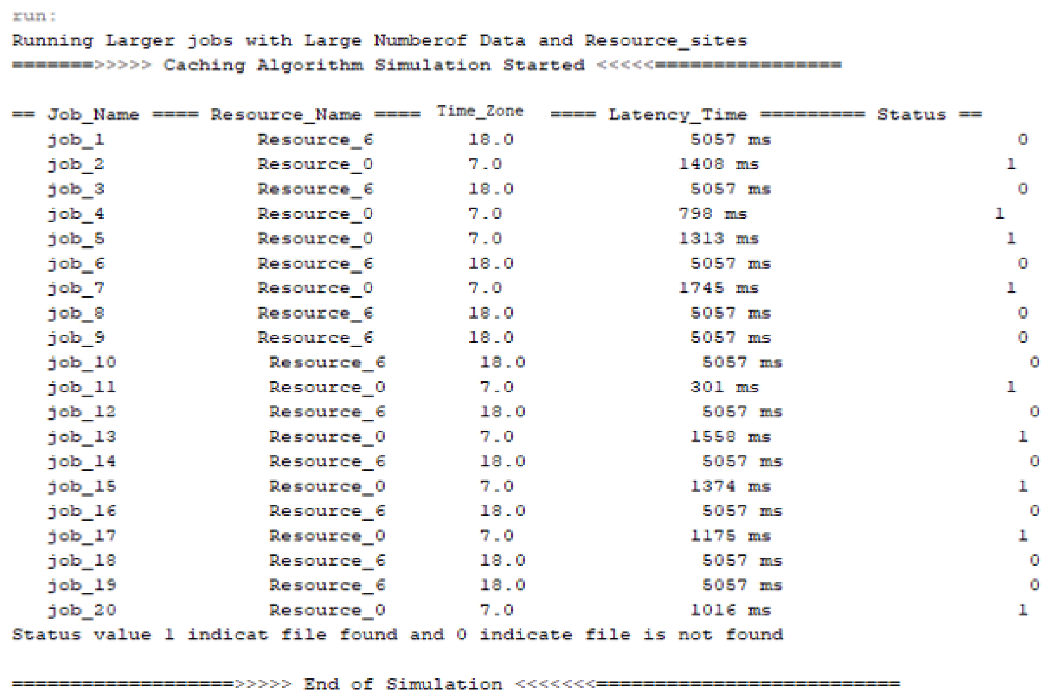
Figure 5.8: Comparison of the above Algorithms - Scenario I

5.4.2 Running Larger Gridlets/Jobs or Users

This model differs from the resource constrained model by the number of jobs created by the users, the number and location of the distributed system (resource sites), and the amount of data within each unconstrained resource (larger). The result of each algorithm is presented as compared to each other. The latency of a job at each algorithm is presented below in detail.

a. Caching Algorithm

As shown in Figure 5.9, 20 jobs are requested by the user. From them, only 9 jobs are executed from cacher storage and the latency is too small for each job. The rest are accessed from the master replica but still not found and the latency is too high as compared to the other algorithms. The average latency for the caching algorithm is 3315.75 milliseconds.

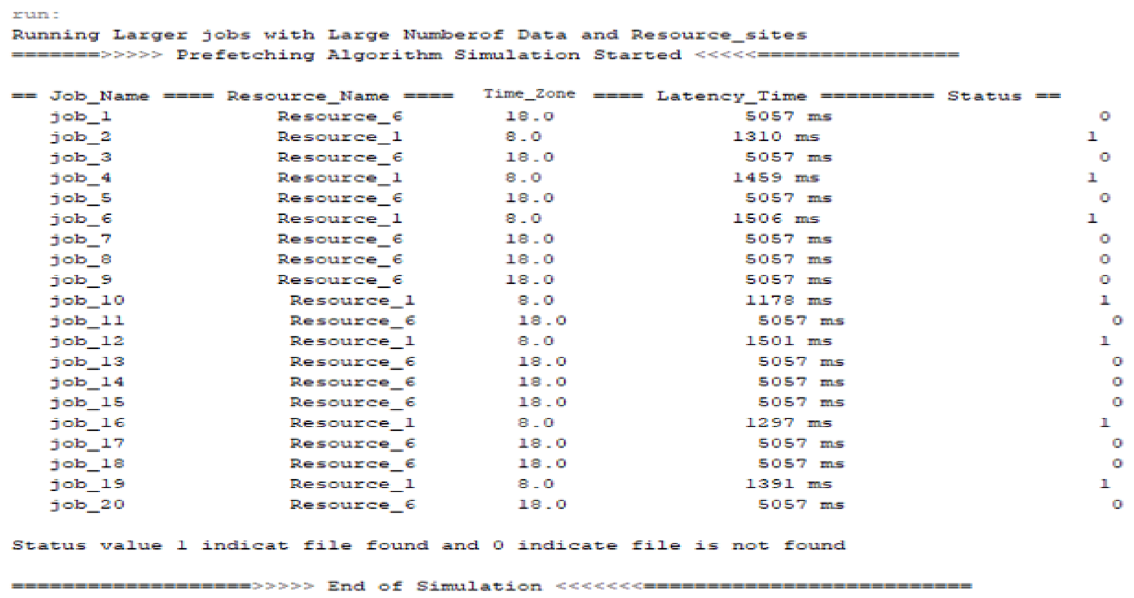


```
run:
Running Larger jobs with Large Numberof Data and Resource_sites
=======>>>>> Caching Algorithm Simulation Started <<<<<================

== Job_Name ==== Resource_Name ====  Time_Zone  ==== Latency_Time ========= Status ==
   job_1           Resource_6        18.0            5057 ms              0
   job_2           Resource_0        7.0            1408 ms              1
   job_3           Resource_6        18.0            5057 ms              0
   job_4           Resource_0        7.0            798 ms              1
   job_5           Resource_0        7.0            1313 ms              1
   job_6           Resource_6        18.0            5057 ms              0
   job_7           Resource_0        7.0            1745 ms              1
   job_8           Resource_6        18.0            5057 ms              0
   job_9           Resource_6        18.0            5057 ms              0
   job_10           Resource_6        18.0            5057 ms              0
   job_11           Resource_0        7.0            301 ms              1
   job_12           Resource_6        18.0            5057 ms              0
   job_13           Resource_0        7.0            1558 ms              1
   job_14           Resource_6        18.0            5057 ms              0
   job_15           Resource_0        7.0            1374 ms              1
   job_16           Resource_6        18.0            5057 ms              0
   job_17           Resource_0        7.0            1175 ms              1
   job_18           Resource_6        18.0            5057 ms              0
   job_19           Resource_6        18.0            5057 ms              0
   job_20           Resource_0        7.0            1016 ms              1
Status value 1 indicat file found and 0 indicate file is not found

===================>>>>> End of Simulation <<<<<<=========================
```

Figure 5.9: Result of Caching Algorithm- Scenario II

b. Prefetching Algorithm

As shown in Figure 5.10, 20 jobs are requested by the user. From them, only 7 jobs are executed from prefetcher storage and the latency is too small for each job. The rest are accessed from the master replica but still not found and the latency is too high. The average latency for the prefetching algorithm is 3769.15 milliseconds.

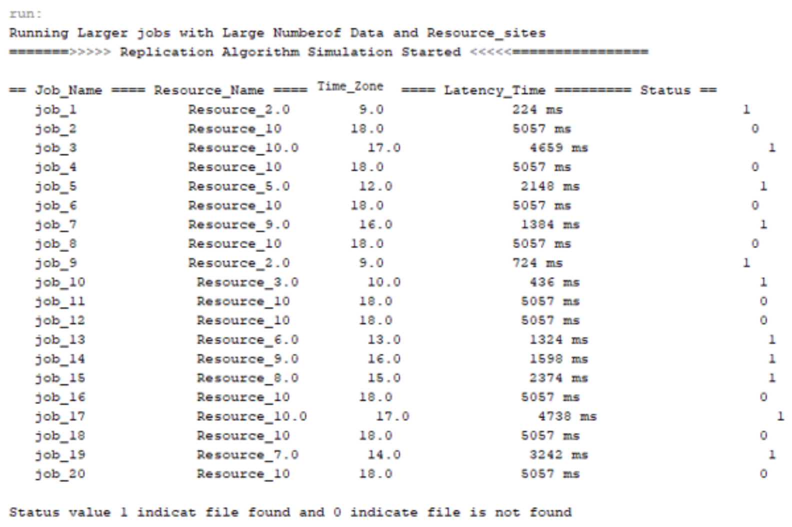


```
run:
Running Larger jobs with Large Numberof Data and Resource_sites
=======>>>>> Prefetching Algorithm Simulation Started <<<<<================

== Job_Name ==== Resource_Name ====  Time_Zone  ==== Latency_Time ========= Status ==
   job_1           Resource_6        18.0            5057 ms              0
   job_2           Resource_1        8.0            1310 ms              1
   job_3           Resource_6        18.0            5057 ms              0
   job_4           Resource_1        8.0            1459 ms              1
   job_5           Resource_6        18.0            5057 ms              0
   job_6           Resource_1        8.0            1506 ms              1
   job_7           Resource_6        18.0            5057 ms              0
   job_8           Resource_6        18.0            5057 ms              0
   job_9           Resource_6        18.0            5057 ms              0
   job_10           Resource_1        8.0            1178 ms              1
   job_11           Resource_6        18.0            5057 ms              0
   job_12           Resource_1        8.0            1501 ms              1
   job_13           Resource_6        18.0            5057 ms              0
   job_14           Resource_6        18.0            5057 ms              0
   job_15           Resource_6        18.0            5057 ms              0
   job_16           Resource_1        8.0            1297 ms              1
   job_17           Resource_6        18.0            5057 ms              0
   job_18           Resource_6        18.0            5057 ms              0
   job_19           Resource_1        8.0            1391 ms              1
   job_20           Resource_6        18.0            5057 ms              0

Status value 1 indicat file found and 0 indicate file is not found

===================>>>>> End of Simulation <<<<<<=========================
```

Figure 5.10: Result of Prefetching Algorithm- Scenario II

c. Dynamic Replication Algorithm

As shown in Figure 5.11, 20 jobs are requested by the user. From them, 11 jobs are executed from replicas and the latency is too small for each job, and the rest are accessed from the master replica but still not found and the latency is too high. The average latency for the dynamic replication algorithm is 3418.2 milliseconds.

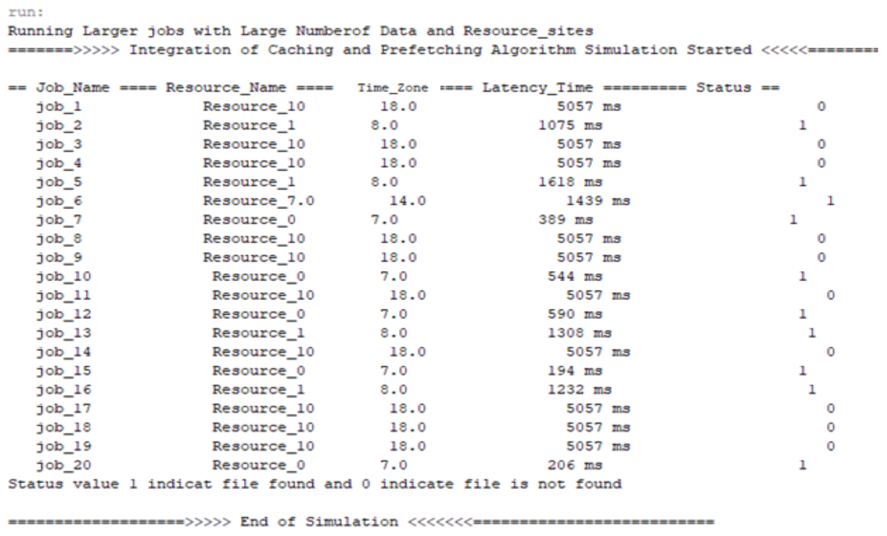


```
run:
Running Larger jobs with Large Numberof Data and Resource_sites
=======>>>>> Replication Algorithm Simulation Started <<<<<=================

== Job_Name ==== Resource_Name ==== Time_Zone  ==== Latency_Time ========= Status ==
   job_1            Resource_2.0        9.0              224 ms                 1
   job_2            Resource_10        18.0              5057 ms                 0
   job_3            Resource_10.0        17.0              4659 ms                 1
   job_4            Resource_10        18.0              5057 ms                 0
   job_5            Resource_5.0        12.0              2148 ms                 1
   job_6            Resource_10        18.0              5057 ms                 0
   job_7            Resource_9.0        16.0              1384 ms                 1
   job_8            Resource_10        18.0              5057 ms                 0
   job_9            Resource_2.0        9.0              724 ms                 1
   job_10            Resource_3.0        10.0              436 ms                 1
   job_11            Resource_10        18.0              5057 ms                 0
   job_12            Resource_10        18.0              5057 ms                 0
   job_13            Resource_6.0        13.0              1324 ms                 1
   job_14            Resource_9.0        16.0              1598 ms                 1
   job_15            Resource_8.0        15.0              2374 ms                 1
   job_16            Resource_10        18.0              5057 ms                 0
   job_17            Resource_10.0        17.0              4738 ms                 1
   job_18            Resource_10        18.0              5057 ms                 0
   job_19            Resource_7.0        14.0              3242 ms                 1
   job_20            Resource_10        18.0              5057 ms                 0

Status value 1 indicat file found and 0 indicate file is not found
```

Figure 5.11: Result of Dynamic Replication Algorithm- Scenario II

d. Integrated Caching and Prefetching Algorithm

As shown in Figure 5.12, 20 jobs are requested by users. From them, 10 jobs are executed from cacher and prefetcher storage and the latency is too small, and the rest are accessed from the master replica but still not found and the latency is too high. The average latency for the integrated caching and the prefetching algorithm is 2958.25 milliseconds.

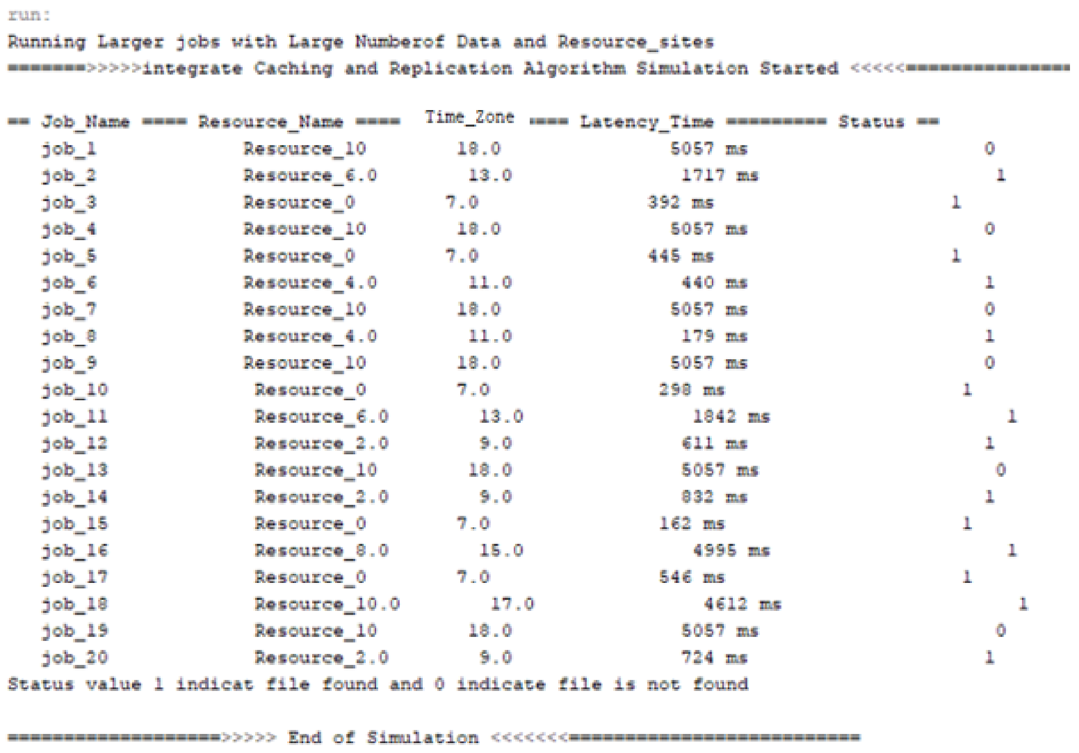


```
run:
Running Larger jobs with Large Numberof Data and Resource_sites
=======>>>>> Integration of Caching and Prefetching Algorithm Simulation Started <<<<<========

== Job_Name ==== Resource_Name ====  Time_Zone ==== Latency_Time ========= Status ==
   job_1            Resource_10        18.0              5057 ms                 0
   job_2            Resource_1        8.0              1075 ms                 1
   job_3            Resource_10        18.0              5057 ms                 0
   job_4            Resource_10        18.0              5057 ms                 0
   job_5            Resource_1        8.0              1618 ms                 1
   job_6            Resource_7.0        14.0              1439 ms                 1
   job_7            Resource_0        7.0              389 ms                 1
   job_8            Resource_10        18.0              5057 ms                 0
   job_9            Resource_10        18.0              5057 ms                 0
   job_10            Resource_0        7.0              544 ms                 1
   job_11            Resource_10        18.0              5057 ms                 0
   job_12            Resource_0        7.0              590 ms                 1
   job_13            Resource_1        8.0              1308 ms                 1
   job_14            Resource_10        18.0              5057 ms                 0
   job_15            Resource_0        7.0              194 ms                 1
   job_16            Resource_1        8.0              1232 ms                 1
   job_17            Resource_10        18.0              5057 ms                 0
   job_18            Resource_10        18.0              5057 ms                 0
   job_19            Resource_10        18.0              5057 ms                 0
   job_20            Resource_0        7.0              206 ms                 1
Status value 1 indicat file found and 0 indicate file is not found

===================>>>>> End of Simulation <<<<<<<==========================
```

Figure 5.12: Result of Caching and Prefetching Algorithm- Scenario II

e. Integrated Caching and Dynamic Replication Algorithm

As shown in Figure 5.13, 20 jobs are requested by the user. From them, 13 jobs are executed from cacher and replica and the latency is too small, and the rest are accessed from the master replica but still not found and the latency is too high. The average latency for the integrated caching and dynamic replication algorithm is 2406.85 milliseconds.

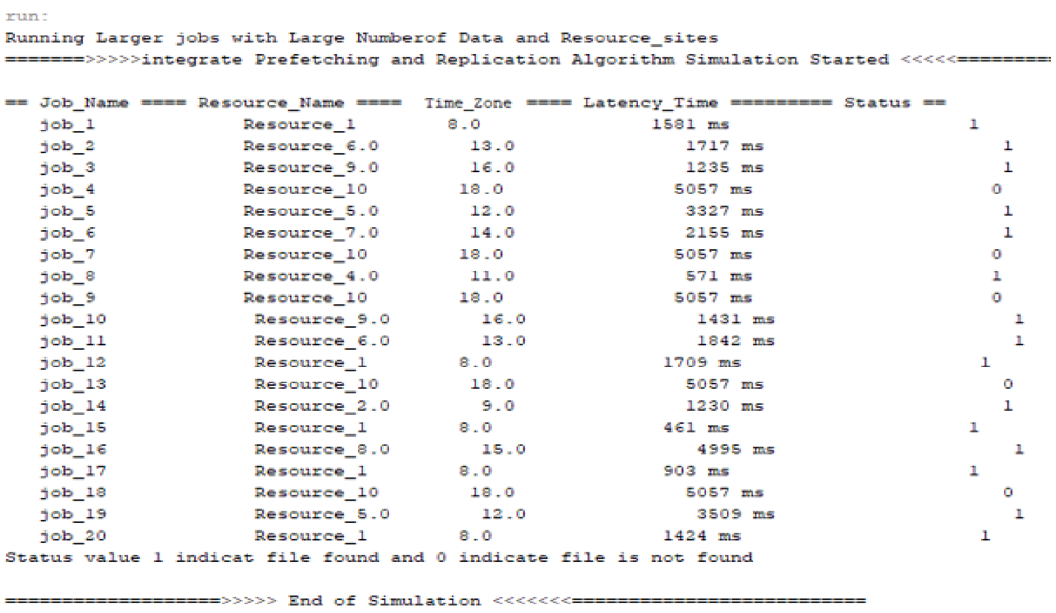


```
run:
Running Larger jobs with Large Numberof Data and Resource_sites
=======>>>>>integrate Caching and Replication Algorithm Simulation Started <<<<<===============

== Job_Name ==== Resource_Name ====  Time_Zone ==== Latency_Time ========= Status ==
   job_1           Resource_10       18.0             5057 ms                  0
   job_2           Resource_6.0       13.0             1717 ms                  1
   job_3           Resource_0       7.0              392 ms                1
   job_4           Resource_10       18.0             5057 ms                  0
   job_5           Resource_0       7.0              445 ms                1
   job_6           Resource_4.0       11.0             440 ms                  1
   job_7           Resource_10       18.0             5057 ms                  0
   job_8           Resource_4.0       11.0             179 ms                  1
   job_9           Resource_10       18.0             5057 ms                  0
   job_10           Resource_0       7.0              298 ms                1
   job_11           Resource_6.0       13.0             1842 ms                  1
   job_12           Resource_2.0       9.0             611 ms                  1
   job_13           Resource_10       18.0             5057 ms                  0
   job_14           Resource_2.0       9.0             832 ms                  1
   job_15           Resource_0       7.0              162 ms                1
   job_16           Resource_8.0       15.0             4995 ms                  1
   job_17           Resource_0       7.0              546 ms                1
   job_18           Resource_10.0       17.0             4612 ms                  1
   job_19           Resource_10       18.0             5057 ms                  0
   job_20           Resource_2.0       9.0             724 ms                  1
Status value 1 indicat file found and 0 indicate file is not found

==================>>>>> End of Simulation <<<<<<==========================
```

Figure 5.13: Result of Integrated Caching and Dynamic Replication Algorithm- Scenario II

b. Integrated Prefetching and Dynamic Replication Algorithm

As shown in Figure 5.14, 20 jobs are requested by the user. From them, 15 jobs are executed from prefetcher storage and replicas and the latency is too small, and the rest are accessed from the master replica but still not found and the latency is too high. The average latency for the integrated prefetching and dynamic replication algorithm is 2668.75 milliseconds.

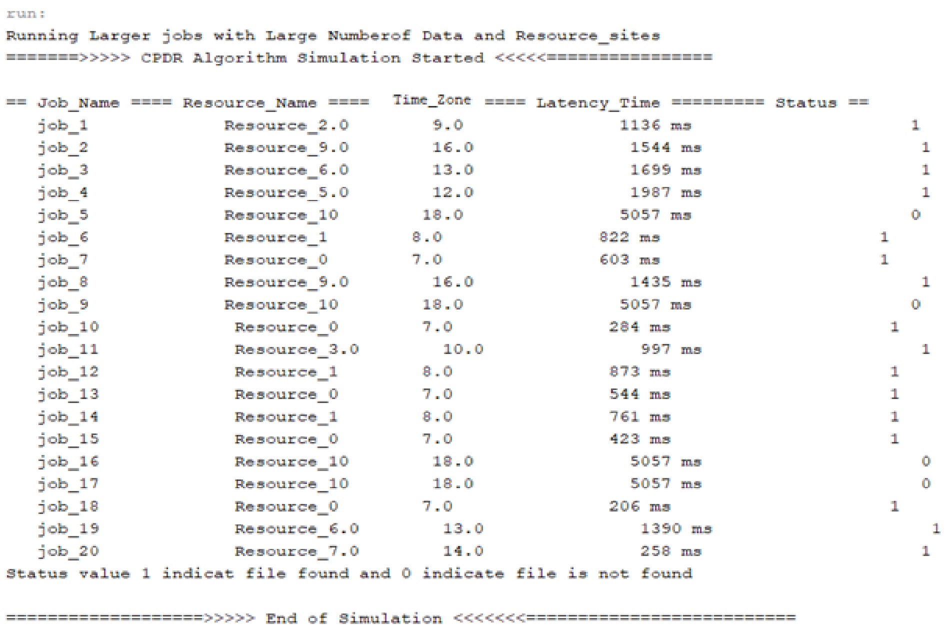


```
run:
Running Larger jobs with Large Numberof Data and Resource_sites
=======>>>>>integrate Prefetching and Replication Algorithm Simulation Started <<<<<=========

== Job_Name ==== Resource_Name ====  Time_Zone ==== Latency_Time ========= Status ==
   job_1            Resource_1         8.0           1581 ms                    1
   job_2            Resource_6.0        13.0            1717 ms                    1
   job_3            Resource_9.0        16.0            1235 ms                    1
   job_4            Resource_10        18.0            5057 ms                   0
   job_5            Resource_5.0        12.0            3327 ms                    1
   job_6            Resource_7.0        14.0            2155 ms                    1
   job_7            Resource_10        18.0            5057 ms                   0
   job_8            Resource_4.0        11.0            571 ms                    1
   job_9            Resource_10        18.0            5057 ms                   0
   job_10            Resource_9.0        16.0            1431 ms                    1
   job_11            Resource_6.0        13.0            1842 ms                    1
   job_12            Resource_1         8.0           1709 ms                    1
   job_13            Resource_10        18.0            5057 ms                   0
   job_14            Resource_2.0         9.0            1230 ms                    1
   job_15            Resource_1         8.0           461 ms                    1
   job_16            Resource_8.0        15.0            4995 ms                    1
   job_17            Resource_1         8.0           903 ms                    1
   job_18            Resource_10        18.0            5057 ms                   0
   job_19            Resource_5.0        12.0            3509 ms                    1
   job_20            Resource_1         8.0           1424 ms                    1
Status value 1 indicat file found and 0 indicate file is not found

===================>>>>> End of Simulation <<<<<<<==========================
```

Figure 5.14: Result of Integrated Prefetching and Dynamic Replication Algorithm- Scenario II

g. CPDR

As shown in Figure 5.15, 20 jobs are requested by the user. From them, 16 jobs are executed from cacher storage, prefetcher storage and replicas and the latency is too small, and the rest are accessed from the master replica but still not found and the latency is too high. The average latency for the CPDR algorithm is 1759.5 milliseconds.



```
run:
Running Larger jobs with Large Numberof Data and Resource_sites
=======>>>>> CPDR Algorithm Simulation Started <<<<<================

== Job_Name ==== Resource_Name ====  Time_Zone ==== Latency_Time ========= Status ==
   job_1            Resource_2.0        9.0            1136 ms                 1
   job_2            Resource_9.0        16.0            1544 ms                 1
   job_3            Resource_6.0        13.0            1699 ms                 1
   job_4            Resource_5.0        12.0            1987 ms                 1
   job_5            Resource_10        18.0            5057 ms                0
   job_6            Resource_1         8.0           822 ms                1
   job_7            Resource_0         7.0           603 ms                1
   job_8            Resource_9.0        16.0            1435 ms                 1
   job_9            Resource_10        18.0            5057 ms                0
   job_10            Resource_0         7.0           284 ms                1
   job_11            Resource_3.0        10.0            997 ms                 1
   job_12            Resource_1         8.0           873 ms                1
   job_13            Resource_0         7.0           544 ms                1
   job_14            Resource_1         8.0           761 ms                1
   job_15            Resource_0         7.0           423 ms                1
   job_16            Resource_10        18.0            5057 ms                0
   job_17            Resource_10        18.0            5057 ms                0
   job_18            Resource_0         7.0           206 ms                1
   job_19            Resource_6.0        13.0            1390 ms                 1
   job_20            Resource_7.0        14.0            258 ms                 1
Status value 1 indicat file found and 0 indicate file is not found

===================>>>>> End of Simulation <<<<<<<=========================
```

Figure 5.15: Result of CPDR Algorithm- Scenario II

h. Summary for Unconstrained Resources

This comparison is done by taking the average latencies of the above algorithms experimented with for unconstrained resources. As shown in Figure 5.16, the average latencies of each algorithm are 3315.75, 3769.15, 3418.2, 2958.25, 2406.85, 2668.75 and 1759.5 for caching, prefetching, dynamic replication, integrated caching and prefetching, integrated caching and dynamic replication. integrated prefetching and dynamic replication, and CPDR respectively. The average latency of prefetching and dynamic replication is also high. This is because the number of users/jobs, the number of data within each resource site and the distribution of resource sites. In addition to the efficiency of each algorithm also have a great impact on the latency for an unconstrained resource. CPDR algorithm also has the lowest latency since the jobs have many options to access instead of requested to the remote server by considering the above parameters in constrained resources.

Note: in section 5.4.2, we also use the words too small and too high to explain the relative latency of each job in a given algorithm. As shown in the Status column of each Figure, too small latency jobs have the status value of 1 and too high latency jobs (rest jobs) have the status value of 0. In this section, the magnitude of too small is less than 5057 ms and the magnitude value of too high (rest jobs) is equal to 5057 ms.

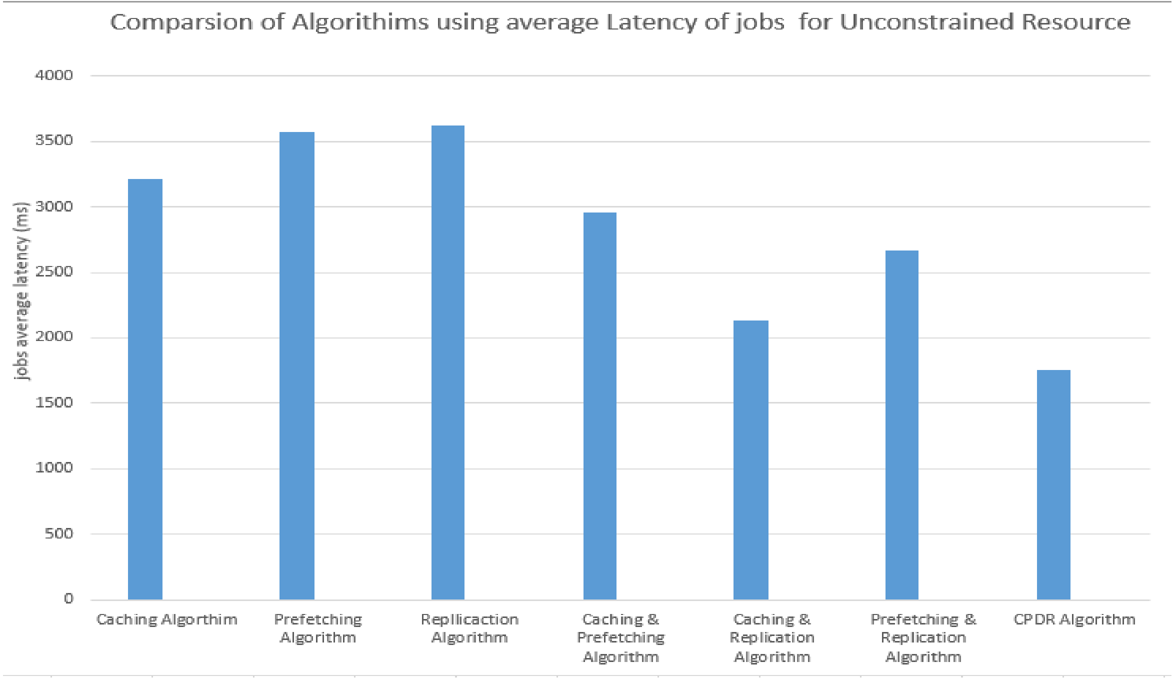


Figure 5.16: Comparison of the above Algorithms- Scenario II

The overall evaluation summary is presented in Table 5.3. The comparison of total latency is based on three perspectives. The first is with the perspective of the parameter within each algorithm (*i.e.*, queuing time, execution time and transfer time) the detail discussed in Equation 2.1. The second is with the perspective to other algorithms and the last is concerning resource constrained and resource unconstrained. Too Small, Small, Medium, High and Too high is given by considering (comparing) algorithms result within each scenario. This has a relatively different range from scenario to scenario. *E.g.*, the range of total latency for constrained scenario high (87-120), medium (60-87), and low (<60), and for unconstrained Too High (>3000), High (2000-3000) and Medium (<2000). This range set is given by considering Figures 5.8 and 5.16. The value of the rest is given by considering the following: Queuing Time depends on the number of jobs, execution Time depends on the number of data within the storage and Transfer time depends on the distance of the resource site from the user.

Table 5.3: Summary of the Above Evaluation metrics

Algorithm	*Resource constrained*				*Resource Unconstrained*			
	Queuing Time	execution *Time*	*Transfer Time*	*Total average Latency*	*Queuing Time*	execution *Time*	*Transfer Time*	*Total average Latency*
Caching	Small	Small	Too Small	High	Medium	Medium	Too Small	Too High
Prefetching	Small	Small	Small		Medium	Medium	Small	
Dynamic replication	Small	Small	Small		Medium	Medium	Medium	
Caching and Prefetching	Small	Medium	Medium	Medium	Medium	High	Medium	High
Caching and Dynamic Replication	Small	Medium	Medium		Medium	High	High	
Prefetching and Dynamic Replication	Small	Medium	Medium		Medium	High	High	
CPDR	Small	Medium	Medium	Low	Medium	High	High	Medium

Note: the total latency is calculated by taking the average latency of each job whether it is accessed from the cacher, prefetcher, or replicas or the remote server. As shown in Table 5.3, the total latency of caching algorithm is high even if the value of queuing time, transfer time and execution time are too small. This result is because only a few jobs are accessed from the cacher and many of the jobs are accessed from the remote server where its latency is too high.

5.5 Summary

In this Chapter, a prototype for Distributed System to reduce the latency of jobs using the CPDR Algorithm is developed using various system development tools including Java and GridSim as a core simulation library. Various packages of the simulation toolkit, as well as some built-in java classes, are overridden to create a conducive computing environment. Moreover, experiments are conducted by creating two Grid models (resource constrained and resource unconstrained) and running a variety of jobs in both models. We also presented the performance of the proposed algorithm in comparison with Caching, Prefetching, Dynamic Replication, integrated Caching and Prefetching, integrated Caching and Dynamic Replication, and integrated Prefetching and Dynamic Replication algorithms. We also discussed the latency of jobs with different factors. The first factor is queuing time. If a large number of users compete to access or execute a job from a single resource site, the queuing time becomes large. The second factor is execution time. When the number of data in given storage becomes too large executing the job by finding from the storage takes too much time. The third factor is the transfer time. When the number of systems (resource sites) becomes too far from the user location (time zone), the transfer time from the user to the systems and then to the users takes much time. In model (scenario) I access latency is too small since the number of users, the amount of data within the storage and the distance of each resource site are small. As shown in Figures 5.8 and 5.16 the CPDR algorithm gives small total latency since each job has three possible storages (*i.e*., cache storage, prefetcher storage and replicator storage) to execute the jobs before going to the remote server with constrained resources. Even if the number of users, data and distance of resource sites becomes too large, the CPDR algorithm has better performance as compared to the other algorithms. In each algorithm, the latency time is the sum of queuing time, transfer time and execution time. In some cases, as shown from the experiment, the total latency from the prefetch storage or replication is smaller than the cacher. This is because of the larger value of queuing time and/or execution time. Generally, the CPDR algorithm has an outstanding performance. The Gridsim simulation result shows that the CPDR algorithm gives a better result than caching, prefetching, dynamic replication, integrated caching and prefetching, integrated caching and dynamic replication, integrated prefetching and dynamic replication to reduce the latency of distributed systems.

6. Conclusion and Future Work

This Chapter summarizes the major findings in this research work. Moreover, the contributions of the proposed CPDR algorithm and future works are outlined.

6.1 Conclusion

Distributed computing, the sharing of resources across machines (sites) on the network rapidly increases from time to time. To improve the performance of the system either we increase or add hardware (storage, processing element, and bandwidth) or we use software (algorithms). Indeed, having a collection of computing machines (sites) with higher processing and storage capacity contributes a lot to the performance of the system. However, the presence of well-equipped machines by itself does not guarantee high performance. Moreover, the absence of an efficient algorithm in a distributed environment affects the performance of the system. On the other hand, a resource constrained distributed computing environment with a relatively efficient latency reduction mechanism has a better performance. Thus, we proposed an efficient latency reduction algorithm by integrating the existing algorithms that could resolve the aforementioned problems and utilizing storage space efficiently.

Our solution architecture consists of various sub-components that are responsible to handle data and give responses to requests of users to increase the performance of the system. The other important component proposed in this work is the Notifier which is found in the Cacher that stores two basic pieces of information about the status of the Prefetcher. The first information is whether the prefetcher is currently active or not and the second information is the ID of the data that is stored in the Prefetcher. If the prefetcher is Active the requested data ID is found in the notifier and the request will access from the prefetcher else the request will forward to the nearest replica with no wasted time by going and finding the data in the prefetcher. In a distributed system, data access latency has three parts: queuing time, transfer time, and execution time. Each part has its impact on access latency. The proposed system improves the performance of the system by considering these three parts. To decrease the transfer time, the job executes from the cacher, prefetcher, or nearest replica. Execution latency and queuing time of a job are handled using FIFO. The proposed algorithm is examined in two scenarios. This algorithm outperforms its counterparts in both scenarios. Specifically, the first scenario is used to show how the proposed algorithm performs in situations where the number of users/jobs, data and resource sites are small (constrained). In this scenario the access latency is minimal since queuing time, transfer time and execution time are small as compared to the second scenario. The second scenario is used to show how well the algorithm performs when the number of users/jobs, data and resource sites are rapidly increased. In each scenario, the proposed algorithm's access latency (compared to caching, prefetching, dynamic replication, integration of caching and prefetching, integration of caching and dynamic replication and integration prefetching and dynamic replication) is less.

6.2 Major Contributions

The main contributions of this research work are summarized as follows.

- **A new algorithm called CPDR is developed**: the major goal of this research work is to improve the performance of a distributed system by reducing the access time. Hence, this research work conceived and implemented various components to achieve this goal. From an exhaustive experiment and evaluation, it was possible to realize that the response time of the proposed CPDR algorithm is by far less than Caching, Prefetching, Dynamic Replication, integration of Caching and Prefetching, Integration of Caching and Prefetching, Integration of prefetching and Dynamic Replication.   Hence, we would say that CPDR has a considerable contribution to the major effort to solve the latency problem of a distributed system.
- **Efficiently Utilizing storage space**: This research focuses on the latency reduction of a distributed system as well as intelligent use of the storage that controls which data should be evicted when the storage becomes full. In a distributed system, storage is used to store basic and relevant data. Instead of increasing storage capacity each time the number of data increases, it is better to use an efficient algorithm to evict irrelevant data.

6.3 Future Work

The CPDR Algorithm proposed in this research could be used in distributed systems. In addition, various research works could use the findings of this research work. We would say the proposed algorithm is fit to the current requirements of most distributed systems though some issues need further work. Hence, this section presents different areas that can be further improved.

- **Adding load balancing**: replication can work as load balancing but it is not efficient as is. One of the major parameters in the latency calculation is queuing time. When the number of jobs and resource sites is rapidly increasing, starvation of jobs may occur if load balancing is not used. So adding load balancing to the CPDR algorithm may further improve the performance of the system.
- **Consistency**: since the same data may be found on more than one site unless synchronized, data may be inconsistent. So adding a consistency mechanism is believed to give a better result.
- **Security**: this is another future work for our research. A single data can be found/ distributed on different sites and accessed by different users whether these users are authorized or not. This has a security issue. So, adding security or access privilege for each data makes the proposed system more reliable.

References

[1] G. Couloris, J. Dollimor, T. Kinberg, and G. Blair, Distributed Systems - Concepts and Design, 5th Edition ed., UK: Addison-Wesley, Pearson Education, 2012, p. 1067.

[2] Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems: Principles and Paradigms, USA: Prentice-Hall, Pearson Education, 2002.

[3] Nadiminti Krishna, Marcos Dias, and Rajkumar Buyya, "Distributed Systems and Recent Innovations: Challenges and Benefits," in *Grid Computing and Distributed Systems Laboratory*, The University of Melbourne, Australia, 2006.

[4] A. Chervenak, R. Schuler, M. Ripeanu, M. Ali, S. Bharathi, A. Iamitchi, and C. Kesselman, "The Globus Replica Location Service: Design and Experience," *IEEE Trans Parallel Distrib System,* p. 1260–1272, 2009.

[5] Wang Tao, Shihong Yao, Zhengquan Xu, and Shaming Pan, "Dynamic Replication to Reduce access Latency Based on Fuzzy Logic System," *State Key Laboratory for Information Engineering in Surveying, Mapping and Remote Sensing,* no. 60, pp. 48-57, 2017.

[6] Shahin David, Pedram Ghodsnia and Khuzaima Daudjee, "Dynamic Data Allocation with Replication in Distributed Systems," in *30th IEEE International Performance Computing and Communications Conference*, Orlando, USA, 2011.

[7] Dushyant Bansal and Paul A.S. Ward, "Reducing Bandwidth Utilization in peer-to-peer Networks," University of Waterloo, 2002.

[8] Edith Cohen and Haim Kaplan, "Prefetching the Means for Document Transfer: a New Approach for Reducing Web Latency," *Computer Networks and Communications Societies,* vol. 39, p. 437–455, 2002.

[9] Jerome H. Saltzer and M. Frans Kaashoek, "Survey on System I/O Hardware Transactions and Impact on Latency, Throughput, and Other Factors," *Reducing Latency,* 2014.

[10] Abdullah Balamash, Marwan Krunz and Philippe Nain, "Performace Analysis of Client-Side Caching/Prefetching System for Web Traffic," *Computer Networks,* no. 13, pp.

3673-3692, 2007.

[11] N. Mansouri and M. M. Javidi, "A New Prefetching-Aware Data Replication to Decrease Access Latency in Cloud Environment," *Journal of System and Software,* vol. 144, pp. 197-215, 2018.

[12] David J. Yates, Abdelsalam A. Heddaya and Sulaiman A. Mirdad, "Method and System for Distributed Caching, Prefetching, and Replication," vol. 19, 1997.

[13] Maarten Steen Van and Andrew S. Tanenbaum, "A brief introduction to distributed systems," *Distributed Systems, Principles and Paradigms,* pp. 967-1009, 2016.

[14] Mohamed Firdhous, "Implementation of Security in Distributed Systems," *A Comparative Study: International Journal of Computer Information Systems,* vol. 2, 2011.

[15] Tang Ming, Bu Sung Lee, Xueyan Tang, and Chai Kiat Yeo, "The Impact of Data Replication on Job Scheduling Performance in the Data Grid," *Future Generation Computer Systems,* vol. 22, p. 254 –268, 2006.

[16] Myungchul Kim, Samuel T. Chanson, and Son T. Vuong, "Concurrency Model for Distributed Systems," *Journal of Parallel and Distributed Computing,* vol. 59, pp. 445-464, 1999.

[17] Kapil Arora and Dhawaleswar Rao Ch, "Web Cache Page Replacement by Using LRU and LFU Algorithms with Hit Ratio: A Case Unification," *International Journal of Computer Science and Information Technologies,* vol. 5, no. 3, pp. 3232 - 3235, 2014.

[18] Irena Balin, "Building P2P network s with the Help of Bluetooth on IOS and Android Devices," 15 Nov. 2013. [Online]. Available: http://www.dbbest.com/blog/buiIding-p2p-networks-with-the-help-of-bluetooth-on-ios-and-android-devices. [Accessed 26 Nov. 2020].

[19] Snehal Nayak, Meera Narvekar and Debajyoti Mukhopadhyay, "Cooperative Caching Technique in Peer to Peer Mobile Environment," in *5th International Conference on System Modeling & Advancement in Research Trends,* Moradabad, 2016.

[20] Nong Xiao, Fang Liu, Yingjie Zhao, and Zhiguang Chen, "Dual Cache Replacement Algorithm Based on Sequentiality Detection," *Science China. Information Sciences,* vol.

55, no. 1, pp. 191-199, 2011.

[21] K.W. Froese and R.B. Bunt, "The Effect of Client Caching on File Server Workloads," in *System Sciences Hawaii International Conference*, Wailea, USA, 2006.

[22] Griffioen J. and Appleton R., "Reducing File System Latency Using a Predictive Approach," in *Proceedings of the 2000 USENIX Annual Technical Conference,* Boston, 1999.

[23] Gwan Hwan Hwang, Hsin Fu Lin, Chun Chin Syf, and Chiu Yang Chang, "The Design and Implementation of Appointed File Prefetching for Distributed file Systems," *Journal of Research and Practice in Information Technology,* vol. 40, no. 2, 2008.

[24] Ming Tang, Bu Sung Lee, Xueyan Tang, and Chai Kiat Yeo, "The Impact of Data Replication on Job Scheduling Performance in the Data Grids," *International Journal of Future Generation of Computer Systems,* vol. 22, no. 3, p. 254–268, 2006.

[25] J.H. Abawajy, H.M. Suzuri and M.Mat Deris, "An Efficient Replicated Data Access Approach or Large-scale Distributed Systems," *IEEE International Symposium on Cluster Computing and the Grid,* pp. 588-594, 2004.

[26] P. A. Bernstein, "Middleware: a model for distributed system services," *Commun. ACM,* vol. 39, no. 2, p. 87–98, 1996).

[27] B. Michael, "Hiding Distribution in Distributed Systems," in *13th International Conference on Software Engineering*, Austin, 2000.

[28] Ajay D. Kshemkalyani and Mukesh Singhal, "Distributed Computing Principles, Algorithms and Systems", Cambridge University Press, 2008.

[29] S. Neeraj, "Distributed System Security Knowledge Area," *The National Cyber Security Centre*, no. 1, 2019.

[30] Jiang Bian and R. Seker, "A Secure Distributed File System Computational Intelligence in Cyber Security," *CICS '09. IEEE Symposium on, Nashville,* pp. 76-82, 2009.

[31] Zhiqian Xu and Hai Jiang, "HASS: Highly Available, Scalable and Secure Distributed Data Storage Systems," in *International Conference on Computational Science and Engineering*, Vancouver, Canada, 2009.

[32] B., Neumann, "Scale in Distributed Systems," *Readings in Distributed Computing*

Systems, pp. 463-489, 1994.

[33] Lampson Butler, "Atomic Transactions," *Distributed Systems and Architecture and Implementation*, pp. 246-265, 2000.

[34] D. Riddoch, "Low Latency Distributed Computing", Cambridge: Downing College, 2002.

[35] Songnian Zhou, Michael Stumm, Kai Li and David Wortman, "Heterogeneous Distributed Shared Memory," *IEEE Transactions on Parallel and Distributed Systems,* vol. 3, no. 5, p. 540–545, 1992.

[36] I. Grigorik, High Performance Browser Networking, O'Reilly Media, 2013.

[37] A. Froehlich, "Network Bandwidth," Network Detection Response, [Online]. Available: https://searchnetworking.techtarget.com/definition/bandwidth. [Accessed 24 March 2021].

[38] S. Vazhkudai, S. Tuecke, and I. Foster, "Replica selection in the globus data grid," *ACM International Symposium on Cluster Computing and the Grid,* pp. 106-113, 2001.

[39] M. Deris, J.H. Abawajy and H. Suzuri, "An Efficient Replicated Data Access Approach for Large-Scale Distributed Systems," in *IEEE/ACM International Symposium on Cluster Computing and the Grid*, Chicago, USA, 2004.

[40] Azarias Reda, Yidnekachew Haile, and Brian Noble, "Distributing Private Data in Challenged Network Environments," in *Proceedings of the 19th International Conference on World Wide Web,* ACM, 2010.

[41] Dharma Nukarapu, Bin Tang, Liqiang Wang and Shiyong Lu, "Data Replication in Data Intensive Scientific Applications with THE Performance guarantee," *IEEE Trans Parallel Distrib. Syst.*, vol. 22, no. 8, p. 1299–1306, 2011.

[42] Tomoya Enokido and Makoto Takizawa, "An Integrated Power Consumption Model for Distributed Systems," *IEEE Transactions on Industrial Electronics,* vol. 60, no. 2, pp. 824 - 836, 2013.

[43] Morris Sloman, Emil Lupu and Jorge Lobo, Policy for Distributed Systems and Networks, Bristol, UK: Springer, 2001.

[44] Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy, "Rate of Change and

Other Metrics: a Live Study of the World Wide Web," *Proceedings of USENIX Symposium on Internet Technologies and Systems,* 1997.

[45] Bzoch Pavel and Jiri Safarik, "Simulation of Client-side Caching Policies for Distributed File System," in *Eurocon*, Zagreb, Croatia, July 2013.

[46] Yifeng Zhu, Hong Jiang, Xiao Qin, D. Feng and D.R. Swanson, "Improved Read Performance in a Cost-Effective, Fault Tolerant Parallel Virtual File System," in *Proceedings 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, Tokyo, Japan, 2003.

[47] Bzoch Pavel, Lubos Matejka, Ladislav Pesicka and Jiri Safari, "Design and Implementation of a Caching Algorithm Applicable to Mobile Clients," *Informatica,* vol. vol. 36, no. 4, pp. 369-378, 2012.

[48] Andrea Araldo, Dario Rossi and Fabio Martignon, "Cost-Aware Caching: Caching More (Costly Items) for Less (ISPs Operational Expenditures," *IEEE Trans. Parallel Distrib. Syst.,* vol. 27, no. 5, p. 1316 –1330, 2016.

[49] M. Chrobak and J. Noga, "LRU is Better Than FIFO," *Algorithmica,* vol. 23, p. 180–185, 1999.

[50] Reed Benjamin and Darrell D., "Analysis of Caching Algorithms for Distributed File Systems," *ACM SIGOPS Operating Systems Review,* vol. 30, no. 3, pp. 12-21, 1996.

[51] Stefan Podlipnig and Laszlo Boszormenyi, "A Survey of Web Cache Replacement Strategies," *ACM Computing Surveys,* vol. 35, no. 4, pp. 374-398, 2003.

[52] Hong-Tai Chou and David J. DeWitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems," *Algorithmica,* vol. 1, p. 311–336, 1986.

[53] A. Boukerche and Raed Al-Shaikh, "Building a Fault Tolerant and Conflict-Free Distributed File System for Mobile Clients," in *Advanced Information Networking and Applications*, USA, 2006.

[54] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, S.H. Noh, Sang Lyul Min, Yookun Cho and Chong Sang Kim, "LRFU: a Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies," *Computers IEEE Transactions,* vol. 50, no. 12, pp. 1352 -1361, 2001.

[55] H. Yang, "An Improved Adaptive Policy Based on Recency and Frequency," in *International Conference on Advances in Mechanical Engineering and Industrial Informatics*, China, 2015.

[56] Padmanabhan V. and Mogul J., "Using Predictive Prefetching to Improve world wide web latency," *ACM Sigcomm Computer Communication Review,* vol. 26, p. 22–36, 1999.

[57] M. Ai Assaf, Jiang X., Abid M. and Qin X, "Eco Storage: A Hybrid Storage System with Energy-Efficient Informed Prefetching," *Journal of Signal Processing Systems,* vol. 72, p. 165–180, 2013.

[58] S. Jiang, X. Ding, Y. Xu and K. Davis, "A Prefetching Scheme Exploiting both Data Layout and Access History on Disk," *ACM Transaction on Storage* , vol. 9, no. 3, p. 23 , 2013.

[59] Jeremy Stribling, Yair Sovran, Irene Zhang, Jinyang Li, M. Frans Kaashoek, Robert Morris and Xavid F. Pretzer, "Flexible, Wide Area Storage for Distributed Systems with WheelFS," *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation,* p. 43–58, 2009.

[60] Jianwei Liao, Francois Trahay, Guoqiang Xiao, Yutaka Ishikawa and Li Li, "Performing Initiative Data Prefetching in Distributed File Systems for Cloud Computing," *IEEE Transactions on Cloud Computing,* vol. 1, no. 1, p. 99, 2015.

[61] Tehmina Amjad, Muhammad Sher and Ali Daud, "A Survey of Dynamic Replication Strategies for Improving Data Availability in Data Grids," *Future Generation Computer Systems,* vol. 28, no. 2, pp. 337-349, 2012.

[62] Atakan Dogan, "A Study on Performance of Dynamic File Replication Algorithms for Real-Time File Access in Data Grids," *Future Generation Computer Systems,* vol. 25, no. 8, pp. 829-839, 2009.

[63] Uras Tos, Abdelkader Hameurlain, Tolga Ayav and Sebnem Bora, "Dynamic Replication Strategies in Data Grid Systems," *Journal of Super Computing,* vol. 71, pp. 4116-4140, 2015.

[64] amidreza Rashidy Kanan, amidreza Rashidy Kanan and Mahsa Beigrezaei, "New Fuzzy

Based Algorithm in Data Grid," in *13th Iranian Conference on Fuzzy Systems*, Iran, 2013.

[65] Sang-Min Park, Jai-Hoon Kim, Young-Bae Ko and Won-Sik Yoon, "Dynamic Data Replication Strategy Based on Internet Hierarchy BHR," *Lecture notes in Computer Science,* vol. 3033, pp. 838-846, 2004.

[66] K. Sashi and Antony Selvadoss Thanamani, "Dynamic Replication in a Data Grid Using a Modified BHR Region Based Algorithm," *Future Generation Computer Systems,* vol. 27, no. 2, pp. 202-210, 2011.

[67] Najme Mansouri, Gholam Hosein and Dastghaibyfard, "A Dynamic Replica Management Strategy in Data Grid," *Journal of Network and Computer Applications,* vol. 35, no. 4, pp. 1297-1303, 2012.

[68] Najme Mansouria, Gholam Hosein, Dastghaiby fardb and Ehsan Mansouric, "Combination of Data Replication and Scheduling Algorithm for Improving Data Availability in Data Grids," *Journal of Network and Computer Applications,* vol. 36, no. 2, pp. 711-722, 2013.

[69] Mahsa Beigrezaei, Abolfazle Toroghi Haghighat, Mohamed Reza Meybodi and Maryam Runiassy, "A New Pre-fetching and Prediction Based Replication Algorithm in DaAta Grid," in *6th International Conference on Computer and Knowledge Engineering,* Mashhad, Iran, 2017.

[70] Eunsam Kima and Jonathan C.L.Liub, "An Integrated Prefetching/Caching Scheme in Multimedia Server," *Journal of Network and Computer Applications,* vol. 88, no. 15, pp. 112-123, 2017.

[71] Feng Luab, Ziqian Shiab, Lin Guab, Hai Jinab and Laurence TianruoYangbc, "An Adaptive Multi-Level Caching Strategy for Distributed Database System," *Future Generation Computer Systems,* vol. 97, pp. 61-68, 2019.

[72] A. Feldmann, R. Caceres, F. Douglis, G. Glass and M. Rabinovich, "Performance of Web Proxy Caching in Heterogeneous Bandwidth Environments," in *Proceedings of the IEEE INFOCOM’99 Conference,* New York, USA, 2002.

[73] Dan Duchamp, "Prefetching Hyperlinks," in *Proceedings of the 2nd USENIX*

Symposium on Internet Technologies and Systems, 1999.

[74] M. Carman, F. Zini, L. Serafini and K. Stockinger, "Towards an Economy Based Optimization of File Access and Replication on a Data Grid," in *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, Berlin, Geremany, 2002.

[75] Anirban Chakrabarti, R. A. Dheepak and Shubhashis Sengupta, "Integration of Scheduling and Replication in Data Grids," in *High Performance Computing* , USA, 2004.

[76] Sonali Warhade, Prashant Dahiwale and M. M. Raghuwanshi, "A Dynamic Data Replication in Grid System," *Procedia Computer Science,* vol. 78, pp. 537-543, 2016.

[77] Pei Cao, "Opportunities and Challenges for Caching and Prefetching on Mobile Devices," in *Third IEEE Workshop on Hot Topics in Web Systems and Technologies*, Washington, USA, 2016.

[78] M. C. Little and S. K. Shrivastava, "A Method for Combining Replication with Caching," in *IEEE Symposium on Reliable Distributed Systems*, England, 2002.

[79] V. Wietrzyk, R. Schmid, R. Lawson and V. Khandelval, "Distributed Replication and Caching: A Mechanism for Architecting Responsive Web Services," in *Advanced Information Networking and Applications*, Tokyo, 2005.

[80] Mahsa Beigrezaei, Abolfazl Toroghi, Mohammad Reza and Maryam Runiassy, "A New Pre-Fetching and Prediction Based Replication Algorithm in Data Grid," in *6th International Conference on Computer and Knowledge Engineering*, Mashhad, Iran, 2017.

[81] Ketan Shah, Anirban Mitra and Dhruv Matani, "An O(1) algorithm for implementing the LFU," dhruvbird.com, New York, 2010.

[82] H. YANG, "An improved adaptive policy Based on Recency and Frequency," in *International Conference on Advances in Mechanical Engineering and Industrial Informatics*, China, 2015.

[83] Liao Jianwei, Francois Trahay, Guoqiang Xiao, Li Li and Yutaka Yishikawa, "Performing Initiative Data Prefetching in Distributed File Systems for Cloud Computing," *IEEE Transactions on Cloud Computing,* 2015.

[84] Oracle, "What is new in Java Development Kit 8.1," Oracle, 17 March 2020. [Online]. Available: https://www.infoworld.com/article/3534133/jdk-15-the-new-features-in-java-15.html. [Accessed 24 March 2021].

[85] NetBeans Org., "What is new in NetBeans IDE 8.2," [Online]. Available: https://netbeans.org/community /releases/81. [Accessed 1 April 2021].

[86] Rajkumar Buyya and Manzur Murshed, "GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing," *Concurrency and Computation: Practice and Experience,* p. 1175–1220, 2002.

[87] Nong Xiao, YingJie Zhao, Fang Liu, and ZhiGuang Chen, "Dual Queues Cache Replacement Algorithm Based on Sequentiality Detection," *Science China Information Science,* vol. 55, no. 1, pp. 191-199, 2011.

[88] Pavel Bzoch, Lubos Matejka, Ladislav Pesicka and Jiri Safarik, "Towards Caching Algorithm Applicable to Mobile Clients," in *Federated Conference on Computer Science and Information Systems (FedCSIS)*, Wroclaw, Poland, 2012.

[89] H. Yang, "An Improved Adaptive Policy Based on Recency and Frequency," in *International Conference on Advances in Mechanical Engineering and Industrial Informatics*, China, 2015.

Annex-Source Code for Resource Creation

This is the sample code used to create one resource site with it characteristics.

```
private static GridResource createGridResource(String name)
{
// Here are the steps needed to create a Grid resource:
// 1. We need to create an object of MachineList to store one/more
Machines
MachineList mList = new MachineList();
//2. Create one Machine with its id, number of PEs and MIPS rating per PE
int mipsRating = 377;      //mips means maximum instruction per second
mList.add( new Machine(0, 1, mipsRating));// First Machine
//3. Repeat the process from 2 if we want to create more Machines
mList.add(new Machine(1, 2, mipsRating)); //Second Machine
mList.add( new Machine(2, 3, mipsRating)); // Third Machine
// 4. Create a ResourceCharacteristics object that stores the
//properties of a Grid resource: architecture, OS, list of   Machines,
//allocation policy: time- or space-shared, time zone and its price
String arch = "X86";       // system architecture
String os = "Window 10”;    // operating system
double time_zone = 9.0;     // time zone this resource located
double cost = 3.0;    // the cost of using this resource
ResourceCharacteristics resConfig = new ResourceCharacteristics(
arch, os, mList, ResourceCharacteristics.SPACE_SHARED,time_zone, cost);
// 5. Finally, we need to create a GridResource object.
double baud_rate = 100.0;      // communication speed
long seed =500;
double peakLoad = 0.0;     // the resource load during peak hour
double offPeakLoad = 0.0;   //the resource load during off-peak hr
double holidayLoad = 0.0;    // the resource load during holiday
// incorporates weekends so the grid resource is on 7 days a week
LinkedList Weekends = new LinkedList();
Weekends.add(new Integer(Calendar.SATURDAY));
Weekends.add(new Integer(Calendar.SUNDAY));
LinkedList Holidays = new LinkedList();
GridResource gridRes = null;
try{
ResourceCalendar resCalendar = new ResourceCalendar(time_zone,
peakLoad, offPeakLoad, holidayLoad, Weekends, Holidays, seed);
// create a storage, which demands the storage size in MB, but the
// description we get is in GB.
 Storage storage = new HarddriveStorage("storage",storage_size * 1);
gridRes = new GridResource(name, baud_rate, resConfig,  resCalendar, obj);
gridRes.addStorage(storage);
} catch (Exception e) {
System.out.println("msg = " + e.getMessage() );
}
System.out.println("Creates Grid resource with name = " + name);
return gridRes;
}
```

Declaration

I, the undersigned, declare that this thesis is my original work and has not been presented for a degree in any other university and that all sources of materials used for the thesis have been duly acknowledged.

Declared by:

Name: Yilkal Binalf Worku

Signature: ___________________________

Date: _______________________________

Confirmed by advisor:

Name: Mulugeta Libsie (PhD)

Signature: ___________________________

Date: _______________________________