Placement of Replicas in Large-Scale Data Grid Environments

by

Mohammad Shorfuzzaman

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Doctor of Philosophy

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
March 2012

Thesis advisor Author

Dr. Rasit Eskicioglu and Dr. Peter Graham Mohammad Shorfuzzaman

Placement of Replicas in Large-Scale Data Grid Environments

Abstract

Data Grids provide services and infrastructure for distributed data-intensive applications accessing massive geographically distributed datasets. An important technique to speed access in Data Grids is replication, which provides nearby data access. Although data replication is one of the major techniques for promoting high data access, the problem of replica placement has not been widely studied for large-scale Grid environments. In this thesis, I propose improved data placement techniques useful when replicating potentially large data files in wide area data grids. These techniques are aimed at achieving faster data access as well as efficient utilization of bandwidth and storage resources. At the core of my approach is a new highly distributed replica placement algorithm that places data in strategic locations to improve overall data access performance while satisfying varying user/application and system demands. This improved efficiency of access to large data will improve the practicality of large-scale data and compute intensive collaborative scientific endeavors.

My thesis makes several contributions towards improving the state-of-the-art for replica placement in large-scale data grid environments. The major contributions are: (i) development of a new popularity-driven dynamic replica placement algorithm for hierarchically structured data grids that balance storage space utilisation and access latency; (ii) creation of an adaptive version of the base algorithm to dynamically adapt the frequency and degree of replication based on such factors as data request arrival rates, available storage capacities, etc.; (iii) development of a new highly distributed algorithm to determine a near-optimal replica placement while minimizing replication cost (access and update) for a given traffic pattern; (iv) creation of a distributed QoS-aware replica placement algorithm that supports multiple quality requirements both from user and system perspectives to support efficient transfers of large replicas.

Simulation results using widely observed data access patterns demonstrate how the effectiveness of my replica placement techniques is affected by various factors such as grid network characteristics (i.e. topology, number of nodes, storage and workload capacities of replica servers, link capacities, traffic pattern), QoS requirements, and so on. Finally, I compare the performance of my algorithms to a number of relevant algorithms from the literature and demonstrate their usefulness and superiority for conditions of interest.

Contents

List of Figures

Acknowledgments

I would like to thank my advisors, Dr. Rasit Eskicioglu and Dr. Peter Graham, for their continuous guidance, encouragement, and contributions in the development of this work. They were a constant source of support, without which this thesis could not have been completed. I would also like to extend my thanks to all my committee members, Dr. Alexander Reinefeld, Dr. Jeff Diamond and Dr. John Anderson for providing excellent reviews of my research, which helped me compile a quality thesis.

I would like to extend my sincere gratitude to the Faculty of Graduate Studies for supporting me with the University of Manitoba Graduate Fellowship throughout my PhD. I also appreciate the financial support from the Faculty of Science, University of Manitoba.

Finally, a special thanks to my family members for their constant support and perseverance to accomplish this endeavor. Above all, I would like to express my profound gratitude to the Almighty and the Majestic for enabling me complete this thesis and granting me continuing success in my life.

Chapter 1

Introduction

This chapter presents an introduction of this thesis. It starts with a high-level overview of the key concepts related to the research problem addressed. Then the fundamental motivations behind this research are stated and the proposed solution to address the research challenges is briefly presented. The chapter ends with a discussion on the research contributions and the organization of the rest of this thesis.

1.1 Grid Computing

The next-generation of large-scale scientific applications in areas as diverse as high-energy physics, molecular modeling, and earth sciences involves the processing of large datasets from simulations or from large-scale experiments [ABB+02; ABB+01; AFN+01; FAC+01; Che02; CFK+00]. Analysis of these datasets and their dissemination among researchers located over a wide geographical area require high capacity resources such as supercomputers, high bandwidth networks, and mass storage systems. Many such applications may also require new paradigms that address issues such as multi-domain applications, co-operation and co-ordination of resource owners and removing system boundaries. Grid computing [KF98] is one such paradigm that enables the aggregation of large scale computing, storage, and networking resources. A grid provides an environment where a widely distributed scientific community can share its resources, across different administrative and organizational domains, to solve large-scale compute- and data-intensive problems and collaborate in a wide variety of disciplines. A grid, therefore, enables the creation of a virtual environment including a pool of physical resources across different administrative domains; these resources are then abstracted into computing or storage units that can be transparently accessed and shared by large numbers of remote users.

The concepts used in grid computing are not new. The invention of networking and the introduction of network operating systems enabled access to resources across geographically distributed locations [CDK01]. More technological advances brought up by parallel processing and distributed computing allowed not only remote access to resources but also the simultaneous sharing of these distributed resources by different remote users [CDK01]. Parallel processing enabled different tasks to be run simultaneously on different, usually homogeneous computers, and to compete for access to computational resources. Using distributed computing, users can employ widely distributed heterogeneous computers to run jobs that require more resources than may be available in local networks and laboratories. The emerging need for more resources and also for collaborative problem solving in cost efficient ways led to the development of middleware that transparently provides access to distributed resources and route data from back-end sources to end-user applications in a seamless and relatively scalable manner; this became known as meta-computing and later "computing on the Grid" [KF98]. Grid computing has the potential to support different kinds of applications. These include compute-intensive applications, data-intensive applications and applications requiring distributed services. Various types of grids have been developed to support these applications and have been categorized as follows [YBdA+07].

1. **Computational Grids.** These provide distributed computing facilities for executing compute-intensive applications, such as Monte Carlo simulations [AGK00], and Bag-of-Tasks (BoT) applications [CBS+03], where each consists of a collection of independent tasks or jobs. UC Berkley leads the SETI@home [ACK+02] project that takes advantage of extra processing power from a distributed collection of volunteered systems. SETI@home allows users to download and install their program which in turn uses the idle cycles of personal computers to get data from a SETI@home server over the Internet, analyze that data, and then report the results in the hope at finding extraterrestrial life forms. Another project, Nimrod-G [BAG00], utilizes grids to schedule compute-intensive applications on available resources.

2. **Data Grids.** These provide the infrastructure to access, transfer and manage large datasets stored in distributed repositories. Data grids typically focus on satisfying the requirements of scientific collaborations, where there is a need for analyzing large collections of data and sharing the results. Such applications are found in areas including astronomy [Hol01], climate simulation [AFN+01; FAC+01], and high energy physics [RAD+02]. The amount

of data required to analyze natural phenomena and conduct experiments in these applications is very large. These experiments in turn generate large data sets that need to be collected and appropriately stored in geographically distributed data centers to be accessed for further processing at different additional sites [BCF03; HJMS$^+$00; TTGD04; Hol01; CFK$^+$00]. Experiments in high energy particle physics such as those running at the European Center for Nuclear Research (CERN) (e.g. the CMS and ATLAS [Hol01; eur01; gri01a; SSA$^+$02] experiments running on the Large Hadron Collider (LHC) produce and collect massive amounts of data and involve thousands of researchers from around the world to analyze the data and initiate future experiments. High performance data grid architectures facilitate these requirements by applying a number of underlying technologies in a coordinated fashion to support data intensive petabyte scale applications.

3. **Interaction Grids.** These provide services and platforms for users to interact with each other in a real-time environment, e.g. Access Grid [CDO$^+$00]. This type of grid is suitable for multimedia applications, such as large-scale video conferencing, and those that otherwise require fast networks.

4. **Application Service Provisioning (ASP) Grids.** These concentrate on providing access to remote applications, modules, and libraries hosted at data centers or on computational grids(e.g. NetSolve [SYAD05]).

5. **Knowledge Grids.** These work on knowledge acquisition, data processing, and data management. Moreover, they provide business analytics services driven

by integrated data mining services. Some projects in this field are Knowledge Grid [CT03] and the EU Data Mining Grid [EU:08].

6. **Utility Grids.** These focus on providing one or more of the above grid services to end-users as information technology (IT) utilities on a pay-to-access basis. In addition, they set up a framework for the negotiation and establishment of contracts, and allocation of resources based on user demands. Existing projects in this area are Utility Data Center [GPS03], at the enterprise level and Gridbus [BV04] at the global level.

This thesis focuses specifically on data grid infrastructures and aims at developing techniques to achieve faster data access as well as efficient utilization of bandwidth and storage resources. Based on existing data grid applications we envision that the size of the data is expected to be multiple terabyte or even petabyte scale for some applications. Maintaining a local copy of such data on each site that needs the data is cost prohibitive due to consistency management and storage requirements. Also, storing such huge amounts of data in a centralized manner is impractical due to the slowness of remote data access and concerns about a single point of failure. Given the high latency of wide-area networks that underlie many grid systems, and the need to access or manage multiple petabytes of data in data grid environments, data availability and access optimization become key challenges to be addressed.

Grid systems typically involve loosely coupled jobs that require access to a large number of datasets. In most situations, the datasets requested by a user's job cannot be found at the local nodes in the data grid. In this case, data must be fetched from other nodes in the grid which incurs high access latency. An important technique to speed up access in data grid systems is to replicate data at multiple locations, so a user can access the data from a nearby site [VBR06]. This thesis presents a family of algorithms that intelligently and transparently places data in strategic locations to improve overall data access performance while satisfying varying user, system, and application demands.

1.2 Motivation for Replication in Data Grids

Data grids primarily deal with providing services and infrastructure for distributed data-intensive applications that need to access, transfer and modify massive datasets stored in distributed storage resources. A data grid aims to present the following capabilities to its users: (a) ability to search through numerous available datasets for the required dataset and to discover suitable data resources for accessing the data, (b) ability to transfer large-sized datasets between resources in minimal time, (c) ability for users to manage multiple copies of their data, (d) ability to select suitable computational resources and process data on them, and (e) ability to manage access permissions for the data. Therefore, data grids aim to combine high-end computing technologies with high-performance networking and wide-area storage management techniques.

To realize these abilities, a data grid needs to provide tools, services and APIs (Application Programming Interfaces) for managing collaborative access to data and computational resources. These include administration tools to manage authenticating and authorizing widely dispersed members for accessing disparate resources and data collections, data search tools to allow users to discover datasets that may be available within a collaboration, intelligent data replication and caching techniques to ensure that the users can access the required datasets in the fastest and/or cheapest manner, resource management services and APIs to allow applications and users to utilize the infrastructure effectively by processing the data at idle resources that offer better turnaround times and reduced costs, and so on. This thesis, however, concentrates only on replication techniques for placing data in strategic locations in the grid.

Availability and efficient accesses are critical requirements in many data intensive applications [Che02; GKL$^+$02]. Delayed accesses due to availability problems or non-responsiveness may cause undesired results. To effectively address these challenges, the need for data replication is apparent. For example, consider the case of data distribution that follows a hierarchical structure. In the CMS and ATLAS experiments, for example, the data grid system spans worldwide distributed locations, and is organized in "Tiers". Tier 0 represents the main site located at CERN (European Center for Nuclear Research), Tier 1 includes national centers, Tier 2 represents regional centers that cover one region of a large country such as a state in the USA or a smaller country, Tier 3 represents workgroup servers, and Tier 4 the (thousands of) researchers' workstations and desktops. In this scenario, all data is collected at CERN, located in Geneva, Switzerland. It is preprocessed online and stored in the CERN computer center, which is Tier 0 in the data grid hierarchy. Subsets of that data are then replicated at national centers in France, Germany, Italy, Canada, the USA, and so on. Meanwhile, smaller subsets of the data are replicated at individual institutions such as individual universities. Existing systems such as the Globus

Replica Location Service (RLS) [Che02] enable users to locate replicas of a given data file. A physicist working at Caltech can use the RLS to find the location of the data originally collected at the experiment site. It is possible that the data is already located on a storage server at Caltech, in which case access to the data is fast. However, if the data is located only at CERN, network latencies are likely to result in slow data access. A good replication strategy can be used to anticipate and/or analyze the users' requests for data and to place subsets of the data and replicas at strategic locations. Given the size of the data sets and number of users, it is difficult to make manual decisions about where the data needs to be placed. An automated data replication system is needed that can take into account the data access patterns by multiple users and applications.

In addition to achieving access efficiency and availability, data replication can also be used to improve data locality and increase robustness and scalability for many applications. Replication of data has been demonstrated to be a practical and efficient method to achieve high network performance in distributed environments, and has been applied in some data grid applications [RF01b; CFK$^+$00]. Creating replicas effectively distributes client requests to different replica sites and offers higher access speed than a single server. At the same time, the workload on the original server is distributed across the replica servers and, therefore, also decreases significantly. Additionally, the network load is also distributed across multiple network paths thereby decreasing the probability of congestion-related performance degradation. In these ways, replication plays a key role in improving the performance of data-intensive computing in data grids. Replication is, of course, limited by the amount of storage available at each site in a data grid and by the bandwidth available between those sites and access locations. Experience from parallel and distributed systems design shows that replication promotes high data availability, lower bandwidth consumption, increased fault tolerance, and improved scalability. However, the replication algorithms used in such systems cannot always be directly applied to data grid systems due to the wide-area (mostly hierarchical) network structures and different data access patterns seen in data grid systems.

1.3 Research Problem and Solution Strategy

A scientist located at a small university may need to run a time consuming processing job on a huge data set. She may choose to get the data from where it exists to the local computing resource and run the job there. Alternatively, it may be better to transfer the job to where the data exists or, both the job specification and the data may be sent to a third location that will perform the computation and return the results to the scientist. My focus here is only on the data distribution aspect of a grid. When a user generates a request for a file, large amounts of bandwidth could be consumed to transfer the file from the server to the client. Furthermore, the latency involved could be significant considering the size of the files involved and the geographic distance between source and sink. New challenges are also faced in the grid. For example, huge data file sizes, system resources belonging to multiple owners, and dynamically changing resources and user behaviors. To address these challenges, my research investigates the usefulness of creating replicas by distributing data sets among the various locations in the grid.

Replication methods can be classified as static or dynamic [TLYT05]. For static replication, after a replica is created, it will be stored in the same place until it is deleted. The drawback of static replication is evident – when client access patterns change, the benefits brought by replicas may decrease. On the contrary, dynamic replication takes into consideration changes in the data grid environment and automatically creates new replicas for referenced data files or moves the replicas to other sites as needed to improve performance. Although the usefulness of replication in data grid systems is evident, it entails a number of issues such as replica placement, resource discovery and management, selecting suitable replicas, the impact of replication on the performance of job scheduling, replica consistency maintenance, and so on. In this thesis, I concentrate on the replica placement issue. The overall file replication problem consists of making the following decisions [RF01b]: (1) which files should be replicated; (2) when and how many replicas should be created; and (3) where the replicas be placed in the system.

Although a substantial amount of work has been done on data replication in grid systems, most of it has focused on infrastructure for replication and mechanisms for creating and deleting replicas [CSK+05; BCC+03; BCCS+03**?** ; HJMS+00; LSsD02; RIF02; SSA+02]. However, to obtain the maximum benefit from replication, a strategic placement of replicas considering many factors is essential. In this thesis, I study replica placement in data grid systems taking into account a number of important issues. The first is to consider the highly dynamic behavior of a data grid environment where resource availability, network latency, and users requests may vary constantly. Such a system needs a replication strategy that uses available storage capacities, data access arrival rates, network state, and other factors to determine the frequency and degree of replication to reduce data access time and to use network and storage resources efficiently. Secondly, in grids where operation control is decentralized and resources are under the control of their own local administrative domains, placing the replicas of an object through a centralized algorithm is undesirable. This motivates decentralized replica placement algorithms for large-scale data grid environments to ensure improved scalability and reliability of the system. Another important issue is to consider the quality of service (QoS) requirements imposed by data requesters and grid infrastructure. As noted earlier, grid computing infrastructure usually consists of various types of resources and the performance of these resources can be quite diverse. Moreover, different sites may have different service quality requirements according to the system performance of the sites. Base quality of service requirements can be specified in the form of a general distance metric or access time deadlines by users, and the system must ensure that there exists a replica of a server within that quality range to answer the request. Finally, workload capacity constraints on the servers must be considered and it must be recognized that the bandwidth capacity of a link is bounded. The average number of requests serviced by a replica server directly affects the average response time of the requesting sites due to queuing delays and network congestion.

In my thesis, I address the aforementioned problems in large-scale data grids to improve the performance of data access while ensuring efficient use of both computational and storage resources. To this end, I propose and evaluate a family of efficient algorithms for replica placement in a hierarchical data grid structure (as is common in current data grid systems [BCCS$^+$03; LCG01; RF01b; RF01a]). My basic popularity-driven dynamic replica placement strategy, called Popularity Based Replica Placement (PBRP), for hierarchical data grids, aims to increase data access performance from the perspective of the clients by dynamically creating replicas for "popular" files. In the real world, some files will be more popular than others (e.g. current or "hot" areas of experimentation in ATLAS or CMS). Moreover, data access patterns may change over time, so any dynamic replication strategy must keep track of file access histories to decide on when, what and where to replicate. The "popularity" of a file is determined by its recent access rate by the clients. It is assumed that recently popular files will tend to be accessed more frequently than others in the near future. The idea behind PBRP is to create replicas as close as possible to those clients that frequently request the corresponding files subject to storage availability. The effectiveness of this algorithm depends on the selection of a threshold value that relates to the popularity of files and is used to determine their placement in the hierarchy. I also propose an adaptive version of this algorithm that determines the threshold dynamically considering data request arrival rates and available storage capacities at the replica servers resulting in faster access and more efficient use of bandwidth and storage in spite of changing access patterns and loading conditions.

Further, I develop a *distributed* popularity based replica placement (DPBRP) algorithm the goal of which is to determine optimal locations for replicas to minimize overall replication cost/overhead (access and update) for a given traffic pattern (i.e. a recurring pattern of access frequencies from clients for different files) while offering enhanced performance and reliability due to its distributed nature. The use of dynamic programming is natural in distributed systems. In particular, when designing a dynamic programming algorithm for a hierarchical structure, results of a function computed at child nodes can be combined to give the result for the parent. I show that my replica placement problem can be formulated as a dynamic programming problem and its solution can be obtained for hierarchical data grids in a distributed fashion.

To satisfy the QoS requirements specified both from the user (imposed by data requests) and system perspectives, I then describe a modified DPBRP algorithm to determine the locations of replicas to improve system performance while satisfying quality of service requirements simultaneously. I consider the replica distance as the user QoS requirement where each client may specify a maximum distance allowable to the nearest replica server. The system must ensure that a replica server exists to answer all user requests. Since the QoS requirement can be specified by distance (e.g. number of hops) between the client and the replica server, this distance constraint can easily be incorporated when the replication costs for clients are calculated considering various distance possibilities for the replica servers. The bandwidth constraint on a link is specified by an upper bound on link capacity. The constraint mandates that the amount of data that can pass through the link over a period of time will be limited to that bound. If the total amount of data passing through a link is greater than its capacity constraint, then the link is congested. Furthermore, it is assumed that the workload capacity of a replica server is bounded which means a node hosting a replica can only process up to a certain number of requests per unit of time from the clients of its subtree. My replication strategy has to ensure that the user requests are satisfied while limiting the workload of each replica server to its capacity and ensuring that none of the communication links is congested.

To evaluate the performance of my algorithms, I use the OptorSim [BCC+03] data grid simulator applied to a hierarchical data grid structure reflecting the characteristics of the Compact Muon Spectrometer (CMS) experiments [Hol01] at CERN. Simulations are done to evaluate the performance of the algorithms and to demonstrate how the effectiveness of replica placement is affected by various factors, such as replica server capacities, link capacities, and data access rates and patterns. I also compare the performance of my algorithms to a number of other existing replica placement algorithms.

1.4 Contributions

This thesis makes several contributions towards improving the understanding of replication for data grid environments, particularly towards advancing the area of replica placement in large-scale hierarchical data grids. The major contributions are as follows:

- A popularity-driven dynamic replica placement algorithm (PBRP) for hierarchically structured data grids that improves the performance of data access while ensuring efficient use of network, and storage resources. Performance of this replica placement algorithm has been evaluated using a modified version of OptorSim and compared against various prior solutions with good results derived from making better trade-offs between storage costs and runtime.

- Since a grid environment is highly dynamic, resource availability, network latency, and user requests vary. To address these issues the system needs a dynamic replica placement strategy that adapts to the dynamic behavior in data grids. A novel adaptive placement algorithm (APBRP) has been designed to use available storage capacities and data access arrival rates to tune the frequency and degree of replication. The results obtained are superior to PBRP and thus, transitively, to the other tested algorithms.

- Due to decentralized operation and resources being under the control of their own local administrative domains, placing the replicas of an object through a centralized algorithm is undesirable. This motivates distributed replica placement algorithms for large-scale grid environments to ensure improved scalability and reliability. Thus, I developed a new highly distributed and decentralized replica placement algorithm for data grids that transparently places data in strategic locations to improve the overall data access performance. The replication mechanism is guided by a replication cost function that takes into account current demand for the data, network resources availability, and storage capacity. A dynamic programming algorithm is presented to determine a near-optimal replica placement for large-scale hierarchical data grids while minimizing replication cost.

- A QoS-aware replica placement problem has been formulated to guide development of a replication strategy that supports QoS requirements from users and involves minimal replication cost.

- An extended QoS-aware distributed placement algorithm has been designed to generate a near-optimal solution of the replica placement problem taking into account workload capacity of replica servers and link capacity constraints while supporting efficient transfers of large replicas. The novelty of this extended QoS-aware algorithm comes from the ability to integrate multiple types of QoS both from user and system (workload and link capacity constraints) perspectives. Performance of this extended QoS-aware algorithm has been evaluated and compared against two prior QoS-aware replica placement algorithms with results.

- Extensive performance evaluation over a wide range of system parameters has been carried out from which a number of useful observations are obtained regarding deploying my replica placement algorithms in a real data grid.

1.5 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 reviews the background material on data grids and data replication in grid environments. Existing dynamic replication strategies in data grids are also discussed. Chapter 3 provides motivation for my work and gives a problem definition. In Chapter 4, I propose a popularity-driven dynamic replica placement algorithm to be used in hierarchically structured data grids and also presented an adaptive version of this replica placement algorithm. I also discuss the performance of these two placement algorithms and compare them to a number of other existing replica placement strategies. Chapter 5 presents my distributed replica placement approach. It starts with a base distributed algorithm and then develops QoS-aware techniques that support QoS requirements both from the user and system perspectives. Simulation results are then presented. Chapter 6 compares my centralized and distributed replica placement algorithms and provides an overall assessment of their applicability. The thesis concludes in Chapter 7 with a summary of contributions made and a brief discussion of some possible areas for future work.

Chapter 2

Background and Related Work

This chapter provides a general overview of data grids and a detailed discussion on the issues and challenges in data replications. Data replication is a well-known technique and has been realized and extensively used within the context of other distributed data-intensive paradigms such as the World Wide Web, peer-to-peer file-sharing networks, and distributed and mobile databases. I discuss the caching or replication strategies in each of these environments and address the similarity and differences with data grid replication. The chapter ends with a discussion on the existing techniques for replica placement in data grids and some of the problems and issues that I will address in this thesis.

2.1 Preliminaries

Data grid applications produce, manipulate or analyse data in the range of hundreds of GegaBytes (GB) to hundreds of PetaBytes (PB) and beyond. The data is organised as collections or *datasets* that are stored on mass storage systems (also called *repositories*). The datasets are accessed by users in different locations who may create local copies or *replicas* of the datasets to reduce the latency involved in wide-area data transfer. A replica can be defined as a complete copy of the original dataset. A *data replication mechanism* allows users to create, register, access and update replicas. The replica management system may also create replicas on its own, guided by *replication strategies* that consider current and expected future demand for the datasets, locality of requests and storage capacity of the repositories. *Metadata*, or “data about data”, is information that describes the datasets and could consist of attributes such as name, time of creation, size on disk and time of last modification. Metadata may also contain specific information such as details of the process that produced the data. A *replica catalog* contains information about the locations of datasets and their associated replicas. Users can query the catalog to locate the nearest replica of a particular dataset. A catalog may be centralized or distributed.

2.2 Data Grids: An Overview

The efficient management of huge distributed and shared data resources across Wide Area Networks (WANs) is a significant challenge for both scientific research and commercial applications. The data grid [CFK$^+$00] as a specialization and extension of the Grid [BBL06] provides one possible solution to this problem. Essentially, data grids deal with providing services and infrastructure for distributed data-intensive applications that need to access, transfer and modify massive datasets stored in distributed storage servers. At the minimum, a realistic data grid must pro-

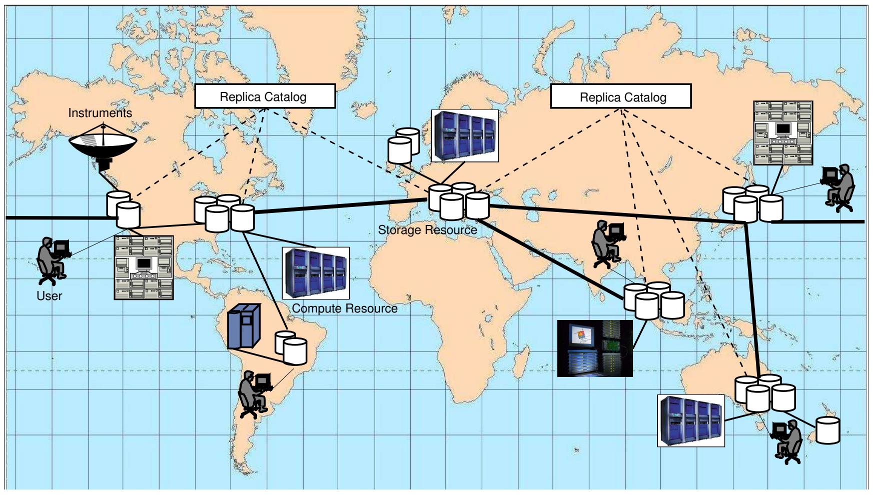


Figure 2.1: A high level view of a data grid [VBR06]

vide two basic capabilities: a high-performance, reliable data transfer mechanism, and a scalable replica discovery and management mechanism [CFK$^+$00]. Depending on application requirements, other services may also be needed (e.g. security, accounting, etc.). Examples of such services include consistency management for replicas, metadata management and data filtering and reduction mechanisms. All operations in a data grid are typically mediated by a security layer that handles authentication of entities and ensures conduct of only authorized operations.

Figure 2.1 shows a high-level view of a hypothetical worldwide data grid consisting of computational and storage resources located in different countries that are connected by high speed networks. The thick lines show high bandwidth networks linking the major data centres and the thinner lines correspond to lower bandwidth network links that connect the subsidiary centres to those major data centres. The data generated from an instrument, experiment, simulation or a network of sensors is usually stored at its principal storage site and transferred to the other storage sites around the world on request through the data replication mechanism. Users can query their local replica catalog to locate datasets that they require. The data is fetched from a repository local to their area, if it is available there; otherwise it must be fetched from a remote repository. As an example, data may be transmitted to a computational site such as a cluster or a supercomputer facility for processing before the results are sent to a visualisation facility, a shared repository or to the desktops of individual users. A data grid, therefore, provides a platform through which users can access aggregated computational, storage and networking resources to execute their data-intensive applications on remote data. It promotes a rich environment for participating users to analyse data, share results with their collaborators and maintain state information about data seamlessly across institutional and geographical boundaries.

Resources in a grid are often heterogeneous in terms of operating environments, capability and availability and are under the control of their own local administrative domains. These domains are autonomous and possesses the rights to grant users access to the resources under their control. Therefore, data grids must also be concerned with such issues as: authentication and authorization of entities, sharing of resources, and resource management and scheduling for efficient use of available resources. However, certain characteristics of data grids are specific to the applications for which they are targeted. For instance, for astrophysics or high energy physics experiments, the principal instrument such as a telescope or a particle accelerator is the single site of data generation. This means that all data is written at a single site, and then replicated to other sites for, typically, read access. Updates to the source are propagated to the replicas either by the replication mechanism or by a separate consistency management service.

Providing access to different types of resources using grid computing faces numerous challenges. Foster et al. [FKT01] have proposed a grid architecture for resource sharing among different entities based on the concept of Virtual Organizations (VOs). A VO is formed when different organisations pool resources and collaborate to achieve a common goal. A VO defines the resources available for the participants and the rules and conditions for accessing and using the resources. A VO also provides protocols and mechanisms for applications to determine the suitability and accessibility of available resources. The use of VOs impacts the design of data grid architectures in many ways. For example, a VO may be standalone or may be composed of a hierarchy of regional, national and international VOs. In the latter case, the underlying data grid may have a corresponding hierarchy of repositories and the replica discovery and management systems must be structured accordingly. More importantly, sharing of data collections is guided by the relationships that exist between the VOs that own each of the collections.

2.2.1 Architecture

Many initiatives were undertaken by different groups of researchers from different institutions and universities towards building frameworks for computational and data grids. The Globus Toolkit [Fos06; KF98; FKT01] project is a leading effort which provides a framework for building grids based on a service-oriented architecture. The framework offers Security, Information, Resource Management, and Data

Management services. The toolkit relies on a grid Security Infrastructure (GSI), which provides a set of security features to allow users to authenticate their communication and use single sign-on to access grid resources and services. The Information Services provide information about the status of grid resources using a notification approach where resources publish their status and subscribers receive updates about specific instruments, machines, or storage components they want to access. This includes the monitoring as well as the discovery of information resources. The Resource Management component uses input from the Information Services to enable users to access available resources and to allow the system to schedule resource allocations. The Data Management component (or data grid) provides the ability to access and manage data and data resources in the grid. The Globus toolkit provides several components to move, copy, and locate data. The major components are: GridFTP, RFT (Replica File Transfer), and RLS (Replica Location Service) [FKT01; Fos06]. GridFTP provides tools for fast, secure, convenient, and parallel data transfers in the grid by extending the FTP protocol. The RFT service provides reliable management of multiple GridFTP transfers. RLS maintains and provides access to information about the location of data available within the data grid. The RLS service uses Replica Catalogs to register, index, and locate data. The Replica Catalog is a registry in which users maintain records for all the shared files in the grid. Each record contains the locations of all file replicas, and provides a mapping between file names and these replicas. The aforementioned components of the data grid can be organized in a layered architecture as shown in Figure 2.2, reproduced from descriptions in [CFK$^+$00; ABB$^+$01]. The layers outlined in Figure 2.2 represent different compo-

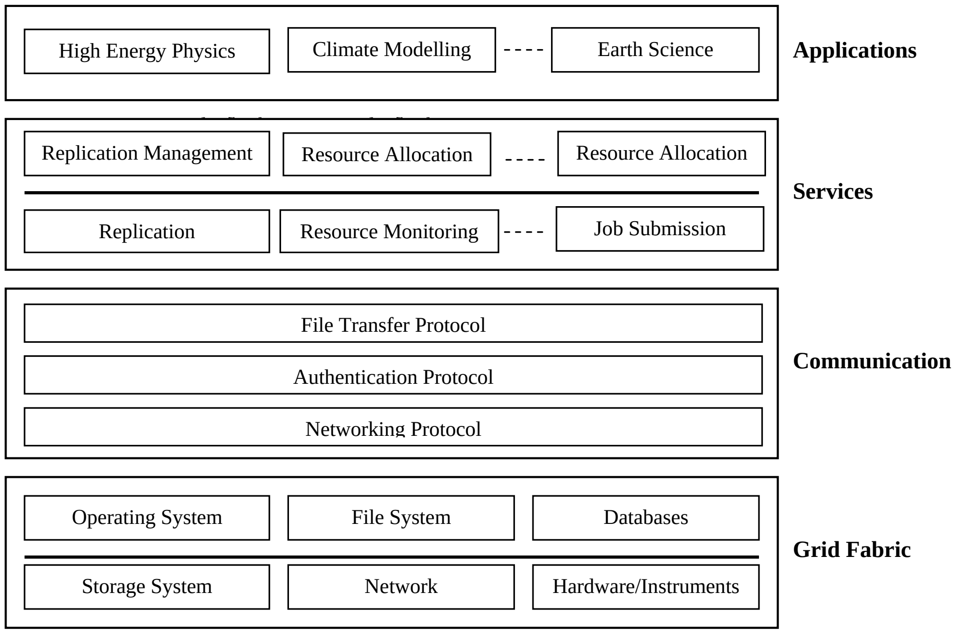


Figure 2.2: A layered view of a data grid architecture

nents of the data grid infrastructure. Components at the same level can co-operate to offer certain services, and components at higher levels use services and components offered at lower levels and build on top of them. The Grid Fabric consists of computing resources such as workstations and supercomputers, storage resources such as disks and tapes, as well as scientific instruments. Most of these resources are widely distributed and are connected by high bandwidth wide area networks. These computing, storage and networking resources represent the physical layer of the data grid. The operating systems and software that manage these heterogeneous resources represent the basic software layer of the data grid. The Communication layer consists of data transfer protocols to copy data from resources in the Grid Fabric layer. The data transfer protocols are typically based on the TCP/IP communication protocol and authentication protocols are used to verify users identities and ensure security and data integrity. The data grid Services layer consists of core services such as replication and resource monitoring that provide transparent discovery, location, and access to data and compute resources. These services can be used to contribute resources to the grid and define access policies for these resources. The monitoring service is equivalent to the Information Services in the Globus architecture and implementation [FKT01]. The next components of the Services layer provide higher level services that use the lower level services to enable efficient management and allocation of replicas and data resources in the data grid. The collective set of services provided at this layer represents the data grid middleware. The middleware abstracts and hides the complexity of managing access to resources and provides API's for users and applications to transparently take full advantage of the utilities available in the data grid. The Applications layer provides services and access interfaces that are specific to a domain (such as high energy physics, biology or climate modelling) or Virtual Organization.

2.2.2 Models

A data grid model represents the manner in which data sources are organized in the grid. A variety of models are in place for the operation of a data grid. They are based on the source of data, whether single or distributed, the size of data and the mode of data sharing. A few of the common models found in data grids are shown in

Figure 2.3 and are discussed below:

a) Central Repository: This is the general form of a data grid in which all the data is gathered at a central repository and users obtain the requested data through queries. The data can be from many sources such as distributed instruments and sensor networks and is made available through a centralized interface such as a web portal which also verifies users and checks for authorization. Figure 2.3(a) shows such a grid model which has been applied by the NEESgrid (Network for Earthquake Engineering Simulation) project [PKG^{+}04] in the United States. The difference between this and other models of data grid organization is that there is only a single point for accessing the data. In contrast, with other models, the data can be wholly or partially accessed at different points where it is made available through replication. The central repository may be replicated in this case for fault tolerance but not for improving locality of data. Thus, this model serves better in scenarios where the overhead of replication is not compensated by an increase in efficiency of data access such as the case where all accesses are local to a particular region.

b) Hierarchical: This model maintains a single source for data but the data is distributed across the world for collaborations. For example, the MONARC (Models of Networked Analysis at Regional Centres) group within CERN has proposed a tiered infrastructure model for data distribution [ea00]. This model is presented in Figure 2.3(b) and specifies requirements for transfer of data from CERN to various groups of physicists around the world. The first tier is the compute and storage farm at CERN which stores the data generated from the detector. This data is then distributed to Regional Centers (RCs) located around the world. From the RCs, the

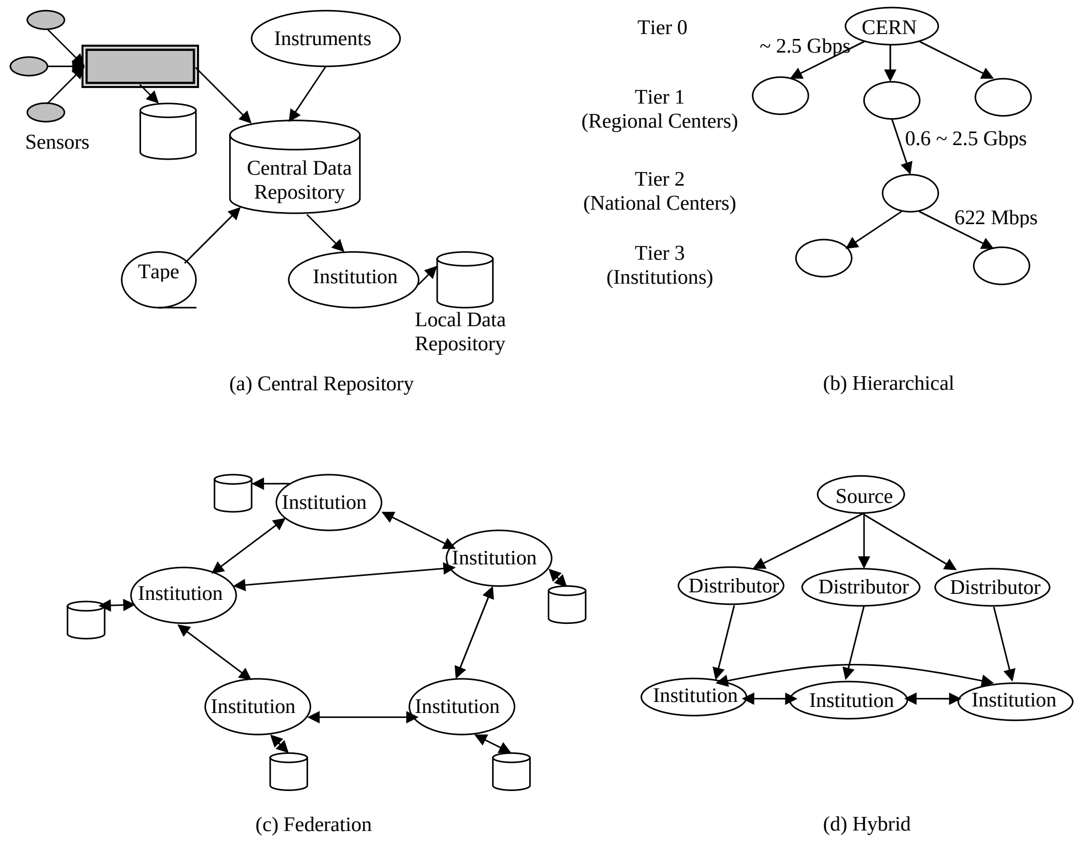


Figure 2.3: Various models for organization of data grids [VBR06]

data is then passed downstream to the national and institutional centers and finally onto the physicists working on the data. The massive amounts of data generated in these experiments motivate the need for a robust data distribution mechanism. Also, researchers at participating institutions may be interested only in subsets of the entire dataset that may be identified by querying using metadata. One advantage of this model is that maintaining consistency is much simpler as there is only one source for the data.

c) Federation: The federation model [RWMS04] is shown in Figure 2.3(c) and is found in data grids created by institutions who wish to share data in existing databases. One example of a federated data grid is the BioInformatics Research Network (BIRN) [bir05] in the United States. Researchers at a participating institution can request data from any of the databases within the federation if they have the proper authentication. Each institution retains control over its local database. Varying degrees of integration can be present within a federated data grid. Moore et al. [MJR$^+$04] pointed out different types of federations that are possible using a Storage Resource Broker (SRB) [BMRW98] in various configurations. The differences are based on the degree of autonomy of each site, constraints on cross-registration of users, degree of data replication and synchronization.

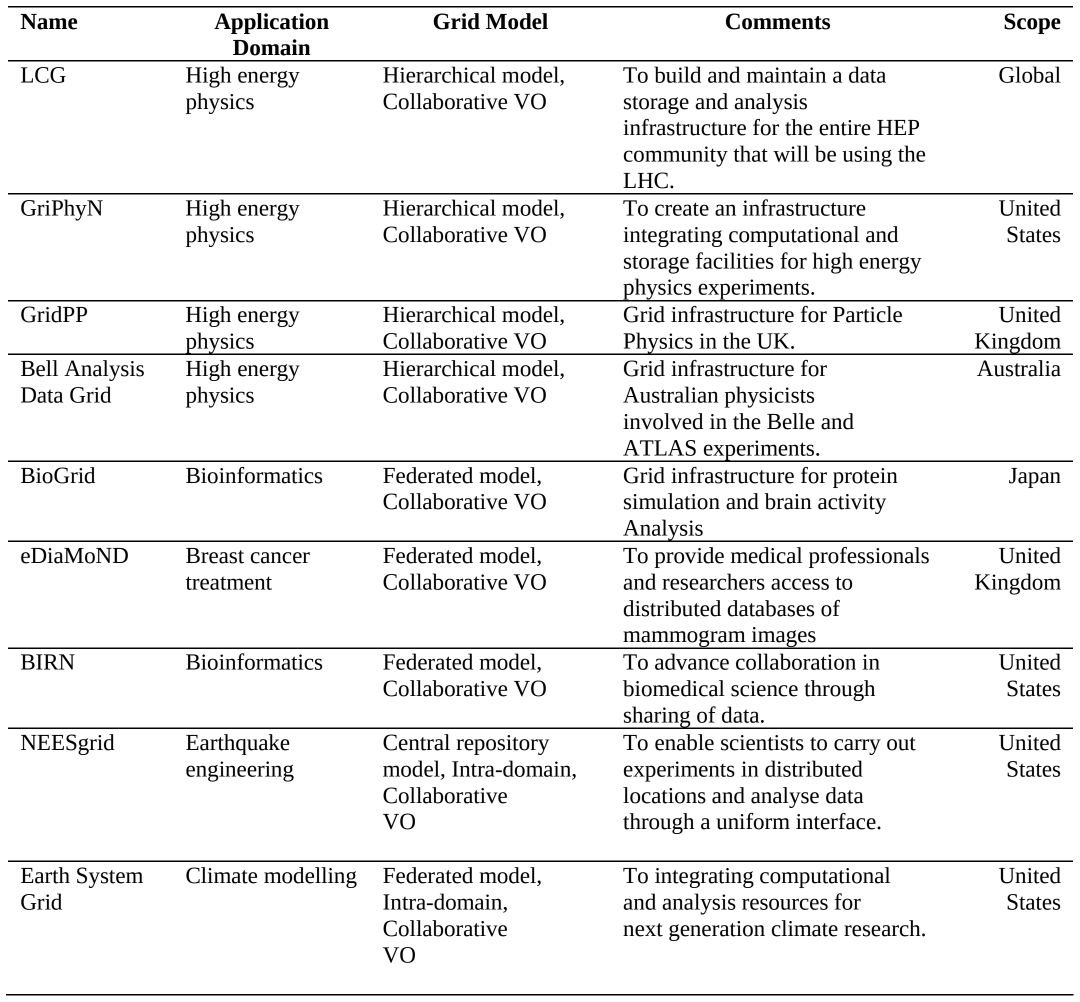
d) Hybrid: Hybrid models that combine the above models are beginning to emerge as data grids are being used by scientists in different domains of applications. These models meet the demand for researchers to collaborate and share the products of their analysis. Figure 2.3(d) shows a hybrid model of a hierarchical data grid with peer linkages at the edges. Lamehamedi et al. [LSsD02] uses a hybrid topology for replica creation in a data grid.

2.2.3 Applications

There are a number of scientific applications in the area of High Energy Physics (HEP), bioinformatics, and earth sciences that are adopting data grid technologies as part of their computing infrastructure. For example, a HEP device called the Large Hadron Collider (LHC) [LCG01], at CERN will produce roughly 15 Petabytes (15 million Gigabytes) of data annually, which thousands of scientists around the world will access. The goal of the LHC Computing Grid (LCG) project is to build and maintain a data storage and analysis infrastructure for the entire high energy physics community that will be using the LHC. ATLAS (A Toroidal LHC ApparatuS) and CMS (Compact Muon Solenoid) are two particle detector experiments that have been constructed at the LHC. The ATLAS project involves approximately 2100 physicists at 167 institutions in 37 countries. At the same time, approximately 2600 physicists from 180 institutions in 32 countries form the collaboration for the CMS project. The data from these LHC experiments will be distributed, world-wide, using a four-tier model with CERN as the root, or tier 0, site. A primary backup of raw data will be stored in tier-0. After initial processing, this data will be distributed to a series of regional tier-1 centers that will make data available to many national tier-2 centers. Individual scientists will access these data through tier-3 centers, which might consist of local clusters in a University Department. All the tier-3 centers of the hierarchy are local sites where users can issue requests to access their required data stored in the data grid system.

There are also a number of other grid projects around the world that are setting up the infrastructure for physicists to process data from HEP experiments. Some of these are the Particle Physics Data Grid (PPDG) [PPD03] and Grid Physics Network (GriPhyN) [gri01a] in the United States, GridPP [Gri01b] in the UK and Belle Analysis Data Grid (BADG) [BAD05] in Australia. These projects have common features such as a hierarchical model for distributing the data, shared facilities for computing and storage and personnel dedicated towards managing the infrastructure.

The increasing importance of realistic modeling and simulation of biological processes along with the need for accessing existing databases has led to data grid solutions being adopted by bio-informatics researchers worldwide. These projects involve federating existing databases and providing common data formats for the information exchange. Among these projects are BioGrid [Bio02] in Japan for online brain activity analysis and protein folding simulation, and the eDiaMoND project [edi01] in the UK for breast cancer treatment and the BioInformatics Research Network (BIRN) [bir05] for imaging of neurological disorders using data from federated databases.



Name	Application Domain	Grid Model	Comments	Scope
LCG	High energy physics	Hierarchical model, Collaborative VO	To build and maintain a data storage and analysis infrastructure for the entire HEP community that will be using the LHC.	Global
GriPhyN	High energy physics	Hierarchical model, Collaborative VO	To create an infrastructure integrating computational and storage facilities for high energy physics experiments.	United States
GridPP	High energy physics	Hierarchical model, Collaborative VO	Grid infrastructure for Particle Physics in the UK.	United Kingdom
Bell Analysis Data Grid	High energy physics	Hierarchical model, Collaborative VO	Grid infrastructure for Australian physicists involved in the Belle and ATLAS experiments.	Australia
BioGrid	Bioinformatics	Federated model, Collaborative VO	Grid infrastructure for protein simulation and brain activity Analysis	Japan
eDiaMoND	Breast cancer treatment	Federated model, Collaborative VO	To provide medical professionals and researchers access to distributed databases of mammogram images	United Kingdom
BIRN	Bioinformatics	Federated model, Collaborative VO	To advance collaboration in biomedical science through sharing of data.	United States
NEESgrid	Earthquake engineering	Central repository model, Intra-domain, Collaborative VO	To enable scientists to carry out experiments in distributed locations and analyse data through a uniform interface.	United States
Earth System Grid	Climate modelling	Federated model, Intra-domain, Collaborative VO	To integrating computational and analysis resources for next generation climate research.	United States

Figure 2.4: Data grid projects in various application domains [VBR06]

Similarly, researchers in earthquake engineering and climate modeling and simulation are also adopting grids to solve their computational and data access requirements. NEESgrid [nee01] is a project to link earthquake researchers with high performance computing and sensor equipment so that they can collaborate on designing and performing experiments. Earth System Grid [ESG00] aims to integrate high-performance computational and data resources to study the petabytes of data resulting from climate modeling and simulation. A list of all these projects and a brief summary of each of them is provided in Figure 2.4.

2.3 Data Replication

This section provides an introduction to data replication and then discusses some general issues and challenges in replication for data grids. Data replication can be applied to provide good performance, high availability, and fault tolerance in a range of distributed systems. When data is stored at a single location on a single data server, that server can be a bottleneck if too many requests need to be served at the same time and consequently, the whole system slows down, i.e. slow response time and limited throughput in terms of requests per second. By providing multiple replicas at multiple locations, requests can be served in parallel with each replica providing data access to a smaller community of users. If multiple users access data over a network from geographically distant locations, data access will be slower than in a small local-area network given that LANs have lower network latencies than WANs. By providing data as close to the user as possible (data locality), the smaller distances over the network can also contribute to higher performance and lower response times.

Moreover, if a single data item is only stored at a single server, this data item cannot be accessed if the server crashes or otherwise does not respond. In contrast, if a copy (replica) of the data item is stored at a different server, this additional server can provide the data item in case of a server or network failure. Thus, the availability of data can be increased even in the event of natural disasters like earthquakes.

To differentiate between an independent copy of a file and a replica of the same, consider the following example. A person creates a file and sends a copy to another person who can then use the private copy. Since the location of the second copy is neither stored in a file catalog nor any guarantee is given that both copies have the same contents after one is changed, these files cannot be regarded as replicas. Unlike "conventional" data items that exist only once in a data store, replicated data items also require an extended naming convention. A set of identical replicas is identified by a *logical name* and each individual replica is identified by a *physical name* indicating a particular replica storage location. A client needs to know the logical name for a file and an additional data structure (i.e. catalog) that maps the logical name to a set of physical names. Thus, a replica is more sophisticated than an independent copy as it requires some data management and special data structures. Each replica is dependent on at least one other data item called, *primary copy*, which is created first. When a replica is created, it is assigned the same logical name as the primary copy and new physical name.

2.3.1 Issues related to replication in data grids

Although the necessity of replication in data grid systems is evident, its implementation entails issues such as selecting suitable replicas, maintaining replica consistency, and so on. The following fundamental issues are identified:

a) Strategic placement of replicas is needed to obtain maximum gains from replication based on the objectives of applications.

b) The degree of replication must be selected to use the minimum possible number of replicas without excessively reducing the performance of applications.

c) Replica selection should identify the replica that best matches the user's quality of service (QoS) requirements and, perhaps, achieves one or more system-wide management objectives.

d) Replica consistency management should ensure that the multiple copies (i.e. replicas) of a given file are kept consistent in the presence of concurrent updates.

e) The impact of replication on the performance of job scheduling must also be considered.

Figure 2.5 presents a visual taxonomy of these issues which will be used in the following subsections.

Replica Placement

Although, data replication is one of the major optimization techniques for promoting high data availability, low bandwidth consumption, increased fault tolerance, and improved scalability, the problem of replica placement has not been well studied for large-scale grid environments. To obtain the maximum possible gains from

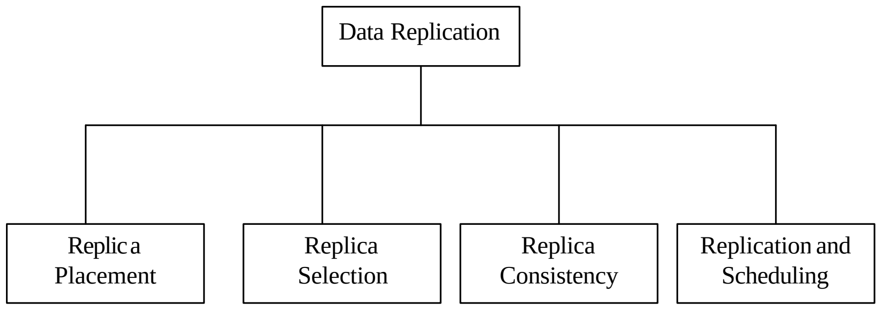


Figure 2.5: Taxonomy of issues in data replication

file replication, strategic placement of the file replicas in the system is critical. The replica placement service is that component of a data grid architecture that decides where in the system a file replica should be placed. In fact, different replication strategies can be defined depending on when, where, and how replicas are created and destroyed. A detailed discussion of existing work in replica placement in data grids will be presented later in this chapter.

Replica Selection

A system that includes replicas also requires a mechanism for selecting and locating them at file access time. Choosing and accessing appropriate replicas are very important to optimize the use of grid resources. A replica selection service discovers the available replicas and selects the "best" replica given the user's location and, possibly, quality of service (QoS) requirements. Typical QoS requirements when doing replica selection might include access time as well as location, security, cost and other constraints. The replica selection problem can be divided into two subproblems [RKA05]: 1) discovering the physical location(s) of a file replica given a logical file name, and 2) selecting the best replica from a set based on some selection criteria.

Network performance can play a major role when selecting a replica. Slow network access limits the efficiency of data transfer regardless of client and server implementation. Correspondingly, one optimization technique to select the best replica from different physical locations is by examining the available (or predicted) bandwidth between the requesting computing element and various storage elements that hold replicas. The best site, in this case, would be the one that has the minimum (predicted)transfer time required to transport the replica to the requested site. Although, network bandwidth plays a major role in selecting the best replica, other factors including additional characteristics of data transfer (most notably, latency), replica host load, and disk I/O performance are important as well.

Replica Consistency

Consistency and synchronization problems associated with replication in data grid systems are not well addressed in the existing research with files often being regarded as being read-only. However, as grid solutions are increasingly used by a range of application types, requirements will arise for mechanisms that maintain the consistency of replicated data that can change over time. The replica consistency problem deals with concurrent updates made to multiple replicas of a file. When one file is updated, all other replicas then have to have the same contents and thus provide a consistent view. Consistency maintenance therefore also requires some form of concurrency control.

Replica consistency is a traditional issue in distributed systems, but it introduces new problems in data grid systems. Traditional consistency maintenance techniques such as invalidation protocols [DPM00], distributed locking mechanisms [WL88], atomic operations [DDW08] and two-phase commit protocols [CP85] are not necessarily suitable for data grid environments because of the long delays introduced by the use of a wide-area network and the high degree of autonomy of data grid resources [DDP$^+$04]. For example, in a data grid, the replicas for a file may be distributed over different countries. So, if one node which holds a replica is not available when the update operation is underway, the whole update process could fail.

The Impact of Data Replication on Job Scheduling

Dealing with the large number of data files that are geographically distributed causes many challenges in a data grid. One that is not commonly considered is the scheduling of jobs to take data location into account when determining job placement. The locations of data required by a job clearly impacts grid scheduling decisions and performance [TLYT06], so, it is important to pick an appropriate job execution site.

Traditional job schedulers for grid systems are responsible for assigning incoming jobs to compute nodes in such a way that some desirable conditions are met, such as the minimisation of the overall execution time of the jobs or the maximisation of throughput or processor utilisation. Such systems generally take into consideration the availability of compute cycles, job queue lengths, and expected job execution times, but they typically do not consider the location of data required by the jobs. Indeed, the impact of data and replication management on job scheduling behaviour has largely remained unstudied.

One must consider not only the abundance of computational resources but also data locations. A site that has enough available processors may not be the optimal choice for computation if it doesn‘t have the required data nearby. (Allocated processors might wait a long time to access the remote data.) Similarly, a site with local copies of required data is not a good place to compute if it doesn’t have adequate computational resources. An effective scheduling mechanism is required that will allow the fastest possible access to the required data, thereby reducing the data access time. Since creating data replicas can significantly reduce the data access cost, a tighter integration of job scheduling and automated data replication could bring substantial improvement in job execution performance.

2.4 Related Data-Intensive Research

This section describes data replication in various distributed data-intensive environments that share similar requirements, functions and characteristics. These have been chosen because of the similar properties and requirements that they share with data grids.

2.4.1 Web Caches

To increase access performance in the web, web documents can be cached on the clients, using either proxies or the servers directly [RSB01]. Client side caching can be accomplished in two ways; either by using a browsers caching facility, or via a web proxy. Upon visiting a website, the client’s browser retains a local copy of the page visited in its cache. The page can be loaded directly from the local browsers cache, if it is requested again. The concept of web proxy [LA94] was first proposed for companies who maintain a firewall for their clients. In this model, the requests from a client are directed to the web server through the proxy server. When a response comes in, the web proxy sends the result back to the requesting client. The proxy also stores a copy of the requested document in its own cache so that it can satisfy requests from other clients in the near future. Caceres et al. and Kroeger et al. [CDF+98; KLM97] show that a web proxy can improve performance significantly by reducing network and server loads and decreasing the latency of web access. However, one of the demerits of a proxy is the extra processing time taken by the proxy when a cache miss occurs. Different co-operative caching approaches [SN02; BO00] have been studied to improve the cache effectiveness (i.e. hit rates and response times) by communicating with other neighboring proxy caches. Moreover, it is possible to maintain a hierarchy of proxies or to distribute the proxies. In hierarchical caching, caches are located at different network levels or tiers. On encountering a cache miss, the proxy makes requests for the missed object to the proxies at the next tier. Such a caching technique has been used by Netscape and MS Explorer and implemented by Squid [squ00]. However, hierarchical caching suffers from a number of disadvantages: each tier introduces additional delay, caches at the higher level becomes overloaded when frequent cache misses occur at lower levels, and high storage overhead due to several copies that exist at different levels. To address these problems, several researchers propose distributed caching where caches cooperate to serve client requests. The Internet Cache Protocol (ICP) [icp97] and the Cache Array Routing Protocol (CARP) [VR07] are distributed techniques that support document discovery and retrieval from neighboring caches. The query based scheme used in ICP can incur substantial bandwidth consumption and increased latency. To overcome this problem, a summary of the contents of other cooperating caches is proposed by Fan et al. [FCAZ00] to avoid the queries or polls needed by ICP. A number of optimistic replication techniques for Web documents have been proposed [PvST02], where weak consistency algorithms are used. However, data accesses in the web and in grids have different patterns. In the web, updates are very frequent and requested data (documents) are small compared to data in data grids. Therefore, these two environments require different validation mechanisms. However, a decentralized optimization model for a general approach to replica placement might be useful in improving the performance of replication techniques for the web.

2.4.2 Content Delivery Networks

A Content Delivery Network (CDN) consists of a collection of servers that attempt to speed up the delivery of web content and reduce the load on origin servers as well as the network generally. That is, within a CDN, client requests are satisfied from other servers distributed around the Internet (also called edge servers) that cache the content stored at the origin server. A client request is redirected from the origin server to an available server close to the client that is likely to host the content required [DMP$^+$02]. If the selected server does not have the requested data then it is retrieved from the origin server or another edge server. The content is replicated either on-demand when users request it, or it can be replicated beforehand, by pushing the content to the content servers [KRR02]. One of the issues in CDNs is to decide where to place site contents in the CDN infrastructure. This problem has been proven to be NP-Complete, and several caching and replica placement algorithms have been proposed and used in the literature based on heuristic and relaxation algorithms [KRR02; KM02]. However, most of the existing algorithms do not scale well for larger systems. CDNs are generally employed by commercial web content providers such as Akamai [aka07] and Digital Island [dil07] who have built dedicated infrastructure to serve web content for multiple clients. However, CDNs have not gained much acceptance for data distribution generally because currently CDN infrastructures are proprietary in nature and owned completely by the providers.

2.4.3 Peer-to-Peer Networks

Another mechanism for content distribution is the use of Peer-to-Peer (P2P) networks that are formed by ad hoc aggregation of resources to form a decentralized system within which each peer is autonomous and depends on other peers for resources, information and forwarding requests [Ora01]. The following key features are found in P2P networks: ensuring scalability and reliability by removing the centralized authority, ensuring reliability via redundancy, sharing resources, and ensuring anonymity. P2P networks have been designed and implemented for many areas such as compute resource sharing (e.g. SETI@Home [ACK+02], Compute Power Market [BV01]), content and file sharing (Napster, Gnutella, Kazaa [CHNB05], etc.) and collaborative applications such as instant messengers (Jabber [jab05]). Milojicic et al. [MKL+02] present a detailed taxonomy and survey of peer-to-peer systems. Their review focuses mostly on content and file-sharing P2P networks as these involve data distribution. Such networks have mainly focused on creating efficient strategies to locate particular files within a group of peers, to provide reliable transfers of such files in the face of high volatility and to manage high load caused due to demand for highly popular files. Currently, major P2P content sharing networks do not provide an integrated computation and data distribution environment.

2.4.4 Distributed Databases

A distributed database (DDB) [OV99] is a logically organized collection of data stored at different sites of a computer network. Each site has a degree of autonomy and thus is capable of executing local applications, and also participates in the execution of a global application (the DDB management system). A distributed database can be formed either by taking an existing single site database and splitting it over different sites (top-down approach) or by federating existing database management systems so that they can be accessed through a uniform interface (bottom-up approach) [SL90]. The latter are also called multidatabase systems. Distributed databases have evolved to serve the needs of large organizations which need to remove the constraints of a centralized computer center, to interconnect existing databases, to replicate databases for increased reliability, and to add new databases as new organizational units are created. Distributed database replication can be divided into two different categories: synchronous/eager and asynchronous/lazy replication. To understand these replication strategies, a brief introduction of transaction management and two phase locking is now given.

A *transaction* allows clients to access and modify data items as a single atomic operation. However, it is possible for two operations from different transactions to access the same data item concurrently. If one or more of those operations is a write, the database may be left in an inconsistent state. To address this problem Two Phase Locking (2PL) [Tho79] has been proposed to guarantee serializability (i.e. equivalence to a serial execution of the transactions). The 2PL algorithm has two phases, one is the growing phase where it acquires locks and accesses data items; and then a shrinking phase where locks are released [EN03]. By following the 2PL rules for lock acquisition and release serializability is guaranteed.

In synchronous replication, an update transaction synchronizes all copies of modified data before it commits whereas in lazy replication [PS00] the update can be done to one copy and the propagation of changes to other copies is deferred until later. A replica copy is characterized as a primary copy if it is updatable [OV99]. In primary copy replication, each data item is assigned a primary or master copy and the replication first takes place in the master copy with updates propagated later to other copies. On the other hand, the update-everywhere approach allows updates to any copy [GHaDO96]. Eager primary copy and eager update-everywhere replication strategies use a 2 Phase Commit (2 PC) [TS02] protocol to ensure that all sites commit the transaction.

The problem of allocating data in a distributed database was investigated by a number of researchers. This problem deviates from the well known file allocation problem. The representative research related to the file allocation problem and the data allocation problem in distributed databases is presented below.

The file allocation problem deals with allocating files to different computer nodes in a network for a given set of queries, updates and executions such that a cost function is minimized. Chu [Chu73] developed a model that allocates files to several computers to minimize the overall storage and transmission cost. They formulate the problem as a linear zero-one programming problem and solved it using standard linear integer programming techniques. Chandy et al. [CH76] proposed a model that allows multiple copies to be stored at different sites. The objective of their model was to minimize the overall cost associated with updating a copy and transmission cost for satisfying a query.

Allocation in distributed file system differs from data allocation in a distributed database. In a distributed database, relations are broken up into logical units called fragments that are distributed to different sites. In distributed query processing, fragments at different sites needs to be accessed to process a query. Therefore, fragments can not be allocated in isolation as in file allocation. There is a relationship between fragments and the placement of one fragment has an impact on the placement of other fragments. Different quantitative data needs to be considered for data allocation [OV99]. For example, the size of fragments, the read and update access frequencies of the fragments, the processing and storage capabilities at each site and the characteristics of the communication network among sites. Distributed queries are decomposed into a set of sub-queries. Each sub-query may need to access fragments stored at different sites and make a join between fragments to answer a query. Further, the cost associated with data integrity, concurrency control, etc., should take into account. Although distributed database systems are very robust and provide distributed transaction processing, distributed query optimization and efficient management of resources, these systems cannot be employed in their current form at the scale of data grids envisioned as they have strong requirements for ACID (Atomicity, Consistency, Isolation and Durability) properties [GR93] to ensure that the state of the database remains consistent and deterministic and these cannot be efficiently provided at grid scale.

2.4.5 Mobile Environments

Mobile computing is gaining popularity with the explosive growth of wireless technology and mobile devices. Wireless technology has several challenges such as narrow bandwidth, frequent disconnections, and low battery power. Therefore, replicating or caching of frequently accessed data items at the client/device side is important because it increases data availability during disconnection as well as saving battery power. Cache invalidation techniques are required to validate the cached data items at the client site. Two approaches are proposed in the literature to manage caches in mobile environments: 1) Stateful [DH95]: where the server tracks the states of mobile clients caches so if the data item is changed the server can broadcast the changed data to the clients that hold that data and 2) Stateless [KL01; Cao00]: where the server has no information about mobile caches. In this case, the server periodically broadcasts invalidation reports to all mobile clients. Sistla et al. [SWH98] propose a sliding window algorithm for replica allocation on the mobile nodes. The algorithm is based on the client access frequency, which determines whether to allocate or de-allocate a data object according to the number of read and update accesses.

Unlike the technique of Sistla et al. [SWH98], which considers minimizing access cost in a network with only a base node and a mobile node, the techniques in [Har01; Har03; HMN04] consider data replica allocation in unstructured mobile ad hoc networks to improve data availability in the presence of network partitioning. Hara [Har01] proposes three methods for allocating replicas in ad hoc networks for data objects shared by multiple mobile hosts. The major goal is to allocate data objects such that they are distributed evenly in the system without having many replicas clustered in a neighborhood. In such a way, the data availability will be improved if the network partitioning probability is random. This work assumes that there is no update request and all hosts have knowledge of the read access probability of each independent data object. In a subsequent effort [Har03], Hara extends the previous work and considers update requests, and replicates the data objects with higher access frequencies and a larger update time period. Hara et al. [HMN04] further improve this work and consider correlation among data objects. They define pairs of data objects that are correlated with each other when the two data objects are accessed by multiple requests in the same session. They make a further assumption that the correlation strength between the two data objects is known in advance and does not change with time and location.

2.4.6 Discussion

I now compare the data-intensive paradigms described in the previous sections with data grids to bring out the uniqueness of the latter by highlighting their respective similarities and differences. Also, each of these areas have their own mature solutions which may be applicable to similar problems in data grids either wholly or with some modification. The following characteristics are important to consider [VBR06]:

a) - Considering the purpose of the network, it is generally seen that P2P content sharing networks are vertically integrated solutions for a single goal (for example, file-sharing). CDNs are dedicated to caching web content so that clients are able to access it faster. DDBs are used for integrating existing diverse databases to provide a uniform, consistent interface for querying and/or replicating existing databases for increasing reliability or throughput. In contrast to these single purpose networks, data grids are primarily created for enabling collaboration (by forming VOs) through sharing of distributed resources including data collections and support various activities including data transfer and computation over the same infrastructure. The overall goal is to bring together existing disparate resources to obtain benefits from aggregation.

b) Access type distinguishes the type of data access operations done in the network. P2P content sharing networks are mostly read-only environments. Write operations occur only when an entity introduces new data into the network or creates copies of existing data. CDNs are almost exclusively read-only environments for end-users and updating of data happens at the origin servers only. In DDBs, data is both read and written frequently. Data grids are similar to P2P networks as they are mostly read-only environments into which either data is introduced or existing data is replicated. However, a key difference is that depending on application requirements, data grids may also support updating of data replicas when the source is modified.

c) A key element of performance in distributed data-intensive networks is the manner in which they reduce the latency of data transfers. Some of the techniques commonly used in this regard are replicating data close to the point of consumption, caching of data, streaming data and pre-staging the data before the application starts executing. Replication is different from caching as the former involves creation and maintenance of copies of data at different places in the network depending on access rates or other criteria while the latter involves creating just a copy of the data close to the point of consumption. Replication is, therefore, done mostly from the source of the data (provider side) while caching is done at the data consumer side. While both replication and caching seek to increase performance by reducing latency, the former also aims to increase reliability by creating multiple backup copies of data.

CDNs employ caching and streaming to enhance performance especially for delivering media content. While several replication strategies have been suggested for a CDN, Karlsson and Mahalingam [KM02] experimentally show that caching provides equivalent or even better performance than replication. In the absence of requirements for consistency or availability guarantees in CDNs, computationally expensive replication strategies do not offer much improvement over simple caching methods. P2P networks also employ replication, caching and streaming of data to various degrees. Replication and caching are used in distributed database systems explicitly for optimizing distributed query processing.

In data grids, all of the techniques mentioned are implemented in one form or another. However, additionally, data grids are differentiated by the requirement for transfer of massive datasets across geographically distributed sites. This is either absent in the other data-intensive networks or is not considered while designing these networks. This motivates use of high-speed data transfer mechanisms that have separation of data communication - that is, sending of control messages happens separately from the actual data transfer. In addition, features such as parallel and striped data transfers among others, are required to further reduce time for data movement. Optimization methods to reduce the amount of data transfered, such as accessing data close to the point of its consumption, are also employed in data grids.

d) Most well-established Data Grid applications are used in scientific and engineering fields and they primarily require large read-only datasets so replica synchronization is not a critical concern in grids [ABB$^+$02]. However, synchronization and consistency are key features for Distributed Database (DDB) replication or web caching. A DDB typically uses short transactions while scientific ones tend to be long, so the issues are very different.

e) Computational requirements in data intensive environments originate from operations such as query processing, applying transformations to data and processing data for analysis. CDNs are exclusively data-oriented environments with a client accessing data from remote nodes and processing it at its own site. While current P2P content sharing networks have no processing of the data, it would be possible to integrate such a capability in the future. Computation within DDBs involves transaction processing which can be conducted in two ways: the requested data is transmitted to the originating site of the transaction and the transaction is processed at that site, or the transaction is distributed among the different nodes which have the data. High volumes of transactions can cause heavy computational load within DDBs and there are a variety of optimization techniques to deal with load balancing in parallel and distributed databases.

Data grids have heavy computational requirements that are caused by workloads often involving extensive analysis of datasets. Many operations in data grids, especially those involving analysis, can take significant time (measured in many hours or even days/weeks). This is in contrast to the situation within DDBs where the turnaround time of requests is short and for applications such as OLTP (On Line Transaction Processing), commonly measured in milliseconds. High performance computing sites, that generally constitute existing data grids, are shared facilities and are oversubscribed most of the time. Therefore, application execution within data grids has to take into account the time to be spent in queues at these sites as well.

f) Autonomy deals with the degree of independence allowed to different nodes in a network. However, there can be different types and different levels of autonomy provided [SL90]. *Access autonomy* allows a site or a node to decide whether to grant access to a user or another node in the network. *Operational autonomy* refers to the ability of a node to conduct its own operations without being overridden by external operations of the network. *Participation autonomy* implies that a node has the ability to decide the proportion of resources it donates to the network and the time it wants to be connected or disconnected from the network. Data grid nodes support all three kinds of autonomy to their fullest extent. While nodes in a P2P network do not have fine-grained access controls against users, they have maximum independence in deciding how much share of resources they will contribute to the network. CDNs are dedicated networks and so, individual nodes have no autonomy at all. Tightly coupled databases retain all control over the individual sites whereas multi-database systems retain control over local operations.

g) Mobile data replication is addressed differently than in a data grid. Due to frequent disconnections and low battery power, the mobile-clients cache frequently accessed data items. The data items in a mobile environment are typically rather small in size whereas the file size in a grid is normally large, on the magnitude of multiple GBs or more. Therefore, caching frequently accessed data items needs a huge amount of space making caching many data items difficult at the client site. In a grid, users submit jobs which in turn request a number of files to run the job. Moreover, the grid environment is usually based on network stability and high transmission bandwidth. These two requirements conflict with the key characteristics found in mobile environments [LS97]; so results from this research are normally not directly applicable to data grids.

2.5 Replica Placement Strategies in Data Grids

Most existing replica placement algorithms for data grids focus on one of two types of objective functions for placing replicas. The first type of replica placement strategy looks towards decreasing the data access latency and the network bandwidth consumption. The other type of replica placement strategy focuses on how to improve system reliability and availability. A combination of strategies is, of course, also possible. Further, other criteria might also be envisioned.

2.5.1 Algorithms Focused on Access Latency and Bandwidth

Ranganathan and Foster [RF01b; RF01a] present and evaluate different replication strategies for a hierarchical data grid architecture. These strategies are defined depending on when, where, and how replicas are created and destroyed in a hierarchically structured grid environment. They assess six different replication strategies: 1) No Replication: only the root node holds files (the base case for comparison); 2) Best Client: a replica is created for the client who accesses the file the most; 3) Cascading: a replica is created somewhere on the path from the root node to the best client; 4) Plain Caching: a local copy is stored at clients upon initial request; 5) Caching plus Cascading: combines plain caching and cascading; 6) Fast Spread: file copies are stored at each node on the path from the root to the best client. They show that the cascading strategy reduces response time by 30% over plain caching when data access patterns contain both temporal and geographical locality. When access patterns contain some locality, Fast Spread saves significant bandwidth over other strategies. These replication algorithms assume that popular files at one site are also popular at others. The client site counts hops for each site that holds replicas, and the model selects the site that is the least number of hops from the requesting client; but it does not consider current network bandwidth and also limits the model to a hierarchical grid. The replication algorithms described can be refined so that time interval and threshold of replication change automatically based on user behaviour such as file access rates from users.

Lamehamedi et al. [LSsD02; LSSD03] study replication strategies where the replica sites can be arranged in different topologies such as a ring, tree or hybrid. Each site or node maintains an index of the replicas it hosts and the other locations that it knows about that host replicas of the same files. Replication decisions are made based on a cost model that evaluates both the data access costs and performance gains of creating each replica. The estimation of costs and gains is based on factors such as run-time accumulated read/write statistics, response time, bandwidth, and replica size. Their replication strategy places a replica at a site that minimises the total access costs including both read and write costs for the datasets. The write cost considers the cost of updating all the replicas after a write at one of the replicas. They show, via simulation, that the best results are achieved when the replication process is carried out closest to the users.

Bell et al. [BCCS$^+$03] present a file replication strategy based on an economic model that optimises the selection of sites for creating replicas. Replication is triggered based on the number of requests received for a dataset. Access mediators receive these requests and start auctions to determine the cheapest replicas. A Storage Broker (SB) participates in these auctions by offering a "price" at which it will "sell" access to a replica if it is available. If the replica is not available at the local storage site, then the broker starts an auction to replicate the requested file onto its storage if it determines that having the dataset is economically feasible. Other SBs then bid with the lowest prices that they can offer for the file. The lowest bidder wins the auction but is paid the amount bid by the second-lowest bidder.

In subsequent research, Bell et al. [BCC$^+$03] describe the design and implementation of a grid simulator, OptorSim which allows the analysis of various replication algorithms. The goal is to evaluate the impact of the choice of an algorithm on the throughput of typical grid jobs. Various algorithms were compared to a novel algorithm [BCCS$^+$03] based on an economic model. The comparison was based on several grid scenarios with various work loads. The results obtained from OptorSim suggest that the economic model performs at least as well as traditional methods. However, the economic model shows marked performance improvements over other algorithms when data access patterns are sequential.

The OptorSim simulator was constructed assuming that the grid consists of several sites, each of which may provide computational and data-storage resources for submitted jobs. Each site consists of zero or more Computing Elements (CEs) and zero or more Storage Elements (SEs). Computing Elements run jobs, which use the data in files stored on Storage Elements. A Resource Broker controls the scheduling of jobs to Computing Elements. Sites without Storage or Computing Elements act as network routing nodes.

Park et al. [PKKY03] propose a dynamic replication strategy, called BHR (Bandwidth Hierarchy based Replication), to reduce data access time by avoiding network congestion in a data grid network. The BHR algorithm exploits 'network-level locality', so that any required file is located at the site that has the broadest bandwidth to the site of the job‘s execution. In data grids, some sites may be located within a region where sites are linked closely. For instance, a country or province/state might constitute a network region. Network bandwidth between sites within a region will be broader than bandwidth between sites across regions. That is, a hierarchy of network bandwidth may appear in the Internet. If the required file is located in the same region, less time will be consumed to fetch the file. Thus, the benefit of network-level locality can be exploited. The BHR strategy reduces data access time by maximizing this network-level locality.

Rahman et al. [RBA05b] present a replica placement algorithm that considers both the current state of the network and available file requests. Their replication strategies are mainly based on “utility” and “risk”. Before placing a replica at a site, expected utility and risk index are calculated for each site by considering current network load and user requests. A replication site is then chosen by optimizing expected utility or risk indexes. Utility is used by their proposed algorithm to select a candidate site to host a replica by assuming that future requests and current load will follow current loads and user requests. The algorithm uses the risk index to expose sites far from all other sites and assumes a worst case scenario whereby future requests will primarily originate from that distant site thereby attempting to provide good access throughout the network. One major drawback of these strategies is that the algorithms select only one site in each simulation interval and place a replica there. Grid environments can be highly dynamic and thus there might be a sudden burst of requests such that a replica needs to be placed at multiple sites simultaneously to quickly satisfy the large spike of requests.

Two dynamic replication mechanisms [TLYT05] are proposed for a multi-tier architecture for data grids: Simple Bottom-Up (SBU) and Aggregate Bottom-Up (ABU). The SBU algorithm replicates any data file that exceeds a pre-defined threshold of access rate as close as possible to the accessing clients. The main shortcoming of SBU is the lack of consideration of the relationship to historical access patterns. To address this problem, ABU was designed which takes into account access histories of files used by sibling nodes and aggregates the access record of similar files so that those frequently accessed files are replicated first. This process is repeated until the root is reached. An example of a data file access history and the network topology of the related nodes is shown in Figure 2.6. The history indicates that node *N1* has accessed file *A* five times, while *N2* and *N3* have accessed *B* four and three times, respectively. Nodes *N1*, *N2* and *N3* are siblings and their parent node is *P1*.

nodeID	*fileID*	*numOfAccesses*
N_1	*A*	5
N_2	*B*	4
N_3	*B*	3

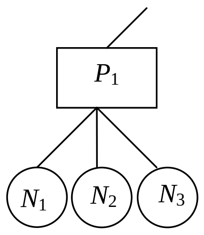


Figure 2.6: An example of the history and the node relations

If we assume that the SBU algorithm is adopted and the given threshold is five, the last two records in the history will be skipped and only the first record will be processed. The result is that file *A* will be replicated in node *P1* if it has enough space, and file *B* will not be replicated. Considering this example it is clear that the decision of SBU is not optimal, because from the perspective of the whole system, file *B*, which is accessed seven times by nodes *N2* and *N3*, is more popular than *A*, which is only accessed five times. Hence, the better solution is to replicate file *B* to *P1* first, then replicate file *A* to *P1* if it still has enough space available. The Aggregate Bottom-Up (ABU) algorithm works in a similar fashion. While access latency can be improved significantly, significant storage space may be needed. Storage space utilization and access latency must be traded off against each other.

Rahman et al. [RBA05a] propose a multi-objective approach to address the replica placement problem in data grid systems. A grid environment is highly dynamic, so predicting user requests and network load, a-priori, can be difficult. Only considering a single objective, variations in user requests and network load will have larger impacts on system performance. Rahman et al. use two models: the p-median and p-center models [Hak64], for selecting the candidate sites at which to host replicas. The p median model places replicas at sites that optimize the request-weighted average response time (which is the time required to transfer a file from the nearest replication site). The response time is zero if a local copy exists. The request-weighted response time is calculated by multiplying the number of requests at a particular site by the response time for that site. The overall average is calculated by averaging the request weighted response times for all sites. The p-center model selects candidate sites to host replicas by minimizing the maximum response time.

2.5.2 Algorithms Focused on Reliability and Availability

Once bandwidth and computing capacity become relatively cheap, data access time can decrease dramatically. How to improve the system reliability and availability then becomes the focal point for replication algorithms. Lei and Vrbsky [LV06] propose a replica placement strategy to improve availability when storage resources are limited without increasing access time.

To better express system data availability, Lei and Vrbsky introduce two new measures: the file missing rate and the bytes missing rate. The File Missing Rate (FMR) represents the number of files potentially unavailable out of all the files requested by all the jobs. The Bytes Missing Rate (BMR) represents the number of bytes potentially unavailable out of the total number of bytes requested by all jobs. Their replication strategy is aimed at minimizing the FMR and BMR. To do this, their proposed strategy makes the replica and placement decisions based on the benefits received from replicating files in the long term. If a requested file is not at a site, it is replicated at the site if there is enough storage space. If there is not enough free space to store the replica, an existing file must be replaced. Their replication algorithm can be enhanced by differentiating between the FMR and BMR in the grid.

Ranganathan et al. [RIF02] present a dynamic replication strategy that creates copies based on trade-offs between the cost and the future benefits of creating a replica. Their strategy is designed for peer-to-peer environments where there is a high-degree of unreliability and hence, considers the minimum number of replicas that might be required given the probability of a node being up. In their approach, peers create replicas automatically in a decentralized fashion, as required to meet availability goals. The aim of the framework is to maintain a threshold level of availability at all times.

Each peer in the system possesses a model of the peer-to-peer storage system that it can use to determine how many replicas of any file are needed to maintain the desired availability. Each peer applies this model to the (necessarily incomplete and/or inaccurate) information it has about the system and replication status of its files to determine if, when, and where it thinks new replicas should be created. The result is a completely decentralized system that can maintain performance guarantees. These advantages come at the price of accuracy since nodes make decisions based on partial information, which sometimes leads to unnecessary replication. Simulation results show that the redundancy associated with distributed authority is more evident when nodes are highly unreliable.

An analytical model for determining the optimal number of replica servers to guarantee a specific overall reliability given unreliable system components is presented by Schintke and Reinefeld [SR03]. Two views are identified: the requester who requires a guaranteed availability of the data (local view), and the administrator who wants to know how many replicas are needed and how much disk space they would occupy in the overall system (global view). Their model captures the characteristics of peer-to-peer-like environments as well as that of grid systems.

Abawajy [Aba04] focuses on the issue of strategic replica placement with the objectives of increased data availability and improved response time while distributing load equally among replica servers. Abawajy proposes a replica placement approach called Proportional Share Replication (PSR). The main idea underlying the PSR policy is that each file replica should serve an approximately equal number of requests in the system. The objective is to place the replicas on a set of sites in such a way that file access parallelism is increased while the access costs are decreased. Abawajy argues that no replication approach balances the load of data requests within the system both at the network and host levels. Simulation results show that his file replication strategy improves the performance of data access but the gains depend on several factors including where the file replicas are located, burstiness of the request arrivals, packet losses and file sizes.

To use distributed replicas efficiently and to improve the reliability of data transfer, Wang et al. [WHCW06] propose an efficient multi-source data transfer algorithm for data replication, whereby a data replica can be assembled in parallel from multiple distributed data sources in a fashion that adapts to various network bandwidths. The goal is to minimize the data transfer time by scheduling sub-transfers among all replica sites. All replica sites must deliver their source data continuously to maximize their aggregated bandwidth, and all sub-transfers of data should, ideally, be fully overlapped throughout the replication. Experimental results show that their algorithm can obtain more aggregated bandwidth, reduce connection overheads, and achieve superior network load balance.

2.5.3 Algorithms Focusing on QoS Requirements

Although a substantial amount of work has been done on data replication in grid systems, most of it has focused on infrastructure for replication and mechanisms for creating and deleting replicas. However, to obtain the maximum benefit from replication, a strategic placement of replicas considering many factors is essential. Notably, different sites may have different service quality requirements. Therefore, quality of service is an important additional factor in overall system performance.

An early effort by Tang and Xu [TX05] describes a QoS-aware replica placement problem to cope with the QoS issues. Every edge uses the distance between the two end points as a cost function for quality assurance. A request must be answered by a server that is within the distance specified by the request. Every request knows the nearest server that has a replica and the request takes the shortest path to reach the server. Their goal has been to find a replica placement that satisfies all requests without violating any range constraint, and which minimizes the update and storage cost at the same time. They show that their QoS-aware replica placement problem is NP-Complete for general graphs, and provide two heuristic algorithms, called l-Greedy-Insert and l-Greedy-Delete, for general graphs, and a dynamic programming solution for tree topologies.

l-Greedy-Insert starts with an empty replication set, R. In the first step, l+1 nodes with maximum Normalized Benefits (NB) are selected and added into R. NB is defined as the increased number of satisfied requests divided by the increased replication cost due to the selection. In each subsequent step, l-Greedy-Insert examines all possibilities of replacing some already assigned replicas with (l+1) replicas. Thus, each step exactly finds one more replica and the process continues until all QoS requirements are fulfilled. l-Greedy-Delete works the opposite way from l-Greedy-Insert. It begins with having a replica in every node, then it deletes replicas whose deletion maximizes the replication cost reduction until there is no replica that can be deleted without violating QoS requirements. There is a trade-off between the time complexity and the quality of solution on l value. Although the time complexity is a polynomial function of the number of nodes, the execution times of these two algorithms are very slow in practice even when $l = 1$.

Jeon et al. [JGN06] propose a simpler formulation of the QoS-aware replica placement problem for arbitrary overlay networks. Their goal is to minimize the number of replicas in the system. They did not consider update cost and assumed each server has identical storage cost. They present simple centralized as well as distributed solutions to the problem and provide a proof of the problem's NP-Completeness. Their centralized algorithm can be described as follows. Let A be a distance matrix, where each entry (i, j) denotes the shortest path distance between node i and j. B is an equal size matrix as A. Every entry in row i in B has identical value that represent the quality requirement of node i. Then each entry of $A - B$ is examined and if it is less than or equal to 0, set the entry to 1, otherwise, set the entry to 0. Let the resulting 0-1 matrix be called C. Column j in C represents which nodes are covered by node j. In each step, the column j with the most rows not covered so far will be selected. The node j will then be added into the replica strategy. This process will continue until all rows in the matrix have been covered.

In Jeon et al.'s first distributed solution, called *Core-selection*, each node continuously monitors whether there is at least one replica either at itself or at one of its neighbor nodes which is reachable within its QoS constraint. If false, this node should either fetch a replica, or it should choose a neighbor node with maximum in-degree for fetching the replica. This approach tries to maximize the number of nodes that can access this new replica. Their other distributed algorithm, called TTL (*Time to Live*)-based method, a node decides when to create a replica at itself when future requests might come to itself from a different node. In this approach, every node containing a replica maintains a *depending-on* list to indicate the nodes relying on it for accessing the required data. Any non-replica node periodically creates a QoS advertising message, called *QoS-advert* that contains its QoS deadline value, δ, and the *depending-on* list. The QoS-advert is then broadcast to each of its immediate neighbors connected by a link with a delay smaller than δ. Each received QoS-advert is re-broadcast by the receiving node, but only if the QoS value (δ) in the QoS-advert message is greater than or equal to the delay on the outgoing link.

When a node has sent out a QoS-advert with a non-empty *depending-on* field, it waits for the node specified therein to send a reply. If no such reply is received (determined by a timeout), another QoS-advert is sent immediately with the *depending-on* field empty. When a node has sent out a QoS-advert with an empty *depending-on* field, it waits to receive replies, makes a decision about whom to depend on, and sends out a QoS-advert with the *depending-on* field set.

Wang et al. [WLW06] propose another solution called, *Greedy-Cover*, with the concept of cover set using the same system model as [TX05]. Again this is a centralized algorithm and the heuristic information is the normalized benefit which has the same meaning as described. The algorithm starts by finding the cover set of every server in the network. The cover set of a node, u is the set of other nodes that are within the QoS requirement from u. Then the algorithm identifies and deletes super cover sets (that contain some other cover set) in the network. In each subsequent step, *Greedy-Cover* chooses the smallest cover set from the sets of cover sets, examines every node in the set, and puts a replica on the node with the highest normalized benefit. If the newly placed replica satisfies other cover sets, these cover sets are also removed. This process continues until all replica sets are removed. Experimental results show that *Greedy-Cover* efficiently finds near-optimal solutions and is more scalable than l-Greedy-Insert and l-Greedy-Delete. However, the authors did not consider any workload capacity constraint on the replica servers or link capacity constraints.

A similar instance of the problem has been studied by Lin et al. [LLW06], who describe a three-fold objective in a hierarchical data grid model. First, the replicas should be placed in server locations so that the workload on each server is balanced. Another important issue is choosing the optimal number of replicas when the maximum workload capacity for each replica server is known. They also consider the issue of service locality. Each user may specify the minimum distance he will accept to the nearest data server. This serves as a locality assurance that users may specify, and the system must make sure that within the specified range there is a server to answer any file request. Lin et al. devise efficient dynamic programming algorithms that find optimal solutions for the placement problems described above. However, they did not consider any replication cost model in their solution. Also, the link capacities are not bounded. In subsequent work [WLL08], Wu et al. evaluate the effectiveness of their algorithms by comparing them with a modified affinity replica placement heuristic algorithm [Aba04] and show that their algorithms consistently outperform the heuristic algorithm both in terms of minimum number of replicas and the actual data transmission time.

In a recent study, Cheng et al. extend the system model in [CWL09] and propose two heuristic algorithms, called, *greedy remove* and *greedy add* to approximate the optimal solution. They study the QoS-aware replica placement problem for general graphs, which in addition to storage and update cost, also takes access cost of replicas into account. They assume that the workload capacity of a replica server is bounded. The goal is to make sure that each request will be serviced by a replica server within its quality requirement and without violating the capacity limits of the replica server.

The algorithm *greedy remove* starts with having a replica on every server. This replication strategy is feasible since every server can serve itself locally, and thus any

QoS constraint is satisfied. Therefore, the service set (i.e. the set of servers receiving services) of each server contains only itself. *Greedy remove* then repeatedly adjusts the service sets of a pair of replica servers and tries to remove replicas to reduce the replication cost. While removing replicas, the algorithm must simultaneously maintain the feasibility of the replication strategy.

The *greedy add* algorithm works the opposite way as *greedy remove* does. The algorithm begins with an empty replica server set R, and adds replicas to R one at a time. The replication process completes in two stages. In the first stage, *greedy add* repeatedly adds the replicas into R until the replication strategy is feasible. In the second stage, replicas are added into the replica server set R until it is impossible to reduce the replication cost.

2.5.4 Summary of Replica Placement Algorithms

In this section, I summarize current and past research on different replica placement techniques for data grid environments. Several important factors such as grid topology, data access patterns, network traffic conditions, and so on are taken into account when choosing a replica placement strategy. In the presence of diverse and varying grid characteristics it is difficult to create a common ground for comparison of different strategies. To gain insight into the effectiveness of different replication strategies, we discuss their usability by considering metrics including access latency, bandwidth consumption, QoS requirements imposed by data requests, and server work load.

Access Latency: This is the time that elapses from when a node sends a request for a file until it receives the complete file. If a local copy of the file exists, the response time is assumed to be zero.

Bandwidth Consumption: This includes the bandwidth consumed for data transfers occurred when a node requests a file and when a server creates a replica at another node.

QoS Requirements: The QoS requirement can be any function that represents, for example, the number of hops between the user and the replica server, the access latency between them, or a combination of things. For example, each user may specify a maximum distance allowable to the nearest replica server. The system must ensure that a replica server exists to answer all user requests.

Server Work Load: This is the amount of work (measured in terms of the number of requests served) done by the servers. Ideally, the replicas should be placed so that the workload on each server is balanced.

We start with the initial work [RF01b] on replication strategies proposed for hierarchical data grids. Among these strategies, Fast Spread shows relatively consistent performance and is best both in terms of access latency and bandwidth consumption given random access patterns. The disadvantage is that it has high storage requirements. The entire storage space at each tier is fully utilized by Fast-Spread. If, however, there is sufficient locality in the access patterns, Cascading would work better than the others in terms of both access latency and bandwidth consumption. The Best Client algorithm is naive and illustrates the worst case performance among those presented in [RF01b].

An improvement to the Cascading technique is the Proportional Share Replica policy [Aba04]. The method is a heuristic one that places replicas at "optimal" locations assuming that the number of sites and the total number of replicas to be distributed are already known. Firstly, an ideal load distribution is calculated and then replicas are placed on candidate sites that can service replica requests slightly greater than or equal to that ideal load. This technique was evaluated based on mean response time (mean access latency). Simulation results show that it performs better than the cascading technique with increased availability of data and considers load sharing among replica servers. Unfortunately, the approach is inflexible once placement decisions have been made and thus is unrealistic for most scenarios.

With the aim of improving the performance of data access given varying workloads, dynamic replication algorithms were presented by Tang et al. [TLYT05]. In their paper, two dynamic replication algorithms, Simple Bottom-Up (SBU) and Aggregate Bottom-Up (ABU), were proposed for a multi-tier data grid. Their simulation results show that both algorithms can reduce the average response time of data access significantly compared to static replication methods. ABU can achieve noteworthy performance improvements for all access patterns even if the available storage capacity of the replication server is relatively small. Comparing the two algorithms to Fast Spread, the dynamic replication strategy ABU proves to be superior. As for SBU, although the average response time of Fast Spread [RF01b] is better in most cases, Fast Spread's replication frequency may be too high to be useful in the real world.

A multi-objective approach to dynamic replica placement exploiting operations research techniques was proposed in [RBA05a]. In this method, replica placement decisions are made considering both the current network status and data request patterns. Dynamic maintainability is achieved by considering replica relocation cost. Decisions to relocate are made when a performance metric degrades significantly in a specific number of recent time periods. Their technique was evaluated in terms of request-weighted average response time, but the performance results were not compared to any of the other existing replication techniques.

The BHR [PKKY03] dynamic replication strategy focuses on 'network-level locality' by trying to place the targeted file at a site that has broad bandwidth to the site of job execution. The BHR strategy was evaluated using the OptorSim [BCC+03] simulator in terms of job execution time (which includes access latency) with varying bandwidths and storage capacities. The performance of BHR was compared with aggressive replication strategies like LRU Delete and Delete Oldest [BCC+03]. In LRU Delete, the least recently accessed file is chosen for deletion whenever replacement takes place. Delete Oldest is another replacement-based approach which deletes the oldest file first when a newly required replica is received and replacement is necessary. The simulation results show that BHR can outperform LRU Delete and Delete Oldest in terms of data access time especially when grid sites have relatively small storage capacity and a clear hierarchy of bandwidths.

Lin et al. [LLW06] is one of the relatively few replication efforts that focuses on overall grid performance. Their proposed placement algorithm, targeted for a tree-based network model, finds optimal locations for replicas so that the workload among the replicas is balanced. They also propose a new algorithm to determine the minimum number of replicas required when the maximum workload capacity of each replica server is known. All their algorithms ensure that QoS requirements from

Authors (Grid Model)	**Replication Technique**	**Replica Placement Method**	**Comments: Pros (+) and Cons (-)**	**Performance Metric**
Average system performance based techniques				
Ranganathan and Foster (2001) (**Tree**)	Best Client	A replica is created at a node that accesses the file the most	- Naïve and shows worst case performance - Not suitable for Grid	Average response time and bandwidth conservation
	Cascading	Replica drips down to lower tiers if number of file requests exceeds threshold	+ Works better for sufficient degree of locality in access patterns - Worse for random access pattern	
	Caching	A replica is stored at the client locally	- Relatively high response time	
	Fast Spread	Replicas are stored at each node on the path to the client	+ Relatively consistent performance + Best for random access pattern - High storage requirement	
Abawajy (2004) (**Tree**)	Proportional Share Replica	Replicas are distributed based on a calculated ideal workload	+ Performs better than the cascading technique + Load sharing among replica servers - The approach is inflexible once placement decisions made	Mean response time
Tang et al. (2005) (**Tree**)	Dynamic Replication Algorithms	Simple Bottom Up (SBU), Aggregate Bottom Up (ABU)	+ ABU proves to be superior to Fast Spread - Fast Spread performs better than SBU	Response time, bandwidth cost, replication frequency
Rahman et al. (2005) (**Tree**)	Multi-objective Approach	Solved as *p*-facility problem to decide on sites for replica placement	+ Considers current network status and data access patterns + Dynamic maintainability through replica relocation - Results are not compared to any existing replication technique	Request-weighted average response time
Park et al. (2003) (**Tree**)	Bandwidth Hierarchy based Replication	Exploits the benefit of 'network level locality' to place replicas	+ Performs better than aggressive replication strategies like LRU Delete, Delete Oldest - Suitable only for grid sites with a clear hierarchy of bandwidth	Job execution time (includes access latency)
Lei and Vrbsky (2006) (**Tree**)	Replication based on data missing rate	Uses two data availability metrics, File Missing Rate (FMR) and Byte Missing Rate (BMR)	+ Aims at improving data availability - Can be enhanced by differentiating between FMR and BMR when file size varies	Replica availability and job execution time
Ranganathan et al. (2002) (**P2P**)	Model-driven P2P Replication	Creates replicas based on trade-offs between cost and benefits, maintain a minimum no. of replicas all the time	+ No single point of failure - Possibility of unnecessary replication due to decisions based on partial information	Replica availability
Lamehamedi et al. (2002) (**Hybrid**)	Hybrid Replica Connection Service	Based on a cost model that evaluates access costs and performance gains of creating and placing replicas	+ Performance gains increase with the size of data - Results based on synthetic workload; real user access patterns could be used	Response time
QoS-aware techniques				
Lin et al. (2006) (**Tree**)	Replica Placement with Locality Assurance	Finds optimal locations for replicas so that workload among replicas are balanced, finds minimum no. of replicas given maximum server workload	+ Ensures QoS requirement from the users + Measures server workload - Doesn't consider any replication cost model or link capacities - Other network topologies should be considered instead of only tree	Number of replicas, QoS from user and system perspectives
Wang et al. (2006) (**General Graph**)	Greedy-Cover	A heuristic algorithm to position replicas based on a calculated normalized benefit	+ Satisfy quality requirements from users + Considers general network graphs - Doesn't consider workload capacity of servers or link capacities	QoS from users
Cheng et al. (2009) (**General Graph**)	Greedy remove and greedy add	Two heuristic algorithms to approximate optimal solution based on incremental addition or deletion of replicas	+ Satisfy quality requirements from users + Considers general network graphs + Consider workload capacity of servers - Does not consider link capacities	QoS from user and system perspectives

Figure 2.7: Summary of replica placement techniques in data grids

the users are satisfied. However, the algorithms did not consider any replication cost model and also the link capacities are not bounded. Wang et al. [WLW06] address the replica placement problem when the underlying network is a general graph, instead of a tree. Their experimental results indicate that their proposed algorithm efficiently finds near-optimal solutions. However, their solution does not take into account workload capacity of replica servers or bandwidth capacity constraints on the communication links.

In a recent work, Cheng et al. [CWL09] study the QoS-aware replica placement problem for general graphs and propose two heuristic algorithms, called *greedy remove* and *greedy add* to approximate the optimal solution. In addition to storage and update cost, the replication model also takes access cost of replicas into account, and assumes that the workload capacity of a replica server is bounded. Simulation results demonstrate that both the algorithms find a near-optimal solution effectively and efficiently and can adapt to various parallel and distributed environments.

Figure 2.7 summarizes the major research work done on replica placement in data grid environments.

Chapter 3

Motivation and Problem Description

The motivation for grids was initially driven by large-scale, resource intensive applications that require more resources than available in a single computing unit, be it a workstation, a supercomputer, or even a cluster within a single administrative domain. The emerging trend in scientific applications in many areas such as high energy physics and large scale simulations suggests that these applications process and produce large amounts of data. The resulting data needs, in turn, to be stored for further analysis and shared with collaborating researchers within scientific communities that are often spread around the world. With increasingly levels of global collaboration between researchers, data storage and collection centers are spreading around the world.

Experiments in high energy particle physics such as those running at the European Center for Nuclear Research (CERN) serve as good example. The Compact Muon

Solenoid (CMS) detector and ATLAS [Hol01; eur01; gri01a] experiment designed to study particle physics, produce and collect massive amounts of data and involve thousands of researchers from around the world. The goal of these experiments is to find rare events that are produced from the decay of massive new particles. Similarly, in Astronomy the Sloan Digital Sky Survey (SDSS) is an ambitious experiment with the goal of producing a detailed image of a quarter of the sky and determining the positions and brightness of more than 100 million celestial objects [sds00]. These experiments and scenarios emphasize the challenges introduced by combining the management of large amounts of data and computing resources in grid environments. Massive amounts of data are currently being produced by the aforementioned research for scientific analysis; some experiments indeed produce over a petabyte of data in one year. To effectively and efficiently address these challenges, a framework that enables transparent access to and sharing of widely distributed large data sets and computing resources is needed.

A general design framework has been proposed for a data grid architecture by Foster et al. in [KF98]. In data grids, data and data management utilities and resources are treated as the most important entities. In addition to accessing large amounts of data, most collaborative applications running across data grid environments require simultaneous and coordinated access to extensive computational power to process and analyze this data. Ensuring efficient and reliable access to such huge and widely distributed data is a major challenge to grid designers. The major barrier to fast data access in a grid is the high latency of communication in the WANs that underlie many grid systems, which impacts scalability and fault tolerance of applications running on the grid. To address these problems and enhance performance, replication has been widely used to place copies of data sets across different domains in the grid.

Clearly a good replication strategy is needed to anticipate and/or analyze the users' requests for data and to place subsets of the data and replicas at strategic locations. Since grids are heterogeneous and dynamic environments consisting of computing, storage and network resources with varying capability and availability, any replication strategy that is based on only static resource information would lead to inefficient placement and degraded application performance. Thus, grid replica placement strategies should incorporate dynamic approaches that adapt to changing resource conditions in grids. Such a replication strategy should also be able to create an appropriate number of replicas in suitable locations. The denser the distribution of replicas is, the shorter the distance between a client site and the closest replica. However, as noted earlier, maintaining multiple copies of data in grid systems is expensive, and therefore, the number of replicas should be bounded. Clearly, minimizing the access time for data requests and reducing the cost of replication are two conflicting goals. Thus, finding a suitable trade-off between them is necessary.

The existing replica placement strategies for grid environments studied in the literature are generally centralized. In grids where operation control is decentralized and resources are under the control of their own local administrative domains, placing the replicas of an object through a centralized algorithm is undesirable. The existing centralized replica placement strategies become unattractive as the number of users and resource providers increase in the system. Further, existing grid systems are expected to provide sufficient support to a large community of users as originally intended by the designers and developers of these systems. This motivates distributed replica placement algorithms for large-scale grid environments. Moreover, when dealing with communication networks distributed algorithms are more desirable.

Although there has been much work done on replica placement, little has focused on quality of service (QoS) issues. Most of the existing work focuses only on the average system performance, for example, minimizing the total access cost, or the total communication cost, etc. Although such factors are important to overall system performance, under some circumstances, they cannot meet individual user or system requirements adequately. Grid computing infrastructure usually consists of various types of resources and the performance characteristics of these resources can be quite diverse. Moreover, different sites may have different service quality requirements according to the system performance of the sites. Therefore, site/user specific quality of service is an important factor in addition to overall system performance.

The average number of requests serviced by a server node directly affects the average response time that the nodes covered by the server will observe due to queuing delays and, possibly, network congestion. Further, most of the earlier research efforts only consider user requests for replica placement and ignore network latencies. While network bandwidth plays a vital role in large file transfers, substantial transfer time can be saved if we place file replicas at neighboring sites even with limited bandwidth in such a way that network congestion can be avoided. Thus, taking into account limited link capacity in replica placement is also an important consideration. All these requirements are examples of different types of QoS issues.

3.1 Problem Description

Data grid infrastructures facilitate the execution of data-intensive computing jobs in a variety of application areas. An end user/client (scientist, researcher, organization, etc.) who wishes to solve a data-intensive problem will typically use a data grid environment by submitting requests for the data that is needed by their application for execution. As discussed earlier, a possible problem that may arise during the course of job execution is the data retrieval time due to high latency of communication in the underlying grid system. One way to deal with this problem is to place replicas close to the client. A challenge in doing this is to ensure that the data read performance from the perspective of the clients is increased while minimizing the overall replica creation cost (including storage and access/bandwidth costs). This is the focus of my thesis research, as applied to data grid environments.

The specific replica placement problem I consider deals with hierarchical data grid structures as is common in current data grid systems [BCCS$^+$03; LCG01; RF01b; RF01a] and assumes a limit on storage size at the replica servers in the grid. Thus, the challenging part of the problem entails identifying appropriate files for replication at appropriate locations (nodes) in the grid aiming to achieve a trade-off between space utilization and access latency of the files.

In the real world, data access patterns may change over time, so any dynamic replication strategy must keep track of the system state, particularly, file access histories to guide the replication by dynamically creating replicas for selective files throughout the grid. A simple "popularity" measure can be used for this purpose. The popularity of a file can be represented by its recent access rate by specific clients. The challenge is to track the exact value of access counts to find out which files are popular. A pre-defined threshold on access counts can be used to determine popularity. If some files have access counts greater than or equal to the threshold, they will be considered to be popular. Identifying potential popular files is a key task of the dynamic replica placement algorithm. Usually, it is assumed that recently popular files will tend to be accessed more frequently than others in the near future.

Once the popular files have been identified they need to be replicated as close as possible to those clients that frequently request the corresponding files. To make replication decisions, the replica placement algorithm needs to be invoked at regular intervals. The interval should be based on the arrival rate of data access requests, for example, a shorter interval will be chosen for higher arrival rates because files become popular in a short period of time due to frequent accesses. Implementing this in practice, however, requires a careful selection of this threshold and other parameters. Depending on this selection, a file may not be replicated or may be replicated on all nodes. This issue is critical to the usefulness of a threshold-based approach. This suggests an adaptive algorithm that uses data access arrival rate from the client, available storage capacities of the replica servers, and other factors to dynamically adjust this threshold and the interval length used for sampling accesses. The replica servers will, of course, become filled with replicas over time so an efficient replacement strategy is also needed so that popular files are retained and not displaced when new files arrive.

As mentioned, the primary goal of the replica placement algorithm is to increase the data read performance from the client's perspective by dynamically creating replicas for “popular” files. At the same time, from the perspective of the whole system, efficient use of bandwidth and storage resources must be considered to ensure that the dynamic replication algorithm does not cause heavy load on the system. We assume that all data are initially located at the root, and a data replica can be placed in any node other than the root. All the leaves of the grid hierarchy are local sites where users can issue requests to access their required data. The data “access cost” is calculated based on accumulated read statistics, network latency (bandwidth), and replica size. It is expected that data updates will be infrequent in the applications being considered. However, it is necessary to guarantee that the updates will be eventually propagated to and that the users will have access to consistent copies of the data. Thus, to maintain data consistency, the root node issues updates to every replica server. The communication involved in this update process is captured as an “update cost”. The number of replicas to be placed will be a trade-off between the cost of data access by users and the cost of data updates from the root. The problem that needs to be solved is therefore the construction of replica server sets among the grid sites that minimizes the total sum of the data access and update costs under a given traffic pattern (i.e. a recurring pattern of access frequencies from clients for different files).

The replica placement problem so described can be modelled as a dynamic programming problem and its solution can be obtained for large-scale hierarchical data grids in a distributed fashion. In dynamic programming when designing an algorithm for a hierarchical structure (i.e. a tree), results for some function computed for children nodes can be combined to produce the result for the parent node. Hence, a distributed replica placement algorithm can be designed for large-scale hierarchical data grids considering the overhead/cost incurred in placing and maintaining the replicas.

The issue of quality of service (QoS) also needs to be factored into the replica placement problem to determine locations of replicas that improve system performance and at the same time satisfy the quality requirements both from the user and system perspectives. Each user in the lowest tier of the data grid hierarchy may have some QoS requirement on retrieving the requested data. The QoS requirement of each user can be specified by an upperbound on retrieval cost. The requirement mandates that all requests generated by the user will be serviced by a server within that bound. The QoS requirement can be any function that represents, for example, the number of hops between the user and the replica server, the access latency between, and so on. Currently, we consider the replica distance as the QoS requirement where each user may specify a maximum distance allowable to the nearest replica server. The system must ensure that a replica server exists to answer all user requests. Since the QoS requirement can be specified by the distance (e.g. number of hops) between the client and the replica server, this distance constraint can easily be incorporated when the replication costs for clients are calculated considering various distance possibilities for the replica servers.

From the system perspective, link and workload capacity constraints should be added to the replica placement problem while satisfying the quality requirements specified by the user. Each link in the data grid hierarchy has some capacity constraint on transferring data down the hierarchy. The bandwidth constraint on each link is specified by an upper bound on link capacity. The constraint mandates that the amount of data that can pass through the link over a period of time will be limited within that bound. The replication strategy has to ensure that the user requests are satisfied while limiting the bandwidth use of each link to its capacity. The bandwidth constraint associated with different links can be different. If the total amount of data passing through a link is greater than its capacity constraint, then the link is congested. Furthermore, it is assumed that the workload capacity of a replica server is bounded which means a node equipped with a replica can process up to a certain number of requests per unit of time from the clients of its subtree. The replication strategy has to ensure that the user requests are satisfied while limiting the workload of each replica server to its capacity. The workload capacity constraint associated with different servers can be different. If the total workload that a server services is greater than its capacity constraint, then the server is overloaded. The goal is to find a replication strategy with the minimal replication cost that limits the workload of each server to its capacity and where none of the communication links is congested.

Finally, to properly evaluate the performance of the above mentioned placement algorithms an extensive assessment designed over a range of parameters is required. The assessment results will be used to demonstrate how the effectiveness of replica placement is affected by numerous factors such as grid network characteristics (i.e. topology, number of nodes, node and link capacities, traffic patterns, etc.), QoS parameters, and so on.

3.2 Positioning the Thesis

From the literature, it's clear that most of the early work done on data replication in grid environments has focused on infrastructures for replication and mechanisms for creating/deleting replicas [RIF02; LSsD02; CSK$^+$05; BCC$^+$02; BCCS$^+$03; DAS04; SSA$^+$02]. However, to obtain maximum benefits from replication, a strategic placement of replicas is essential. A large part of the work that deals with replica placement concerns optimizing the average system performance [KDW01; KRW01; TLM$^+$05; UC04; WM91], for example, to minimize the total access cost, or to minimize the total communication cost, etc. In addition, to address the resource heterogeneity and varied user requests, quality of service issue are also considered by a number of researchers [TX05; JGN06; WLW06; LLW06; CWL09].

The replica placement problem studied in this thesis differs from the previous work in several ways. First, since a grid environment is highly dynamic, resource availability, network latency, and users requests may change frequently. My replica placement algorithms provide adaptivity to these changing characteristics by dynamically adapting the degree of replication based on data access arrival rate and available storage capacities. Further, unlike earlier work, my algorithms support constraints on the storage size of the replica servers. Second, a data grid system usually consists of multiple data servers connected by routers that enable concurrent data transfer between independent pairs of users and servers. Therefore, in a data grid environment, the overall performance of the system is affected by the workload capacity of replica servers and by link capacities. Although it is believed that workload capacity constraints on replica servers and link capacity constraints are the key optimization criteria for data grid systems, it is also important to consider these two factors and user quality of service simultaneously. To date, this aspect has not been adequately addressed in the literature. My solution to the dynamic programming problem supports multiple QoS parameters both from the user and system perspectives. Each user/client can specify its own acceptable QoS, in terms of the number of hops towards the root of the hierarchy or access time deadlines to ensure timely retrieval of the requested data. In addition, from the system perspective, link and workload capacity constraints can also be added to the problem. Finally, due to decentralized operation control and resources being under the control of their own local administrative domains, placing the replicas of an object through a centralized algorithm is unattractive. This motivates my distributed replica placement algorithm for large-scale grid environments to ensure improved scalability of the system. The related work mentioned in the literature (except the model in [JGN06]) only provides a centralized solution. My proposed distributed model emphasizes minimizing the replication cost (update and data access cost) for optimal replica placement while the model in [JGN06] emphasizes minimizing storage use for overlay networks. Apart from the differences mentioned above, among the studied QoS-aware replica placement scenarios in the literature, the replication cost model in [CWL09] seems closest to my replication model. However, their heuristic algorithms are targeted for general network graphs and link capacities are not bounded. Also, the replication cost includes storage cost in addition to update and data access cost.

My proposed model focuses on tree topologies in which the requests can only go up towards the root. In real-world grid systems, like LCG [LCG01], the requests go from tier-2 to tier-1 sites, then to tier-0 sites if necessary, searching for data. The grid hierarchy usually reflects the structure of administrative organizations, or the geographic locality, so the assumption that the requests go up towards the root is reasonable.

The next chapters describe in detail my contributions towards the research problem in this thesis by defining a family of related algorithms, starting with a simple (baseline) popularity based replica placement algorithm which is centralized in nature. This will evolve into a number of distributed replica placement algorithms that will also be described with added QoS constraints both from the user and system perspectives.

Chapter 4

Centralized Replica Placement

Recall that the over-arching goal of this thesis is to address the problem of replica placement in large-scale data grids to improve the performance of data access while ensuring efficient use of both computational and storage resources. In this chapter, a centralized "popularity-driven" dynamic replica placement algorithm is proposed for use in the hierarchically structured data grids. I also propose an adaptive version of the basic replica placement algorithm which considers both data access arrival rates from the clients and the storage capacities of the replica servers to select the best candidate sites at which to place replicas. The algorithms presented in this chapter are centralized– the replication decisions are made by a single entity in the data grid system which invokes the algorithms at regular intervals. The performance of the proposed placement algorithms is evaluated with a set of carefully designed simulation experiments over a range of data access patterns and replica server capacities and is compared to a number of other existing replica placement algorithms. The results constitute a preliminary investigation into the placement problem and has provided helpful formation into placement strategies for real data grids.

4.1 Assumed Data Grid Structure

Data on the grid needs to be easily accessible to users regardless of the location of data. Data access models are formed by community organizations and are affected by the direction of data flow and users' access patterns. The location and number of data sources as well as data size play important roles in shaping the map of the community sharing access to that data. The scope of this thesis covers scientific data grids, where a number of researchers from different institutes share their resources to collaborate on solving scientific problems. The most prevalent data model used in these settings is the hierarchical model. This model is used in an environment where there is a single source of data, and that data has then to be distributed to multiple locations to be shared by a large community of collaborators. This model is shown in Figure 2.3(b) as described in Section 2.2.2. It shows the data distribution model for the CERN (LHC) experiments where data is first generated and stored at CERN, and later copied to different distribution and regional centers. From these centers the data is then distributed to different labs worldwide to give access to scientists from around the world. Such hierarchical grid management is frequently found in other current data grid systems [BCCS+03; LCG01; RF01b; RF01a].

The hierarchical data grid has many advantages. First, it allows hundreds or even thousands of scientists everywhere to access the resources in a common and efficient way. Second, the datasets can be distributed to appropriate resources and accessed by multiple sites. The network bandwidth for access will be used efficiently because

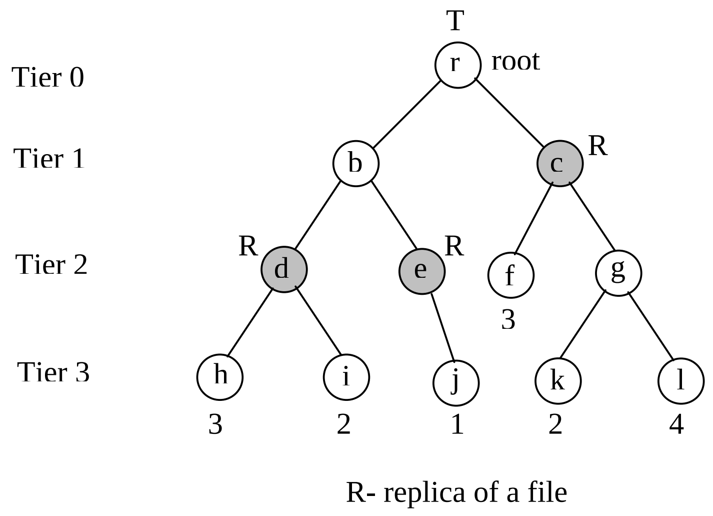


Figure 4.1: An example hierarchical data grid

most of the data transfers will only use local or national network resources, hence alleviating the workload of international network links. Third, with the support of grid middleware, the resources located in different centers and even end-users machine can be utilized to support data-intensive computing. Furthermore, the hierarchical structure enables flexible and scalable management for datasets and users.

I begin with a detailed discussion of my data grid model before addressing the placement problem itself. I use a hierarchical structure (i.e. a tree) T as shown in Figure 4.1 to represent a data grid system. The root of the tree is denoted by r. I assume that all data are initially located at the root. A data replica can be placed on any node except the root. To minimize data access time and network load, replicas must be spread from the root to regional, national, and institutional centers. All the leaves of the tree are local sites where users can issue requests to access the required data stored in the data grid system.

In such a system, a user at a local site will try to locate a replica locally. If a replica is not present, the request will go to the parent node to find a replica there. Generally, a user request goes up the hierarchy and uses the first replica encountered along the path towards the root. If, after traveling up the path, a replica is not found, the root will service the request. For example, in the four-tiered hierarchical data grid in Figure 4.1, the user at node k tries to access a data file required by the job running on it. The user cannot find the data locally, so the request goes to the parent node g, where the data is not available either. Finally, the request reaches node c, and is served there. Each leaf node is associated with a non-negative number which is the access frequency of a data file of interest by that node during a certain period of time.

The goal of my dynamic replica placement strategy is to place the replicas such that various objectives (e.g. fast access) can be satisfied. This raises a number of issues. For example, if we can accurately estimate how frequently a client (leaf node) accesses a specific data file, where do we place the replicas to improve data read performance from the client's perspective. At the same time, from the perspective of the whole system, efficient use of bandwidth and storage resources must be considered to ensure that the dynamic replica placement does not cause heavy load on the system. For example, every file could be replicated at every site if the replica servers have sufficient storage. However, in reality, this is not possible due to limited storage capacity of the replica servers. My dynamic replica placement algorithms will aim to balance the space utilization and access latency trade-off by selectively replicating files among grid sites.

4.2 Basic Popularity Based Replica Placement Algorithm

In this section, I describe my basic popularity-driven dynamic replica placement strategy, called Popularity Based Replica Placement (PBRP), for hierarchical data grids. The primary goal of the algorithm is to increase data access performance from the perspective of the clients by dynamically creating replicas for "popular" files. As mentioned, in the real world, some files will be more popular than others and data access patterns may change over time, so any dynamic replication strategy must keep track of file access histories to decide on when, what and where to replicate. The "popularity" of a file is determined by its recent access rate by the clients. Identifying popular files is thus a key task of PBRP. We assume that recently popular files will tend to be accessed more frequently than others in the near future (i.e. will continue to be popular). In PBRP, popular data files are identified by analyzing file access histories. The replica placement algorithm is invoked at regular intervals and it processes the access histories to determine new replica locations based on file popularity. Old replicas are retained at those replica locations that are common between the new and old replica sets. The rest of the replicas from the old set are deleted and new replicas are created for the remaining new locations. New replica locations are determined after each interval since file popularity varies with time. The access history logs are cleared at the beginning of each replication interval to capture access pattern dynamics. The interval chosen is determined by the arrival rate of data accesses so a shorter interval will be chosen for higher arrival rates. This incurs greater overhead but adapts more rapidly to changing access patterns.

Unlike most of the existing data replication techniques, the proposed replica placement strategy assumes limited storage capacity on each replica server. Over time, the replica servers will become filled so a replacement strategy is needed. Such a scheme must ensure that popular files are retained and not displaced when new files arrive. I will use a popularity-based form of the Least Recently Used (LRU) replacement policy with a constraint added to ensure that replicas created in the current replication interval will not be replaced. This additional constraint is needed to avoid the deletion of newly created replicas by the dynamic replication algorithms. If the available space on a given replica server is less than the size of the new replica, some removable replicas need to be deleted to make room for the new replica. Removable replicas are defined as those replicas that were created before the current replication interval and which were not recently used by any client. Let R_s be the set of all replicas in server s, then the set of removable replicas R'_s is:

$R'_s = \{r | r \in R_s$, r is created before the current replication interval, and r is not recently used$\}$.

The first condition prevents the deletion of any newly created replicas. The second condition chooses (hopefully) unneeded replicas for replacement and also avoids the interruption of any ongoing data accesses.

The idea behind PBRP is to create replicas as close as possible to those clients that frequently request the corresponding files or more specifically to the clients that request the files with high rates exceeding a pre-determined replication threshold. The file access count as a measure of popularity is, in itself, not new. The novelty

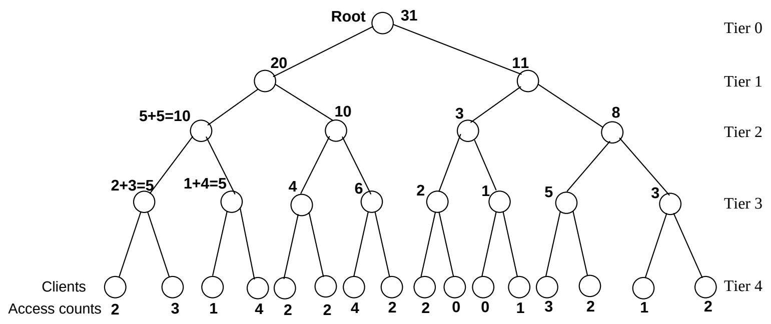


Figure 4.2: Bottom-up aggregation of access counts

with the (PBRP) algorithm is its aim to balance the space utilization and access latency trade-off by selectively replicating files. All the replication algorithms from the literature except ABU process the records in the access history individually for a client and do not study the relations among these records. In hierarchical data grids, every node accesses replicas only from its ancestor nodes. Thus, the relationship among the access records of the clients that are siblings can be used to determine the effective utilization of various replicas. For example, in Figure 4.2, if the replication threshold value is set to five and no aggregation of access counts is considered, no replica will be created in tier-3. However, with the aggregation of access counts, four replicas are created in different nodes in tier-3 because their new access counts exceed or become equal to the threshold value. The whole replication process is done in two phases as described below.

4.2.1 Bottom-Up Access Aggregation

The bottom-up aggregation phase aggregates access history records for each file to upper tiers, step by step, until the root is reached. The computation simply adds

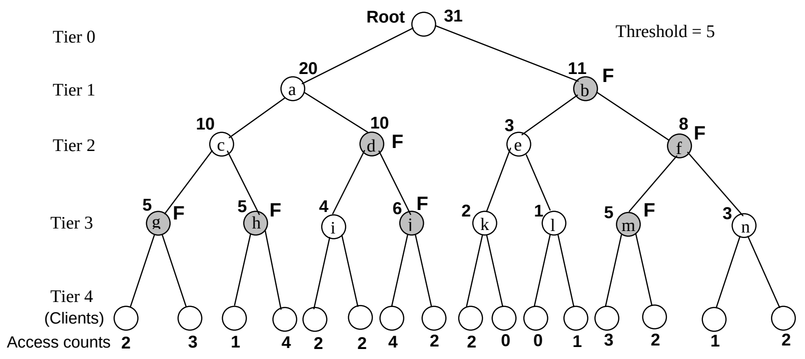


Figure 4.3: Top-down placement of replica

up the access counts for records whose nodes are siblings and which refer to the same files. The result record after aggregation is stored in the parent node. An example of the computation of access counts of a file (F) by different clients is shown in Figure 4.2 (client counts shown at leaves).

4.2.2 Top Down Replica Placement

Using the aggregated access information, replicas are placed from the top to the bottom of the tree. The idea is to traverse down the hierarchy as long as the aggregated access count is greater than or equal to the pre-defined threshold that is used to determine sufficiently popular files. A replica is placed at a node if the threshold prevents further traversal through *one or more of its children*. An example of replica placement is shown in Figure 4.3 where, for example, we traverse down the tree from the root to node *d* through node *a* since both nodes have access counts greater than the threshold value of five. From node *d* further traversal through node *i* does not occur since its access count is less than the threshold. Hence, a replica is placed on node *d*. Node *j* is also traversed through since its access count is six. A replica is also placed on this node since none of its descendants' counts exceed the threshold. The rationale for this approach is to ensure that the replicas are created close to the clients accessing popular files and at the same time to facilitate placement of replicas in locations relatively closer than the root for clients accessing unpopular files.

Algorithm 4.1 Top-Down-Replica-Placement(*threshold*)

```
1: $recentHistory \leftarrow$ getAccessHistory()
2: $recentHistory \leftarrow$ Bottom-up-compute($recentHistory$)
3: for $tier \leftarrow RootTier + 1$ to $ClientTier - 1$ do
4:    $records \leftarrow$ getRecords($recentHistory$, $tier$)
5:    $records \leftarrow$ sortRecords($records$)
6:    for all record $r$ of $records$ do
7:       if $r.accessCount \geq threshold$ then
8:          if checkChild($r.nodeID, r.fileID$) then
9:             if $r.nodeID$ does not contain a replica of $r.fileID$ then
10:               requestForReplication($r.nodeID$, $r.fileID$)
11:            end if
12:         end if
13:      end if
14:   end for
15: end for
```

The replica placement procedure is shown in Algorithm 4.1. The input to the algorithm is a threshold value on access counts of files. A single threshold is used for all files that are accessed by the clients. Function *getAccessHistory* is called in line 1 to scan the most recent access history covering the last pre-defined interval and the results are stored in *recentHistory*. Each record in the history is a tuple of <*nodeId, fileID, accessCount*>, which means that the client of *nodeID* has accessed the data in file *fileID* with a frequency of *accessCount*. In line 2 *Bottom-up-compute* aggregates the access records in *recentHistory* for all tiers. The details of this processing will be discussed later in the section. Once the aggregation of access counts is done, the replica placement algorithm processes the new aggregated history in the **for** loop (lines

3-12) for each middle-tier from the tier below the root to the tier above the clients (which execute at the leaves). Function *getRecords* (line 4) retrieves the aggregated records of the current tier. In line 5, the records are then sorted in descending order by *accessCount* for different files so that the replicas are created for files with higher access counts first. For each record *r* in *records* if the *accessCount* is greater than or equal to the threshold it will be processed in lines 7-9. Function *checkChild* checks if any of the children of *r.nodeID* have *accessCount* less than the threshold or if the children are client nodes. If the result is true, a request for replication is submitted to the replica handler to carry out replication of the data file at the current node.

Before creating new replicas on the determined node (replica server), its available storage capacity needs to be checked. At a given time, the available storage of a replica server is the maximum space that the server can provide for the new replicas. It includes the unused space and the space used by potentially removable replicas. Thus, the available storage capacity of a replica server can be obtained by determining the removable replicas as described earlier. If the available storage capacity of the replica server is greater than or equal to the size of the file *r.fileID*, the replica handler will actually replicate the file from the root or the grid site which is the closest ancestor node that contains the replica.

Bottom-up-compute is shown in Algorithm 4.2. The input parameter *recentHistory* contains the file access records whose *nodeIDs* are in the same tier. The variables *history* and *temp* are buffers with the same structure as *recentHistory*. The outer **for** loop (lines 3-13) processes the access records from the tier above the clients to the root. The inner **for** loop (lines 4-10) aggregates all records in *recentHistory*. Lines

Algorithm 4.2 Bottom-Up-Compute(*recentHistory*)

```
1: $history \leftarrow$ NULL
2: $temp \leftarrow$ NULL
3: for $tier \leftarrow ClientTier - 1$ to $RootTier + 1$ do
4:     for all record $r$ of $recentHistory$ do
5:         $parent \leftarrow$ findParent($r.nodeID$)
6:         if $parent$'s record has the same file as in $temp$ then
7:             Increase the $accessCount$ of $parent$ record by $r.accessCount$
8:         else
9:             Put the $parent$ record in $temp$
10:        end if
11:    end for
12:    Append($history$, $temp$)
13:    $recentHistory \leftarrow temp$
14: end for
15: return $history$
```

5-7 try to merge each record r with a record in *temp*. If there is a record in *temp*, whose *nodeID* is the parent of *r.nodeID* and whose *fileID* is the same as r's, then record r is merged with the corresponding record in *temp* by adding corresponding *accessCounts*. If merging is not possible, a new record is added to *temp* where the new record's *nodeID* is the parent of *r.nodeID* and its *fileID* and *accessCount* is the same as r's. After all records in *recentHistory* are processed, the aggregated history *temp* becomes the new *recentHistory* for the next tier up in the hierarchy and will processed as stated above. Finally, *Bottom-up-compute* returns *history* as the overall aggregated access records for the considered sampling interval.

PBRP improves on ABU by making replicas accessible nearer to clients with lower access counts (which don't exceed the threshold value). This, in turn, reduces overall data access latency and bandwidth consumption at the cost of greater storage use. A bottom-up replica placement is also done in ABU but if the access count of a file at a replica server becomes greater than or equal to the threshold value, a replica is created there and the corresponding access record for that file is deleted. Consider Figure 4.3. A replica would be created in node j (with access count 6) using ABU but due to the deletion of this access count at node j, the aggregated count in d would become 4 which is less than the threshold value and hence no other replica would be created. However, PBRP retains all the aggregated access records and place replicas in a top-down fashion. A replica is placed at a node if the threshold prevents further traversal through one or more of its children. In the example, this creates a replica at node d.

4.3 Adaptive Replica Placement Algorithm

In this section, I describe my new adaptive dynamic replica placement strategy, called Adaptive Popularity Based Replica Placement (APBRP) for hierarchical Data grids. Creation of the replicas is guided by dynamic adjustment of the threshold value in PBRP. The effectiveness of PBRP depends on careful selection of this threshold value that determines the popularity of a data file. Hence, APBRP aims at determining the threshold value dynamically by using such factors as data access arrival rate from the client and available storage capacities of the replica servers. APBRP tries to balance the storage utilization and access latency trade-off by selectively replicating data files through this dynamic adjustment method.

The replication process in APBRP is similar to its non-adaptive counterpart (PBRP) and is, again, done in two phases. The bottom-up aggregation phase aggregates access history records for each file to upper tiers, step by step, till the root is reached. In the second phase, replicas are placed from the top to the bottom of the tree using this aggregated information. Determining the threshold value dynamically is the key new task of APBRP. Like PBRP, the APBRP algorithm is invoked at regular intervals and replication decisions are made based on file popularity.

I consider data access arrival rate and the available storage capacities of replica servers as two logical factors that should affect the value of the threshold. A high access arrival rate indicates frequent access requests from the clients which results in an increased number of popular data files (i.e. their access counts become greater than or equal to the threshold). This will in turn require creating a large number of replicas which will incur substantial replication overhead (in terms of both bandwidth and storage usage). In such a scenario, increasing the threshold will lessen the number of replicas that will be created. On the other hand, if the access arrival rate becomes low, the number of popular data files will be less which in turn will result in the creation of a reduced number of replicas even though the system might be capable (in terms of available bandwidth and storage) of creating more replicas to improve access latency. In this case, decreasing the threshold will increase the number of replicas that will be created.

Likewise, the available storage capacities of replica servers should play an important role in deciding to change the threshold value. The required threshold value can be decreased to create more replicas if replica servers have sufficient available storage space. Also, the threshold value can be increased to lessen the number of replicas if the replica servers become heavily filled with existing replicas. Figure 4.4 shows the flow of replica creation with a threshold controller. First, the variation in data access rate from users is calculated to determine the "initial" change in threshold value (which will be discussed later). Then, the replica placement algorithm uses the new threshold value to determine the replica locations for creating new replicas. A storage checker then verifies the percentage of storage usage for all replica servers to decide whether any further adjustment in threshold value is necessary or not. Feedback information is sent to the threshold controller in case of a heavily loaded or underutilized (in terms of storage capacities of replica servers) system. Otherwise, the replica servers will retain the newly created replicas to serve data requests from the users.

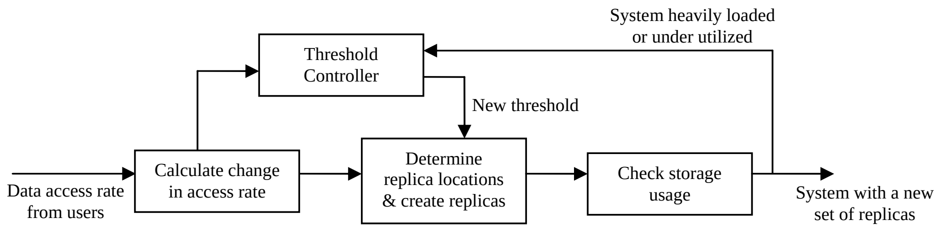

System heavily loaded or under utilized
Threshold Controller
New threshold
Data access rate from users
Calculate change in access rate
Determine replica locations & create replicas
Check storage usage
System with a new set of replicas

Figure 4.4: Replication process through dynamic adjustment of the threshold value

Now, the challenge is to determine how much change (increase or decrease) in the threshold value the system can tolerate based on the change in data access arrival rate or available storage capacities of replica servers. If we consistently increase the threshold value, at some point the number of replicas will drop drastically which will decrease the data access performance and storage utilization of the replica servers. On the contrary, if the threshold value is decreased consistently, the increased number of replicas will eventually make the system non-viable to manage. Hence, the threshold value has to be adjusted in such a way that the space utilization and access latency trade-off is effectively balanced using access arrival rate and available storage capacities of replica servers. The details of how the determination of an initial value of threshold and the adjustment in the threshold value afterwards are done in an adaptive manner are now described.

4.3.1 Determining the Initial Threshold Value

The initial threshold value is set based on the average aggregated access counts at the replica servers in the lower tier of the data grid. The value of the average aggregated access count is calculated by dividing the total number of aggregated access counts for a file at the replica servers in the second to the lowest tier of the hierarchy by the number of replica servers at that tier. This is done during the bottom-up aggregation phase of APBRP. The initial value is then adjusted dynamically (described in the next subsection) based on available storage at the replica servers and the request arrival rates.

4.3.2 Dynamic Adjustment of the Threshold Value

The threshold value is adapted dynamically to ensure efficient utilization of storage at replica servers while satisfying both bandwidth and access latency constraints. As mentioned, the increase or decrease in the threshold value is done based on an increase or decrease in the data access arrival rate, respectively. However, this change in the threshold is not applied immediately to avoid over reacting to transient changes in the access arrival rate. Rather, if the increasing or decreasing trend of access arrival rate is steady for a sufficient period of time, the effect of the new threshold takes place. The challenge is to determine this time interval and the relationship between the change in the threshold value and the access arrival rate. The time interval is calculated as a fraction of the interval length for sampling access histories of clients. The threshold value is increased or decreased by the difference between the current and previous average aggregated access counts of replica servers in the lower tier of the data grid. For example, the threshold value for the ith sampling interval, $threshold_i$, is calculated as follows:

Assume, R_i = New average aggregated access rate in ith interval and R_{i-1} = Average aggregated access rate in (i-1)th interval

$$threshold_i = threshold_{i-1} + \triangle \tag{4.1}$$

where $\triangle$ is defined as:

$$\triangle = \begin{cases} 0, & \text{for the first interval of sampling access histories of clients} \\ R_i - R_{i-1}, & \text{otherwise} \end{cases} \tag{4.2}$$

The relationship in equation 4.1 can easily be justified since the initial threshold value is determined by the average aggregated access rate for the first sampling period. For any subsequent sampling interval, the threshold value will either be increased or decreased based on the difference in aggregated access rates for the current and immediately preceding intervals. Once the threshold value is changed, the available storage capacities of the replica servers are checked. Further adjustment of the threshold value is done based on the available storage at the replica servers to ensure efficient space utilization. More specifically, the threshold controller (as shown in Figure 4.4) updates the threshold value on the basis of the ratio between the desired percentage of storage usage and the actual percentage of storage usage by replica servers. The new threshold value for the ith interval is thus calculated as follows:

$$threshold_i(updated) = threshold_i \times offset \tag{4.3}$$

where *offset* is the described ratio. To facilitate this, I have defined three states of the system in terms of storage utilization by the replica servers. The system is lightly, moderately, and highly loaded if the percentage of storage size used by replica servers is less than 50%, 50% to 70%, and greater than 70%, respectively. If the system is heavily loaded or under utlilized then the calculated ratio contributes in updating the threshold value. Like PBRP, APBRP also uses a modified form of the Least Recently Used (LRU) replacement policy based on popularity for replica replacement.

4.4 Simulation Setup

Often, the evaluation of complex scenarios can not feasibly be carried out using a real grid environment due to issues of scale, cost, and availability. It is also difficult to do performance evaluation in a repeatable and controlled manner due to the dynamic nature of grids. Grid testbeds are thus limited, and further creating an adequately-sized testbed is not only expensive but also time consuming. Moreover, the testbed requires the handling of different administration policies at each resource. Therefore, I chose to assess the effectiveness of my replica placement algorithms through simulation.

Simulation has been used extensively for modeling and evaluation of real world systems, from business process and factory assembly lines to computer systems design. Consequently, modeling and simulation has emerged as an important discipline and many standard and application-specific tools and technologies have been built. These include simulation languages (e.g. Simscript [Sim08]), simulation environments (e.g. Parsec [BMT+98]), simulation libraries (e.g. SimJava2 [Sim02]), and application specific simulators (e.g. the NS-2 network simulator [ns208]). While there exists a large body of knowledge and tools, there are only a few well-maintained tools available for simulation in grid computing environments. The SimGrid toolkit [CLQ08], developed at the University of California at San Diego (UCSD), is a C language based toolkit for the simulation of application scheduling. It supports modeling of resources that are time-shared and the load can be injected as constants or from real traces. SimGrid is a powerful system that allows creation of tasks in terms of their execution time and resources, with respect to a standard machine capability. GangSim [DF05], developed at the University of Chicago, is targeted towards the study of usage and scheduling policies in a multi-site and multi-VO (Virtual Organization) environment. It is able to combine discrete simulation techniques and modeling of real grid components to achieve scalability to simulated grids of substantial size. GridSim [gri08], developed at the University of Melbourne, supports simulation of various types of grids and application model scheduling. Finally, OptorSim [BCC+03] was developed as part of the European Data Grid project and is a data grid simulator written in Java that is designed to allow experiments and evaluations of various replication strategies in a data grid environment. To evaluate the performance of my replica placement algorithms I extended the OptorSim to support the necessary information gathering needed for my algorithms. The basic architecture and detailed simulation configurations, including the data grid topology, data file access patterns, and storage capacities of replica

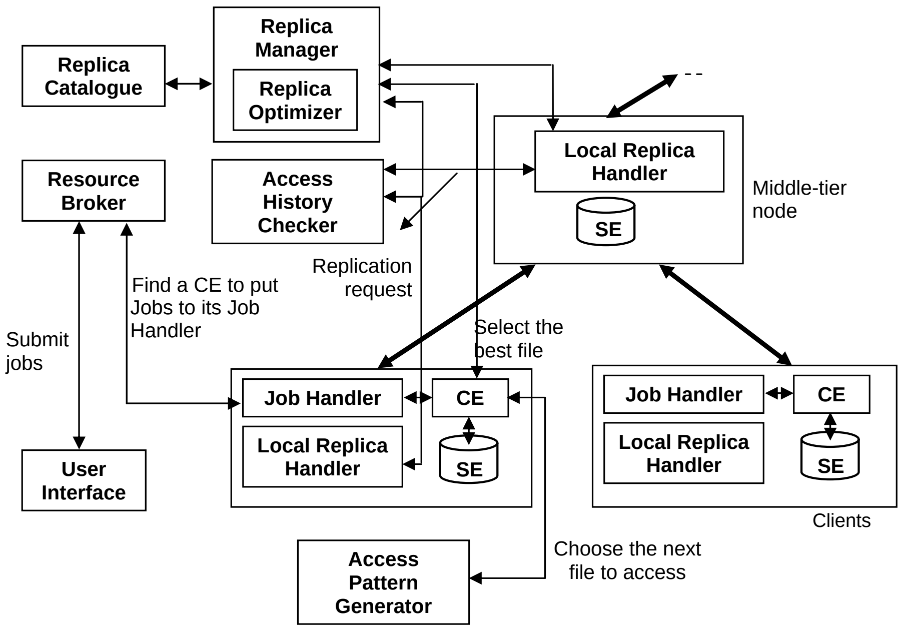


Figure 4.5: System architecture (built on top of OptorSim Architecture)

servers considered, are now described.

4.4.1 OptorSim System Architecture

Figure 4.5 shows the architecture of OptorSim. The simulator is designed assuming that the grid structure consists of a number of sites, each consisting of zero or more computing elements (CEs) and zero or more storage elements (SEs) [BCC$^+$03]. The computing elements provide computational resource and the storage elements serve as data storage resources for submitted jobs. A resource broker acts as a meta-scheduler that controls job scheduling to different CEs. A job accesses a set of files that may be located remotely. A logical filename (LFN) is an abstract reference to a file irrespective of its location in the storage elements. A physical file name (PFN) refers to a specific replica of a LFN stored to a specific storage element. There may be many physical locations of a logical file. To get the physical locations of a logical file, each CE consults a replica manager. After getting information from the replica catalogue (RC), the replica manager returns the locations of the physical files for the requested logical file to the computing element. In general, the best replica of a file would be the one that has the minimum transfer time given current network state. The network bandwidths simulated by OptorSim are shared equally by all connections. The available bandwidth for a connection is determined by the lowest bandwidth along the transmission path. Replica update management is also supported in the system. The locks of all data files in the grid are stored in the RC and managed by the replica manager. If a node wants to update a data file, it must obtain the file write-lock from the replica manager. Once the write is done, the replica manager will propagate the updates to all replicas of the data file and release the lock.

In OptorSim, optimization is seen as an ongoing activity, which is performed at two points during the lifetime of a job. The first optimization phase occurs when the CE where the job should be run is chosen. Various job scheduling strategies are supported by OptorSim including: a) *Random scheduling*: Schedules a job to candidate sites picked randomly; b) *Shortest queue scheduling*: Schedules a job to a candidate site that has the shortest job queue; c) *Access cost scheduling*: For each file listed in the job configuration file, get the access cost of each file depending on the current network state. By adding the times to access the best replica of each file, the method returns the estimated file access time the job would have if scheduled to a given computing element. The scheduling algorithm then schedules a job to the computing element that has the minimum access cost; d) *Queue access cost scheduling*: Schedules the job to the computing element that has the lowest access cost for the job itself and the access costs of all jobs in the queue.

In the second optimization phase, optimal dynamic replica selection is achieved during the run time of a job; in this phase creation of replicas can be triggered by the replica placement algorithms. In this thesis, I consider only optimization that occurs after a job has been scheduled to a CE. This allows us to focus on the performance of the proposed replication algorithms.

In the European Data Grid project, the grid optimization service is called the replica optimizer and it is embedded into the replica manager as shown in Figure 4.5. The replica optimizer makes decisions about data movement associated with jobs between sites and creation or deletion of replicas. I added Access History Checker (AHC) and Local Replica Handler (LRH) modules to the OptorSim architecture as shown in Figure 4.5. The AHC keeps track of data access histories that collectively contain the information of client access patterns. The AHC also invokes the necessary replica placement algorithms to process the histories at regular intervals, to discover popular files, and to decide on replication. If the AHC determines that a replica should be created in a certain node, it will send a request to the LRH of the node and ask it to carry out the replication. When the new replica is created successfully, the LRH will notify the replica manager and let it update the RC to ensure consistency of the directory information.

4.4.2 Simulator Internals

The simulator starts by distributing the master files (original copies) to the storage elements specified. These files are then registered in the replica catalog. A job is

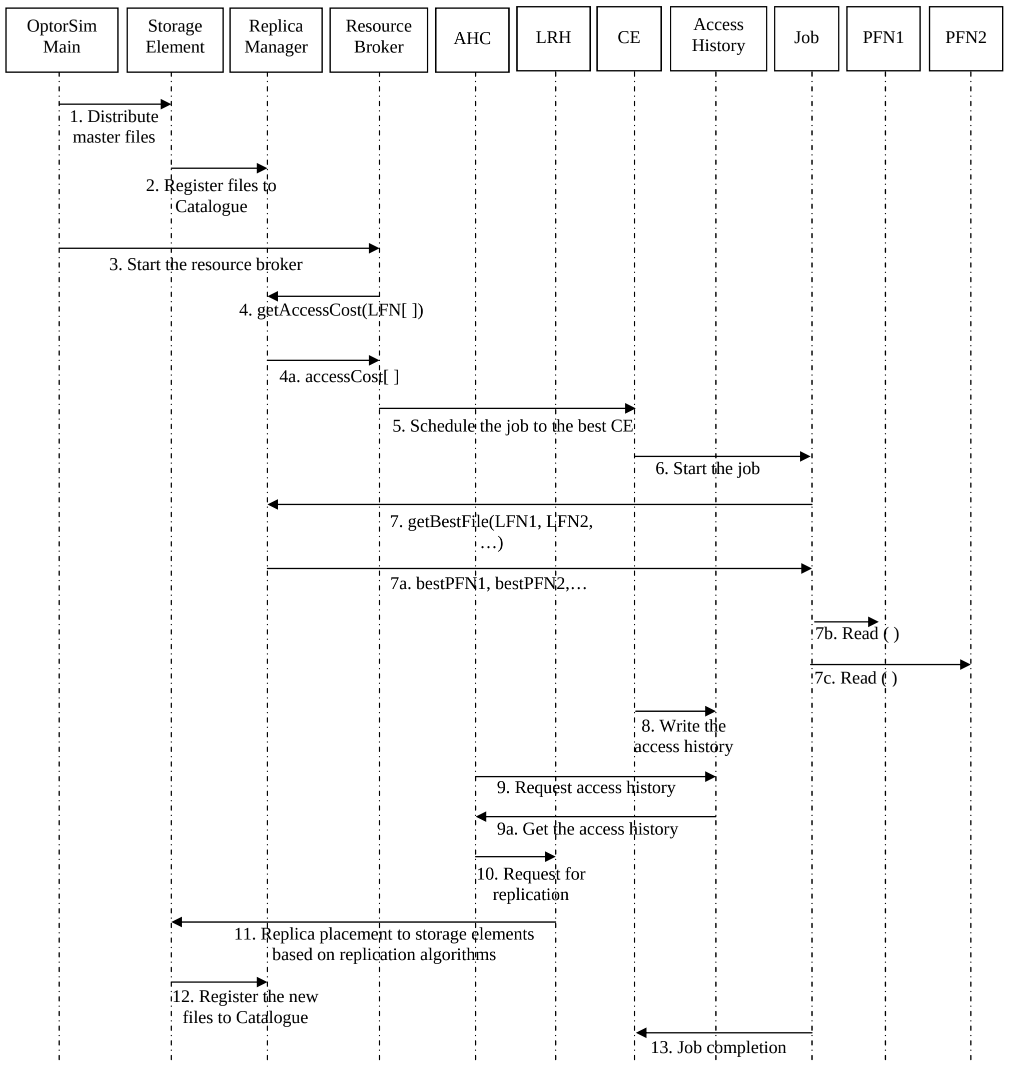

OptorSim Main
Storage Element
Replica Manager
Resource Broker
AHC
LRH
CE
Access History
Job
PFN1
PFN2
1. Distribute master files
2. Register files to Catalogue
3. Start the resource broker
4. getAccessCost(LFN[ ])
4a. accessCost[ ]
5. Schedule the job to the best CE
6. Start the job
7. getBestFile(LFN1, LFN2, …)
7a. bestPFN1, bestPFN2,…
7b. Read ( )
7c. Read ( )
8. Write the access history
9. Request access history
9a. Get the access history
10. Request for replication
11. Replica placement to storage elements based on replication algorithms
12. Register the new files to Catalogue
13. Job completion

Figure 4.6: A typical interaction between different components of OptorSim, Numbers indicate the sequence of operations

picked up by the resource broker from the user interface, and scheduled to a suitable computing element by following any of the four scheduling strategies described earlier. Once the scheduling of the job is done to a specific computing element, the computing element requests the files that are needed by the job. Since there might be multiple copies of a file available on the grid, the computing element calls the *getBestFile*()

method that locates the best file given current network conditions. When a file transfer is made, the logical filename, the requesting grid site, and file access count are written into the history log file which is later used for replica placement decisions. The execution flow (shown in Figure 4.6) where a typical interaction between different components of the simulator has taken place is as follows:

1. OptorSimMain initializes the storage elements and distributes the master files among different storage sites as specified in the configuration file. In my experiments, the master files are initially stored in the root node.

2. Storage elements register those master files in the replica catalogue.

3. OptorSimMain starts the resource broker.

4. For Access cost scheduling strategy, the resource broker calls *getAccessCosts*([LFN1 ... LFNN], CE[], ...) to get the minimum access costs for all candidate CEs with respect to physical locations of the data corresponding to each logical file name (LFNx). This information is used to determine the best CE for job execution.

5. The resource broker schedules the job to the best CE.

6. The computing element starts the job.

7. The job calls *getBestFile*([LFN1, LFN2],) to determine the best physical locations of logical files considering current network state.

8. While reading the files from other storage sites, the CE saves the access histories of the files.

9. The access history checker gets the history for a given file from the log and determines how to place replicas to the appropriate grid sites based on the replica placement algorithms.

10. If a replication decision is made for a grid site, the access history checker sends a request for replication to the local replica handler of the site.

11. The local replica handler carries out the replication to the storage element of the chosen site.

12. The replica catalog is then updated about the new replica that has been created in the storage element.

Figure 4.7 shows the sequence of events that happen when the resource broker calls the function $getAccessCosts()$:

1. The replica manager's *listReplicas*() method is called to find the physical locations of the logical files. A number of physical filenames (PFN) will be found for each logical filename (LFN).

2. Network cost is estimated for each site that holds a PFN corresponding to the LFN.

3. The access costs for each CE are calculated based on the values returned by $getNetworkCosts()$.

4. The scheduler schedules the job to the computing element that has the lowest expected access costs.

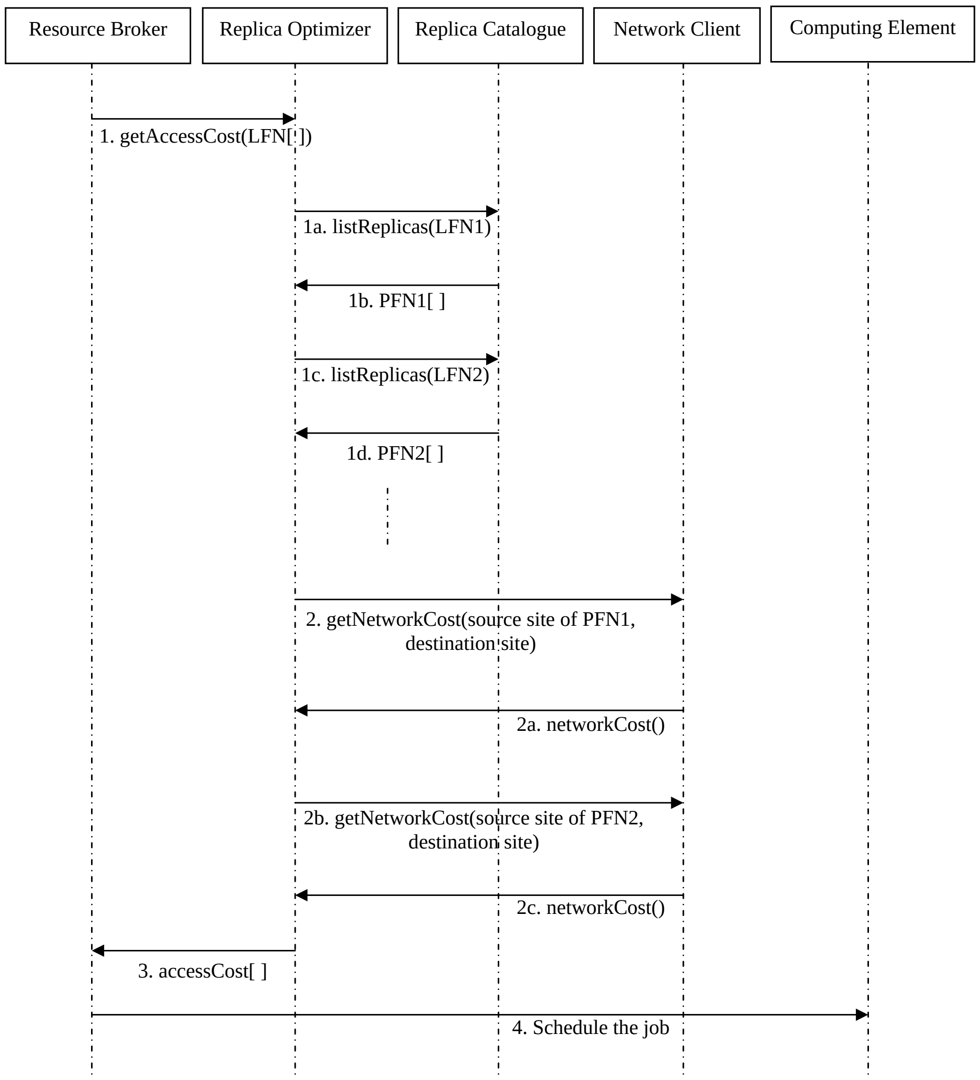
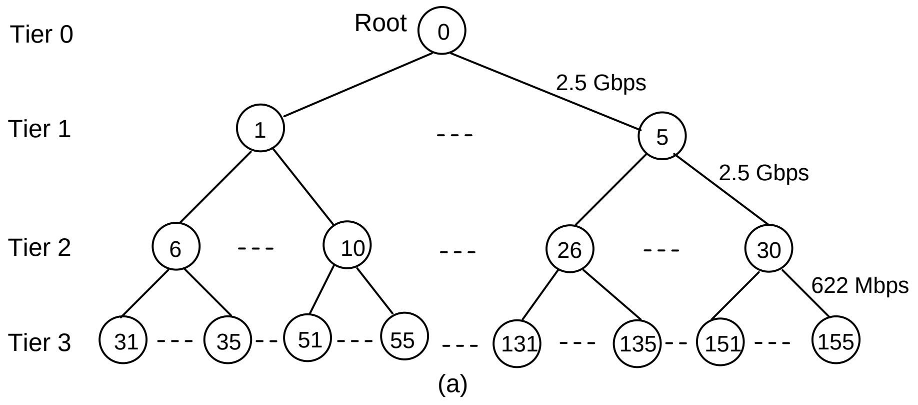


Figure 4.7: Details of access cost determination

4.4.3 Data Grid Topology

The simulated data grid structure and network bandwidths between sites used in my experiments are based on estimates for the CMS experiment [Hol01]. The topology of the simulated data grid is shown in Figure 4.8 (a) which has four tiers, a root at tier-0, five regional centers in tier-1, 25 national centers in tier-2, and 125



Config	Tier 1 (GB per node)	Tier 2 (GB per node)	Tier 3 (GB per node)	Relative storage capacity (%)
1	1000	300	50	75
2	500	200	50	55
3	250	100	50	40
4	125	50	20	17.5
5	50	25	20	13.75

(b)

Figure 4.8: (a) Simulated data grid topology, (b) Different storage configurations

institutional centers in tier-3. Each center in tier-0, -1 and -2 serves five centers in the lower tier. Each center represents a node in the resulting hierarchical topology. All data requests are generated by the leaf nodes. The links between nodes show the available network bandwidth.

The effectiveness of my replica placement algorithms are studied for a wide range of storage resource configurations based on the relative storage capacity of the replica servers, defined as a ratio between the total capacity of the replica servers (S) and the total size of all data files in the system (F). There are 2500 data files in the system and each is 10 GB, so F is approximately 25 TB. The relative storage capacity of the replica servers in the data grid is varied from 75% down to 13% across five simulation cases to reflect the varying amount of storage available at the replica servers. The specific storage configurations for all five cases are shown in Figure 4.8 (b). For example, in configuration 2, each node in tier-1, tier-2, and tier-3 has 500 GB, 200 GB, and 50 GB storage available for replicas, respectively. The absolute capacities are actually of little interest but the relative capacities affect placement decisions. The total storage size of the replica servers is 13.75 TB and the relative storage capacity is 55% ($\frac{13.75}{25} \times 100$). To assess the impact of file size distribution on the performance of the replication algorithms, I also considered file sizes of 2 GB and 20 GB. I adjusted the storage capacity of each server in different tiers so that the relative storage capacity of the servers was unchanged.

4.4.4 Simulation Inputs and Data Access Patterns

The OptorSim simulator reads input from three configuration files. The first describes the network topology (i.e. available network links between grid sites and the bandwidth of each defined link) and the components of each site (i.e. number of computing and storage elements, as well as their sizes). The second configuration file contains information about simulated jobs and the names of logical files that the jobs need to access. It also contains the index of each logical file and a schedule table for each computing element specifying which jobs each computing element will run. The third configuration file specifies different simulation parameters including information such as the total number of jobs to be run, file processing time, delays between each job submission, maximum queue size in each computing element, file access pattern, and the replication algorithm used.

A resource broker acts as a meta-scheduler that controls job scheduling to different client nodes with computing resources based on the estimated data access cost for the current and all queued jobs. As described, a job will request from a predefined set of logical filename(s) for data access. The order in which the files are requested is determined by the selected access pattern. The following access patterns were considered: sequential (files are accessed in the order stated in the job configuration file), random (files are selected randomly from a set with uniform probability distribution), unitary random walk (file requests are one element away from previous file request but the direction is random), Gaussian random walk (as with unitary random walk, but files are selected from a Gaussian distribution centered on the previous file request), and Zipf. The Zipf distribution is given by: $P_i = \frac{K}{i^s}$, where P_i is the frequency of the ith ranked item, K is the popularity of the most frequently accessed data item and s determines the shape of the distribution. The frequency of a request is the inverse of its rank in the frequency. This means that some file requests will occur frequently while many others will occur rarely. Examples of the first four file access patterns and the probability mass function for the Zipf distribution are shown in Figures 4.9 and 4.10 respectively (Note the logarithmic scales for the Zipf graph).

Using the sequential access pattern, every file in the job will be accessed in the order stated in the job configuration file. For all other access patterns, any file in the job may be accessed zero or many times. However, the number of file requests always corresponds to the number of file requests in the job description. For example, using the unitary random walk access pattern (example from Figure 4.9),the job requests ten files so ten requests were made in total. The first file was randomly selected as the sixth in the list of possible files. The next file had an equal probability of being

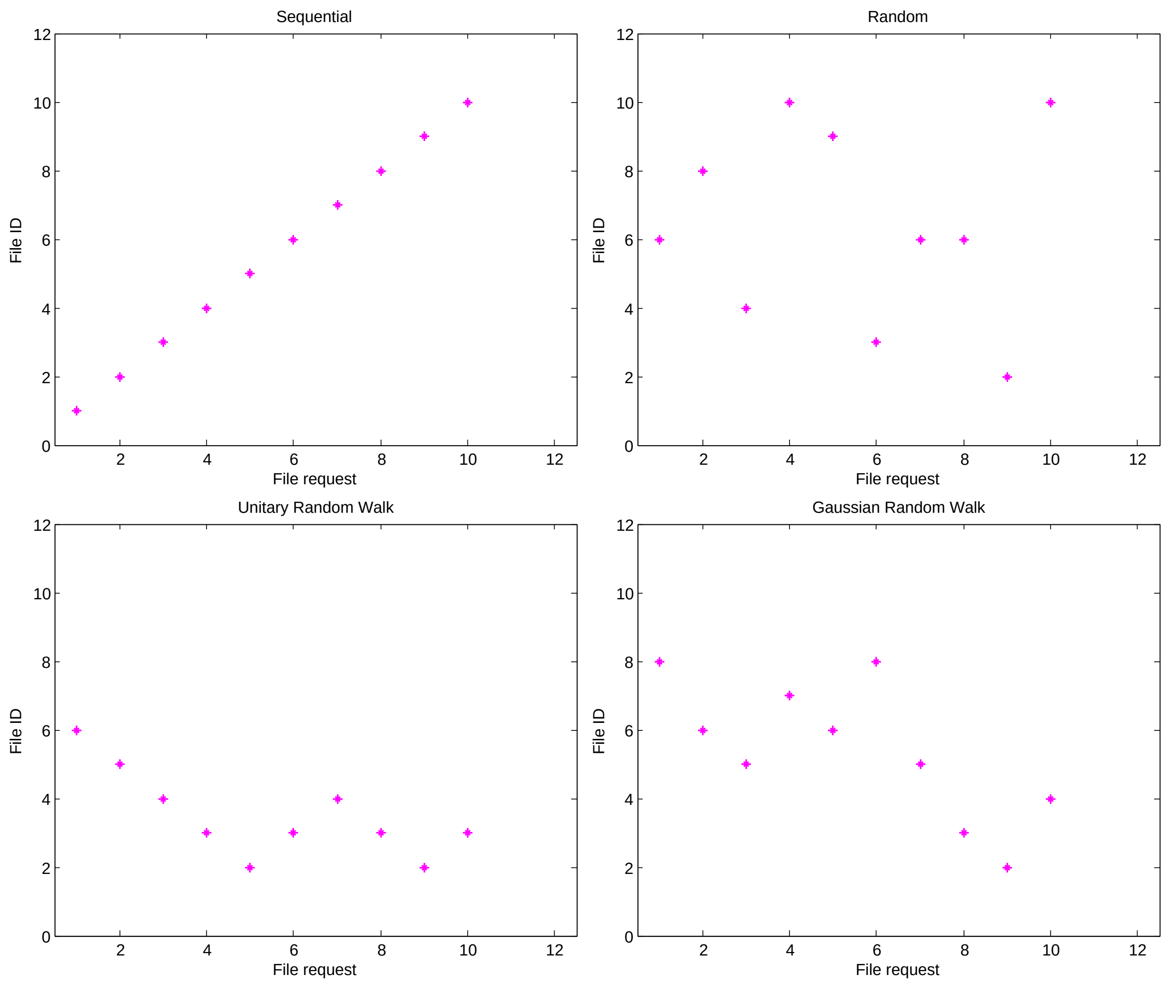


Figure 4.9: Example access patterns for a job containing ten files

file 5 or file 7. As 5 was chosen, the next file had an equal probability of being file 4 or file 6, etc. In this example, the job requested file 3 four times whereas files 1, 7, 8, 9, and 10 were never requested.

The Gaussian and Zipf data access patterns are expected to reflect the behavior of real world applications that will use data grids. Moreover, the Gaussian distribution is the most widely used family of distributions in statistics and many statistical tests are based on the assumption of normality. As such, it is a good base measure which

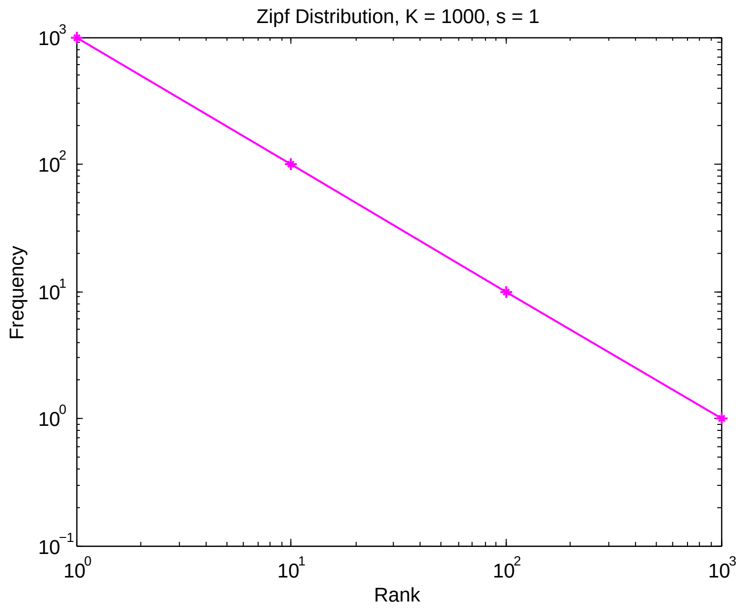


Figure 4.10: Zipf distribution

can be used for easy informal comparison to known applications. Further, in Zipf, the frequency of a request is the inverse of its rank in the frequency. This means that some file requests will occur frequently while many others will occur rarely. Zipf-like distributions exist widely in the Internet. In a system that is designed to react to file popularity, the Zipf distribution offers a natural testing ground.

The data access requests from the clients follow Poisson arrivals. On average, each client sends one request per 2500 milliseconds. According to the properties of a Poisson process, if all the clients are assigned jobs for execution, the merging of 125 Poisson streams results in a Poisson stream with about 50 requests per second for the whole system. Also, it is assumed that request patterns for the files can exhibit various locality properties such as *temporal* and *spatial* locality. Temporal locality implies that recently accessed files are likely to be accessed again. On the other hand, spatial or "file locality" infers that files near a recently accessed file are likely to be accessed. In this definition of spatial locality, we need to specify what "near" means. This definition involves a study of the nature of the data in the files and how we can relate files to each other. This thesis focuses on temporal locality of file accesses and leaves the study of relationships between data files for future work.

As yet we do not know to what extent file access patterns will exhibit the locality properties described above and whether there will be any locality. We can only make reasonable guesses at this point. The worst-case scenario is when the access patterns do not exhibit any locality at all; generating random access patterns (Gaussian random walk, unitary random walk, and flat random distributions) can simulate this situation. More realistic access patterns that contain varying amounts of temporal locality can be generated using the Zipf distribution. The index used to measure the amount of locality in the patterns is denoted by s, which determines the shape of the distribution. The observed parameter values are in the range of $0.65 < s < 1.24$. A higher value of s indicates an increased degree of locality. In this research, parameters of $s = 0.85$ and $s = 1.0$ are used individually, and hereafter we refer to them as Zipf-0.85 and Zipf-1.0 distributions, respectively. The selected values of s will allow Zipf distribution to show a relatively low and high degree of temporal locality.

4.5 Results and Observations

In this section, the performance results of my centralized replica placement algorithms are presented and discussed. The studied performance metrics include job execution time, average bandwidth cost, storage use, and the number of replicas created.

- *Job execution time*: Job execution time is the total time to execute all the jobs. Execution time for each job includes the *response time* for accessing the files contained in the job. The response time for a data file access is represented by the interval between the beginning of the data request sent by the client and the end of the data transmission. If a local copy of the file exists, the response time is assumed to be zero. Job execution time is the primary consideration from the perspective of the data consumer.

- *Average bandwidth cost*: Usually, the price of an international network link is higher than that of a national link [AAR89], and the prices for LAN and campus networks are negligible. In this research, for the sake of simplicity, the unit price is set as one for link between all the tiers. For each data transmission, the bandwidth cost is the data size multiplied by the summation of costs along the data transmission path. The average bandwidth cost is the total bandwidth cost divided by the number of client data requests. This includes the bandwidth consumed for data transfers incurred when a client requests a file and when a replica is created at a replica server down the hierarchy. This is an issue both for network providers and end-users (since excessive use of bandwidth can lead to slow downs due to network congestion).

- *Storage use*: Storage use is the percentage of the available storage used by the replicas. The total use of storage in a data grid is important to grid providers but due to its relatively low cost can be effectively traded-off for improvements in job execution time and network bandwidth consumed, as needed.

- *Number of replicas*: The metric of number of replicas represents the total number of replicas created for all data accesses requested by the clients in a simulation session. An increased number of replicas implies a higher replication frequency which is the value of how many replications occur per data access. For each replication operation, not only is the network bandwidth resource consumed, but also replica server load is increased because of the disk I/O and CPU use. Therefore, the frequency of replication operations must be controlled to avoid heavy network and server load.

First, I compared my basic placement algorithm (PBRP) with five other replication algorithms: aggregate bottom-up (ABU) [TLYT05], Fast Spread, Cascading, Best Client, and Caching with LRU replacement [RF01b; RF01a]. Then to determine the effectiveness of my *adaptive* algorithm I compared APBRP with its non-adaptive counterpart PBRP considering three scenarios: when the data access rate is consistently increasing, consistently decreasing, and when it fluctuates. In real worlds, data access arrival rate from clients may vary with time. With variation in data access rate, the number of popular data files will change over the simulation session. To generate the data access pattern with dynamically changing rate, every simulation session is partitioned into a number of evenly spaced sub-sessions or sampling intervals. The first sampling interval in the simulation is considered as a warm-up period for PBRP and other considered replication algorithms. The reason is that in this period the replication algorithms collect data access statistics from clients to use them in replica placement in the second sampling interval. However, the performance data collected over the warm-up period is included in the overall performance results of the replication algorithms. In each sub-session, the data access requests from the clients follows a Poisson arrival. The access arrival rate (which is expressed as the average number of arrivals during a unit of time) is changed across sub-sessions to reflect the three scenarios mentioned. The arrival rate is regularly increased and decreased throughout the simulation session to simulate a regular fluctuation in access rate. Likewise, the arrival rate is continuously increased or decreased to a certain level during the entire simulation period to simulate a consistent increase and a consistent decrease in access rate, respectively.

4.5.1 Job Execution Time

Figure 4.11 (left) shows the job execution times of PBRP and the pre-existing replica placement algorithms using storage configuration one (significant storage throughout the hierarchy, refer to Figure 4.8 (b)). Since, in this case, the clients have substantial storage capacity for replicating files locally the Caching technique significantly reduces the access time. In a more realistic grid scenario, the client nodes (having both computing and storage resources) which are primarily meant for job execution are likely to have less available space for large scale data storage compared to the middle-tier replica servers. For example, the Large Hadron Collider (LHC) [LCG01] project at CERN is expected to produce roughly 15 Petabytes (15 million Gigabytes) of data annually, which thousands of scientists around the world will access. Individual scientists (clients) will access these data through the lowest tier centers, which might consist of local servers or small clusters in a University Department having storage capacity far less than the total produced data size. Due to this limited expected storage capacity of clients, using caching files will get replaced quickly which in turn will increase access time causing an increase in job execution time (as can be seen in Figure 4.12 and 4.15). The same is also true for sequential data access due to frequent replacement of replicas. Apart from this, Best Client consistently performs worse than other algorithms for all data access patterns. Among the other four strategies, for sequential and flat random data, Fast spread shows the least job execution time though the difference in job time is marginal (2%) in the latter case when compared to PBRP. For Zipf-0.85, Gaussian, and Unitary Random Walk access patterns, PBRP gives the best job execution time. This is due to the fact that the

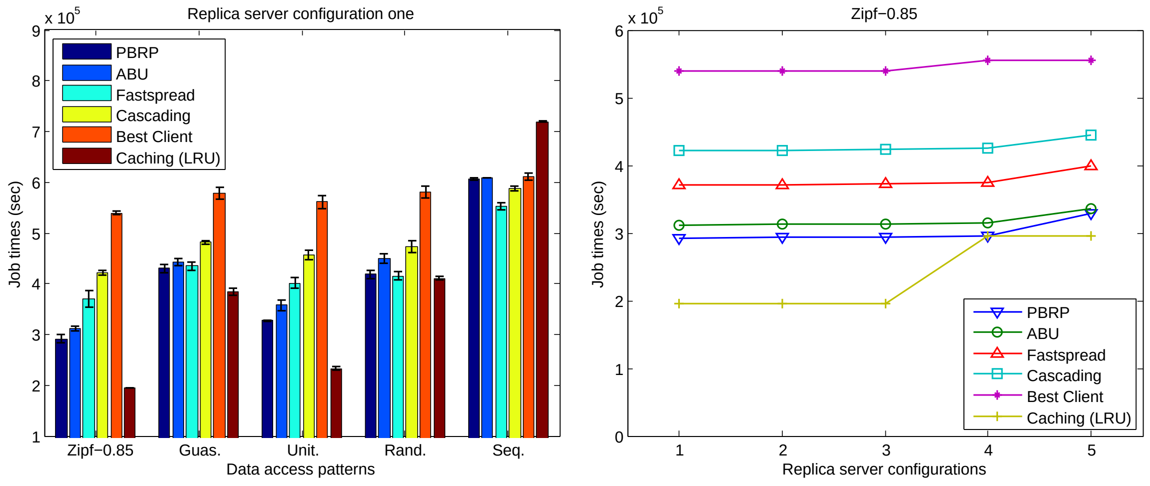


Figure 4.11: Job execution times for different replication methods using storage configuration one (left) and comparison of execution times for Zipf-0.85 distribution (right)

data access latency in PBRP is significantly reduced by selectively replicating popular (likely to be accessed) data files close to the clients. In the case of Zipf-0.85 (with a small amount of temporal locality), the advantage of PBRP over the other algorithms increases. Once the data contains more locality, PBRP has a significant improvement in performance; its job execution time is almost 6% and 25% less than that for ABU

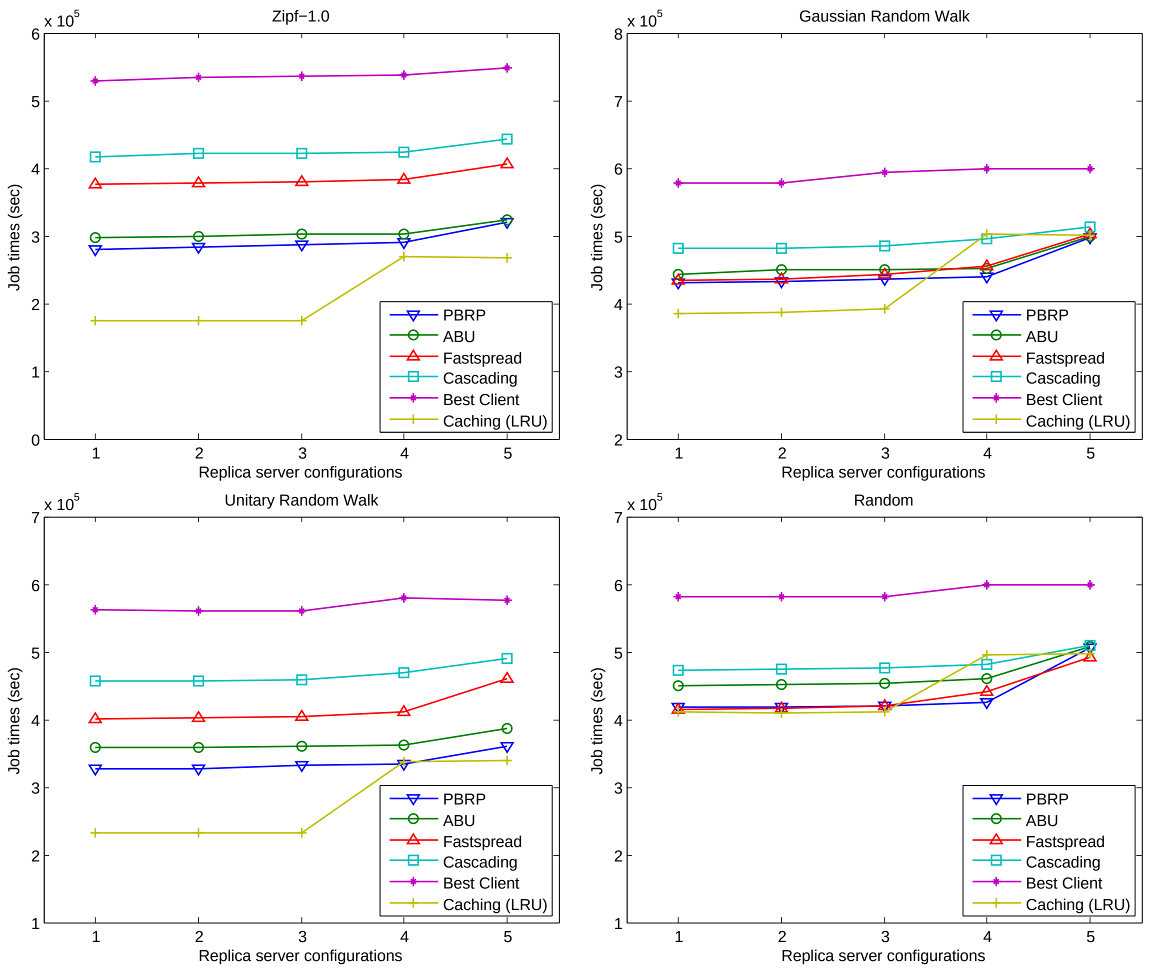


Figure 4.12: Comparison of execution times for different access patterns using various storage configurations

and Fast spread, respectively. This is because when the data access patterns follow Zipf, some file requests occur frequently making them "popular" where many others occur rarely. Thus, the clients focus on a smaller range of data files with higher frequencies compared to other access patterns. Cascading also reduces the performance difference in execution time compared to Fast spread for patterns that contain temporal locality. PBRP requires only a small number of replicas compared to Fast Spread and a slightly higher number of replicas than ABU. As an example, using a

Zipf-0.85 distribution, the approximate numbers of replicas created by PBRP, ABU, and Fast Spread are 194, 149, and 274, respectively for replica server configuration one. Figure 4.13 shows the number of replicas created by different strategies using various data access patterns. The storage capacity of the replica servers has a major

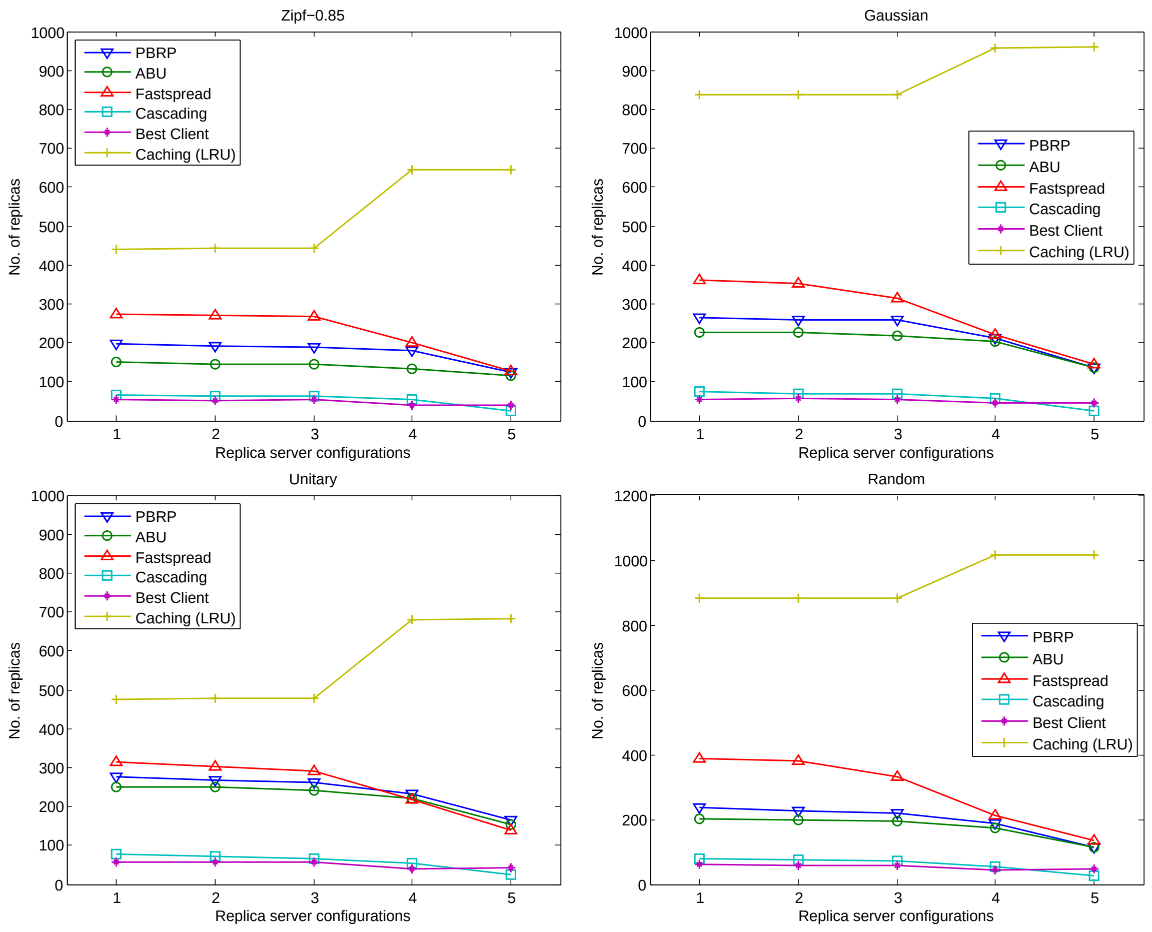


Figure 4.13: Number of replicas created for different access patterns using various storage configurations

impact on the performance of the placement algorithms. With decreasing capacity, the execution times of all methods are increased but by different degrees. Relatively frequent replacement of replicas due to the scarcity of storage capacity in replica servers causes an overall increase in data access latency which, in turn, increases the job execution time. Figure 4.11 (right) and 4.12 show the job execution times of different strategies for the five storage configurations considered using various data access patterns. For all configurations, PBRP shows consistently better performance than the other algorithms except Caching using Zipf, Gaussian, and Unitary data. When the available storage capacity is smaller, the benefit of PBRP over ABU decreases slightly. When the relative storage capacity is 17%(Configuration 4), PBRP shows improvement over Fast spread for flat random access pattern. The job execution time for Caching increases drastically as the storage capacities of the client sites decrease (Configurations 4 and 5). In particluar, for random data access patterns, the job execution time is higher than PBRP for most cases. Since the storage size in the client-tier is the same in both of these configurations, the job execution times for Caching and Best Client remain almost the same. This is because files are replicated on the client sites in both strategies.

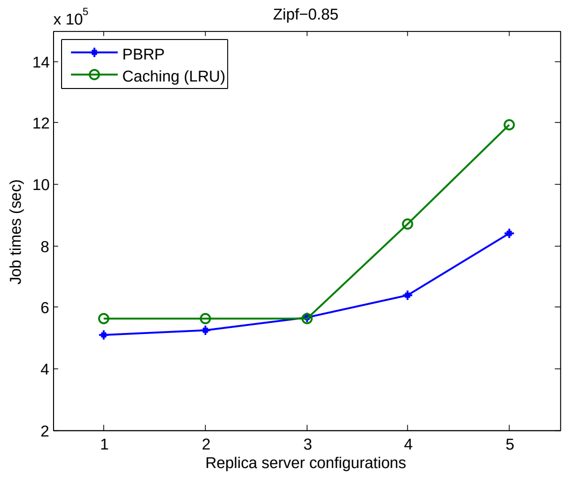


Figure 4.14: Comparison of job times from an increased simulation period using Zipf-0.85 access pattern

As seen from the foregoing discussion, Caching outperforms PBRP and other algorithms when the clients at tier-3 have sufficient storage (e.g. Configurations 1, 2 and 3) although the performance gap in terms of job time between Caching and PBRP almost diminishes in case of flat random access behavior. This can be attributed to the fact that Caching with sufficiently high storage at clients experiences minimal cache misses compared to the cases when storage is limited (e.g. Configurations 4, and 5). Also, once the simulation is started, Caching can take advantage of temporal locality relatively quickly to reduce the cache miss which in turn reduces job execution time, whereas the other methods need a little longer to collect statistics and place replicas down the hierarchy.

That said, I made a number of changes in the simulation that highlight the advantages of PBRP over Caching. First, the simulation length is doubled by increasing total number of jobs from 100 to 200. This increased simulation period facilitates a large pool of files to be accessed by clients and gives simulation enough time to reach a point where Caching starts getting more cache misses (similar to what happens in Configurations 4 and 5 when storage is limited) which increase its job time. This is due to the fact that on a cache miss Caching needs to retrieve the file all the way from the root causing an increased access latency compared to PBRP which also experiences misses as simulation progresses. However, due to the placement of replicas down the hierarchy PBRP can retrieve the files from relatively close locations which reduces its file access latency thus improving job execution performance. Also, increasing total simulation length reduces the impact of warm-up period (first simulation sub-session) on the performance of PBRP. This means that, as the length of the simulation increases, the effect of having a warm-up period that is too short is ameliorated somewhat because the initial period of the simulation has less weight than it would in a shorter simulation. Second, increasing warm-up period for better reflection of file access history in making replica decisions and decreasing the subsequent sampling intervals to take advantage of temporal locality gives PBRP little improvement in job time though not significant. It is important to note that substantial increase in warm-up period affects the performance of PBRP adversely since it further delays the placement of replicas close to the clients. Also, if sampling intervals are too short the algorithm incurs an increased overhead due to the frequent creation and deletion of replicas. Third, storage capacity of each node at tier-3 in Configuration 5 (refer to Figure 4.8(b)) has been changed to 10 GB resulting in the relative storage capacity of 8.5%.

As shown in Figure 4.14 and 4.15, PBRP consistently performs better than Caching in terms of job execution time for the random data access patterns (such as Gaussian and flat Random) where data requests from clients occur for a wider range of files. This causes an increased cache misses thereby resulting in high job execution times for Caching compared to PBRP. The performance improvement of PBRP in terms of job time is not terribly significant compared to Caching when clients have sufficient storage (i.e. Configurations 1, 2 and 3) and data access patterns contain temporal locality (i.e. Zipf-0.85 and Zipf-1.0). In fact once the access patterns contain more locality Caching in some cases shows somewhat better job execution time than PBRP though the difference in this case is up to 6%. This is because when the data access patterns follow Zipf, some file requests occur frequently where many others occur rarely. Thus, the clients focus on a smaller range of data files with higher frequencies

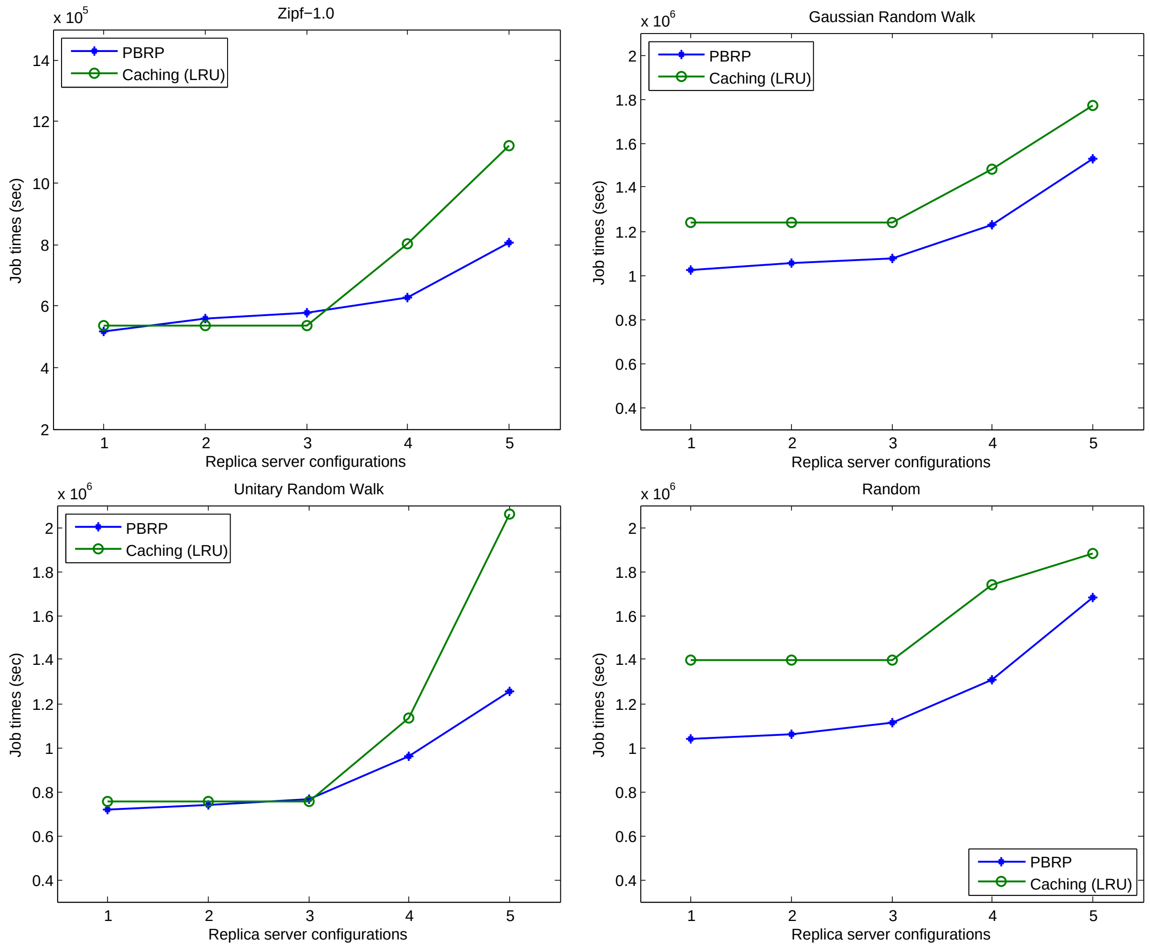


Figure 4.15: Comparison of job times from an increased simulation period for different access patterns

compared to other access patterns. Caching takes advantage of this temporal locality provided clients have sufficient storage. This leads us to conclude that the performance of Caching in terms of job time will be better if there is sufficient temporal locality in the access patterns and the clients have enough storage.

Given the fact that in e-science applications, a large number of files (not necessarily with frequent repetition) will be accessed by clients over the time, the use of relatively longer simulations to highlight the benefit of PBRP is justified. Although file access may not be frequently repeated by a single client, a collection of clients are likely to reuse access to copies above the leaves in the hierarchy. This gives the benefit of PBRP for multiple clients in a sub-tree. These simulation results now help us better understand the performance of Caching and its relative performance with other algorithms as claimed by Ranganathan and Foster [RF01b].

Figure 4.16 shows the approximate job times for storage configuration one for all data access patterns using different file sizes of 2 GB and 20 GB. The relative performance of the algorithms in terms of job execution time and average number of replicas created is almost the same as the performance obtained using a file size of 10GB. The reason is that the performance of the replication strategies is directly dependent on the percentage of files that can be stored at each node. This can be achieved by scaling both the file size and the storage capacity at each node.

To demonstrate the relative performance of PBRP, Figure 4.17 compares its approximate run times and the number of replicas created using different file sizes for all data access patterns. The job time using a file size of 20 GB is much higher than the other two cases due to the increased data transfer latency. The algorithm shows the lowest execution time for 2GB files, as expected. The average number of replicas required does not differ significantly since the number of files in the system remains unchanged in all three cases.

As discussed earlier, the effectiveness of the PBRP algorithm depends on careful selection of the threshold value that determines the popularity of a data file. In PBRP, once the threshold value is initially calculated, as described in Section 4.2, it remains constant during the entire simulation period irrespective of variation in data

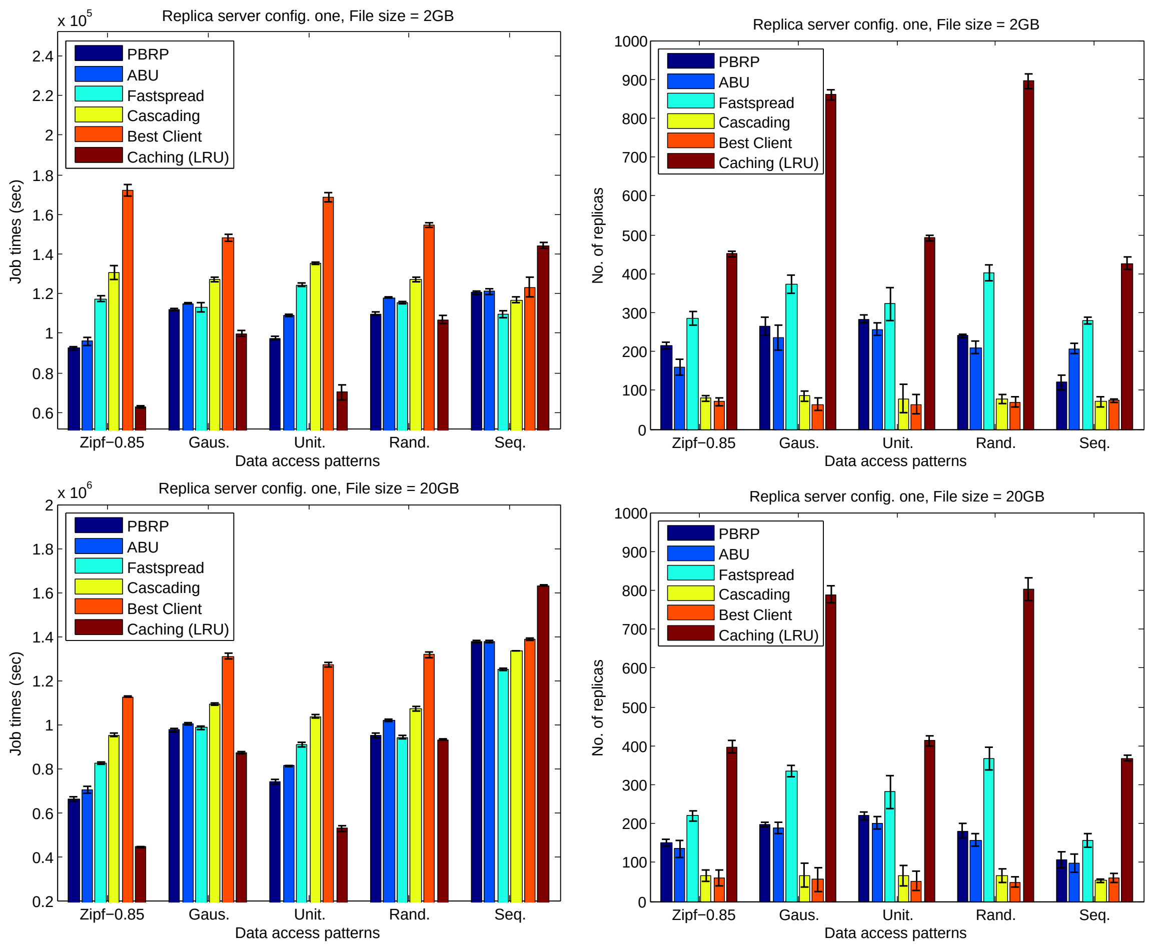
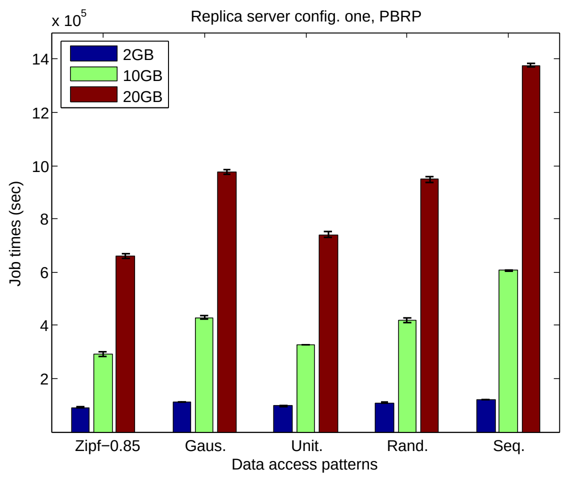


Figure 4.16: Execution times by file size (left), Number of replicas created by file size (right)

access arrival rate and the available storage capacities of the replica servers. APBRP addresses this issue by dynamically changing the threshold value based on the data access rate and storage availability. To determine the effectiveness of this adaptive technique I considered three different scenarios as described earlier; when the data access rate is consistently increasing, consistently decreasing, and when it fluctuates. The following subsections discuss the results obtained for each of these cases by comparing the job execution time of APBRP with its non-adaptive counterpart,



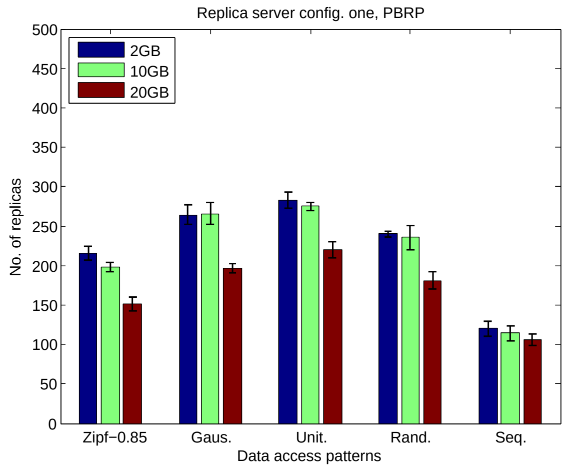
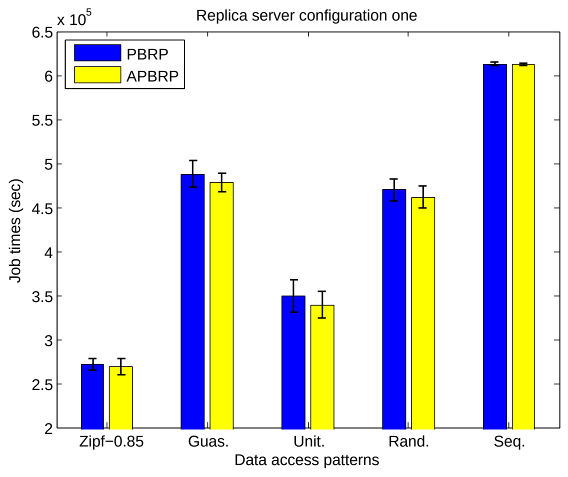


Figure 4.17: Execution times and number of replicas: PBRP by file size, Configuration one



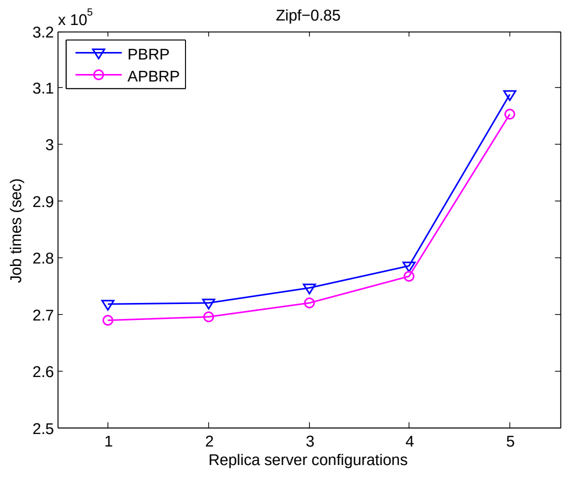


Figure 4.18: Job execution times using resource configuration one (left) and comparison of execution times for Zipf-0.85 distribution (right) when the access rate fluctuates

PBRP (and transitively, to the other non-adaptive algorithms previously compared to PBRP).

Job times when the access rate fluctuates

Figure 4.18 (left) compares the job execution times of APBRP and PBRP using storage configuration one when the data access arrival rate from the clients fluctu-

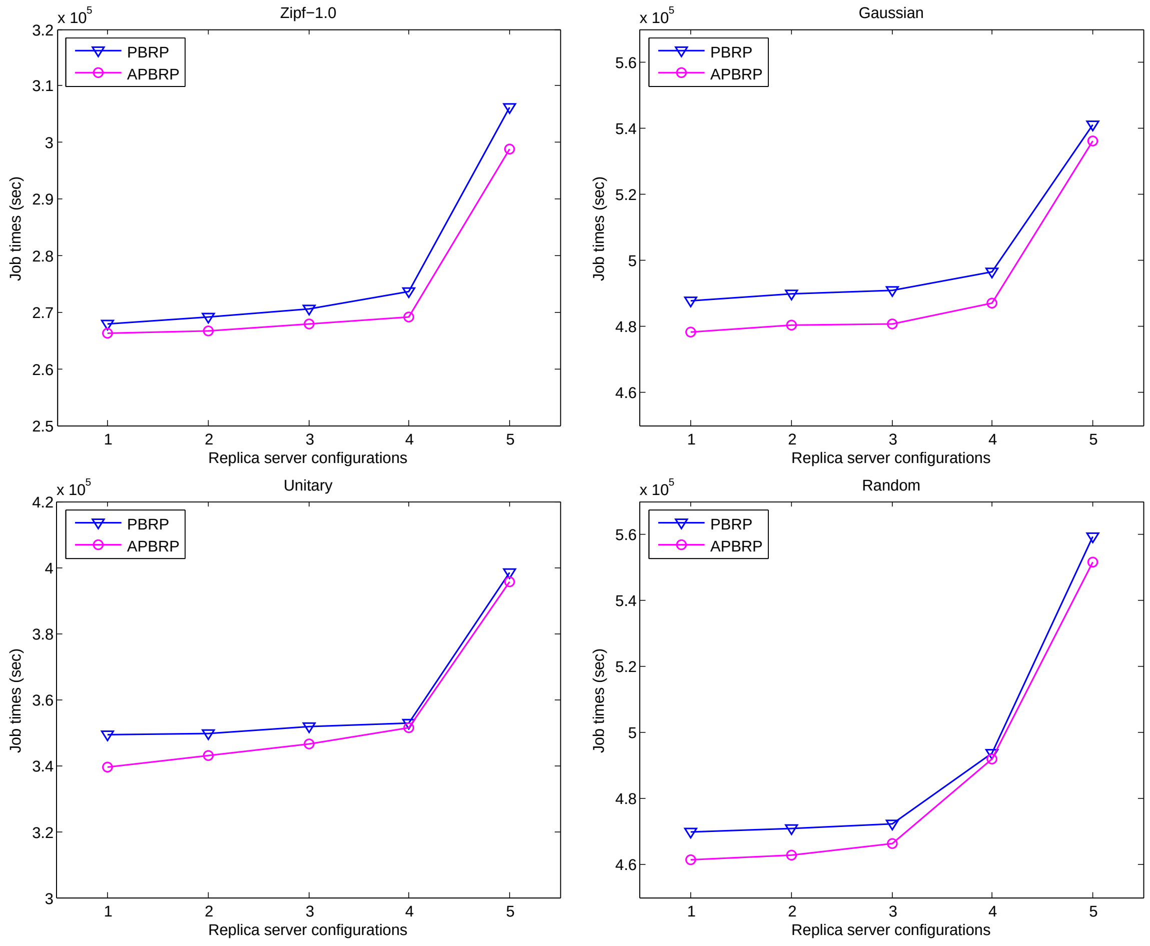


Figure 4.19: Comparison of execution times for different access patterns when the access rate fluctuates regularly

ates regularly. APBRP gives better job execution times for all data access patterns. APBRP adjusts the threshold value based on changes in access arrival rate which leads to the creation of an increased number of replicas compared to PBRP. This in turn decreases the job execution time. As an example, using a Gaussian distribution, the average number of replicas created by APBRP and PBRP are 237 and 215, respectively for Configuration one. Figure 4.20 shows the number of replicas created by both strategies using various data access patterns and replica server configura-

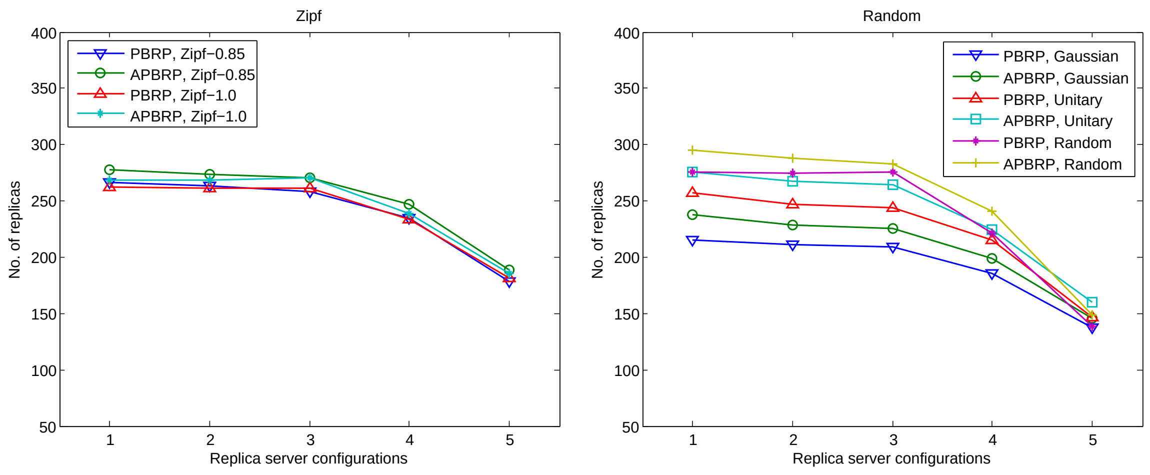


Figure 4.20: Number of replicas created for various storage configurations when the access rate fluctuates regularly

tions. The number of replicas gradually decreases due to the storage constraints in the replica servers. Like PBRP, APBRP shows the lowest job execution time for the Zipf distribution compared to other access patterns. The benefit of APBRP is most significant when the data access pattern shows a degree of randomness. ABRP clearly adapts well to fluctuating accesses.

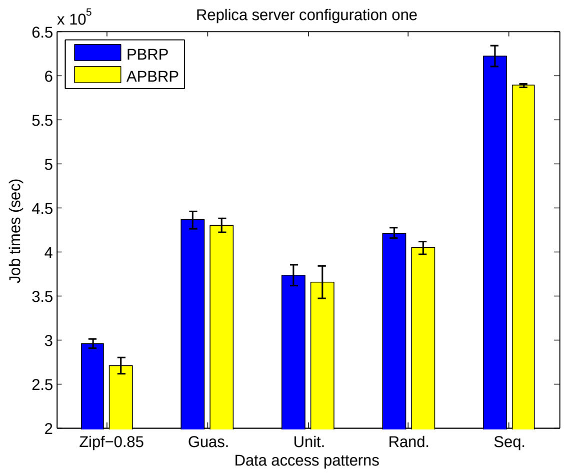


Figure 4.21: Job execution times using resource configuration one when the access rate consistently decreases

Figures 4.18(right) and 4.19 compare the job execution times for all access patterns and storage configurations as the access rate fluctuates. With decreasing storage size, the execution times of both techniques are increased but by different amounts. APBRP shows shorter job times for all storage configurations. Overall, the percentage of job time improvement by APBRP for access patterns with some randomness is more than the case when access patterns contain temporal locality (i.e. Zipf-0.85 and Zipf-1.0). This makes the adaptive technique a suitable choice when files are accessed using random distribution given storage constraints on the replica servers. However, when the available capacity is smaller, the benefit of APBRP over PBRP decreases since there is less available space to allocate additional replicas via adaptation of the popularity threshold. Hence, the increased total number of replicas provides little benefit in term of access latency.

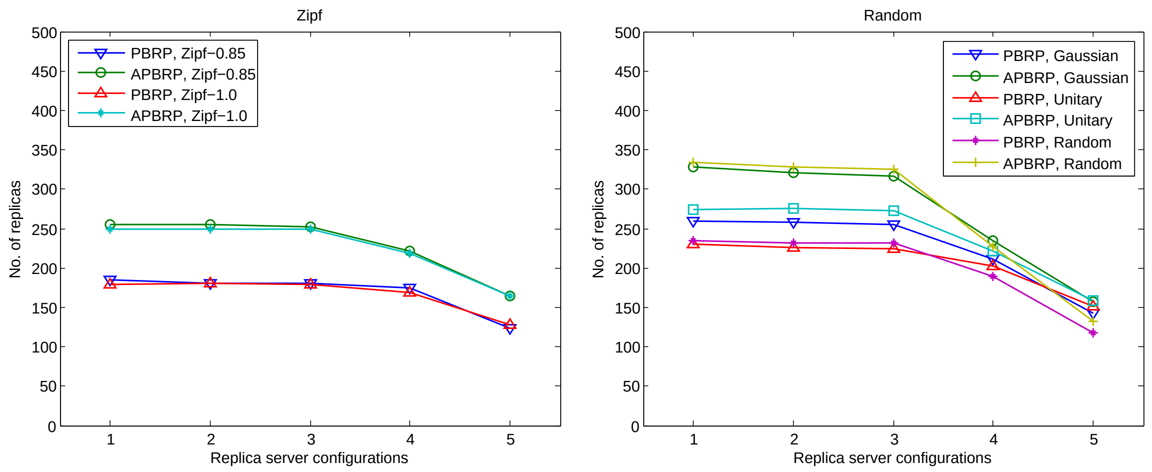


Figure 4.22: Number of replicas created for various storage configurations when the access rate consistently decreases

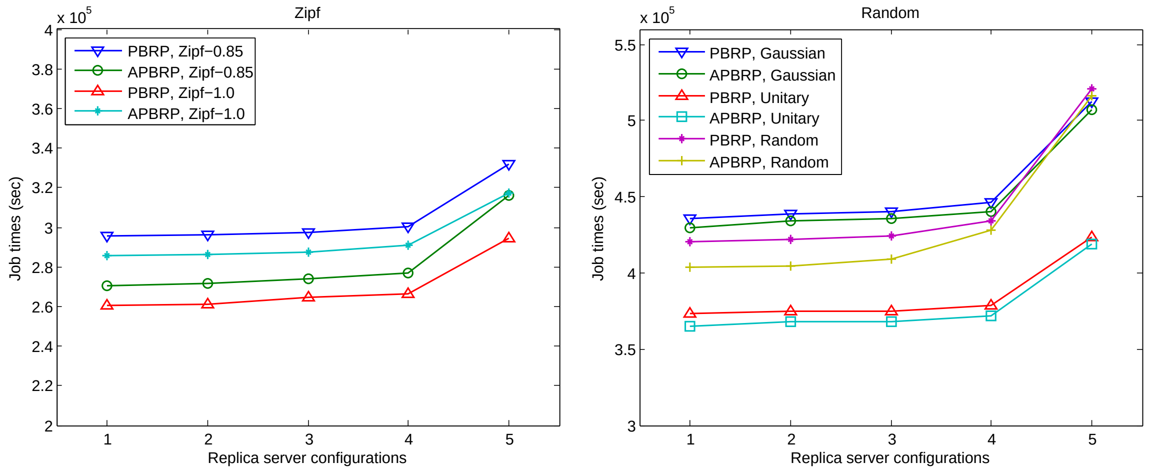


Figure 4.23: Comparison of execution times for different access patterns when the access rate consistently decreases

Job times when the access rate is consistently decreasing

Figure 4.21 shows the job execution times of APBRP and PBRP, again for Configuration one, as the client request arrival rate decreases. APBRP shows consistently better performance for all data access patterns. This is due to the fact that when the access arrival rate decreases consistently, APBRP also decreases the threshold value accordingly which maintains an increased number of replicas compared to PBRP. This, in turn, decreases the data access latency since files are accessed closer and hence faster. Figure 4.22 shows the number of replicas created by both strategies using various data access patterns and replica server configurations. As before, the number of replicas gradually decreases due to the storage constraints in the replica servers. Again, the Zipf distribution shows the lowest execution time for APBRP compared to other access patterns. The benefit of APBRP over PBRP is particularly significant for the Zipf-0.85 and sequential access patterns. Figure 4.23 compares the job execution times of the two placement algorithms using different storage resource

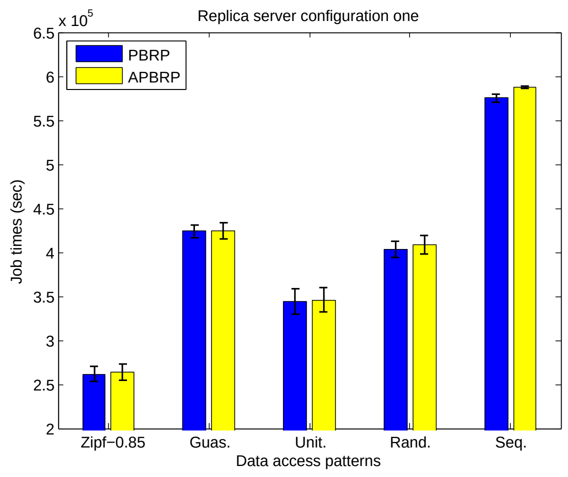


Figure 4.24: Job execution times using resource configuration one when access rate increases consistently

configurations as the access rate decreases consistently. With decreasing capacity, the execution times for both algorithms increase but, again, by different amounts. APBRP shows better job execution times for all storage configurations except for configuration four (relative storage capacity is 17%) where the Gaussian and flat random access patterns have shorter job times for PBRP by almost 2% over APBRP. When the available storage capacity is smaller, the benefit of APBRP over PBRP decreases as before. This happens because of frequent replica replacement in APBRP when the replica servers have limited storage.

Job times when the access rate is consistently increasing

Figure 4.24 compares the run times of APBRP and PBRP, again for configuration one, as the request arrival rate increases. When the request arrival rate increases, APBRP increases the threshold value based on the increase in the rate and tries to limit the number of replicas to a reasonable level based on the storage available (as described in Section 4.3) while PBRP experiences a drastic increase in the number of

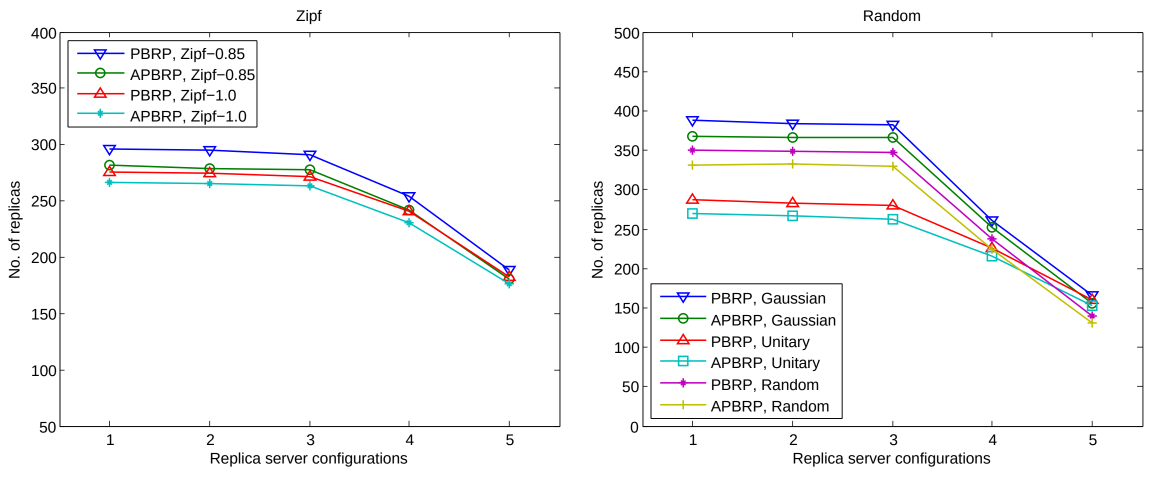


Figure 4.25: Number of replicas created for various storage configurations when the access rate consistently increases

replicas (that eventually fills the replica servers). PBRP performs as well as APBRP or somewhat better for all access patterns due to a significantly increased number of replicas when using PBRP with sufficient storage at the replica servers. Figure 4.25 shows the number of replicas created by both strategies using various data access patterns and replica server configurations. PBRP consistently maintains a larger number of replicas compared to APBRP and, as before, the number of replicas gradually decreases due to the storage constraints in the replica servers. Overall, APBRP does not show any significant benefit over PBRP, for the tested scenarios, when the replica servers have sufficient storage and the client request rate is consistently increasing.

Figure 4.26 compares the job execution times of the two placement algorithms using different storage resource configurations with consistently increasing access rate. With decreasing capacity, the job execution times using both algorithms increase but by different amounts. PBRP maintains better job execution times for all five storage configurations except for the Unitary random walk distribution and when the access

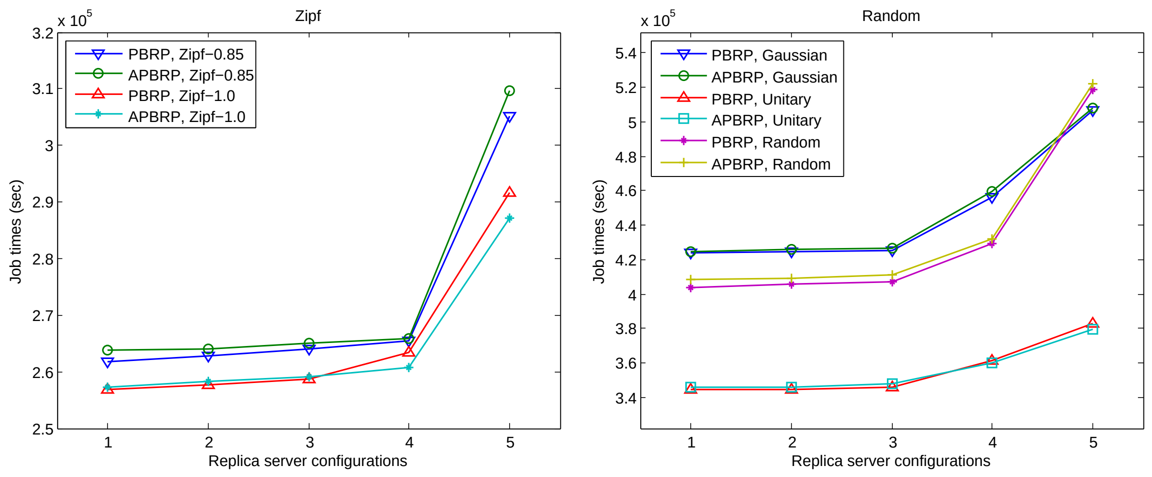


Figure 4.26: Comparison of execution times for different access patterns when access rate consistently increases

pattern contains an increased amount of temporal locality (i.e. Zipf-1.0). For these two access patterns, when the relative storage capacity is less than 40%(Config. 3), APBRP shows improvement over PBRP.

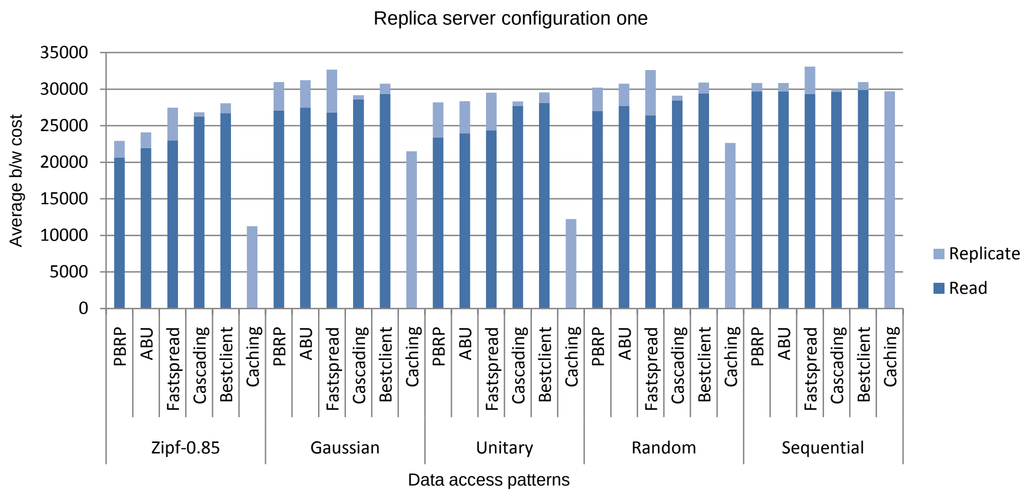


Figure 4.27: Average bandwidth costs for different replication methods using resource configuration one

4.5.2 Average Bandwidth Cost

We now discuss the bandwidth consumption for different cases. Figure 4.27 shows the average bandwidth cost (as defined earlier) for the non-adaptive algorithms and storage configuration one. For all access patterns, the costs of Caching are, naturally, the lowest because the amount of bandwidth consumed is due only to replication as there are no read costs other than the first access. Apart from this, none of the algorithms performs best for all data access patterns. PBRP gives the best average bandwidth cost among the studied placement methods for the Zipf and Unitary random walk access patterns. This is due to the fact that a relatively larger number of data access requests from the clients are served by the middle-tier nodes. As a result, the workloads of the upper tier links are alleviated and the read cost is reduced which in turns makes average bandwidth costs comparatively low. ABU does not differ significantly from PBRP in terms of bandwidth consumption. Fast spread on the other hand leads to the highest bandwidth usage, up to 17% more compared to PBRP when the access pattern contains some locality (Zipf-0.85). This is due to higher replication cost as shown in Figure 4.27 even though in some cases such as for flat random data the read cost is lower compared to others which gives an improvement in job time, as discussed before. Concerning the impacts of the data access patterns, PBRP's bandwidth cost for the Zipf pattern (Zipf-0.85 and Zipf-1.0) is the lowest as a whole. As mentioned before, this is because when the data access pattern follows Zipf, the clients focus on fewer data files with higher frequencies of use. As a consequence, the popular data files can be easily identified by PBRP and the performance is improved. The difference between PBRP and the other strategies is thus more pronounced for

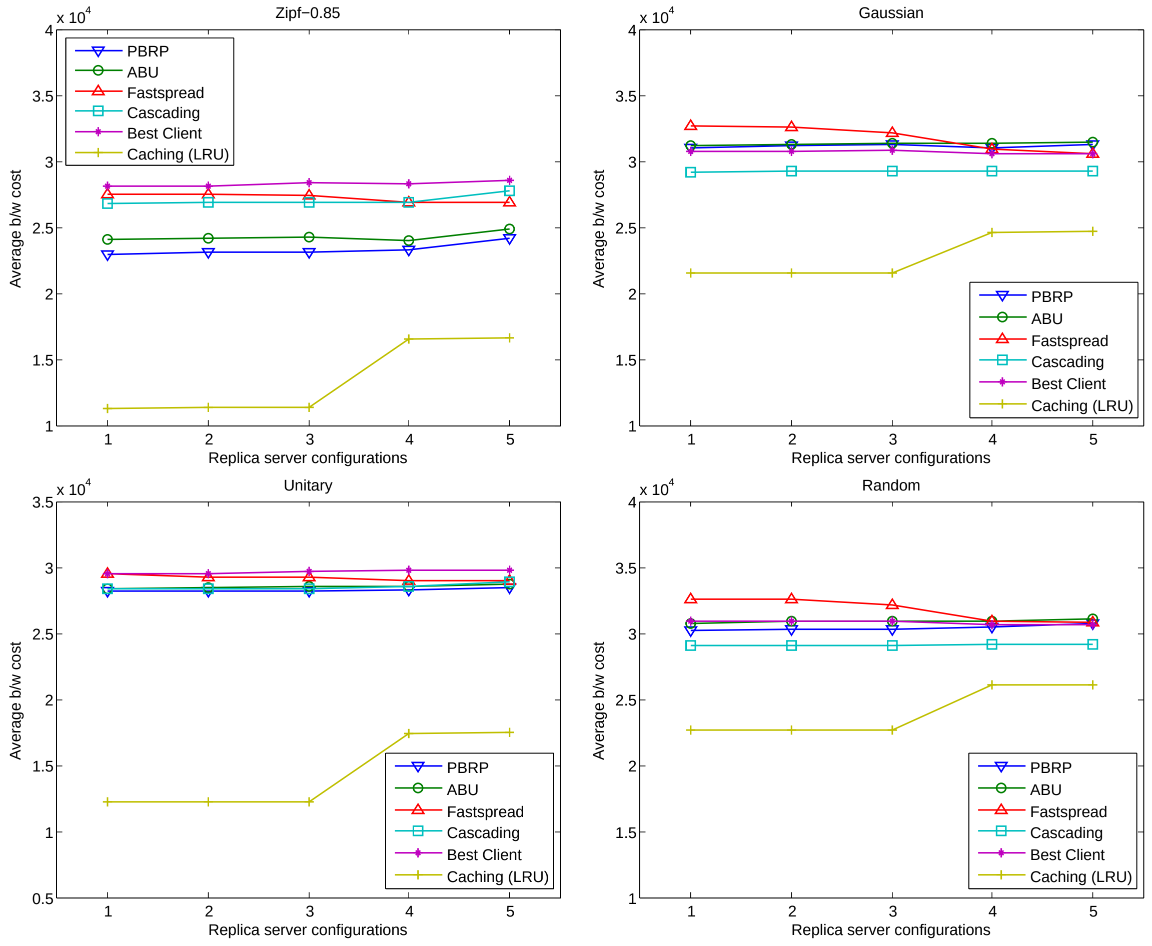


Figure 4.28: Comparison of average bandwidth costs for different access patterns

the Zipf distribution.

Figures 4.28 compares the average bandwidth costs of all replication strategies for the different storage configurations. As the storage size is decreased, the bandwidth costs of all the strategies are generally increased by different amounts due to an increase in read costs. Fast spread shows an exception to this trend where the overall bandwidth usage drops with the decrease in storage capacity. This happens due to a lesser frequency of replica creation resulting in a lower replication cost. This fact is reflected in Figure 4.29 for the Zipf access patterns.

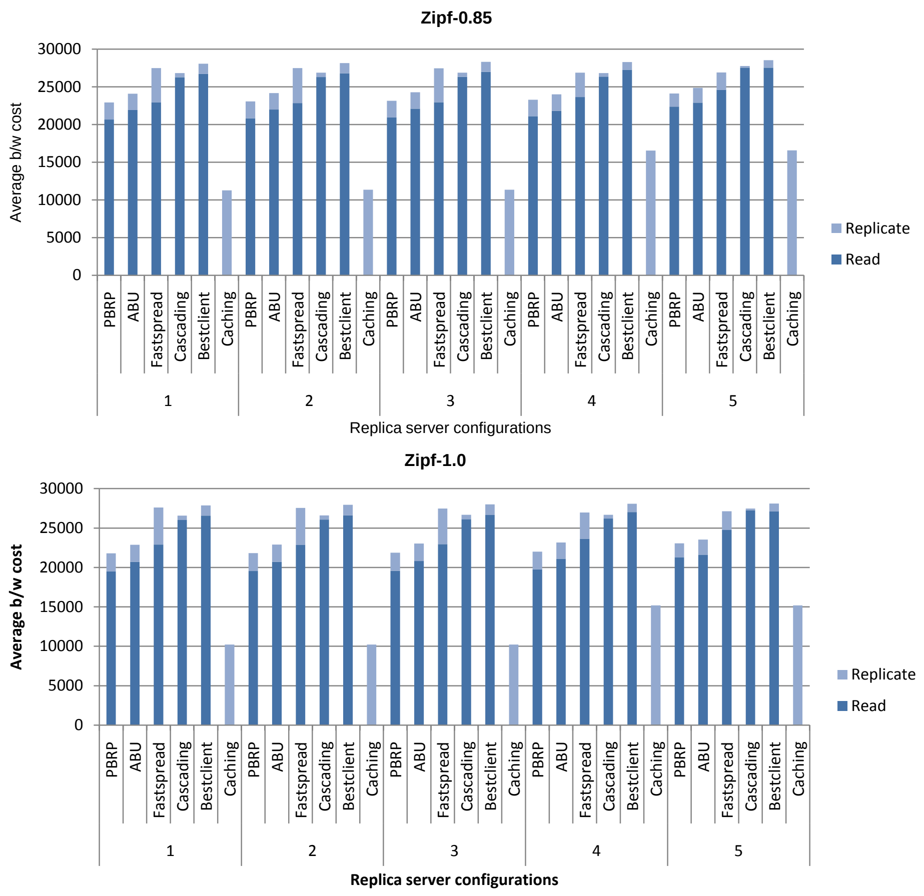


Figure 4.29: Average bandwidth costs (read and replicate) for Zipf access patterns

Figure 4.30 compares the average bandwidth use by PBRP and Caching when the simulation length is increased as mentioned in Section 4.5.1. Caching leads to higher bandwidth use compared to PBRP except the cases when clients have sufficient storage and data access pattern follows Zipf-1.0 and Unitary distributions.

Figure 4.31 shows the approximate average bandwidth costs for all data access patterns using file sizes of 2 GB and 20 GB. For both cases, the relative performance of the algorithms in terms of bandwidth cost is almost the same as the performance obtained using a file size of 10GB. To assess the relative performance of PBRP, Figure 4.32 compares the approximate average bandwidth costs of PBRP using different

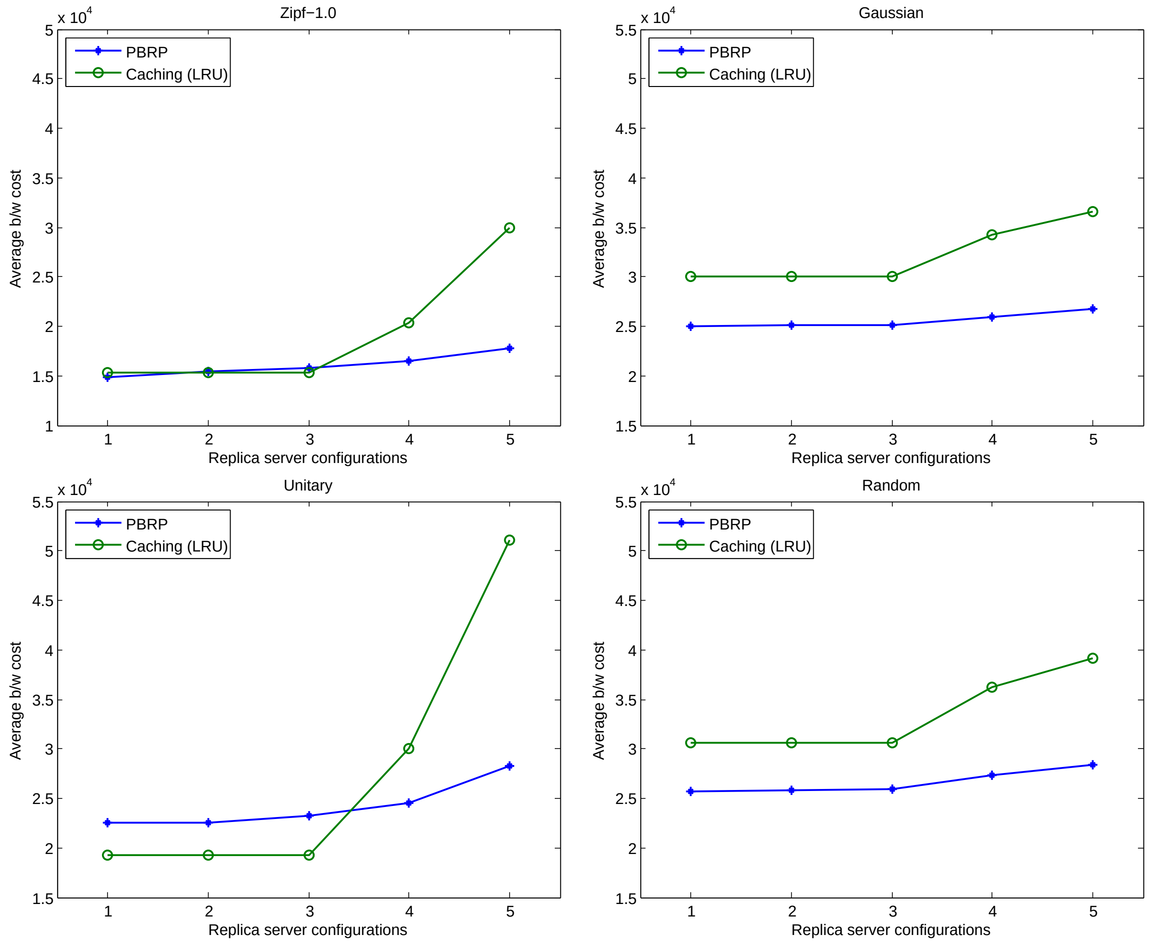
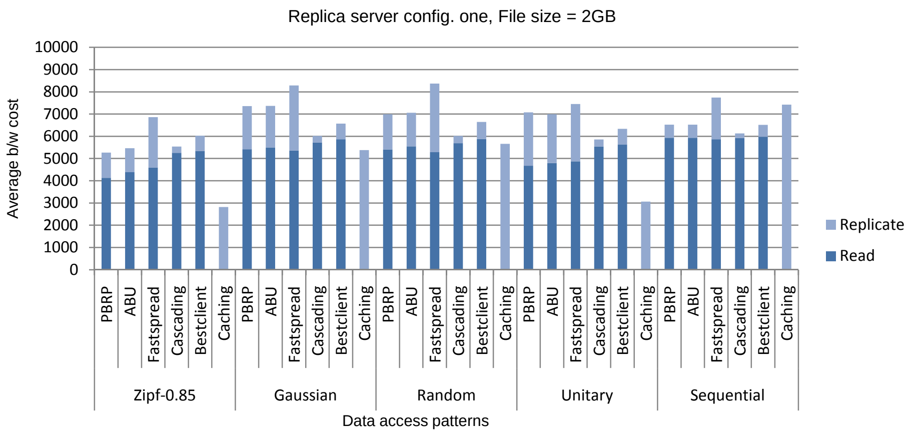


Figure 4.30: Comparison of average bandwidth costs from an increased simulation period for different access patterns

file sizes for all data distributions. As expected, the bandwidth cost using 20 GB files is much higher than the other two cases due to the increased data transfer cost. The bandwidth costs are medium for 10 GB files and low when 2 GB files are used.

I now compare the performance of APBRP and its non-adaptive counterpart in terms of average bandwidth cost when the data access rate is consistently increasing, consistently decreasing, and when it fluctuates.



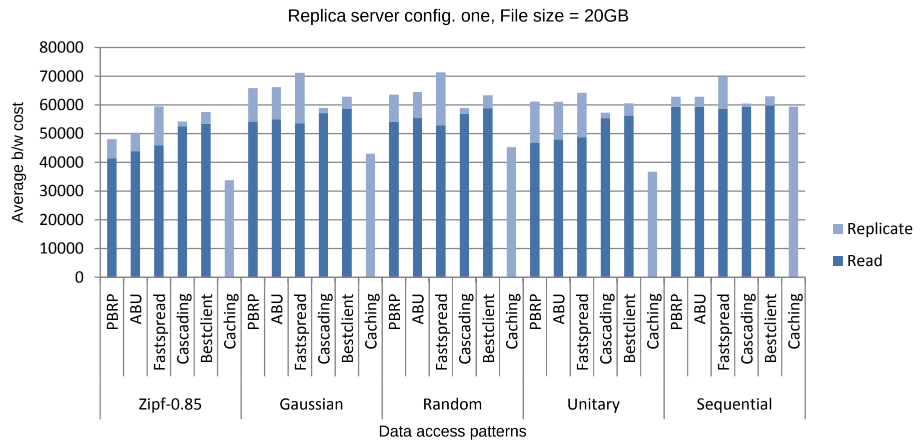


Figure 4.31: Average bandwidth cost by file size, Replica server configuration one

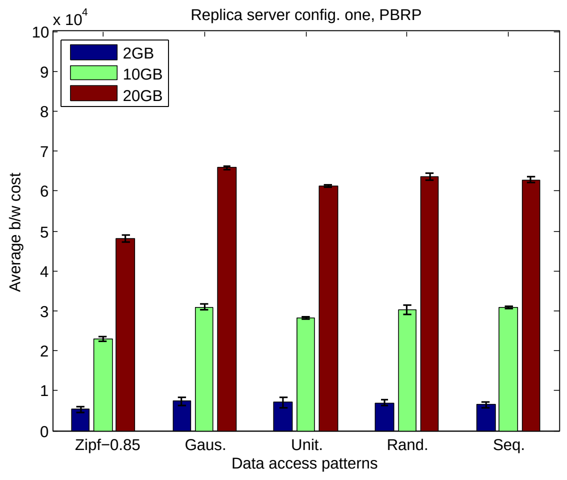


Figure 4.32: Average bandwidth cost: PBRP by file size, Replica server configuration one

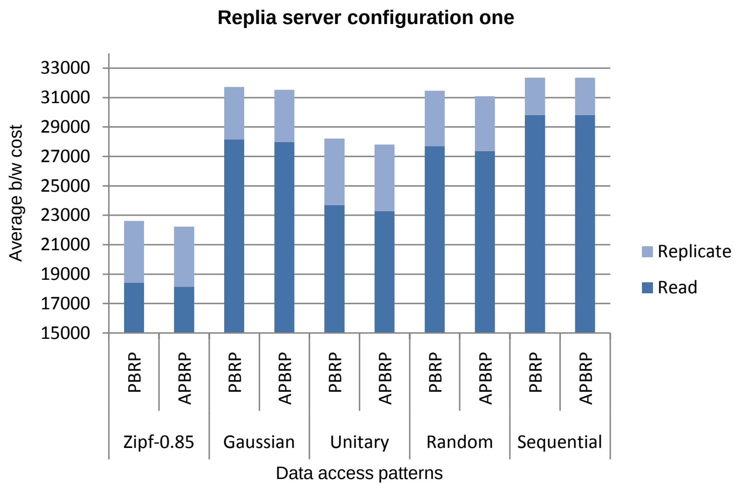


Figure 4.33: Average bandwidth costs using resource configuration one when access rate fluctuates

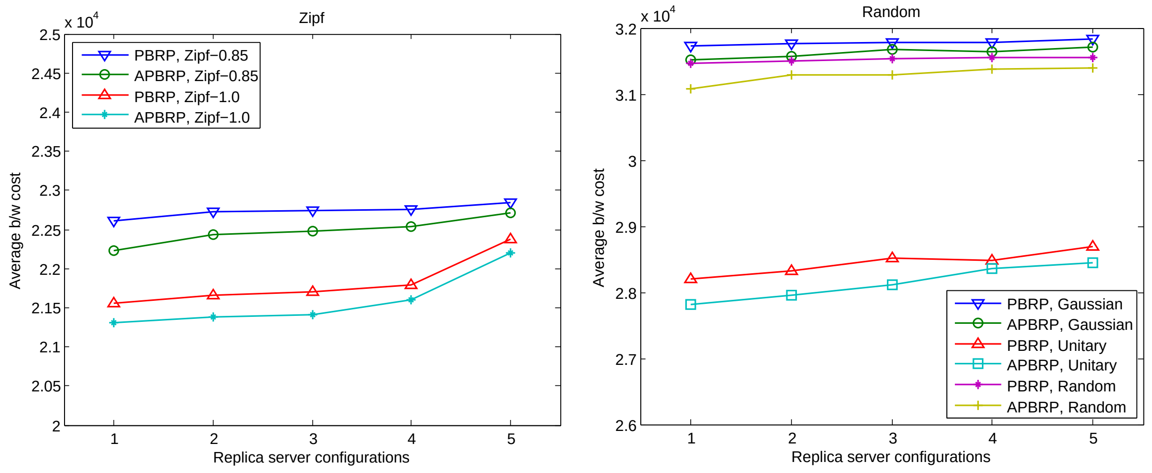


Figure 4.34: Comparison of average bandwidth costs for different access patterns when the access rate fluctuates

Average bandwidth cost when access rate fluctuates

Figure 4.33 shows the average bandwidth cost of APBRP and PBRP using storage configuration one when the data access arrival rate from the clients fluctuates regularly. APBRP gives better average bandwidth cost between the two placement methods for all cases except the sequential access pattern. APBRP adjusts the threshold value based on the varying access arrival rate which leads to the creation of an increased number of replicas close to the clients compared to PBRP. This decreases the (dominant) read cost and in turn lowers overall bandwidth consumption. Like PBRP, APBRP shows the lowest bandwidth consumption for the Zipf distributions compared to other access patterns.

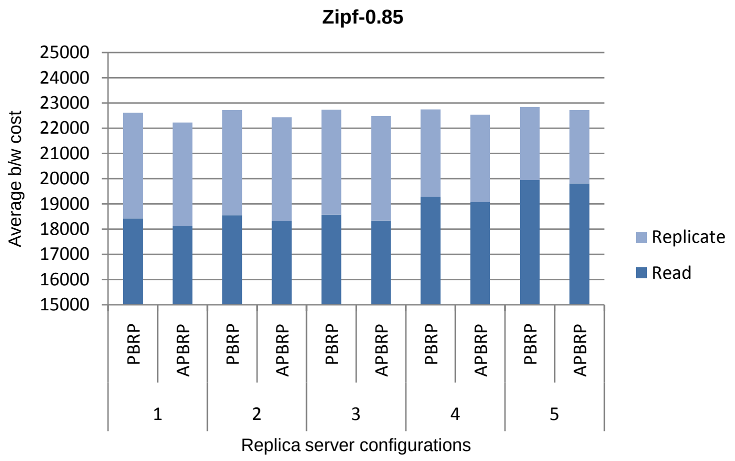


Figure 4.35: Average bandwidth costs (read and replicate) for the Zipf-0.85 access pattern when the access rate fluctuates

Average bandwidth costs for the three different storage configurations are compared in Figure 4.34. With decreasing storage size, the bandwidth costs of both techniques are increased but by different amounts. APBRP shows better bandwidth costs for all storage configurations. However, the benefit of APBRP over PBRP decreases when the storage capacities of the replica servers are reduced as the difference in read costs between the two strategies drops as shown in Figure 4.35.

Average bandwidth cost when the access rate is consistently decreasing

Figure 4.36 shows the bandwidth costs for storage configuration one when the request arrival rate from the clients is decreasing. APBRP shows considerable im-

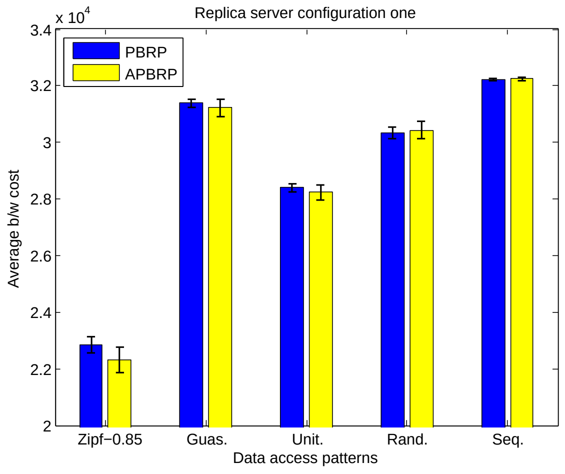


Figure 4.36: Average bandwidth costs using resource configuration one when the access rate is consistently decreasing

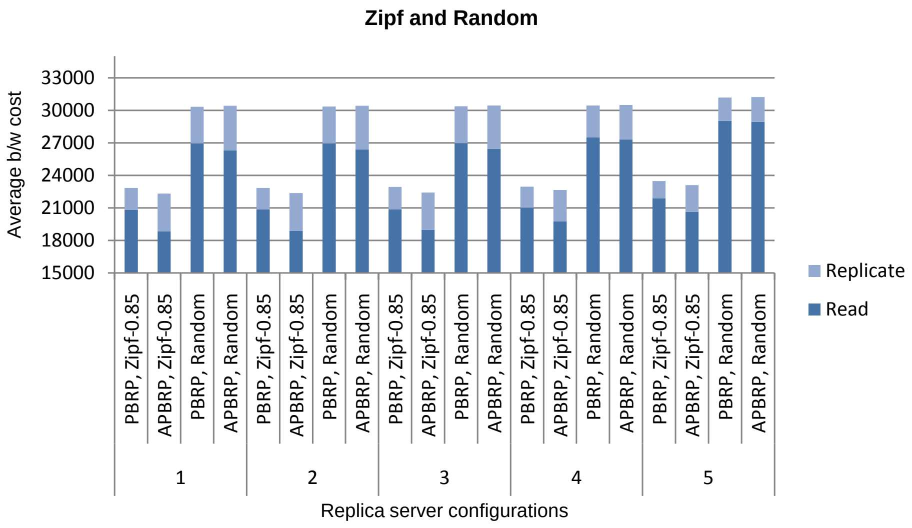


Figure 4.37: Average bandwidth costs (read and replicate) for the Zipf-0.85 and flat random access patterns when the access rate is decreasing

provement in bandwidth savings when the access pattern contains temporal locality. When the access arrival rate decreases consistently, APBRP also decreases the threshold value accordingly which maintains an increased number of replicas compared to PBRP resulting in reduced read cost. This in turn decreases the overall bandwidth

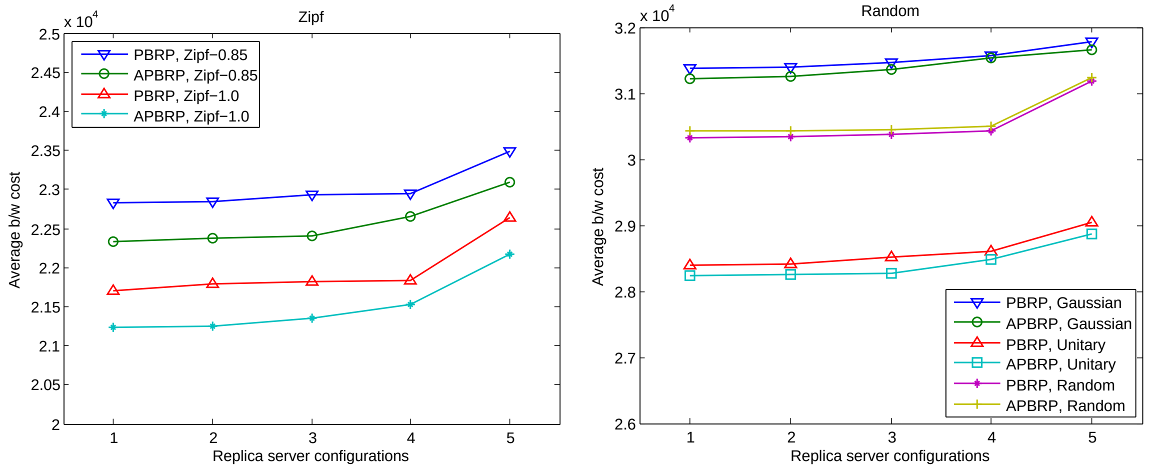


Figure 4.38: Comparison of average bandwidth costs when the access rate is consistently decreasing

costs even though replication cost increases (but by a lesser amount) due to the creation of additional replicas. Figure 4.37 shows the read and replicate components of bandwidth costs when access patterns contain a degree of temporal locality and randomness as well. The overall bandwidth costs for the flat random and sequential distributions are very similar. The benefit of APBRP over PBRP is insignificant when the data access pattern follows the Gaussian and Unitary distributions.

Figure 4.38 shows the average bandwidth costs of both strategies for all storage configurations. APBRP has less bandwidth cost compared to PBRP for the Zipf-0.85, Gaussian, and Unitary distributions. On the other hand, the random access pattern leads to more bandwidth consumption by APBRP for all configurations though the difference is not terribly significant. As the storage size is decreased, the bandwidth costs of both strategies are increased proportionally.

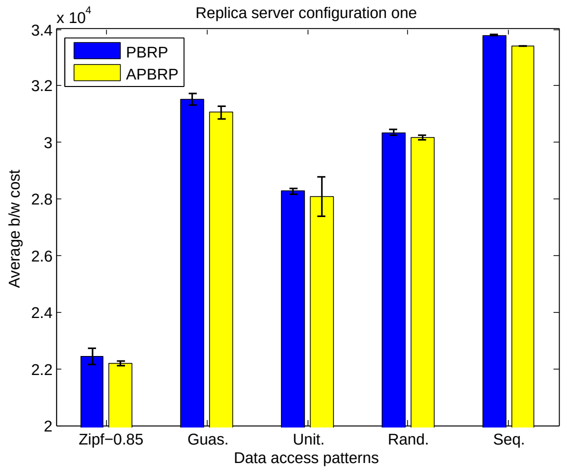
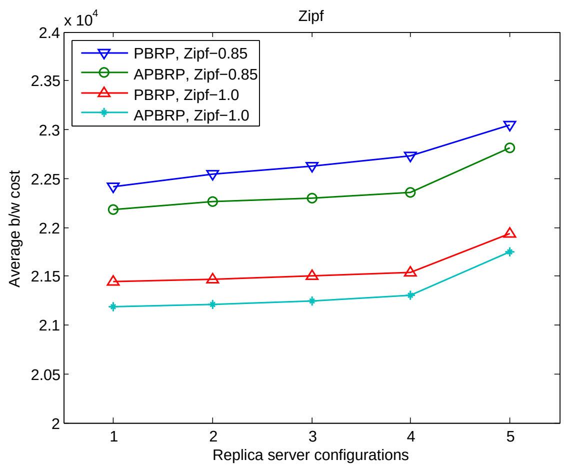


Figure 4.39: Average bandwidth costs using resource configuration one when the access rate is increasing consistently

Average bandwidth cost when the access rate is consistently increasing

Figure 4.39 shows the bandwidth costs for configuration one when the access arrival rate from the clients consistently increases. When the access arrival rate consistently increases, PBRP experiences a drastic increase in the number of replicas. This in turn results in consumption of extra bandwidth for unnecessary data movement. Under the same conditions, APBRP adjusts the threshold value to reduce the number of replicas and thus saves significant link bandwidth consumption which is evident from Figure 4.39. For all data access patterns, APBRP shows lower bandwidth use compared to PBRP.

Figure 4.40 compares the average bandwidth costs of the two placement algorithms for different storage configurations for all data access patterns. With decreasing capacity, the bandwidth costs of both methods are increased but by different amounts. APBRP maintains lower average bandwidth costs for all access patterns for all storage configurations. However, the performance difference is more pronounced when access



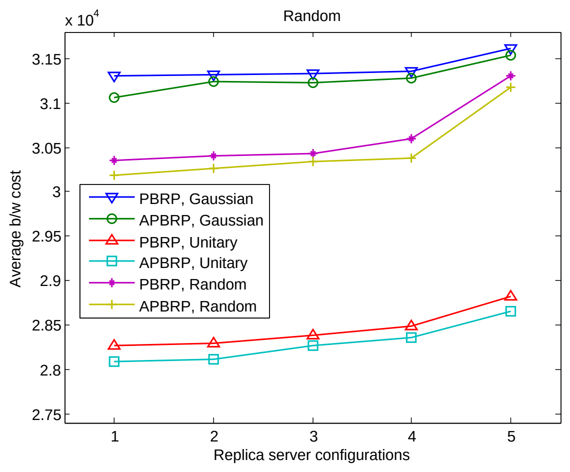
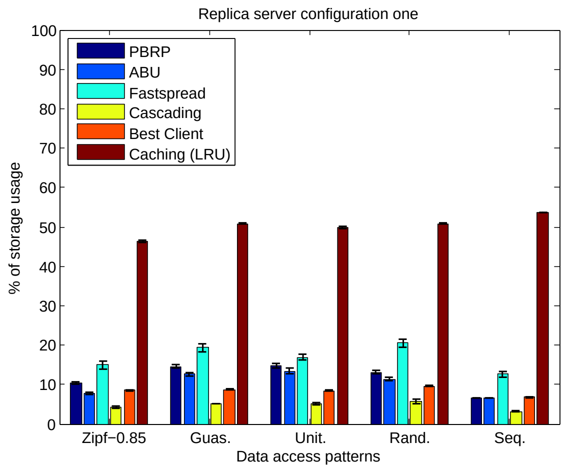


Figure 4.40: Comparison of average bandwidth costs when the access rate is increasing consistently

patterns show randomness.



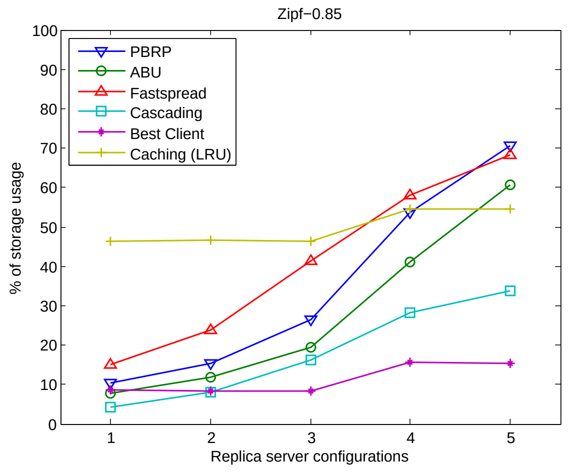


Figure 4.41: Storage use for different replication methods using resource configuration one (left) and comparison of storage costs for the Zipf-0.85 distribution (right)

4.5.3 Storage Use

The storage used by the various non-adaptive placement schemes for storage configuration one are shown in Figure 4.41. For all data access patterns, the storage costs of Caching and Fast Spread are the highest, PBRP and ABU are medium, and Best Client and Cascading are the lowest. The storage used by PBRP is due to its moderate number of replicas created down the hierarchy of the Data Grid which gives its performance benefits in terms of shorter execution times and lower bandwidth consumption. The capacity of the storage resources has a clear impact on the percentage of storage used by different replication strategies. Naturally, reducing the storage size leads to a significant increase in the percentage of available storage used as shown in Figure 4.41 (right) and 4.42. For example, 15.25% of storage is used by PBRP for Zipf in configuration two and it increases to 70.54% in configuration five. One might expect that the reduction of storage size should lead to 100% use of storage but in the simulations some grid sites use the storage completely while others might not use the space at all depending on the data access pattern. It is worthwhile to note that PBRP shows higher storage use compared to other algorithms for access patterns containing temporal locality when the server storage capacities are most constrained (e.g. Configuration 5). Figure 4.43 shows the storage use by PBRP and Caching when the simulation length is increased as mentioned in Section 4.5.1. Caching leads to higher storage use compared to PBRP except the cases when clients have limited storage and data access patterns show temporal locality.

Figure 4.44 shows the approximate storage use for all data access patterns using file sizes of 2 GB and 20 GB. For both cases, the relative performance of the algorithms in terms of storage use is almost the same as the performance obtained using the file size of 10GB. As before, this is due to the fact that the number of files and the relative storage capacity in all three cases are the same. To demonstrate the relative

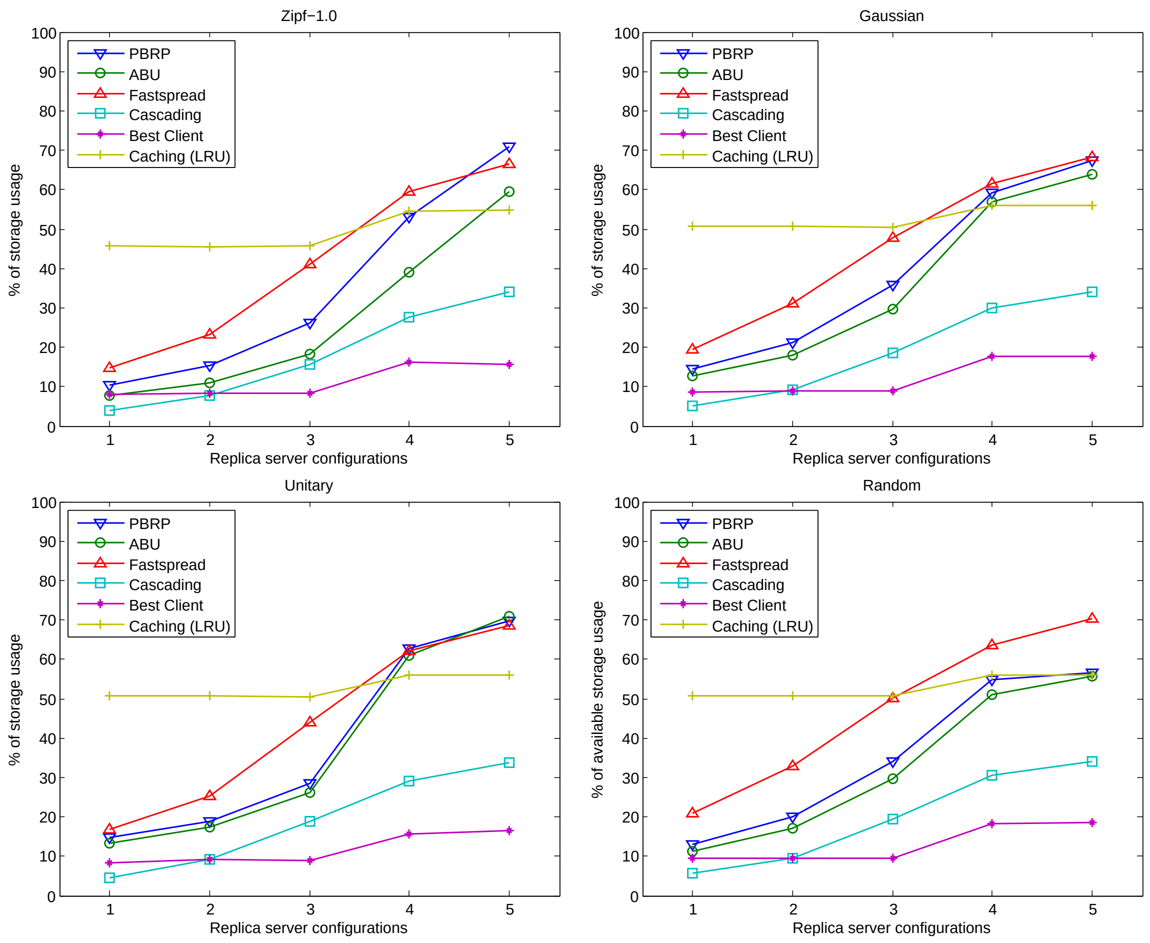


Figure 4.42: Comparison of storage costs for different access patterns

performance of PBRP, Figure 4.45 compares the approximate storage costs of PBRP using different file sizes for all data access patterns. The percentage of storage use does not differ significantly since the average number of replicas created by the system in all three cases is almost the same.

Storage use when the access rate fluctuates

Figure 4.46 shows the storage use of APBRP and PBRP using configuration one when the data access arrival rate from the clients fluctuates regularly. APBRP re-

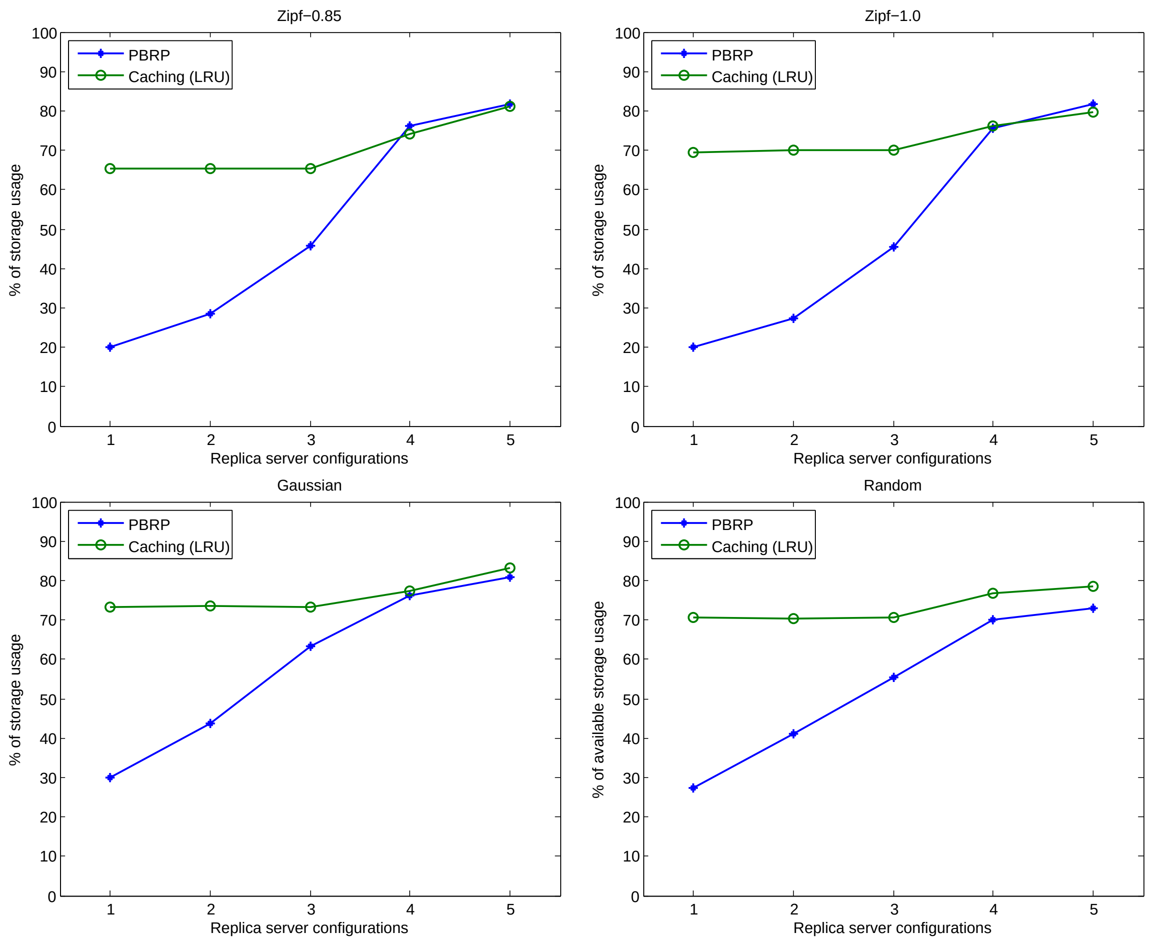
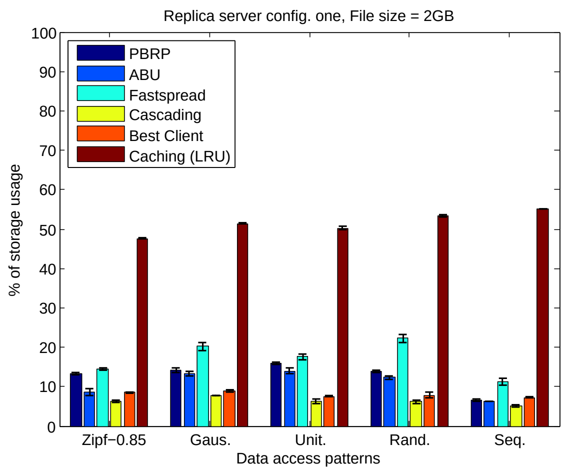


Figure 4.43: Comparison of storage costs from an increased simulation period for different access patterns

quires more storage resources than PBRP for all data access patterns. This is due to the creation of an increased number of replicas in APBRP compared to PBRP but this yields corresponding runtime and bandwidth benefits in most cases.

Figure 4.47 compares the storage use of the two placement algorithms using different storage configurations for all data access patterns. With decreasing storage size, the percentage of storage use for both techniques increases but by different amounts. When the available storage capacity is smaller, the difference in storage use for AP-



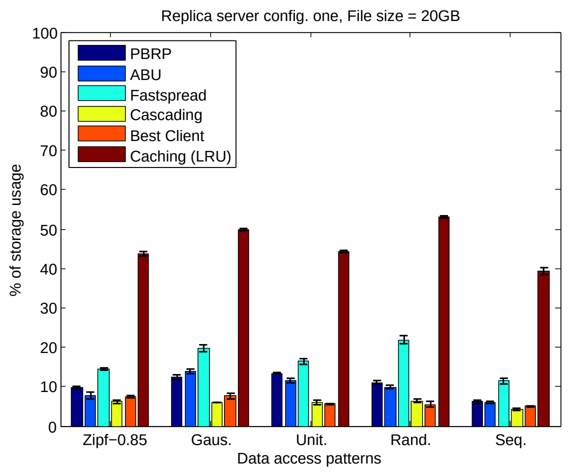


Figure 4.44: Storage use by file size, Replica server configuration one

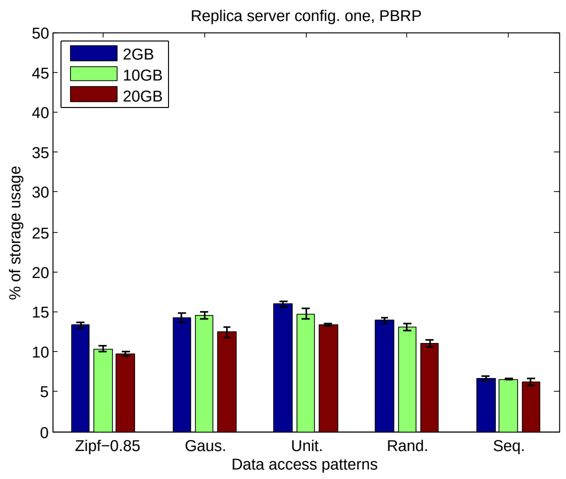


Figure 4.45: Storage use: PBRP by file size, Replica server configuration one

BRP and PBRP increases for most access patterns.

Storage use when the access rate is consistently decreasing

Figure 4.48 (left) shows storage costs for APBRP and PBRP using storage configuration one when the access arrival rate from the clients is decreasing. In this situation, APBRP decreases the threshold value accordingly which maintains an increased number of replicas compared to PBRP. This in turn increases the storage use

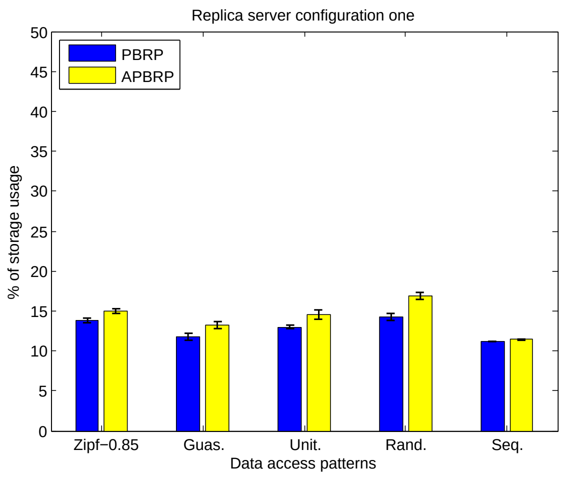


Figure 4.46: Storage costs using resource configuration one when the access rate fluctuates

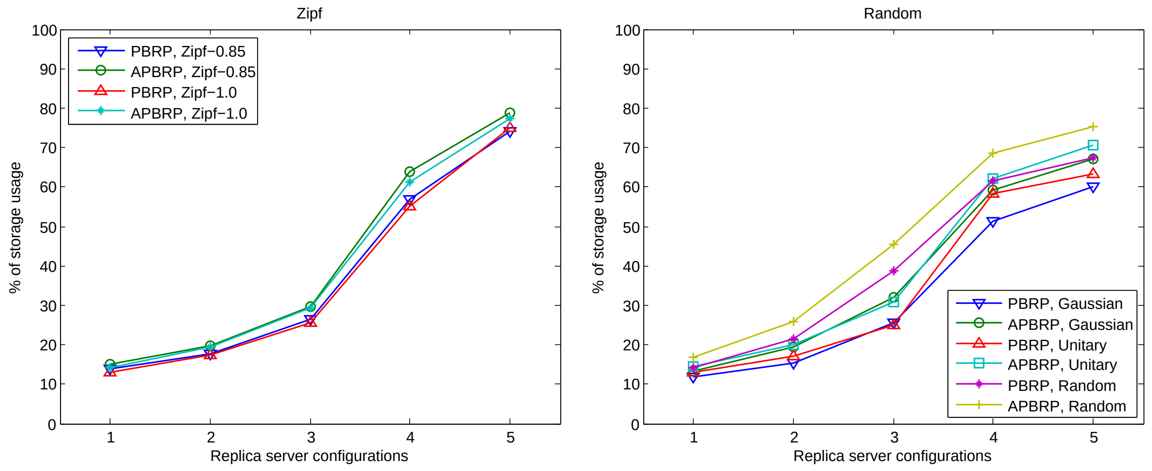
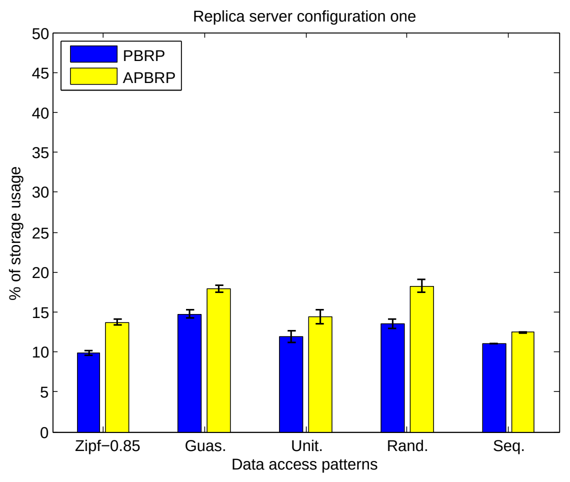


Figure 4.47: Comparison of storage costs for different access patterns when the access rate fluctuates

(with corresponding benefits). Figure 4.49 compares the storage use of the two algorithms for all storage configurations. As the storage size is decreased, the storage costs of both strategies are increased proportionally. However, when the available storage capacity is smaller, the difference in storage use for APBRP and PBRP increases as before for all access patterns.



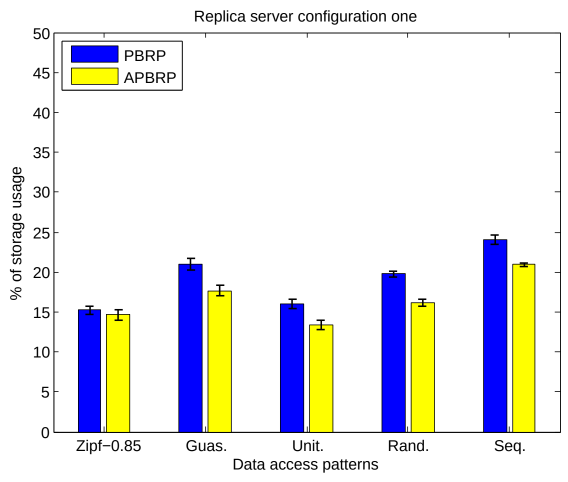
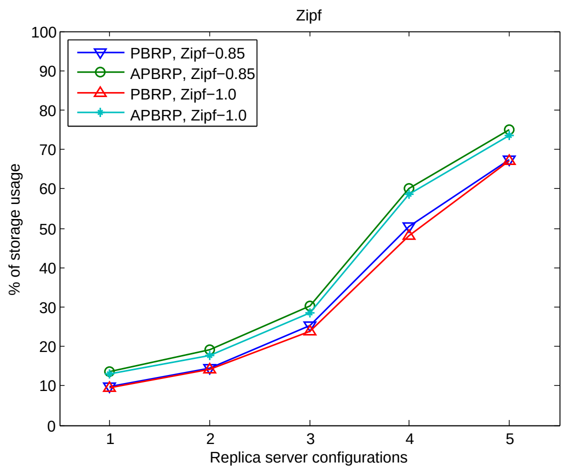


Figure 4.48: Storage costs using resource configuration one when the access rate is consistently decreasing (left) and increasing (right)



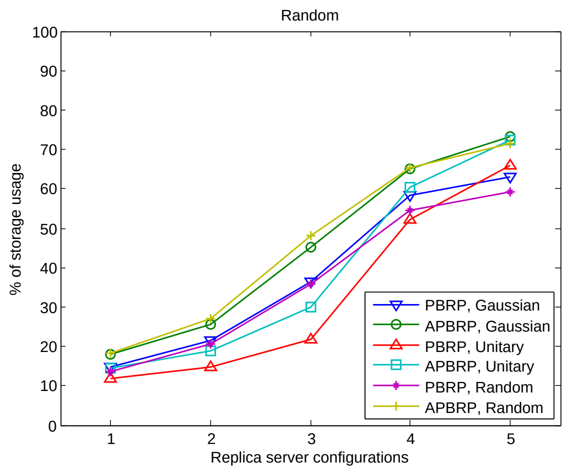


Figure 4.49: Comparison of storage costs for different access patterns when the access rate is consistently decreasing

Storage use when the access rate is consistently increasing

Figure 4.48 (right) shows the percentage of storage use for APBRP and PBRP using replica server configuration one when the data access arrival rate from the clients is consistently increasing. In PBRP, the number of replicas increases drastically with the increase in data access rate whereas APBRP tries to limit the number of replicas to a reasonable level by increasing the threshold value. This in turn results

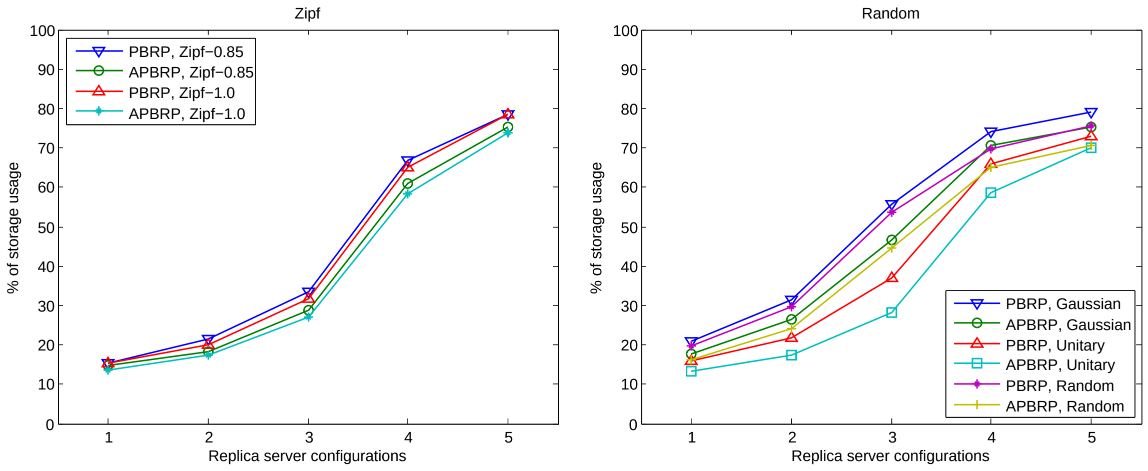


Figure 4.50: Comparison of storage costs as the access rate increases consistently

in higher storage costs for PBRP compared to the adaptive technique. Figure 4.50 compares the storage costs of the two algorithms using all storage configurations for all access patterns. With decreasing capacity, the percentage of storage used by both algorithms increases but by different amounts. APBRP maintains lower storage costs for all access patterns using any storage configuration yet still delivers mostly better execution times and lower bandwidth costs.

4.6 Summary of Centralized Algorithms

The primary goal of dynamic replication is to reduce the job execution time experienced by the end-user (which is impacted by data access latency). At the same time, from the perspective of the whole system, the performance metrics of bandwidth consumption and storage use need to be considered to ensure that the dynamic replications do not incur heavy load on the system.

Initial simulation results show that Caching's performance in terms of job execution time is better than that of the other algorithms for access patterns containing both randomness and temporal locality provided client sites have sufficient storage. However, in cases of limited storage capacity of clients, cached files will get replaced quickly which in turn will increase access time causing an increase in job execution time (refer to Figure 4.12). Relatively longer simulation runs highlight the advantage of PBRP over Caching. PBRP consistently performs better than Caching in terms of job execution time for the random data access patterns (such as Gaussian and flat Random) where data requests from clients occur for a wider range of files. This causes an increased cache misses thereby resulting in high job execution times for Caching compared to PBRP. The performance improvement of PBRP in terms of job time is not terribly significant compared to Caching when clients have sufficient storage (i.e. Configurations 1, 2 and 3) and data access patterns contain temporal locality (i.e. Zipf-0.85 and Zipf-1.0). In fact once the access patterns contain more locality Caching in some cases shows somewhat better job execution time than PBRP though the difference in this case is up to 6%. This is because when the data access patterns follow Zipf, some file requests occur frequently where many others occur rarely. Thus, the clients focus on a smaller range of data files with higher frequencies compared to other access patterns. Caching takes advantage of this temporal locality provided clients have sufficient storage. This leads us to conclude that the performance of Caching in terms of job time will be better than PBRP if there is sufficient temporal locality in the access patterns and the clients have enough storage. In such cases, Caching also shows somewhat better performance in terms of bandwidth consumption. For all other cases except Unitary Random Walk with sufficient storage at the clients, Caching leads to the higher bandwidth use due to higher replication cost compared to PBRP. On the other hand, the naive Best client algorithm consistently performs worse than other algorithms in most situations.

Considering PBRP and the remaining strategies Fast spread, Cascading, and ABU there is no sure best strategy for all possible scenarios. Fast spread consistently performs better than Cascading in terms of job execution time for all access patterns. The job time savings of Fast spread are up to 13% more than that of Cascading. In spite of the advantage Fast spread has, it's overhead in terms of high frequency of replication is obvious. It also has high storage requirement (up to 36% more) and consumes more bandwidth (up to 12%) compared to Cascading.

PBRP consistently performs better in terms of job execution time and bandwidth consumption compared to ABU, Fast Spread, and Cascading for data access patterns that contain temporal locality (i.e. Zipf-0.85 and Zipf-1.0). The advantage of PBRP over other algorithms increases once the access patterns contain more locality and this is the case of interest. The job time savings of PBRP are up to 6% and 25% more than that for ABU and Fast spread, respectively. This is because when the data access patterns follow Zipf, some file requests occur frequently making them "popular" where many others occur rarely. Thus, the clients focus on a smaller range of data files with higher frequencies compared to other access patterns. Consequently, PBRP can identify the popular files effectively based on past data access histories. It also considers the locations of replica servers and clients for determining the replication destinations, so that the replica server storages are properly utilized.

For the random data access patterns, the performance improvement of PBRP in terms of job time is not terribly significant compared to Fast spread and ABU and in fact for the flat random distribution Fast spread shows somewhat better job execution time than PBRP though the difference in this case is marginal (about 2%). This can be attributed to the fact that the overhead in creating additional replicas in PBRP is not offset by the advantage of moving them closer to the clients. ABU does not differ significantly from PBRP in terms of bandwidth consumption in this case. Fast spread on the other hand leads to the higher bandwidth use (up to 8% more) due to higher replication cost compared to PBRP.

Compared to the other algorithms PBRP shows moderate requirements for storage utilization. This moderate use of (relatively cheap) storage space by PBRP makes it a good candidate when data access performance and bandwidth use are of primary concern. Overall, the storage capacities of the replica servers have a major impact on the performance of replication techniques. Increasing the replica server capacity leads to performance improvement for job execution time and average bandwidth cost as more (nearby) replicas are possible.

These results lead us to conclude that if there is sufficient temporal locality in the access patterns then the strategy that would work best is PBRP. With a moderate amount of storage utilization, PBRP lowers job times significantly, while judiciously using network resources. If, however, grid users exhibit total randomness in accessing data, then depending on what is more important in the grid scenario, lower access times or lesser bandwidth consumption, a trade-off between PBRP and Fast spread might be warranted. If the chief aim is to elicit faster responses from the system, Fast spread might work better. On the other hand if conserving bandwidth is of top priority, PBRP is a better grid replication strategy in this scenario.

It is clear that the relative performances of the studied replication algorithms are not impacted by the data file size distributions. Furthermore, the files can be chopped into uniform sized blocks to facilitate system deployment. It is thus reasonable to assign the same size to all the files in the simulation experiments.

In PBRP, the threshold value remains constant irrespective of variation in data access arrival rate and the available storage capacities of the replica servers. APBRP addresses this by dynamically changing the threshold value based on the data access rate and storage availability. Simulation results show that APBRP is able to further reduce the job execution time and bandwidth use in most cases when the data access rate regularly fluctuates and decreases at the expense of additional storage cost. PBRP performs as well as APBRP or somewhat better in terms of job execution time and bandwidth use for all access patterns due to a significantly increased number of replicas when the replica servers have sufficient storage and the client request rate is consistently increasing.

In my algorithms, I assume that recently popular files will tend to be accessed more frequently than others in the near future. Violation of this assumption will lead to decreased performance until the next sampling period commences. Also, my algorithms discussed (in this chapter) don't consider any limit on the amount of data a replica server can serve or on the link capacity. Taking these into account will be important for ensuring the usefulness of my algorithms in real Grid systems. Furthermore, scientific collections of data comprise tens to hundreds of millions of files. It will be necessary to determine the feasibility of aggregating access information about files cached at multiple hierarchical levels for real scenarios to ensure efficient implementation.

My Popularity Based Replica Placement (PBRP) algorithm for hierarchical data grids and a performance comparison of PBRP to other dynamic replication methods including ABU, Fast Spread, and Cascading are published in [SGE08]. APBRP and its performance compared to the non-adaptive counterpart (PBRP) are presented in [SGE10a; SGE09].

Chapter 5

Distributed Replica Placement

In grids where operation control is decentralized and resources are under the control of their own local administrative domains, placing the replicas of an object through a centralized algorithm may be impractical. In this chapter, I introduce a family of new highly distributed replica placement algorithms for hierarchical data grids. As with my centralized algorithms, my distributed algorithms exploit data access histories to identify popular files and determines replication locations to improve access performance by minimizing overall replication overhead (access and update) for a given traffic pattern. The replica placement problem is formulated using a dynamic programming method and its solution is obtained for large-scale hierarchical data grids in a distributed fashion.

I also now consider the issue of quality of service (QoS) in the replica placement problem to determine the locations of the replicas to improve system performance and at the same time satisfy the quality requirements both from the user and system perspectives. Each user in the lowest tier of the data grid hierarchy may have some QoS requirements on retrieving requested data. Such a requirement mandates that all requests generated by the user will be serviced by a server within the QoS bound.

From the system perspective, link and workload capacity constraints are added to the replica placement problem while still satisfying the quality requirements specified by the users. Each link in the data grid hierarchy has some capacity constraint on transferring data down the hierarchy. Furthermore, it is assumed that the workload capacity of each replica server is bounded. The goal is to find a replication strategy with the minimal replication cost that limits the workload of each server to its capacity and ensures that none of the communication links is congested. Thus, the novelty of my QoS-aware algorithm comes from the ability to integrate multiple types of QoS both from user and system (workload and link capacity constraints) perspectives. The performance of my algorithms is evaluated with simulation experiments over a wide range of parameters and is compared to a number of other similar existing replica placement algorithms. The results demonstrate how the effectiveness of replica placement is affected by numerous factors such as grid network characteristics (i.e. number of nodes, node and link capacities, traffic patterns, etc.), QoS parameters, and so on.

5.1 Base System Model

As mentioned, the replica placement problem addressed in this thesis focuses on hierarchical data grids. As in the centralized case, such a data grid is modeled as an undirected tree $T = (V,\ E)$, where V is a set of grid sites and E is the set of communication links among the sites. Each link $(u, v) \in E$ is associated with a cost $d(u, v)$ that denotes the communication cost of the link. The communication cost of a path is defined as the sum of the communication cost of the links along the path. As before, it is assumed that all data are initially located at the root, and a data replica can be placed in any node other than the root. All the leaves of the tree are local sites where users can issue requests to access their required data. Each leaf node v has associated non-negative numbers $\{c(v, f)\}$, which are the access frequencies (representing the popularities) of each data file, f, at v during a certain period of time. These data access frequencies contribute to the weighted communication cost for servicing the data requests from the clients. To maintain data consistency, the root node r issues updates to every replica server. The update frequency $\mu(f)$ denotes the number of update operations during a certain time period for the data file f. I assume that update operations must first be performed on the authoritative/original copy stored at the root, where they are then propagated to the replica servers. That is, updates only come from the root and the replica servers act as repositories for data retrieval. The communication involved in this update process is captured as an "update cost". The number of replicas to be placed will be a trade-off between the cost of data access by clients and the cost of data updates from the root.

I use *replication cost*, in part, to evaluate my replication strategy. Let R be a set of replica servers. For the replica servers in R the replication cost, $cost(R)$, is defined as the sum of the *read cost* $RC(R)$ and the *update cost* $UC(R)$:

$$cost(R) = RC(R) + UC(R) \tag{5.1}$$

a) Read cost: Given a set of replica servers, R, the read cost is defined to be the overall cost of accessing data required by the clients. Let $F(v, R)$ denote the lowest ancestor of v which is contained in R, (i.e. the first replica server that is met while going from the client v to r). Again, $F(v, R)$ could be v itself when a replica is placed

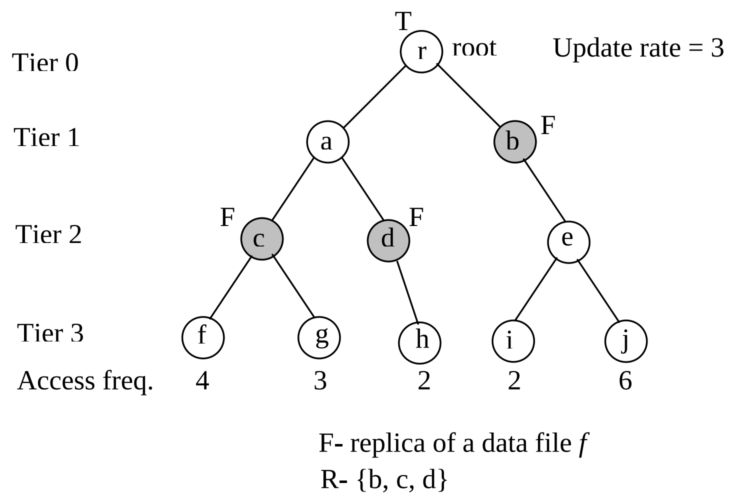


Figure 5.1: An example hierarchical data grid with access and update frequencies

at v, or it could be r when no replica is met on its way to r.

Given the access frequency $c(v, f)$ of a client v for a file f, the read cost for client v to access f is:

$$c(v, f).d(v, F(v, R)) \tag{5.2}$$

The total read cost for all clients in T to access f is therefore:

$$RC(R) = \sum_{v \in T} c(v, f).d(v, F(v, R)) \tag{5.3}$$

I still assume that both reads and updates require one bandwidth unit per data file transfer per hop of the grid network[1]. For example, in Figure 5.1 if the replication strategy R is $\{b, c, d\}$ for file f, then the read cost is $4+3+2+(2\times 2)+(2\times 6) = 25$.
b) Update cost: By using a multicast communication model, the root, r, can multicast data to all replica servers via a shortest path tree (SPT) [Cah98]. The SPT is an induced "update distribution tree" for T, which contains all the replica servers. Multicasting involves sending a *single copy* of the data over the SPT to all replica servers.

[1]This assumption generalizes easily.

Given an update frequency $\mu(f)$ for a file f, the update cost of replicas in R is defined as follows. Let $p(v)$ be the parent of node v in the update distribution tree, and T_v be the subtree rooted at node v. If $T_v \cap R \neq \phi$, the link $(v, p(v))$ contributes to the update multicast. Therefore, the update cost is the sum of the data transfer costs from the links $(v, p(v))$ when $v \neq r$.

$$UC(R) = \sum_{T_v \cap R \neq \phi} \mu(f).d(v, p(v)) \tag{5.4}$$

If we assume that the number of updates issued by the root is 3 for a particular time period then, for example, in Figure 5.1 (where the replication strategy R for file f is $\{b, c, d\}$) the update cost is 6 + 6 + 3 = 15.

The total replication cost of all clients in T to access file f with a set of replicas R can thus be defined as:

$$cost(R) = \sum_{v \in T} c(v, f).d(v, F(v, R)) + \sum_{T_v \cap R \neq \phi} \mu(f).d(v, p(v)) \tag{5.5}$$

Considering Figure 5.1, the total replication cost (read and update) will be 25 + 15 = 40. The problem that needs to be solved is therefore the construction of some R among the grid sites that minimizes this cost under a given traffic pattern (i.e. a recurring pattern of access frequencies from clients for different files).

5.2 Base Distributed Replica Placement Algorithm

I now describe the base form (no QoS support) of my new distributed popularity based replica placement (DPBRP) algorithm for allocating replicas in a hierarchical data grid to minimize the total replication cost. We show that the replica placement problem described can be modeled as a dynamic programming problem and its solu-

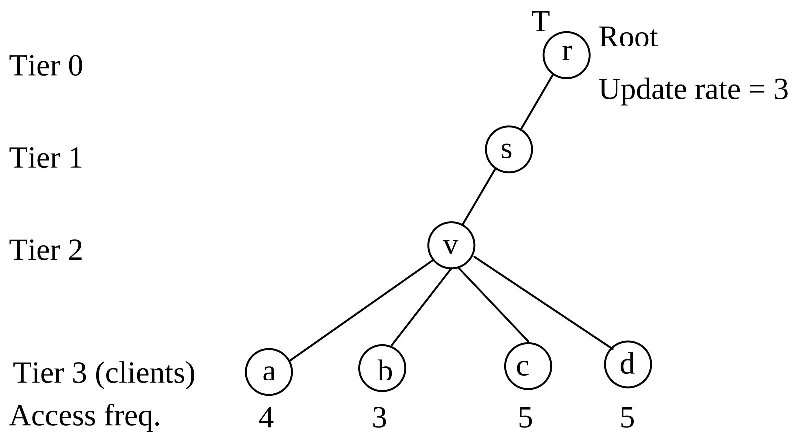


Figure 5.2: Example sub-tree for illustrating cost function

tion can be obtained for large-scale hierarchical data grids in a distributed fashion. Using the replication cost function, it is possible for each node to calculate the cost of creating a local replica and the cost of transferring data from a server higher up in the hierarchy. In DPBRP, a parent node uses the results obtained by its children to assess the cost of creating a local replica versus the cost of transferring data to access a remote replica. This process starts at the leaves and continues till the root is reached. The process of actually placing the replicas begins at the root and ends at the leaves where each node determines (based on the previously calculated results) whether to create a replica locally or not. The details of the algorithm follow.

5.2.1 Cost Function

We begin by defining some terminology. Consider a hierarchical data grid T with root r. If v is a node in T, then we use T_v to denote the subtree rooted at v and $T'_v = T_v - v$, namely the descendants of v. Also, we use $a(v, i)$ to denote the i-th ancestor of v while traversing towards the root of T.

We now define a cost function C. $C(v, i)$ indicates the replication cost contributed by T_v when a replica is placed at $a(v, d)$, $0 \leq d \leq i$. $C(v, 0)$ represents the replication cost for subtree T_v when the replica is placed at v. In this case, the replication cost includes the read cost of all the descendants of v and the update cost for the replica at v. For $1 \leq d \leq i$ when the replica is placed in any of the ancestors of v, $C(v, d)$ represents the replication cost for subtree T_v which includes only the read cost of all the nodes in T_v. Figure 5.2 illustrates an example sub-tree for discussion of the cost function. Assume that both read and update operations require one bandwidth unit per data file transfer per hop of the data grid and the number of updates issued by the root is 3 for a particular time period. If a replica is placed at v, the replication cost for T_v is $C(v, 0) = 17$(read cost) + 6(update cost over 2 hops) = 23. However, if we place a replica at s, the replication cost for T_v is $C(v, 1) = 34$(read cost over 2 hops) = 34.

5.2.2 Bottom-up Computation Phase

Replica placement in DPBRP is done in two phases. Phase one, bottom-up computation, begins at the clients in the lowest tier of the hierarchy and ends at the root. It computes the C functions for every node. Each grid node v calculates the replication cost, $C(v, d)$, for the subtree rooted at it for each value of d when $0 \leq d \leq i$ and $a(v, i) = r$. Thus, node v calculates the cost function for each possible distance up to the root. For each case it also determines the optimal location of the replica, as described below. A node calculates its cost functions only after all its children have done so.

We need to calculate an initial (replication) cost for each node considering each replica distance (d) possibility towards the root as described in Section 5.2.1. The optimal replication cost for that node when $d = 0$ is then calculated by comparing its initial cost with the cost of placing replicas at all of its children nodes (if there is any). For $d \geq 1$, the optimal replication cost is determined by comparing its initial cost for each value of d with the cost of placing replica at the node itself. We start by calculating an initial cost for each client (i.e. leaf) node which then calculates its optimal cost as described above. The initial cost for a non-client (i.e. non-leaf) node is then computed by combining the optimal replication costs of its children. The details of the computation follow.

For a client node, v, an initial cost function, iC, is calculated for each value of d as:

$$iC(v,d) = \begin{cases} UC(v), & \text{if d} = 0 \\ RC(v,d), & \text{if } 1 \leq d \leq i \text{ and } a(v,i) = r \end{cases} \tag{5.6}$$

When $d = 0$, the replication cost does not include a read cost since the replica is at v itself. For $d \geq 1$, the replication cost (as explained earlier in Section 5.2.1) for subtree T_v (in this case only v) includes only the read cost of v. For a non-client node, v, iC, is calculated for each value of d as:

$$iC(v,d) = \begin{cases} \sum_{n \in ch(v)} C(n, d+1) + UC(v), & \text{if d} = 0 \\ \sum_{n \in ch(v)} C(n, d+1), & \text{if } 1 \leq d \leq i \text{ and } a(v,i) = r \end{cases} \tag{5.7}$$

where $ch(v)$ denotes the children of node v. iC represents the initial replication cost for subtree T_v. When $d = 0$ (which means the potential replica is at v itself), the initial cost consists of the replication costs of its children nodes considering the replica one hop away and the update cost from the root to v. For $d \geq 1$, the initial cost includes only the replication costs of its children since the potential replica is not at v.

We then calculate the optimal replication cost, C, and the corresponding replica location in the data grid for v considering each distance possibility, d. We first need to calculate the sum of costs for all children of v when $d = 0$ as:

$$chC(v,0) = \begin{cases} \sum_{n \in ch(v), ch(v) \neq \phi} C(n,0), & v \text{ is not a client} \\ iC(v,0) & v \text{ is a client} \end{cases} \tag{5.8}$$

If v is a non-client node, the above sum includes the replication costs for all of its children considering potential replicas at themselves (i.e. $d = 0$). On the other hand, if v is a client node, the sum includes only the initial replication cost for placing the replica at v itself since it does not have any children.

When $d = 0$, the dynamic programming equation for optimal replication cost for T_v and replica location is then formulated as:

$$C(v,d) = \begin{cases} chC(v,0), & \text{if } iC(v,d) > chC(v,0) \\ iC(v,d), & \text{otherwise} \end{cases} \tag{5.9}$$

$$loc(v,d) = \begin{cases} -1, & \text{if } iC(v,d) > chC(v,0) \\ v, & \text{otherwise} \end{cases} \tag{5.10}$$

This means that if the sum of costs for all children of v is less than the initial cost calculated, it is cheaper to have a replica in each child of v and thus the replica location loc is set to -1 to indicate that the replicas should be placed in the children of v. If this is not true, the replica will be created locally at v and the replication cost will be the initial cost as calculated.

When $1 \leq d \leq i$ and $a(v,i) = r$,

$$C(v,d) = \begin{cases} C(v,0), & \text{if } iC(v,d) > C(v,0) \\ iC(v,d), & \text{otherwise} \end{cases} \tag{5.11}$$

$$loc(v,d) = \begin{cases} loc(v,0), & \text{if } iC(v,d) > C(v,0) \\ a(v,d), & \text{otherwise} \end{cases} \tag{5.12}$$

Here, the cost calculated for $d = 0$ is compared to the cost obtained for each distance possibility, $d \geq 1$, to determine the optimal cost and the replica location. If the cost at $d = 0$, $C(v,0)$, is less than the cost for higher values of d, $C(v,d)$ is set to $C(v,0)$ and the replica location is set to the location given by $loc(v,0)$. Otherwise, the replica will be created higher in the hierarchy based on the value of d and the location will be d-th ancestor of v. The pseudo code for bottom-up computation of replication cost for a client (leaf) and a non-client (middle-tier) node is shown in Algorithms 5.1 and 5.2, respectively. The computation is triggered upon receiving a start message by a client node at the beginning of same sampling interval. The computation in a middle-tier node only starts upon receiving the costs from its children.

Theorem 5.2.1 Consider a hierarchical data grid T with r as the root, $T'_r = T_r - r$, namely the descendants of r, and an overall cost function C. There exists an optimal replica set R in T'_r that minimizes C for a given traffic pattern.

Proof We show the correctness of Equations 5.9 - 5.12 to illustrate the correctness of the theorem. We consider two different types of grid nodes: leaf nodes and non-leaf nodes.

Algorithm 5.1 Compute-Replication-Cost-Client(v)

```
1: Receive START message
2: $C_v[0] \leftarrow UC_v$
3: $loc_v[0] \leftarrow v$
4: for $d \leftarrow 1$ to $distanceToRoot$ do
5:     if $RC_v[d] < C_v[0]$ then
6:         $C_v[d] \leftarrow RC_v[d]$
7:         $loc_v[d] \leftarrow a(v, d)$
8:     else
9:         $C_v[d] \leftarrow C_v[0]$
10:        $loc_v[d] \leftarrow loc_v[0]$
11:    end if
12: end for
13: Send $C_v$ to $parent$
```

Algorithm 5.2 Compute-Replication-Cost-Non-Client(v)

```
1: Receive $C_{ch}$ from a child $ch$ of $v$ // $C_{ch}$ refers to the cost for child $ch$ for each
   distance value towards the root
2: if $C_{ch}$ of all children of $v$ received then
3:     $iC_v[0] \leftarrow sum(C_{ch}[1]) + UC_v$
4:     $chC_v[0] \leftarrow sum(C_{ch}[0])$
5:     if $chC_v[0] < iC_v[0]$ then
6:         $C_v[0] \leftarrow chC_v[0]$
7:         $loc_v[0] \leftarrow$ -1
8:     else
9:         $C_v[0] \leftarrow iC_v[0]$
10:        $loc_v[0] \leftarrow v$
11:    end if
12:    for $d \leftarrow 1$ to $distanceToRoot$ do
13:        $iC_v[d] \leftarrow sum(C_{ch}[d + 1])$
14:        if $C_v[0] < iC_v[d]$ then
15:            $C_v[d] \leftarrow C_v[0]$
16:            $loc_v[d] \leftarrow loc_v[0]$
17:        else
18:            $C_v[d] \leftarrow iC_v[d]$
19:            $loc_v[d] \leftarrow a(v, d)$
20:        end if
21:    end for
22:    Send $C_v$ to $parent$
23: end if
```

For a leaf node when $d = 0$, T contains only a single node which is the root. Since there is no cost for updating replicas and also no read cost in this case, the optimality of the algorithm is trivially true.

For a non-zero distance between the client (a leaf node) and the root, i.e. when $d \geq 1$, the cost calculated for $d = 0$ (replica is in the client itself) is compared to the cost obtained for each replica distance possibility, $d \geq 1$, to determine the optimal cost and replica location. For a client v, if the cost while $d = 0$ (update cost for the replica in v) is less than the cost for $d \geq 1$ (read cost of v considering replica at any ancestor $a(v, d)$) it is cheaper to use the replica from v. Otherwise, the replica is created at $a(v, d)$ higher up in the hierarchy based on the value of d.

To complete the proof, it remains to show that the algorithm produces optimal results for non-leaf nodes in the grid. Recall that a grid node only starts to calculate the cost functions after all its children have completed their calculations. We showed that the cost calculated for a leaf node is optimal for all replica distance possibilities. So, the non-leaf nodes one up from the bottom of the grid hierarchy have children (leaf nodes) whose computed replication costs are optimal. More generally, we can assume that the cost for a subtree rooted at a child of a non-leaf node is optimal for all possible replica distances. We need to show that that the algorithm optimally allocates replicas in the tree rooted at any non-leaf node, v, for all distance values. When $d = 0$, we find from Equations 5.9 and 5.10 that the cost for the subtree rooted at the non-leaf is the minimum of the cases where:

1. a replica is placed at v. The cost is the sum of the costs of the subtrees that are rooted at the children of v with $d = 1$ and the update cost at v.

2. no replica is placed at v. The cost is the sum of the costs of the subtrees that are rooted at the children of v with $d = 0$.

The minimum cost thus obtained is optimal for any node v. ∎

5.2.3 Top-down Replica Placement Phase

In the second phase, DPBRP places replicas from the top to the bottom of the grid. The process begins at the root and ends at the leaves with each node independently determining whether a replica should be created locally or not. From calculations in the bottom-up computation phase, the root node contains the optimal replication cost $C(r, 0)$ for the entire data grid and the location of the replica $loc(r, 0)$. The value of $loc(r, 0)$ can be either r (the root itself) or -1 (children nodes at tier-1). The value of $loc(r, 0)$ being r indicates that the replica is zero distance away from the root. So, the root sends a message, either $distance = 0$ or $distance = -1$ to its children. A child node, v, which receives such a *distance* message increases the value of *distance* by one and then checks $loc(v, distance)$. If the value of $loc(v, distance)$ is v, a replica is placed locally and v sets $distance = 0$ and sends the *distance* message to its children . If the value of $loc(v, distance)$ is -1 v sets $distance = -1$ before sending it to its children. Finally, if the value of $loc(v, distance)$ is $a(v, distance)$ v sends the *distance* message to its children without modifying the value. When the message *distance* is received by all clients at the lowest tier of the hierarchy, the replica placement process ends. The pseudo code for replica placement for root, a non-client (middle-tier), and a client (leaf) node is shown in Algorithms 5.3, 5.4 and 5.5, respectively.

Algorithm 5.3 Place-Replica-Root(r)

```
1: Receive $C_{ch}$ from a child $ch$ of $v$
2: if $C_{ch}$ of all children of $v$ received then
3:    $iC_r[0] \leftarrow sum(C_{ch}[1]) + UC_v$
4:    $chC_r[0] \leftarrow sum(C_{ch}[0])$
5:    if $chC_r[0] < iC_r[0]$ then
6:       $C_r[0] \leftarrow chC_v[0]$
7:       $loc_r[0] \leftarrow$ -1
8:    else
9:       $C_r[0] \leftarrow iC_r[0]$
10:      $loc_r[0] \leftarrow r$
11:   end if
12:   if $loc_r[0] = r$ then
13:      $distance \leftarrow 0$
14:   else if $loc_r[0] = -1$ then
15:      $distance \leftarrow$ -1
16:   end if
17:   Send $distance$ to all children of $r$
18: end if
```

Algorithm 5.4 Place-Replica-Non-Client(v)

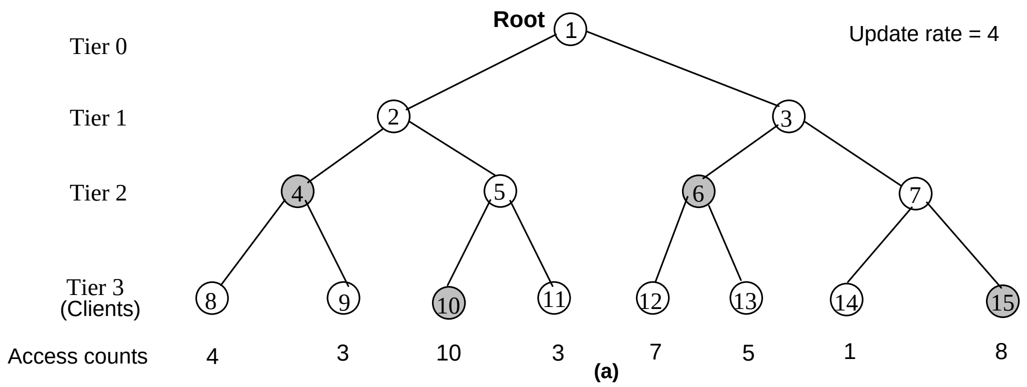
```
1: Receive $distance$ message from parent
2: $distance \leftarrow distance + 1$
3: if $loc_v[distance] = v$ then
4:    $distance \leftarrow 0$
5:    copy a replica at node $v$
6: else if $loc_v[0] = -1$ then
7:    $distance \leftarrow$ -1
8: end if
9: Send $distance$ to all children of $v$
```

5.2.4 Placement Example

Figure 5.3(a) shows an example data grid with 15 nodes. The access counts for a given data file are shown in the client-tier of the hierarchy and the number of updates issued by the root is assumed to be 4 for the time period considered. We continue to assume that both read and update operations require one bandwidth unit per data file

Algorithm 5.5 Place-Replica-Client(v)

```
1: Receive $distance$ message from parent
2: $distance \leftarrow distance + 1$
3: if $loc_v[distance] = v$ then
4:     copy a replica at node (leaf) $v$
5: end if
```



(a)

Grid sites (v)				{cost(v,d), loc(v,d)}			
Tier 3	Tier 2	Tier 1	Tier 0	d = 0	d = 1	d = 2	d = 3
8				(12, 8)	(4, 4)	(8, 2)	(12, 1)
	4			(15, 4)	(14, 2)	(15, 4)	
9				(12, 9)	(3, 4)	(6, 2)	(9, 1)
		2		(36, 2)	(36, 1)		
10				(12, 10)	(10, 5)	(12, 10)	(12, 10)
	5			(21, 5)	(18, 2)	(21, 1)	
11				(12, 11)	(3, 5)	(6, 5)	(9, 1)
			1	(71, 1)			
12				(12, 12)	(7, 6)	(12, 12)	(12, 12)
	6			(20, 6)	(20, 6)	(20, 6)	
13				(12, 13)	(5, 6)	(10, 3)	(12, 13)
		3		(37, -1)	(35, 1)		
14				(12, 14)	(1, 7)	(2, 3)	(3, 1)
	7			(17, 7)	(14, 3)	(15, 1)	
15				(12, 15)	(8, 7)	(12, 15)	(12, 15)

(b)

Figure 5.3: (a) A hierarchical data grid with client access counts, (b) Calculation of replication costs and replica locations for the example data grid

transfer per hop of the grid network. Figure 5.3(b) shows the calculation of *cost* and *loc* functions for each grid node considering all possible distance for the replication location. The calculation starts from the client tier (tier 3) of the hierarchy and ends at the root. Each node sends its *cost* and *loc* values to its parent which receives these values from its children and then calculates its own *cost* and *loc* functions.

In the top-down replica placement process, the root determines that the optimal replication cost for the entire data grid is 71 and sets *distance* to zero before sending messages to its children. The placement process is then carried out as described in Section 5.2.3. For example, node 2 receives $distance = 0$ and increments it by one and checks the entry where $d = 1$. The value of *loc* is 1, therefore, it sends distance to all its children. Node 4 receives $distance = 1$ and increments it by one and checks the entry where $d = 2$. The value of *loc* is 4 which is the node itself. So, node 4 copies a replica (marked in grey in Figure 5.3(a)) and sets $distance = 0$ before sending the message to its children. Similarly, a replica is placed in nodes 6, 10, and 15 in tier 2 and tier 3 based on the *distance* and *loc* values. The total resulting replication cost for the replication strategy (4,6,10,15) is 31(read cost) + 40 (update cost) = 71 which is the same as the cost calculated at the root node in Figure 5.3(b).

5.2.5 Computational Complexity

I now analyze the computational complexity of my algorithm by focusing on the bottom-up computation of the cost function since this dominates the total computation time. For each node v in the data grid we need to compute its $C(v, d)$ for the subtree rooted at it for each value of d, $0 \leq d \leq i$, and $a(v, i) = r$. Node v computes the cost function for each replica distance possibility d by combining the results from all its children. Node v's computation consists of $(i + 1)$ sums of $|ch(v)|$ elements where i is the distance of v from the root, r, and $|ch(v)|$ denotes node v's number of children. Thus, the number of computations for node v is $(i+1).|ch(v)|$. The overall number of computations for all nodes in the grid is:

$$\sum_{v \in V} (i_v + 1).|ch(v)|.$$

where i_v denotes the distance of v from the root.

If the total number of grid nodes, $|V| = N$, it is clear that $i_v \leq N - 1$ for all v. So, we have

$$\sum_{v \in V} (i_v + 1).|ch(v)| \leq N. \sum_{v \in V} |ch(v)| = N(N-1)$$

The above equality holds because the total number of children in the data grid over all the nodes is $N - 1$, since each node, except the root, has a parent. Therefore, the total cost of computing C functions for all nodes is $O(N^2)$. Now, as a special case, if the data grid structure is a balanced binary tree, we get $i_v \leq log_2 N$ for all v. Hence, the overall complexity becomes $O(N log_2 N)$.

As mentioned, the computation of replication cost and location begins at the clients in the lowest tier of the hierarchy and ends at the root. This computation is done concurrently on individual client nodes and is sent to the parent nodes. A parent node calculates its cost functions once the cost information from all of its children are received.

5.3 Performance Evaluation of the Base Distributed Algorithm

To determine the effectiveness of the DPBRP algorithm two different scenarios were considered: when the data request arrival rate remains constant, and when it regularly fluctuates, as described earlier. I compared DPBRP with PBRP (non-adaptive) when the access rate was constant for the different server configurations. For the other case, I compared DPBRP, PBRP, and its adaptive counterpart, APBRP. As it is already shown that the file size distribution does not impact the relative performances of the replication algorithms, the remaining experiments use a file size of 10GB only.

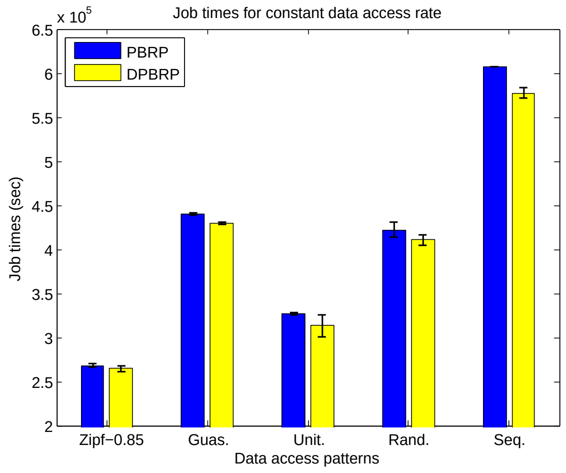


Figure 5.4: Execution time for DPBRP and PBRP, Configuration one, Constant rate

5.3.1 Job execution time

We begin by considering DPBRP's performance in terms of job execution time, the primary consideration from the perspective of the data consumer.

Constant data access rate

Figure 5.4 compares the job execution times for the DPBRP and PBRP algorithms using storage configuration one (refer to Figure 4.8) when the access rates from clients remain constant. In most cases, DPBRP shows somewhat shorter execution times than PBRP. This is because the replica locations selected by DPBRP

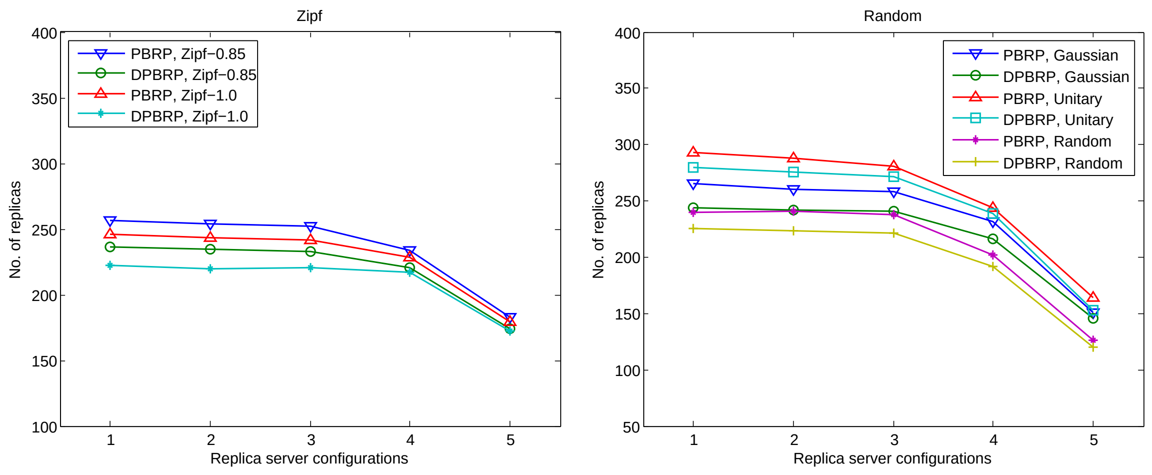


Figure 5.5: No. of replicas for all storage configurations, Constant rate

reduce the data access latency experienced which in turn reduces the overall job execution time. It is noteworthy that DPBRP also requires fewer replicas than PBRP. As a representative example, using the Zipf-0.85 distribution and storage configuration one, the approximate number of replicas created by DPBRP and PBRP are 230 and 265, respectively. Figure 5.5 shows the number of replicas created for different storage configurations for all data access patterns. The storage capacity of the

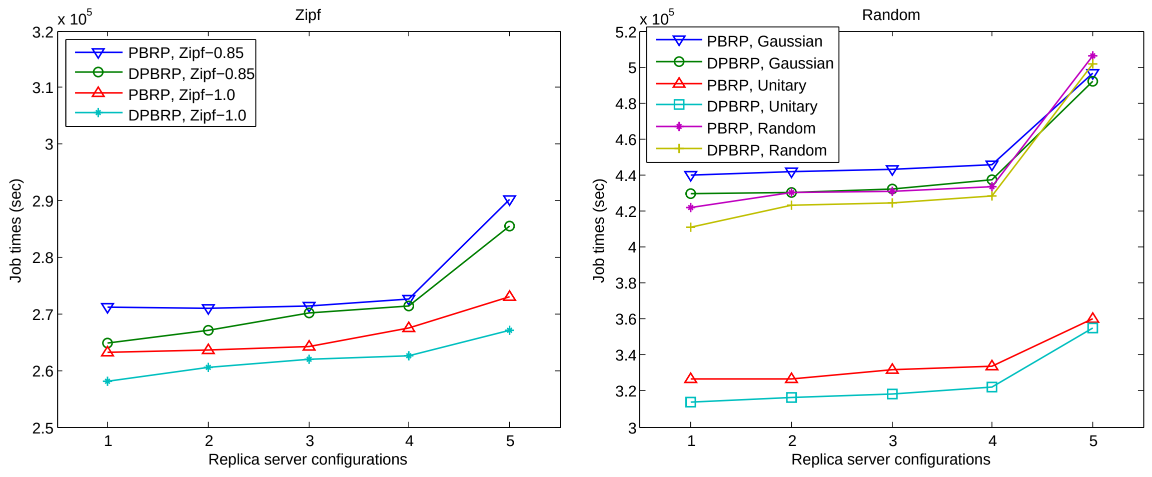


Figure 5.6: Execution time for all access patterns and configurations, Constant rate

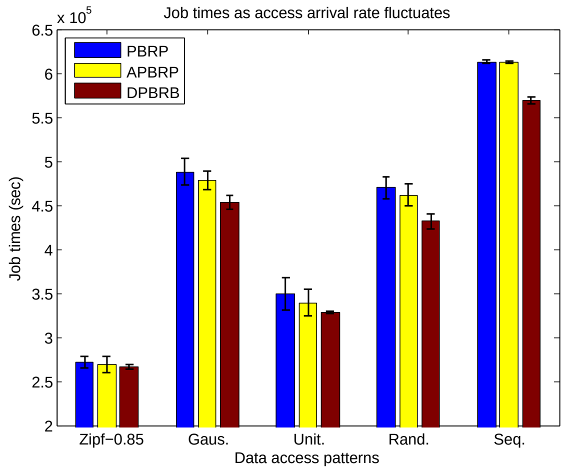


Figure 5.7: Execution time, Configuration one, Fluctuating rate

replica servers has a noticeable impact on the performance of the placement algorithms. With decreasing capacity, the execution times of all methods are increased but by different amounts. Figure 5.6 shows the execution times for all distributions for all five storage configurations. DPBRP shows shorter run times for all the cases.

Fluctuating data access rate

Figure 5.7 shows the times for DPBRP, APBRP, and PBRP using configuration one when the request arrival rate from the clients fluctuates. DPBRP shows shorter execution times for all access patterns. Among the other two algorithms APBRP displays shorter execution times in most cases. This is because APBRP adjusts the threshold value based on the varying request arrival rate which leads to the creation of more replicas which decreases the run time. Using the Zipf-0.85 distribution and storage configuration one, the approximate number of replicas created by APBRP and PBRP are 275 and 240 respectively. Figure 5.8 shows the number of replicas created for different storage configurations for all Zipf-0.85 and Random data access

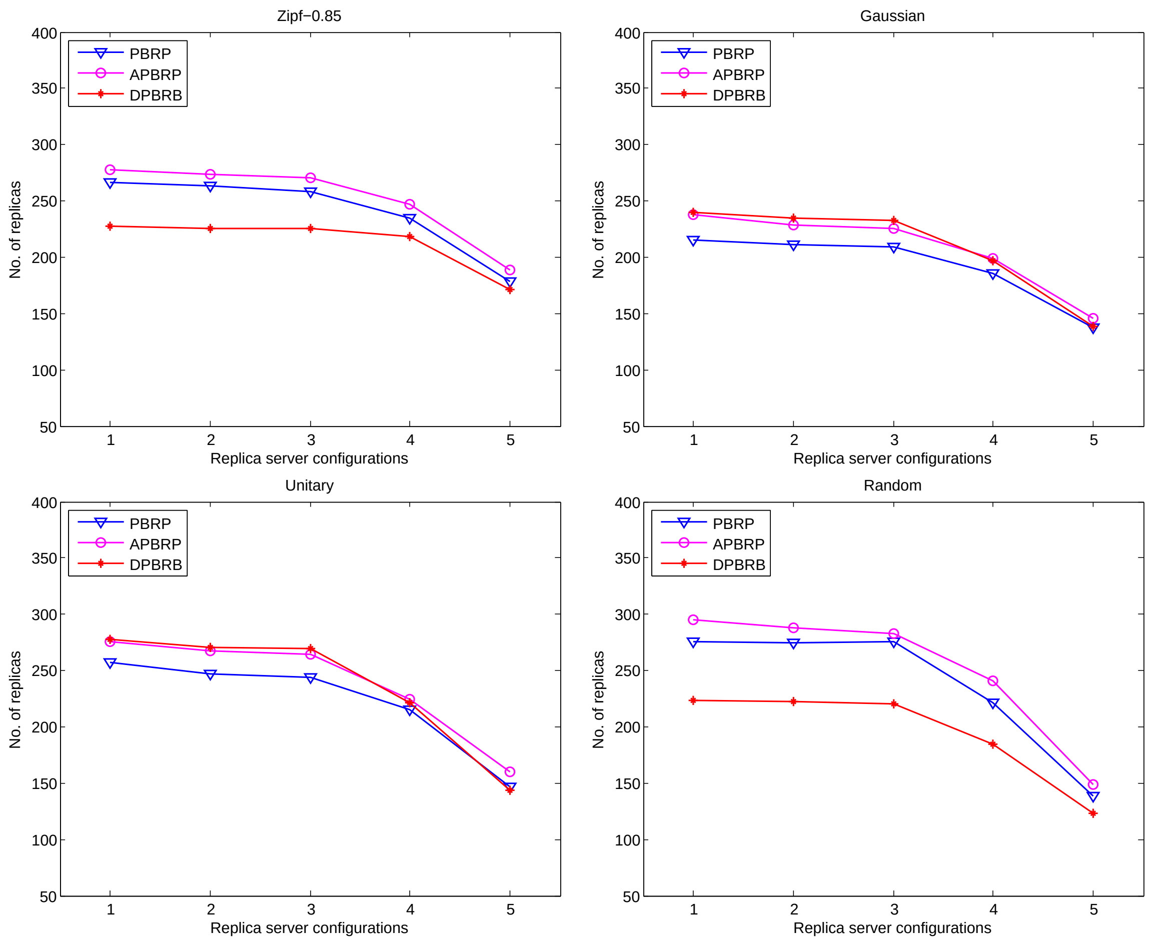


Figure 5.8: No. of replicas for all storage configurations, Fluctuating rate

patterns. The benefit of DPBRP over the other two algorithms reflects the fact that, like, APBRP, it is able to adapt to changing conditions. This is due to fact that the DPBRP takes into account access frequencies from clients while determining the replica locations by calculating replication costs.

Figure 5.9 compares the job execution times for all access patterns and storage configurations as the access rate fluctuates. With decreasing storage size, the execution times are increased in most cases for all the algorithms but by different amounts. DPBRP tends to show shorter execution times for all storage configurations and dis-

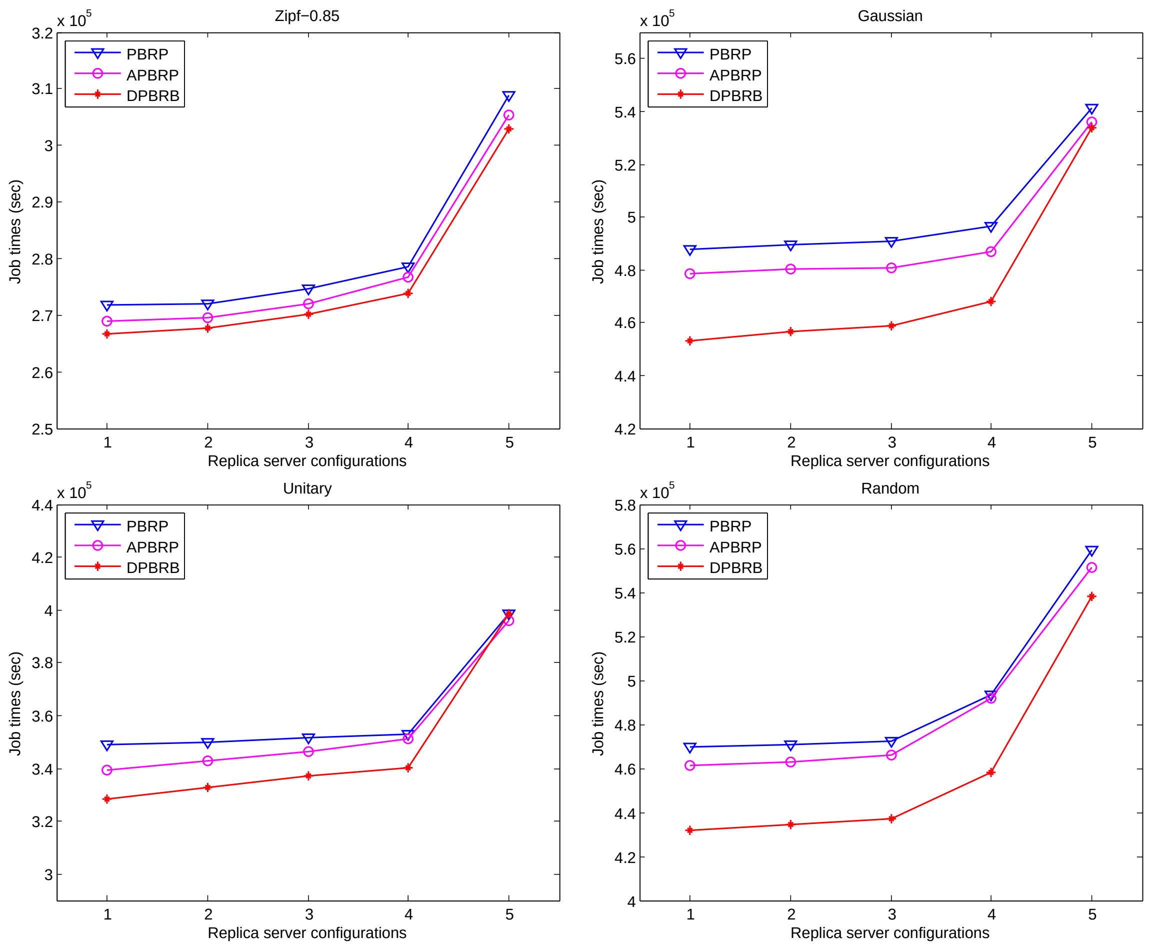


Figure 5.9: Execution time for all access patterns and configurations, Fluctuating rate

tributions. The benefit of DPBRP over others decreases when data accesses exhibit a low degree of temporal locality and it is therefore harder to select a lasting optimal placement.

5.3.2 Average Bandwidth Cost

Bandwidth consumption is an issue both for network providers and end-users (since excessive use of bandwidth can lead to slow downs due to network congestion). In almost all cases, DPBRP consumes less bandwidth than its predecessors while also

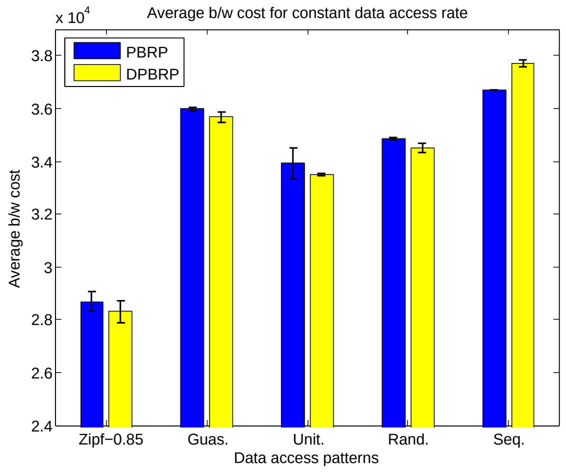


Figure 5.10: Average bandwidth for DPBRP and PBRP, Configuration one, Constant rate

providing shorter run times for accessing jobs.

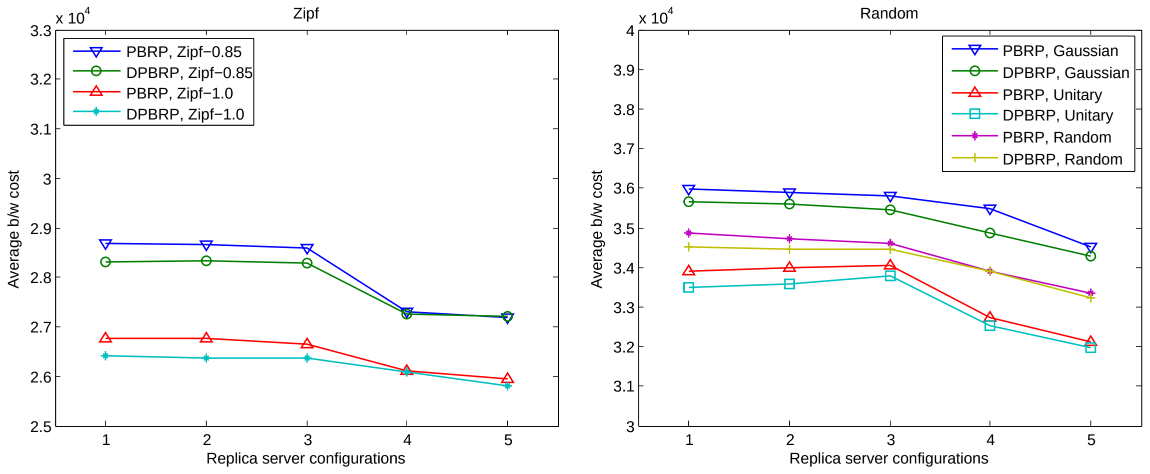


Figure 5.11: Average bandwidth for DPBRP and PBRP, All configurations, Constant rate

Constant data access rate

Figure 5.10 shows the average bandwidth consumption for configuration one. For all access patterns, the costs of DPBRP are less since more client requests are served by mid-tier nodes. As a result, the bandwidth used in the upper tier links is decreased

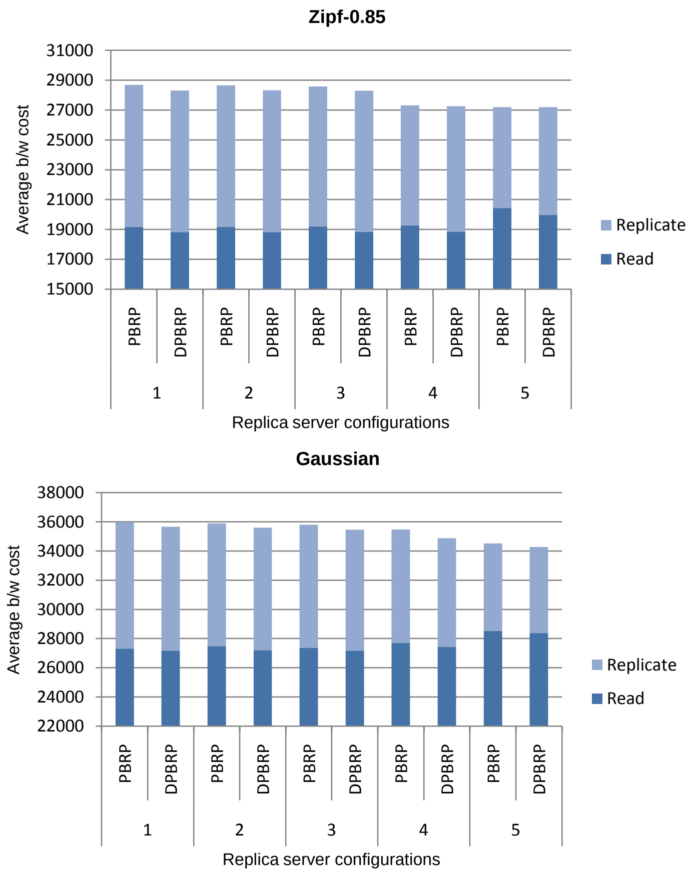


Figure 5.12: Average bandwidth costs (read and replicate) for Zipf and Gaussian access patterns

and the average bandwidth used by DPBRP is lower. As mentioned, PBRP requires more replicas than DPBRP. Thus, PBRP also results in more bandwidth consumption for updating the additional replicas. For Zipf-0.85, the difference between DPBRP and its counterpart is more pronounced. Figure 5.11 shows the bandwidth cost for all access patterns and storage configurations. DPBRP shows less average bandwidth consumption for all storage configurations. As the storage capacity decreases, the bandwidth costs of both strategies tend to decrease. This happens due to a lesser frequency of replica creation resulting in a lower replication cost. This fact is reflected in Figure 5.12 for the Zipf-0.85 and Gaussian access patterns.

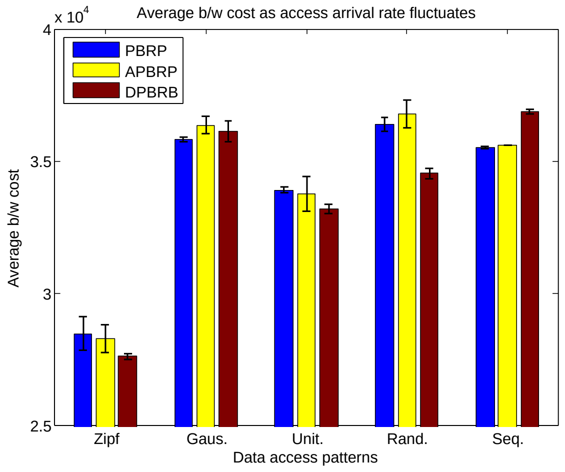


Figure 5.13: Average bandwidth, Configuration one, Fluctuating rate

Fluctuating data access rate

Figure 5.13 shows the average bandwidth costs of DPBRP, APBRP, and PBRP for configuration one as the request arrival rate fluctuates. DPBRP requires less bandwidth for most file access patterns. DPBRP's increased number of client requests served by mid-tier nodes compared to the other algorithms results in decreased bandwidth consumption. APBRP normally shows less and in some cases slightly higher bandwidth consumption compared to its non-adaptive counterpart due to the additional replicas in APBRP that incur some extra data transfer cost.

The average bandwidth costs for all storage configurations and access patterns under regular fluctuation in request rates are shown in Figure 5.14. DPBRP shows less average bandwidth consumption for all storage configurations but the benefit of DPBRP over other algorithms decreases as the storage capacities of replica servers

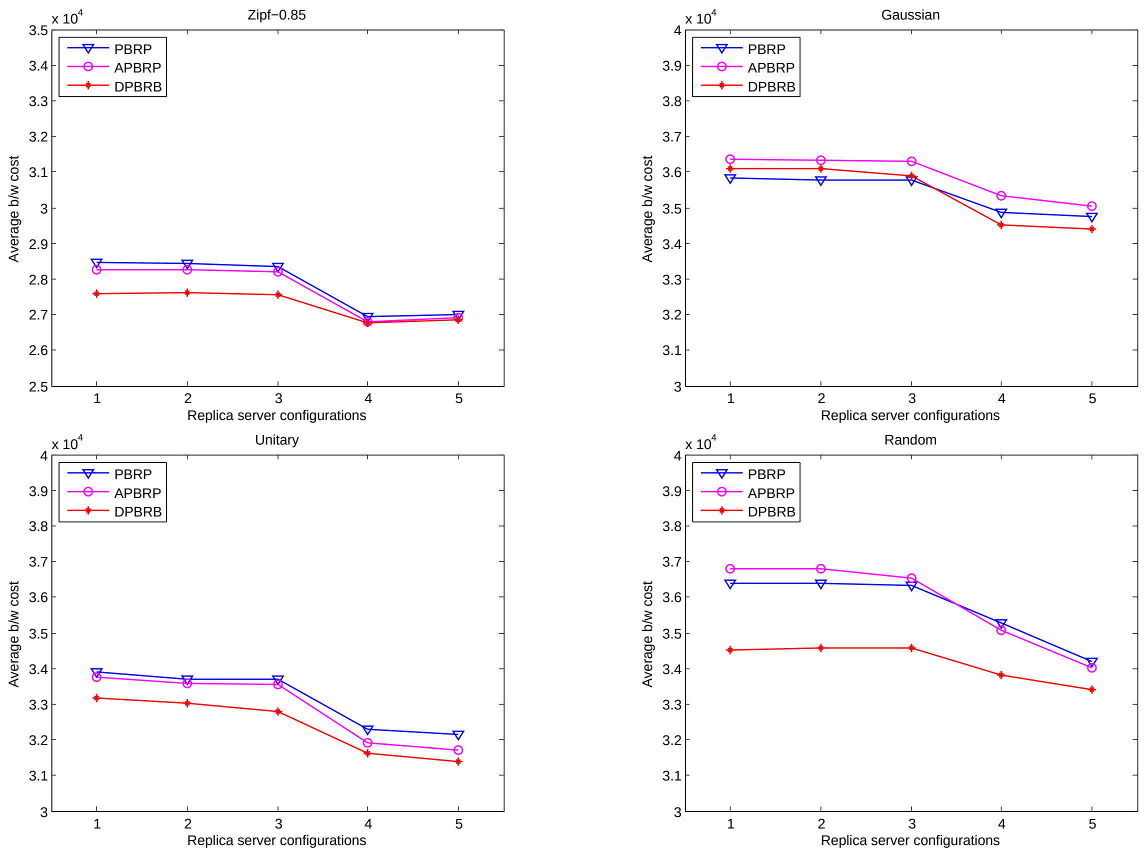


Figure 5.14: Average bandwidth, All algorithms, Configuration One, Fluctuating Rate

are reduced and it too is hard pressed to select replica locations. Figure 5.15 shows read and replication cost components for the Zipf-0.85 and random access patterns. As before, both read and replication costs for DPBRP are less compared to the centralized counterparts.

5.3.3 Storage Use

The total use of storage in a data grid is important to grid providers but due to its relatively low cost can be effectively traded-off for improvements in job execution time and network bandwidth consumed, as needed.

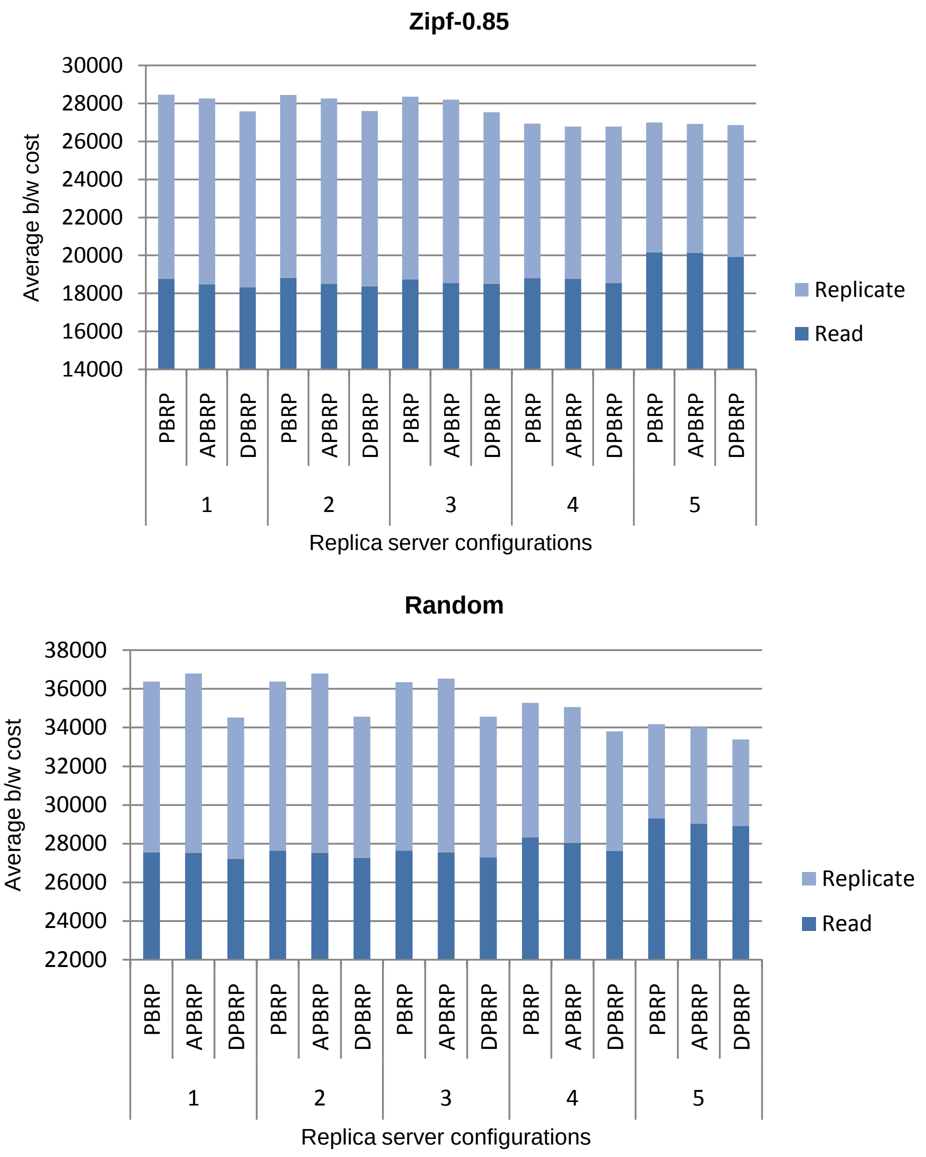


Figure 5.15: Average bandwidth costs (read and replicate) for Zipf and random access patterns

Constant data access rate

The storage used by the placement schemes for configuration one are shown in Figure 5.16. For all access patterns, the storage used by DPBRP is lower than PBRP due to the fewer number of replicas created down the hierarchy. This does

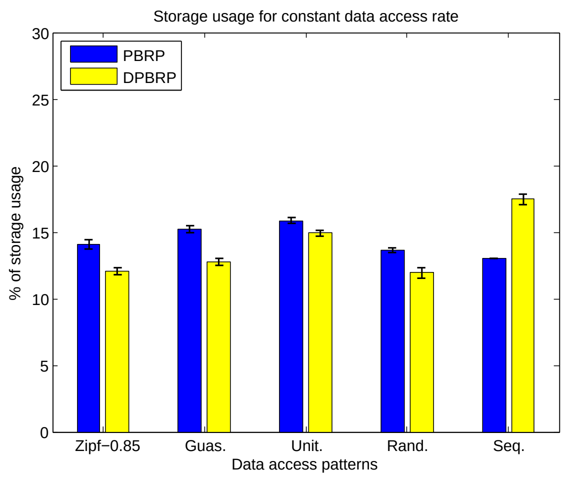


Figure 5.16: Storage use for DPBRP and PBRP, Configuration one, Constant rate

not, however, impact performance negatively as was seen in earlier results for PBRP using consistently decreasing data access rate (Figures 4.21 and 4.38).

The capacity of the replica servers has a clear impact on the percentage of storage used by different strategies. Reducing the available storage naturally leads to an increase in the percentage of storage used as shown in Figure 5.17. One might expect that a significant reduction in storage size should lead to 100% use of storage but, this was not commonly seen. Depending on access patterns, some nodes' storage was used completely while others were not used at all. Despite this, DPBRP consumed less overall storage in most cases while also providing faster execution.

Fluctuating data access rate

Figure 5.18 shows the storage use of DPBRP, APBRP, and PBRP using server configuration one as the request arrival rate fluctuates. DPBRP often requires less storage than the other algorithms for all data access patterns. In most cases, the storage consumption of PBRP is moderate while APBRP is higher due to its creation

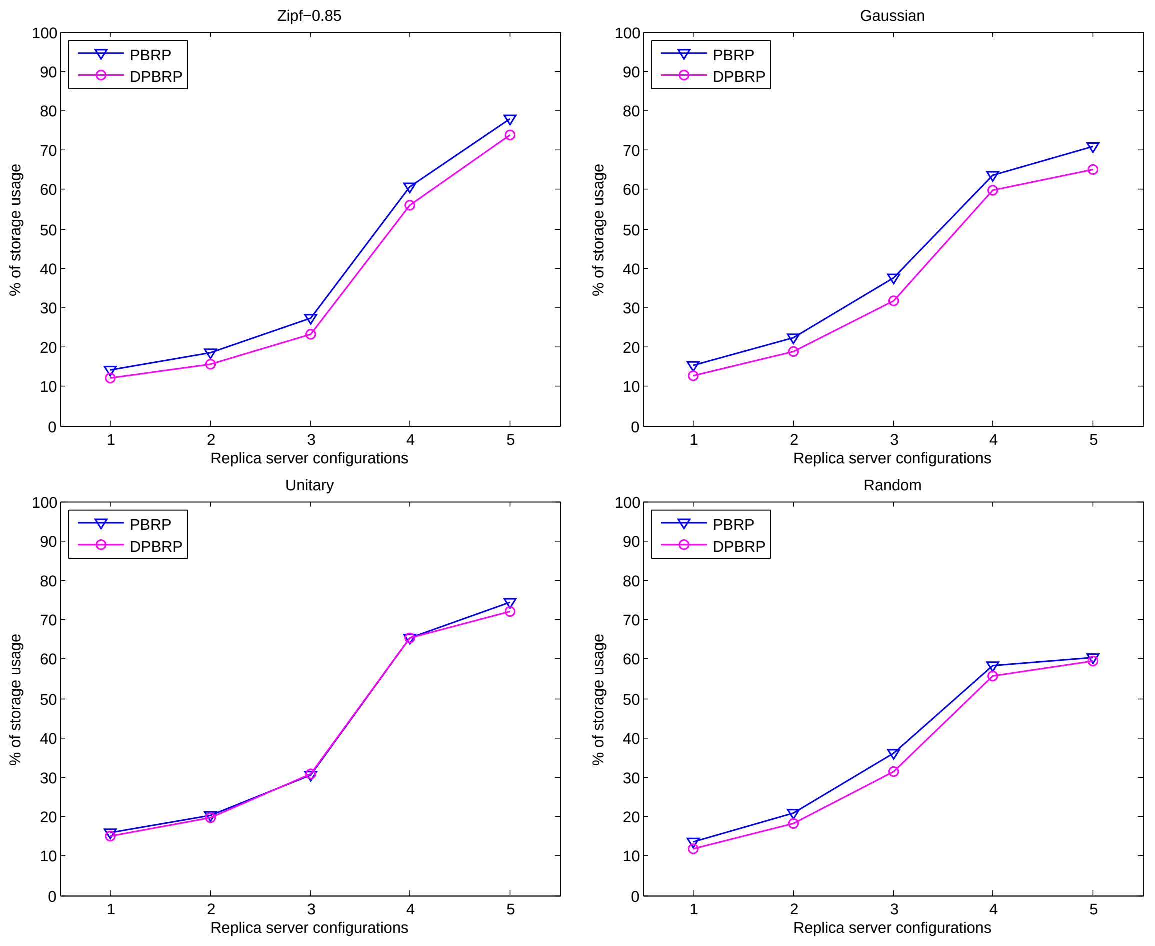


Figure 5.17: Storage use for DBRP and PBRP, All configurations, Constant rate

of an increased number of replicas.

Finally, Figure 5.19 compares the storage use of the placement algorithms for all storage configurations and access patterns as the request rate fluctuates. With decreasing storage availability, the percentages of storage used by each algorithm increase but by different amounts. APBRP shows relatively high storage requirements for all storage configurations among the studied algorithms. DPBRP and PBRP demonstrate somewhat varied relative consumption based on the data access pattern used.

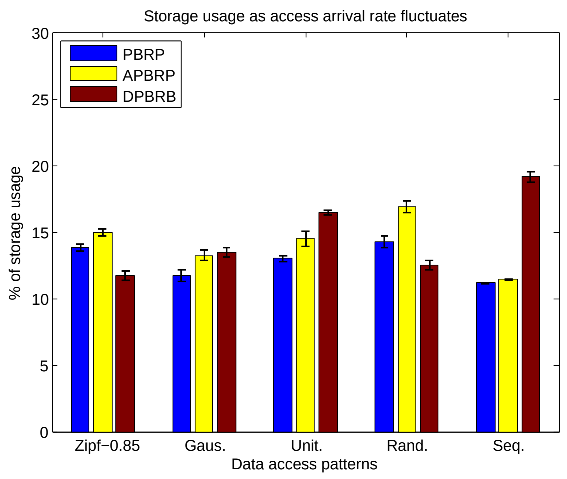


Figure 5.18: Storage use, Configuration one, Fluctuating rate

5.3.4 Discussion

The primary goal of my distributed dynamic replication algorithm is to reduce the job execution time experienced by the end-user (by decreasing data access latency). At the same time, from the perspective of the whole system, the performance metrics of bandwidth consumption and storage use need to be managed to ensure that DPBRP does not induce unduly heavy load on the system.

In most situations, DPBRP shortens job execution time, sometimes significantly, and reduces bandwidth consumption compared to my other algorithms and those described in [RF01a]. Further, the storage costs incurred by DPBRP are less than the other algorithms in most cases. The benefits of DPBRP are achieved by creating an appropriate number of well-placed replicas. My results suggest that my algorithm is successful in deciding which data files should be replicated and where the replicas should be placed. The available storage capacities of the replica servers naturally has a major impact on the performance of replication techniques. Increasing the replica

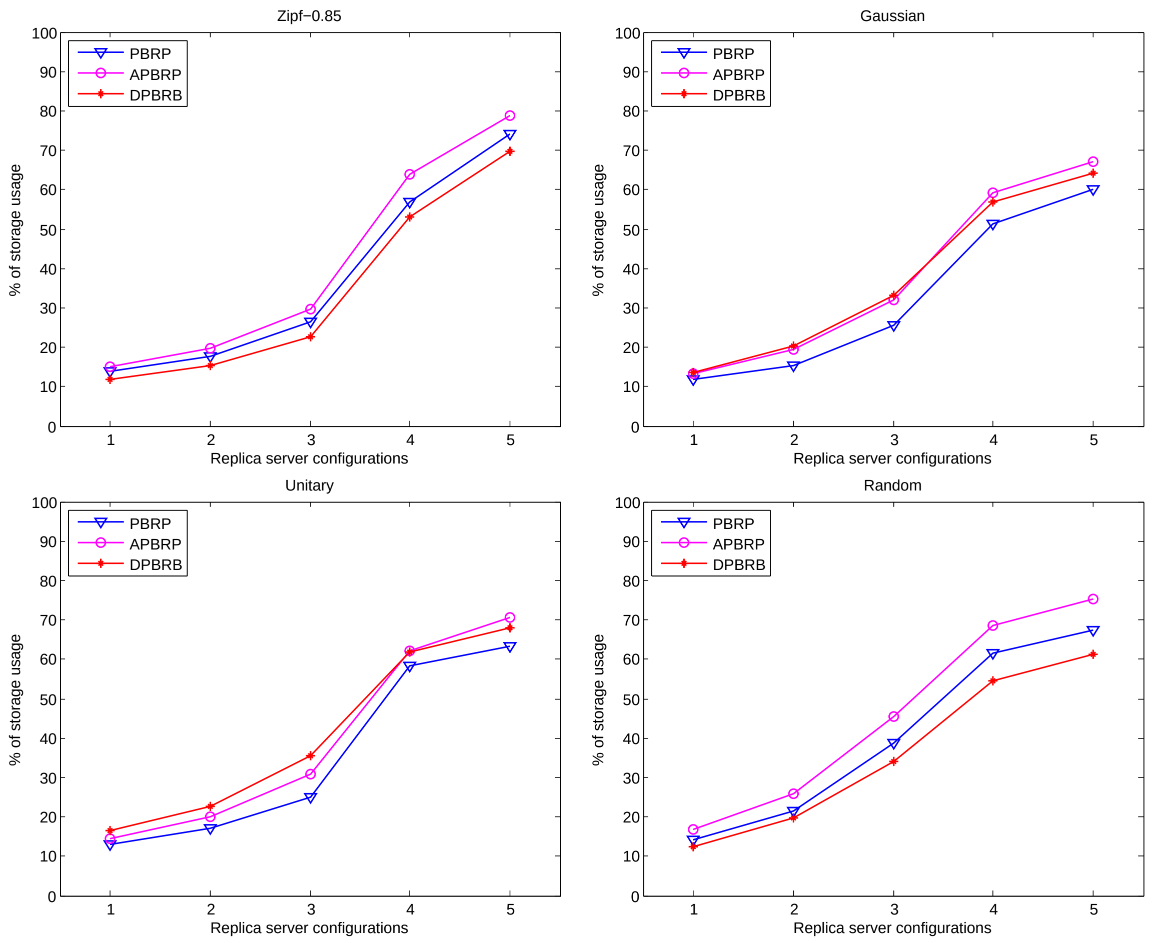


Figure 5.19: Storage use, All configurations, Fluctuating rate

server capacity leads to performance improvement in terms of job execution time and average bandwidth cost. DPBRP appears to be better able to exploit increases in available capacity than other algorithms.

In my original PBRP algorithm, the threshold value remained constant irrespective of variation in access request rate and the available storage capacities of the replica servers. APBRP addresses this by dynamically changing the threshold value based on request rates and storage availability. Simulation results show that APBRP is able to further reduce the job execution time and, in some cases, bandwidth use when the data access rate regularly fluctuates at the expense of additional storage cost. My earlier results show that PBRP can shorten job execution time significantly and reduce bandwidth consumption compared to other dynamic methods including ABU, Fast Spread and Cascading placement. Thus, transitively, DPBRP also performs better than these other non-adaptive dynamic replication methods. Further, as DPBRP is distributed, it does not suffer from the limitations associated with centralized algorithms (e.g. reliability issues and performance bottlenecks). DPBRP and its performance results compared to my centralized algorithms (PBRP and APBRP) have been published in [SGE10b].

5.4 QoS-Aware Distributed Replica Placement

Although assessing overall system performance is important, this does not meet the performance requirements of individuals. Meeting Quality of Service (QoS) constraints is an important consideration in addition to overall system performance. In this section, I present a modified distributed replica placement algorithm for hierarchical data grids that determines replica locations by minimizing overall replication cost (read and update) while satisfying certain QoS requirements both from the user and system perspectives for a given traffic pattern. Users can specify their QoS preferences in terms of an upper bound or a range on the distance (e.g. number of hops) to the nearest replica server. The system can also enforce workload capacity constraints on the replica servers and assumes that the bandwidth capacity of a link is bounded. The goal is to make sure that each request from a user can be serviced by a replica server within its quality requirement and without violating the capacity limits of the replica server(s) and network links. Each of these QoS constraints will be added incrementally to the unconstrained DPBRP algorithm described in the previous section. The details of the QoS constrained DPBRP follow.

5.4.1 Hop Count

My modified, QoS-aware, replica placement algorithm, QoS-DPBRP is designed to simultaneously improve system performance and satisfy the quality requirements of users. The QoS requirement $q(v)$ can be any function that represents, for example, the number of hops between the user and the replica server, the access latency between them, or a combination of things. Responsiveness, which refers to how fast the users can access the requested data, is an important type of QoS requirement needed by a wide range of applications. I address the challenges introduced by responsiveness QoS requirements in data replication to improve the performance of replication in large-scale data grid systems. The effectiveness of replication, to a large extent, depends on the replica locations in the system. In general, a client would experience shorter access latency if a replica of the requested data is placed in its closer proximity. Therefore, I consider replica distance as a fundamental QoS requirement in this thesis (though the implementation strategy generalizes to other QoS measures). Thus, each user may specify a maximum distance allowable to the nearest replica server. This means a request must be served by a replica, or the root, within a fixed number of hops towards the root. In other words, a request can reach a server if the number of communication links between it and the nearest replica on the path to the root is no more than its hop count limit. Formally, I define my objective which is to find a replication strategy, R, with the minimal replication cost such that $R \cup \{r\}$ satisfies the QoS requirement of every client v, i.e. $min_{s \in R \cup \{r\}} d(v,s) \leq q(v)$, where $d(v,s)$ is the distance between v and s.

Algorithm

QoS-aware replica placement in hierarchical data grids can be performed by modifying the existing DPBRP algorithm. As described in DPBRP, each node v calculates the cost function for each distance possibility up to the root and also for each case it determines the optimal location of the replica. Since the QoS requirement, $q(v)$, can be specified by the distance (i.e. number of hops) between the client and the replica server, this distance constraint can easily be incorporated without any additional complexity when the replication costs for clients are calculated considering various distance possibilities for the replica servers. For a client node, v with QoS requirement, $q(v)$, an initial replication cost, iC, is calculated (extending Equation 5.6) as:

$$iC(v,d) = \begin{cases} UC(v), & \text{if d} = 0 \\ RC(v,d), & \text{if } 1 \leq d \leq q(v),\ q(v) \leq i, \text{ and } a(v,i) = r \\ \infty, & \text{if } d > q(v) \end{cases} \tag{5.13}$$

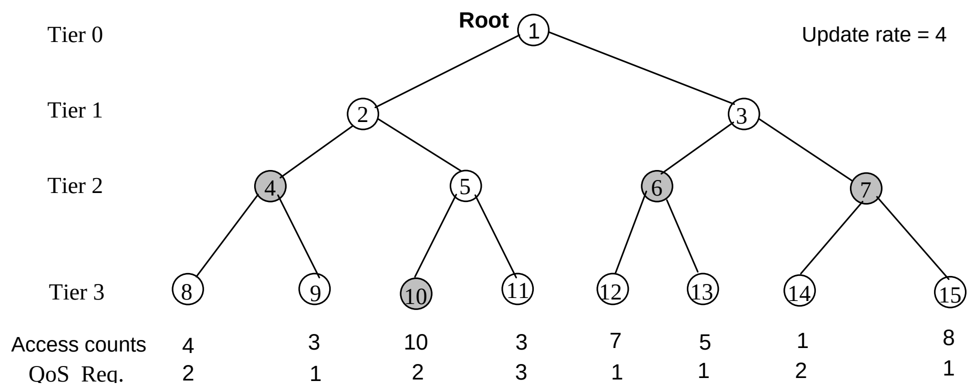
In the above equation, when $d = 0$, replication cost does not include a read cost since the replica is in v itself. For $d \leq q(v)$, i.e. the distance of the replica server is within the QoS requirement of the client, the replication cost for subtree T_v (in this case only v) includes only the read cost of v. When $d > q(v)$, the placement of a replica cannot satisfy the client's requirement and hence the cost is set to infinity. The replication cost, C, and the corresponding replica locations can now be calculated using Equations 5.8 - 5.12. Essentially, for the case, $d > q(v)$, the replication cost,

$C(v,d)$ will be $C(v,0)$ (from Equation 5.11) which is the cost if the replica is created locally.

Placement Example

Consider the data grid in Figure 5.20(a) which consists of 15 nodes. The access counts and QoS requirements for a particular file are shown beneath the leaves (e.g. the QoS requirement is 2 hops for node 8). Assume the number of updates from the root is 4 and, again, that both accesses and updates require one bandwidth unit per file per hop. Figure 5.20(b) shows calculations of *cost* and *loc* functions, done bottom-up for each node and all distance possibilities considering QoS. The calculation starts from the client tier (tier 3) of the hierarchy and ends at the root. Each node sends its *cost* and *loc* values to its parent which receives these values from its children and then calculates its own *cost* and *loc* functions.

In the example, during top-down placement, the root determines that the optimal replication cost for the entire data grid is 73 and sets *distance* to zero before sending it to its children. The placement process is then carried out as described in Section 5.2.3. Replicas are placed in nodes 4, 6, 7, and 10 in tier 2 and tier 3 based on the *distance* and *loc* values. The total replication cost considering the replication strategy (4,6,7,10) is 37(read cost) + 36 (update cost) = 73 which is the same as the cost calculated at the root node in Figure 5.20(b) and all QoS constraints were satisfied. Note that the total replication cost, 73, considering the QoS requirements of the clients is greater than the optimal cost (71) obtained in Figure 5.3(b). This is due to the creation of a replica in node 7 to satisfy the QoS requirements of both nodes 14 and 15. This shows that certain QoS constraints specified by the clients can have



(a)

Grid sites (v)				{cost(v,d), loc(v,d)}			
Tier 3	Tier 2	Tier 1	Tier 0	d = 0	d = 1	d = 2	d = 3
8				(12, 8)	(4, 4)	(8, 2)	**(12, 8)**
	4			(15, 4)	**(15, 4)**	(15, 4)	
9				(12, 9)	(3, 4)	**(12, 9)**	**(12, 9)**
		2		**(36, -1)**	(36, 1)		
10				(12, 10)	(10, 5)	(12, 10)	(12, 10)
	5			(21, 5)	(18, 2)	(21, 1)	
11				(12, 11)	(3, 5)	(6, 2)	(9, 1)
			1	**(73, 1)**			
12				(12, 12)	(7, 6)	(12, 12)	(12, 12)
	6			(20, 6)	(20, 6)	(20, 6)	
13				(12, 13)	(5, 6)	**(12,13)**	(12, 13)
		3		(37, -1)	**(37, 1)**		
14				(12, 14)	(1, 7)	(2, 3)	**(12, 14)**
	7			(17, 7)	(14, 3)	**(17, 7)**	
15				(12, 15)	(8, 7)	(12, 15)	(12, 15)

(b)

Figure 5.20: (a) A hierarchical data grid with client access counts and QoS requirements, (b) Calculation of replication costs and replica locations for the example data grid. Highlighted entries indicate the modified costs and locations due to QoS support.

an overall impact on the optimal placement of replicas and the optimal replication cost.

5.4.2 Workload Capacity

I modified my QoS-aware replica placement problem to determine the locations of the replicas considering workload capacity constraints while at the same time satisfying the quality requirements specified by the users. Every server, v, has a workload capacity constraint $W(v)$ which is specified as an upper bound on the number of user requests processed by v for a particular period of time. The replication strategy has to ensure that the user requests are satisfied while limiting the workload of each replica server to its capacity. The workload capacity constraint associated with different servers can, of course, be different. If the total workload that a server services is greater than its capacity constraint, then the server is overloaded. The goal is to find a replication strategy, R, with the minimal replication cost such that $R \cup \{r\}$ satisfies the QoS requirement of every client v, i.e. $min_{s \in R \cup \{r\}} d(v, s) \leq q(v)$, where $d(v, s)$ is the distance between v and server s and none of the servers in R is overloaded.

Algorithm

Workload capacity constraints on the replica servers can be easily incorporated by modifying the existing DPBRP and QoS-DPBRP algorithms. Each node v calculates the replication cost, $C(v, d)$, for the subtree rooted at it for each value of d for the replica location when $0 \leq d \leq i$ and $a(v, i) = r$. While calculating the cost function for each replica distance possibility, node v can verify that the total aggregated access counts for its subtree does not exceed the workload capacity of each node on the path from node v to the root. For example, for a node v with children $v_1, ..., v_n$, the aggregated access counts, $AC(v) = \sum_{1 \leq i \leq n} c(v_i)$ should be less than $W(v)$ where v represents each node (potential server) on the path from v to the root. For a client node v, $AC(v)$ will include only its own access frequency which should be less than $W(v)$. If $AC(v)$ is greater than $W(v)$, we have to place some replicas on the children, v_i's (for a non-client node) or on v itself (for a client node) to bound the incoming request on $W(v)$. In the former case, this can be done by repeatedly considering a potential replica at v_i's that has the largest access count, until the remaining aggregated request from the children is within $W(v)$. Let this set of v_i's be *potrep*$(ch(v))$, which is the set of children of v that are potentially determined to be equipped with a replica.

Now, we need to calculate the initial replication cost using Equations 5.7 and 5.13 for a node v (considering the cost for placing potential replicas to limit the total request to the workload capacity). For a client node, v, with access count, $c(v)$, an initial replication cost, iC, is calculated (extending Equation 5.13) as:

$$iC(v,d) = \begin{cases} UC(v), & \text{if d} = 0 \\ RC(v,d), & \text{if } 1 \le d \le q(v),\ q(v) \le i,\ c(v) \le W(v), \text{ and} \\ & a(v,i) = r \\ iC(v,0), & \text{if } 1 \le d \le q(v),\ q(v) \le i,\ c(v) > W(v), \text{ and} \\ & a(v,i) = r \\ \infty, & \text{if } d > q(v) \end{cases} \tag{5.14}$$

In the above equation, when $d = 0$, replication cost does not include a read cost since the replica is in v itself. For $d \ge 1$ and $c(v) \le W(v)$, i.e. the replica server is up in the path from v to the root and the access count is within the workload capacity, the replication cost for subtree T_v (in this case only v) includes only the read cost of v. When $c(v) > W(v)$, the replication cost is the cost of placing a replica in v itself. Both of these cases consider $d \le q(v)$, i.e. the distance of the replica server is within the QoS requirement of the client. When $d > q(v)$, the placement of a replica cannot satisfy the client's requirement and hence the cost is set to infinity.

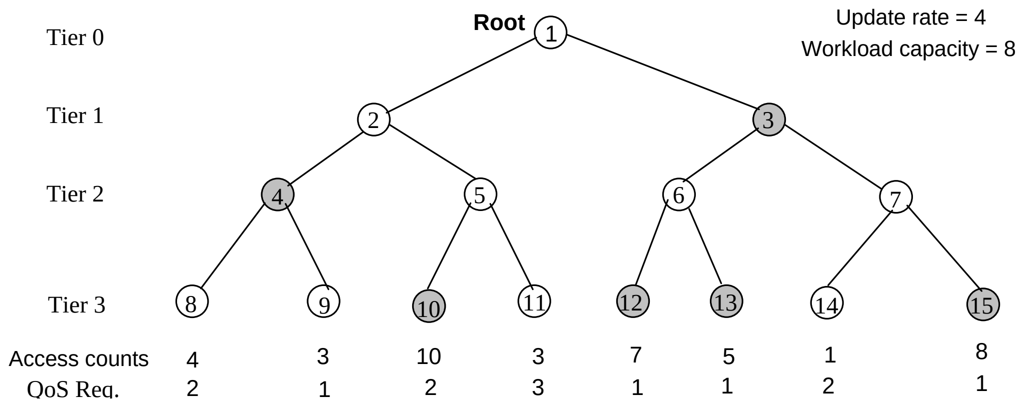
For a non-client node, v, iC, is calculated (extending Equation 5.7) as follows when $AC(v) > W(v)$:

$$iC(v,d) = \begin{cases} \sum_{n \notin ch(v) \bigcap potrep(ch(v))} C(n, d+1) & \\ + \sum_{n \in potrep(ch(v))} C(n,0) + UC(v), & \text{if d} = 0 \\ \sum_{n \notin ch(v) \bigcap potrep(ch(v))} C(n, d+1) & \\ + \sum_{n \in potrep(ch(v))} C(n,0), & \text{if } 1 \leq d \leq i \text{ and } a(v,i) = r \end{cases} \tag{5.15}$$

In this case, the cost for placing potential replicas on the $potrep(ch(v))$ nodes is added to the cost of the rest of the children nodes of v. The replication cost, C, and the corresponding replica locations can now be calculated using Equations 5.8 - 5.12.

Placement Example

Consider the data grid in Figure 5.21(a) which consists of 15 nodes. The access counts for a particular time period and QoS requirements for a data file are given in the client-tier of the hierarchy. Assume the number of updates from the root is 4 and, again, that both accesses and updates require one bandwidth unit per file per hop. The workload capacity of each node is 8. Figure 5.21(b) shows the calculation of *cost* and *loc* functions for each grid node considering all distance possibilities for the replication location. The calculation starts from the client tier (tier 3) of the hierarchy and ends at the root. Each node sends its *cost* and *loc* values to its parent which receives these values from its children and then calculates its own *cost* and *loc* functions.



(a)

Grid sites (v)				{cost(v,d), loc(v,d)}			
Tier 3	Tier 2	Tier 1	Tier 0	d = 0	d = 1	d = 2	d = 3
8				(12, 8)	(4, 4)	(8, 2)	(12, 8)
	4			(15, 4)	(15, 4)	(15, 4)	
9				(12, 9)	(3, 4)	(12, 9)	(12, 9)
		2		(37, 2)	(36, 1)		
10				(12, 10)	(12, 10)	(12, 10)	(12, 10)
	5			(23, 5)	(18, 2)	(21, 1)	
11				(12, 11)	(3, 5)	(6, 2)	(9, 1)
			1	(78, 1)			
12				(12, 12)	(7, 6)	(12, 12)	(12, 12)
	6			(24, -1)	(24, 3)	(24, 1)	
13				(12, 13)	(5, 6)	(12,13)	(12, 13)
		3		(42, 3)	(42, 3)		
14				(12, 14)	(1, 7)	(2, 3)	(12, 14)
	7			(21, 7)	(14, 3)	(21, 7)	
15				(12, 15)	(8, 7)	(12, 15)	(12, 15)

(b)

Figure 5.21: (a) A hierarchical data grid with client access counts, QoS requirements, and workload capacity constraint, (b) Calculation of replication costs and replica locations for the example data grid.

During top-down placement, the root determines that the optimal replication cost for the entire data grid is 78 and sets *distance* to zero before sending it to its children. The placement process is then carried out as described in Section 5.2.3. Replicas are placed in nodes 3, 4, 10, 12, 13, and 15 in tier 1, tier 2, and tier 3 based on the *distance* and *loc* values. Now, the total replication cost for the replication strategy (3,4,10,12,13,15) is 18(read cost) + 60(update cost) = 78 which is the same as the cost calculated at the root node in Figure 5.21(b).

5.4.3 Link Capacity

Taking into account limited link capacity in replica placement is an important consideration in addition to user QoS requirements. User QoS constraints are local to a node, hence, each client has to cope with its own limitation. On the other hand, bandwidth constraints have a global influence on the resources as a link may be shared by multiple clients and consequently all of them are concerned. To this end, I modify the QoS-aware replica placement problem to determine the locations of the replicas considering link capacity constraint while at the same time satisfying the quality requirements specified by the user. Each link in the data grid hierarchy has some capacity constraint on transferring the data down the hierarchy. The bandwidth constraint on each link l is specified by an upper bound $bc(l)$ on link capacity. Such a constraint mandates that the amount of data that can pass through the link, l, over a period of time will be limited within $bc(l)$. The replication strategy has to ensure that the user requests are satisfied while limiting the bandwidth usage of each link to its capacity. The bandwidth constraint associated with different links can, of course, be different. If the total amount of data passing through a link is greater than its capacity constraint, then the link is congested. The goal is therefore to find a replication strategy, R, with the minimal replication cost such that $R \cup \{r\}$ satisfies the QoS requirement of every client v, i.e. $min_{s \in R \cup \{r\}} d(v, s) \leq q(v)$, where $d(v, s)$ is the distance between v and server s and none of the communication link $l(u, v) \in E$ is congested.

Algorithm

Bandwidth capacity constraints on the communication links can also be incorporated by modifying the existing DPBRP and QoS-DPBRP algorithms. Each node v calculates the replication cost, $C(v, d)$, for the subtree rooted at it for each value of d for the replica location when $0 \leq d \leq i$ and $a(v, i) = r$. While calculating the cost function for each replica distance possibility, node v can verify that the total aggregated bandwidth requirement for its subtree does not exceed the bandwidth capacity of each link on the path from node v to the root in a manner similar to checking that aggregate server load is not exceeded. For example, for a node v with children $v_1, ..., v_n$, the aggregated bandwidth, $ABW(v) = \sum_{1 \leq i \leq n} bw(v_i)$ should be less than $bc(l)$ where l represents each link on the path from node v to the root. For a client node v, $ABW(v)$ will include only its own bandwidth requirement which is assumed to be less than $bc(l)$. If $ABW(v)$ is greater than $bc(l)$, we have to place some replicas on the v_i's (for a non-client node) or on v itself (for a client node) to bound the incoming request on $bc(l)$. In the former case, this can be easily done by repeatedly considering a potential replica at v_i's that has the largest bandwidth requirement, until the request is within $bc(l)$. Let this set of v_i's be $potrep(ch(v))$, which is the set of children of v that are potentially determined to be equipped with a replica.

Now, we need to calculate the initial replication cost using Equations 5.7 and 5.13 for a node v (considering the cost for placing potential replicas to limit the bandwidth requirement to the link capacity). For a client node, v with bandwidth requirement,

$bw(v)$, an initial replication cost, iC, is calculated (extending Equation 5.13) as:

$$iC(v,d) = \begin{cases} UC(v), & \text{if d} = 0 \\ RC(v,d), & \text{if } 1 \leq d \leq q(v),\ q(v) \leq i,\ bw(v) \leq bc(l), \text{ and} \\ & a(v,i) = r \\ iC(v,0), & \text{if } 1 \leq d \leq q(v),\ q(v) \leq i,\ bw(v) > bc(l), \text{ and} \\ & a(v,i) = r \\ \infty, & \text{if } d > q(v) \end{cases} \tag{5.16}$$

In the above equation, when $d = 0$, replication cost does not include a read cost since the replica is in v itself. For $d \geq 1$ and $bw(v) \leq bc(l)$, i.e. the replica server is up in the path from v to the root and the requested bandwidth is within the link capacity, the replication cost for subtree T_v (in this case only v) includes only the read cost of v. When $bw(v) > bc(l)$, the replication cost is the cost of placing a replica in v itself. Both of these cases consider $d \leq q(v)$, i.e. the distance of the replica server is within the QoS requirement of the client. When $d > q(v)$, the placement of a replica cannot satisfy the client's requirement and hence the cost is set to infinity.

For a non-client node, v, iC, is calculated (extending Equation 5.7) as:

$$iC(v,d) = \begin{cases} \sum_{n \in ch(v)} C(n,d+1) + UC(v), & \text{if d} = 0 \\ \sum_{n \in ch(v)} C(n,d+1), & \text{if } 1 \leq d \leq i, ABW(v) \leq bc(l) \text{, and} \\ & a(v,i) = r \\ \sum_{n \notin ch(v) \bigcap potrep(ch(v))} C(n,d+1) & \\ + \sum_{n \in potrep(ch(v))} C(n,0), & \text{if } 1 \leq d \leq i,\ ABW(v) > bc(l) \text{, and} \\ & a(v,i) = r \end{cases} \tag{5.17}$$

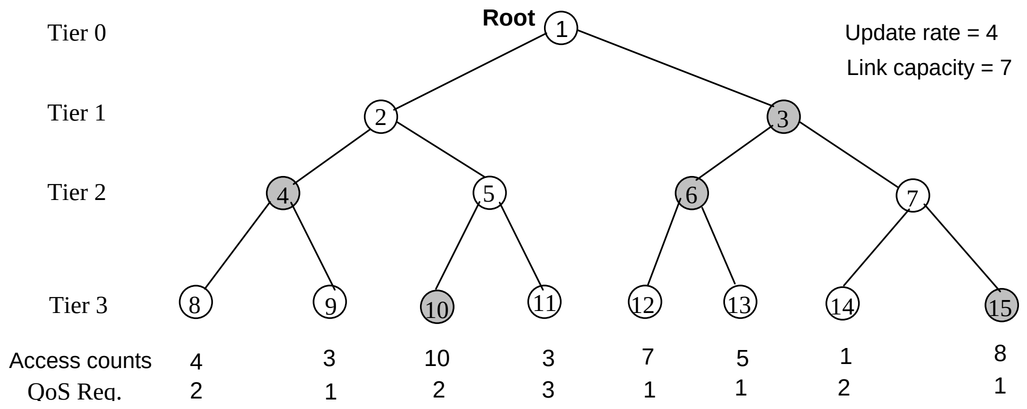
For $ABW(v) > bc(l)$, the cost for placing potential replicas on the $potrep(ch(v))$ nodes is added to the cost of the rest of the child nodes of v. The replication cost, C, and the corresponding replica locations can now be calculated using Equations 5.8 - 5.12.

Considering link and workload capacity constraints together in the QoS-aware replica placement algorithm requires modifications in Equations 5.16 and 5.18 where the initial costs, iC's, are calculated for a client and a non-client node respectively. As for the client node, the second condition in Equation 5.16 must ensure that both $c(v) \leq W(v)$ (i.e. the replica server is up in the path from v to the root and the access count is within the workload capacity) and $bw(v) \leq bc(l)$ (i.e. the replica server is up in the path from v to the root and the requested bandwidth is within the link capacity) are true. The other change is in the third condition of Equation 5.16 which requires that either $c(v) > W(v)$ or $bw(v) > bc(l)$ should be true. As for the non-client node, $iC(v,d)$ can be calculated by merging Equations 5.15 and 5.18:

$$iC(v,d) = \begin{cases} \sum_{n \in ch(v)} C(n,d+1) + UC(v), & \text{if } d = 0 \text{ and } AC(v) \leq W(v) \\ \sum_{n \notin ch(v) \bigcap potrep(ch(v))} C(n,d+1) & \\ + \sum_{n \in potrep(ch(v))} C(n,0) + UC(v), & \text{if } d = 0 \text{ and } AC(v) > W(v) \\ \sum_{n \in ch(v)} C(n,d+1), & \text{if } 1 \leq d \leq i, ABW(v) \leq bc(l), \\ & AC(v) \leq W(v) \text{ and } a(v,i) = r \\ \sum_{n \notin ch(v) \bigcap potrep(ch(v))} C(n,d+1) & \\ + \sum_{n \in potrep(ch(v))} C(n,0), & \text{if } 1 \leq d \leq i,\ (ABW(v) > bc(l) \text{ or} \\ & AC(v) > W(v)) \text{ and } a(v,i) = r \end{cases} \tag{5.18}$$

Placement Example

Consider the data grid in Figure 5.22(a). The access counts for a particular time period and QoS requirements for a data file are given in the client-tier of the hierarchy. Assume the number of updates from the root is 4 and, again, that both accesses and updates require one bandwidth unit per file per hop. Moreover, each link has a capacity of 7 bandwidth unit for data transfer per time period. Figure 5.22(b) shows the calculation of *cost* and *loc* functions for each grid node considering all distance possibilities for the replication location. The calculation starts from the client tier (tier 3) of the hierarchy and ends at the root. Each node sends its *cost* and *loc* values to its parent which, again, receives these values from its children and then calculates its own *cost* and *loc* functions.



(a)

Grid sites (v)				{cost(v,d), loc(v,d)}			
Tier 3	Tier 2	Tier 1	Tier 0	d = 0	d = 1	d = 2	d = 3
8				(12, 8)	(4, 4)	(8, 2)	(12, 8)
	4			(15, 4)	(15, 4)	(15, 4)	
9				(12, 9)	(3, 4)	(12, 9)	(12, 9)
		2		(37, 2)	(36, 1)		
10				(12, 10)	(12, 10)	(12, 10)	(12, 10)
	5			(23, 5)	(18, 2)	(21, 1)	
11				(12, 11)	(3, 5)	(6, 2)	(9, 1)
			1	(74, 1)			
12				(12, 12)	(7, 6)	(12, 12)	(12, 12)
	6			(20, 6)	(20, 6)	(20, 6)	
13				(12, 13)	(5, 6)	(12,13)	(12, 13)
		3		(38, 3)	(38, 3)		
14				(12, 14)	(1, 7)	(2, 3)	(12, 14)
	7			(21, 7)	(14, 3)	(21, 7)	
15				(12, 15)	(12, 15)	(12, 15)	(12, 15)

(b)

Figure 5.22: (a) A hierarchical data grid with client access counts, QoS requirements, and link capacity constraint, (b) Calculation of replication costs and replica locations for the example data grid.

During top-down replica placement, the root determines that the optimal replication cost for the entire data grid is 73 and sets *distance* to zero before sending it to its children. The placement process is then carried out as described in Section 5.2.3. Replicas are placed in nodes 3, 4, 6, 10, and 15 in tier 1, 2, and 3 based on the *distance* and *loc* values. Now, the total replication cost using the replication strategy (3,4,6,10,15) is 30(read cost) + 44(update cost) = 74 which is the same as the cost calculated at the root node in Figure 5.22(b).

QoS-DPBRP has been shown to satisfy QoS requirements both from the user and system perspectives. However, the algorithm does not guarantee that all the QoS requirements (especially when the QoS requirements become more stringent) can be met at all times due to the fact that the actual replica placement depends on factors such as storage availability at the replica servers, file popularity, and so on. For example, if all the clients need replicas to be accessed locally, the requests from all of them may not be satisfied due to the storage limitation. This issue will further be discussed with experimental results in Section 5.5.1.

5.5 Performance Evaluation of the QoS-Aware Distributed Algorithms

The QoS preferences from users were taken in terms of an upper bound (number of hops) to the nearest replica server or from a uniform distribution over a range on the acceptable distance the replica server. To determine the effectiveness of my QoS-DPBRP algorithm, user satisfaction rate was also included as a performance metric. This refers to the percentage of users whose QoS requirements are met. Two different scenarios were considered: when the data request arrival rate remains constant, and when it regularly fluctuates as described earlier. First, I compared DPBRP with QoS-DPBRP using different replica server configurations while considering only user QoS requirements for both scenarios. Then I compared QoS-DPBRP with QoS-aware replica placement algorithms *Greedy Add* and *Greedy Remove* by Cheng et al. [CWL09] considering both user QoS and workload constraints on replica servers when the data access rate remains constant. Finally, link capacity constraint was added to QoS-DPBRP and was compared to its unconstrained counterpart, DPBRP for constant data access rate.

5.5.1 Job Execution Time

I begin by considering DPBRP's performance in terms of job execution time. The results are presented by demonstrating the effects of user QoS constraints, workload capacity, and link capacity constraints on job execution time of DPBRP.

Effects of user QoS, constant data access rate

Figure 5.23 compares job times for DPBRP and QoS-DPBRP using storage configuration one (refer to Figure 4.8) when the access rates from clients remain constant. User QoS requirements/constraints are taken from a constant (i.e. a distance to the server) and a uniform distribution (i.e. the distance requests are uniformly distributed over the range). System QoS (server load and link capacity) constraints are not considered here. For the constant distribution, QoS-DPBRP shows shorter

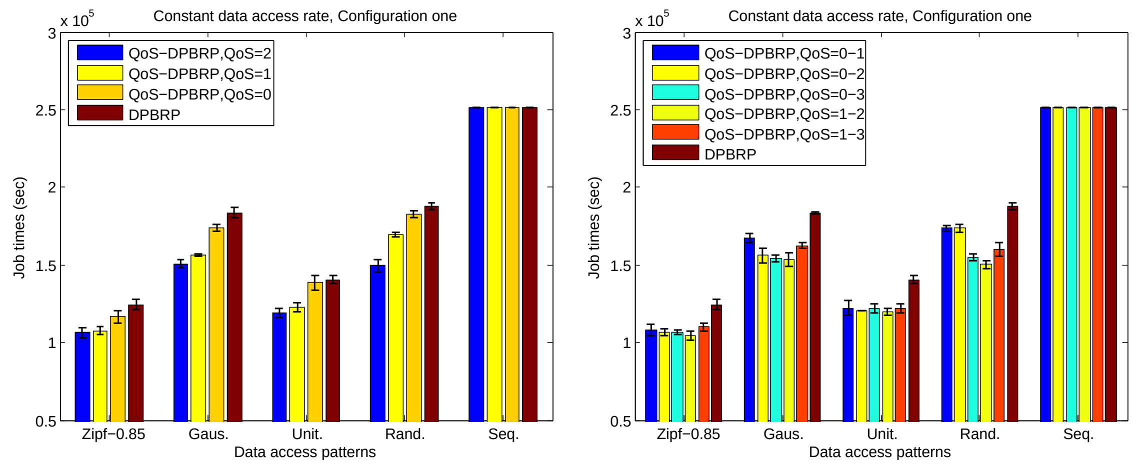
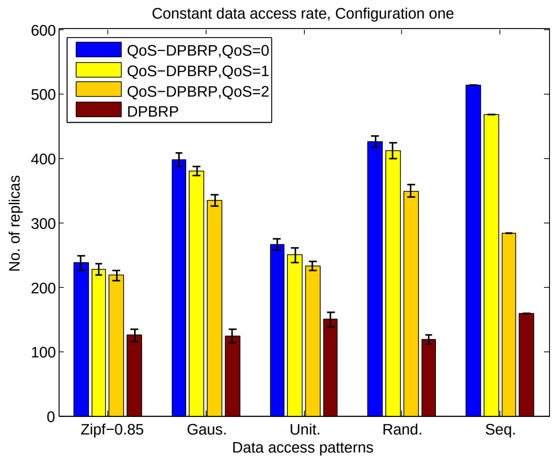


Figure 5.23: Run time, Configuration one, Constant and uniform QoS constraints

execution times when the QoS request is 2 (hops) for most data access patterns. This is because the replica locations selected in this case reduce the data access latency experienced which in turn reduces the overall job execution time. It is noteworthy that for the QoS value of 2, QoS-DPBRP requires (Figure 5.24) fewer replicas than with QoS= 0 or 1, but more replicas than DPBRP.



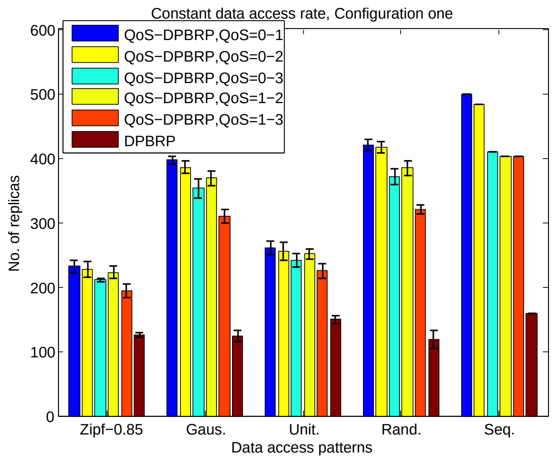


Figure 5.24: No. of replicas, Configuration one, Constant and uniform QoS constraints

The increased number of replicas in QoS-DPBRP compared to DPBRP serve to satisfy more requests which in turn reduces the access latency. Even though QoS=0 results in more replicas, the execution time is still higher in this case. The reason is that replicas are being created mostly in the lower tiers and cannot satisfy an increased number of user requests from the subtrees of the nodes containing replicas. Also, in this case, not all replicas can be created locally to the clients (as requested, QoS=0) due to storage capacity limitations and consequently the satisfaction rate reduces. This is evident from Figure 5.25. Thus, for a “reasonable” QoS request, QoS-DPBRP performs better in terms of execution time and satisfaction rate by creating

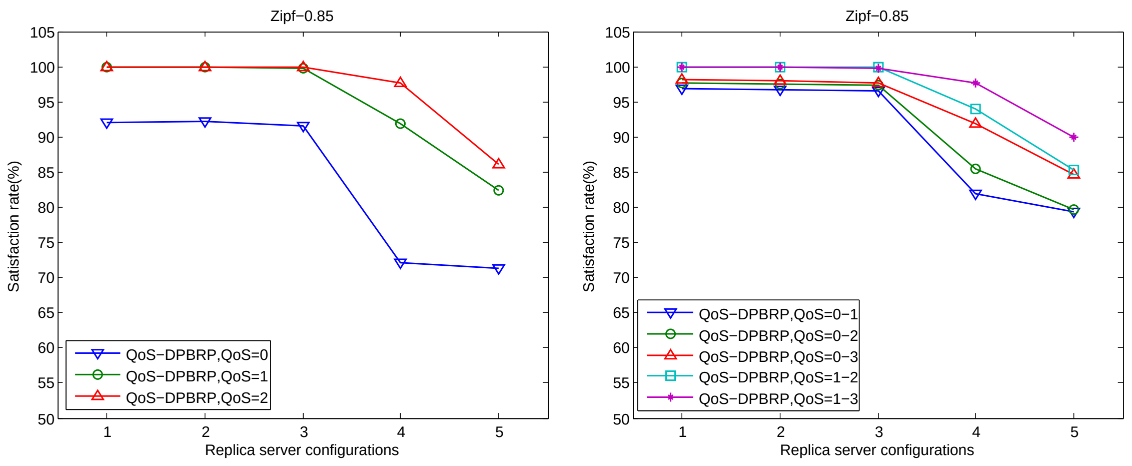


Figure 5.25: Satisfaction rates, Zipf-0.85 access, Constant and uniform QoS constraints

more replicas than DPBRP. Somewhat surprisingly, both the algorithms show almost the same execution times for a sequential distribution. When QoS values are taken from a uniform distribution, QoS-DPBRP shows the lowest job time for a QoS range of [1–2]. A moderate number of replicas, as shown in Figure 5.24, are created in the middle of the hierarchy reducing the access latency experienced which in turn reduces the overall job execution time. As before, QoS-DPBRP shows a somewhat increased job time when QoS ranges include requests for creating replicas locally (for example, [0–1], [0–2], and [0–3]).

The storage capacity of the servers has a noticeable impact on the performance of the placement algorithms. With decreasing capacity, the execution times of all methods are increased but by different amounts. Figure 5.26 shows the execution times for the Zipf-0.85 and Gaussian access patterns for the five storage configurations with constant and uniform QoS distributions. QoS-DPBRP shows shorter run times for all the cases when a QoS value of 2 and a range of [1–2] are used. The benefit

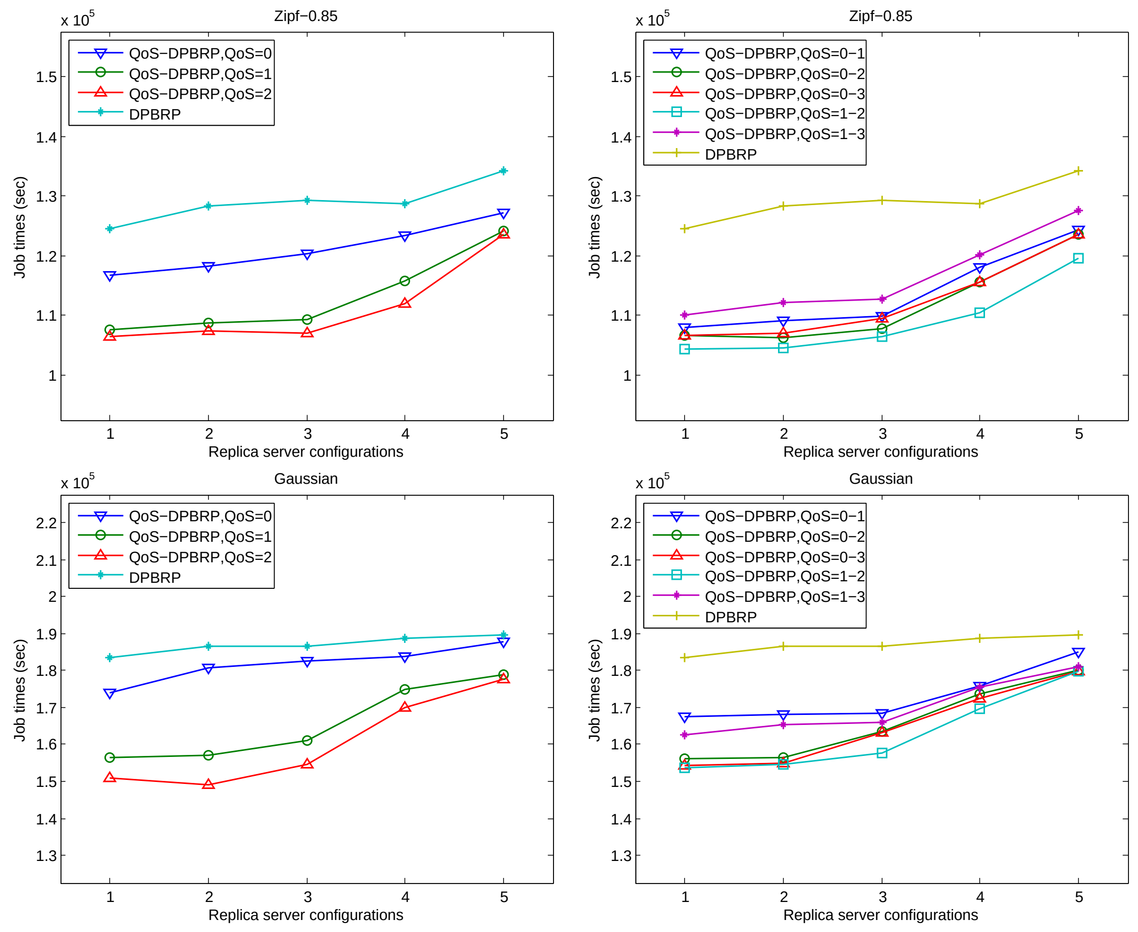


Figure 5.26: Run time, Zipf-0.85 and Gaussian access, Constant and uniform QoS constraints

of QoS-DPBRP over DPBRP in terms of job time reduces with more constrained storage configurations. The number of replicas created also reduces (Figure 5.27) with decreasing storage capacity. Limited storage capacity also results in lower QoS satisfaction rates. Figure 5.25 shows the satisfaction rates of client requests for QoS values from constant and uniform distributions using the Zipf-0.85 distribution. The satisfaction rates are higher for QoS requests of wider range and not including requests for creating local replicas (for example, QoS requests of [1–3], [1–2], and [2]). With

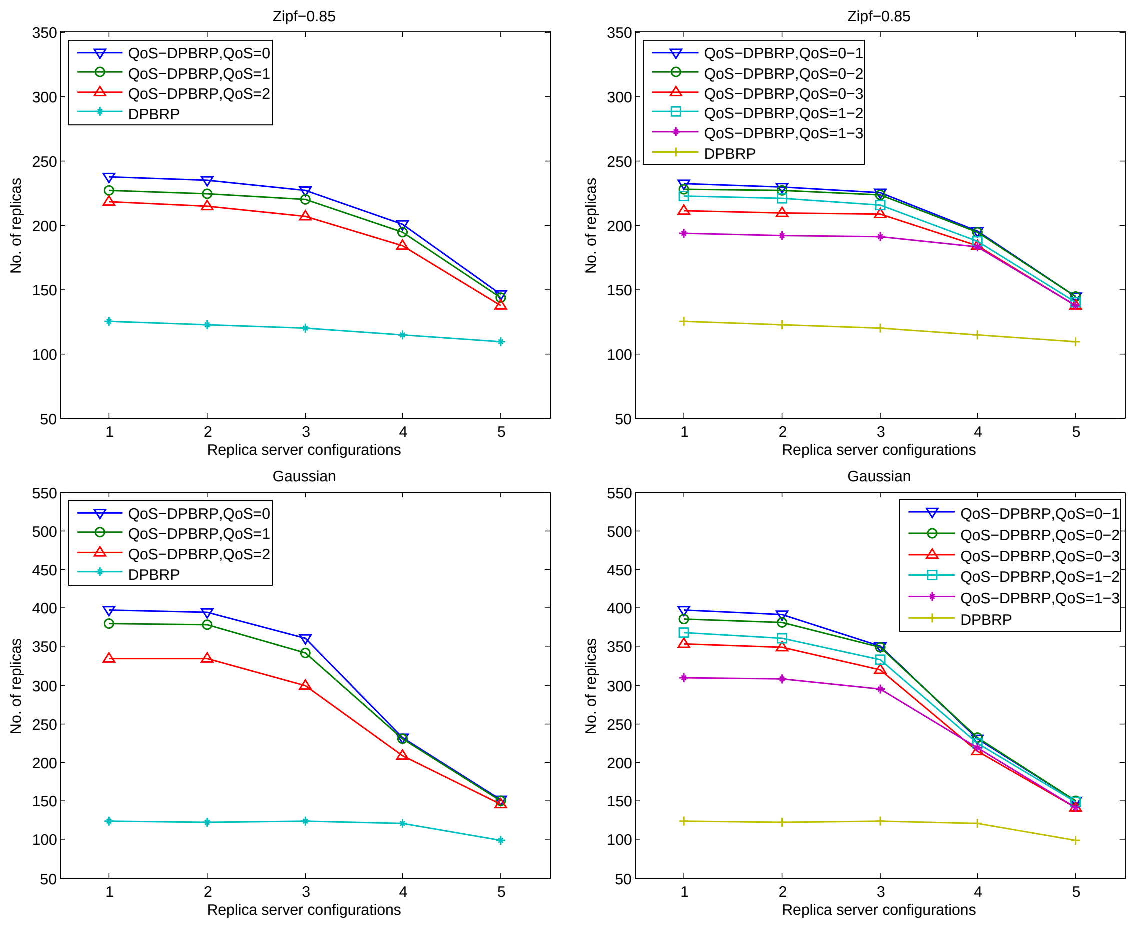


Figure 5.27: No. of replicas, Zipf-0.85 and Gaussian access, Constant and uniform QoS constraints

limited storage (i.e. configurations 4 and 5), the satisfaction rate drops significantly.

Effects of user QoS, varying data access rate

Figure 5.28 compares job execution times for all storage configurations for QoS values from constant and uniform distributions as the access rate fluctuates. With decreasing storage size, the execution times are increased in most cases for all the algorithms but by different amounts. As before, QoS-DPBRP shows shorter run times for all storage configurations for QoS values of [1–2] and 2.

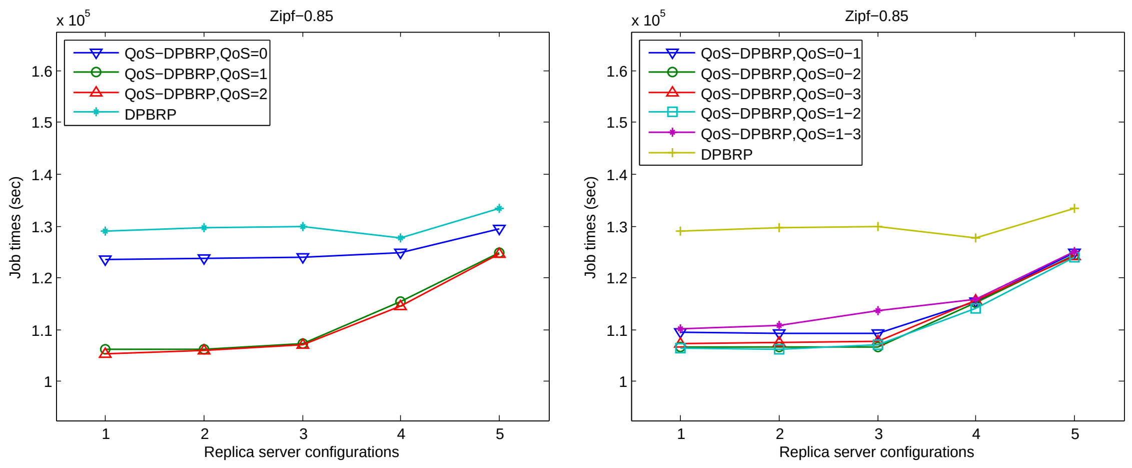


Figure 5.28: Run time, Varying Zipf-0.85 access, Constant and uniform QoS constraints

The popularity of files varies due to the fluctuation in data access rate. The resulting performance is largely unchanged from the performance when the access rate is constant as the fluctuation in access rate does not significantly affect the replicas that need to be created. Figure 5.29 shows this for all storage configurations for the Zipf-0.85 distribution.

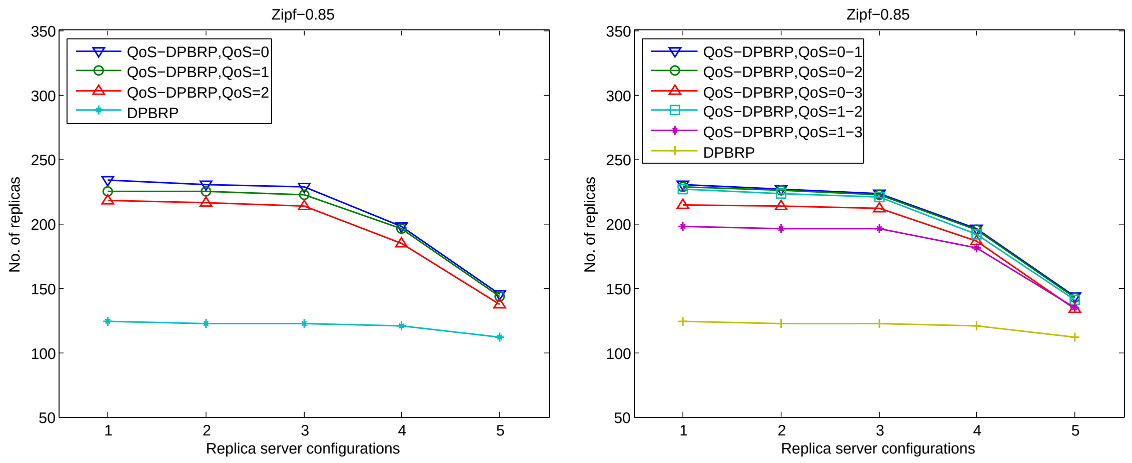


Figure 5.29: No. of replicas, Zipf-0.85 access, Fluctuating rate

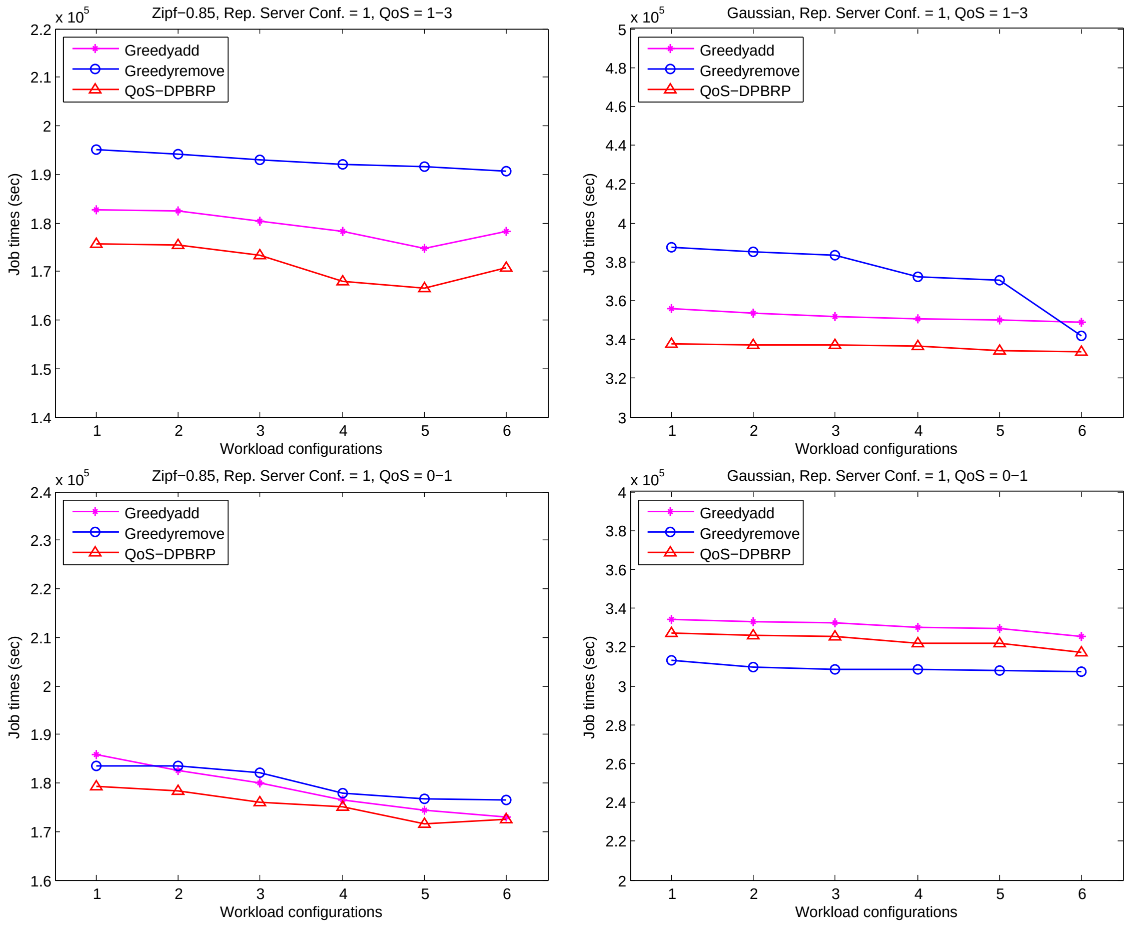


Figure 5.30: Run time, Zipf-0.85 and Gaussian access, Replica server configuration one

Effects of workload capacity constraints for constant access rate

I have chosen *Greedy Add* and *Greedy Remove* [CWL09] to compare with DPBRP since both the methods take update and access costs of replicas into account, and also consider that the workload capacity of a replica server is bounded as DPBRP does. For a fair comparison of the algorithms, a number of assumptions were made while simulating *Greedy Add* and *Greedy Remove*. First, only a hierarchical grid topology was considered even though these two algorithms were originally targeted for a general

Workload config.	Tier 1 (GB per node)	Tier 2 (GB per node)	Tier 3 (GB per node)
1	350	300	250
2	300	250	200
3	250	200	150
4	200	150	100
5	300	150	80
6	150	100	50

Figure 5.31: Different workload configurations

grid model (being aware of the fact that this may cause performance impact on the algorithms). Second, the storage capacities of the replica servers were assumed to be limited and the same replica replacement policy was used as discussed in Section 4.2. The detailed workload configurations for the replica servers in different tiers of the grid hierarchy are shown in Figure 5.31. For example, in configuration 1, each server in tier-1, tier-2, and tier-3 have workload capacities of 350 GB, 300 GB, and 250 GB, respectively. This means that a tier-2 server can process client requests of data for a maximum size of 300GB for a sampling period.

Now I compare (Figure 5.30) the job execution times of DPBRP, *Greedy Add*, and *Greedy Remove* using storage configuration one for the Zipf-0.85 and Gaussian distributions as the workload capacity of replica servers is varied. User QoS requirements are taken from both relaxed and relatively more constrained ranges of [1–3] and [0–1], respectively. DPBRP shows the shortest execution times compared to the other algorithms for almost all cases. This is because the replica locations selected in DPBRP reduce the data access latency experienced which, in turn, reduces the overall job execution time. *Greedy Add* shows better execution time than *Greedy Remove* in most cases. For the QoS constraint [0–1], *Greedy Remove* surprisingly shows better performance than the other two algorithms when the Gaussian access pattern is used. In this case, the replica locations determined by *Greedy Remove* is relatively more scattered in the lower tiers of the hierarchy due to the constrained QoS requirements. Consequently, the randomness in data access pattern results in reduced job execution times. The performance difference between DPBRP and the other algorithms is more evident when relaxed user QoS requirements are considered.

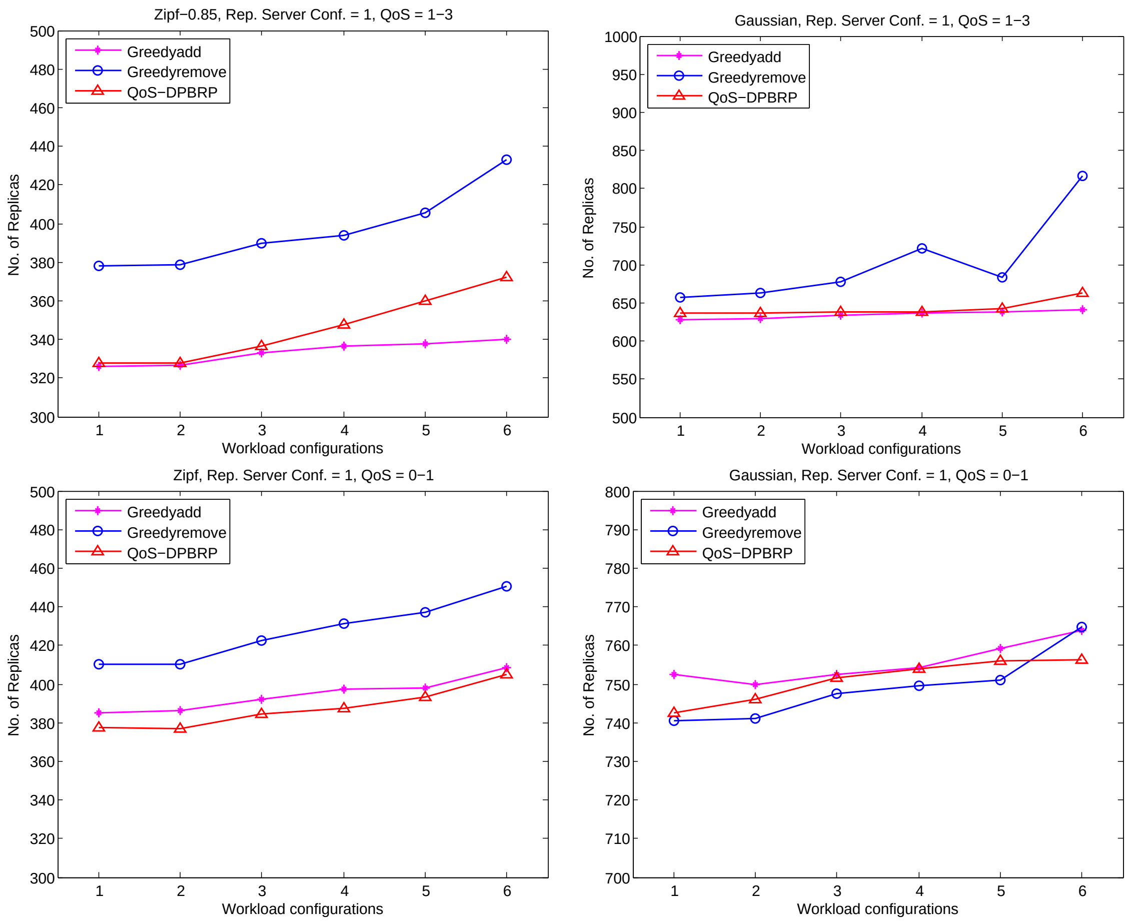


Figure 5.32: No. of replicas, Zipf-0.85 and Gaussian access

DPBRP creates (Figure 5.32) a moderate number of well-placed replicas compared to *Greedy Remove* and *Greedy Add* which, as a result, satisfies more user requests and reduces the access latency. Figure 5.33 shows the user satisfaction rate for all the algorithms. DPBRP consistently shows better performance in all cases. The satisfaction rates are higher for QoS requests of wider range. When the workload capacity of replica servers decreases, the average number of replicas is increased and consequently job times are improved in all cases but by varying amounts.

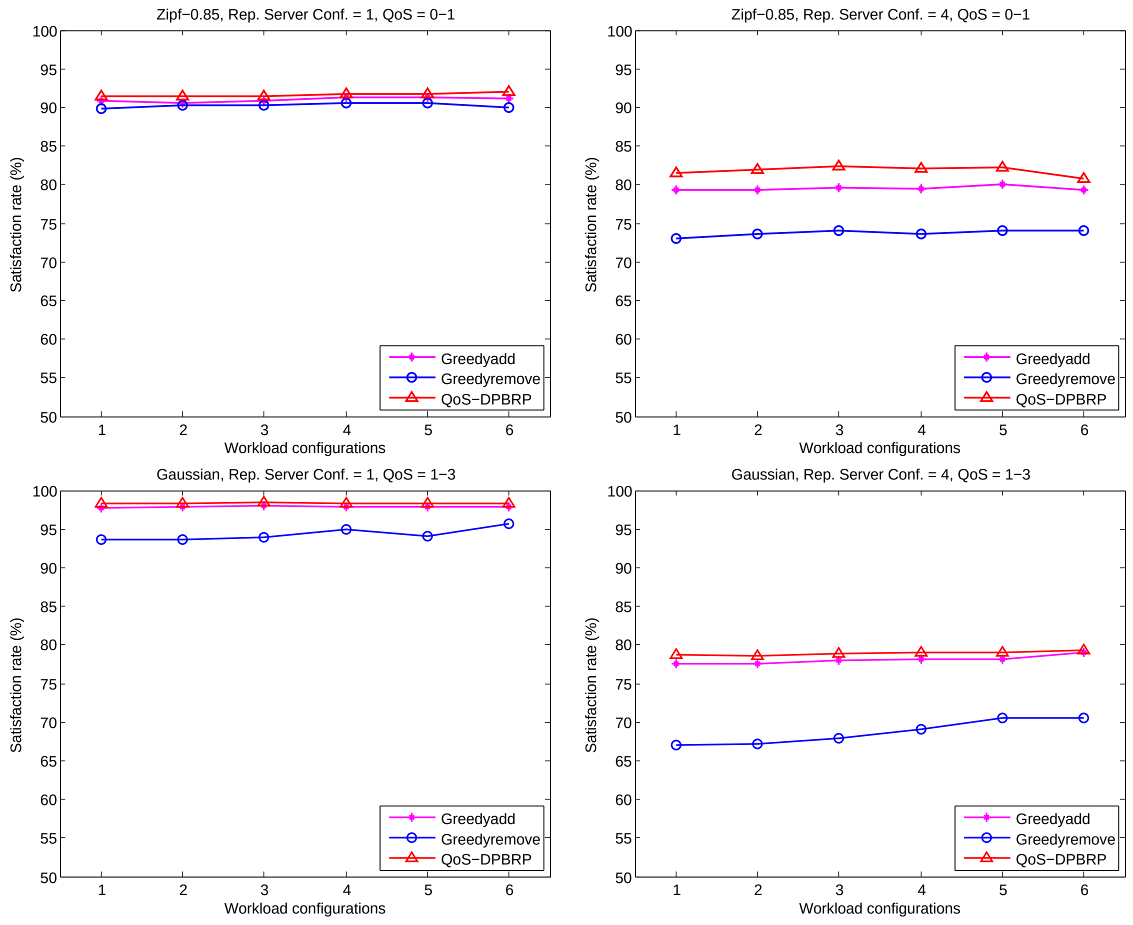


Figure 5.33: Satisfaction rates, Zipf-0.85 and Gaussian access

Figure 5.34 shows the job execution times under different user QoS requirements from constant and uniform distributions for the Zipf-0.85 and Gaussian access patterns using replica storage and workload configurations one (sufficient storage in all tiers) and four (moderate workload capacities in all tiers) respectively. In the case of Zipf-0.85, DPBRP outperforms *Greedy Remove* and *Greedy Add* for both types (constant and uniform) of QoS requirements. The performance difference between DPBRP and the other algorithms increases for relaxed QoS requirements (i.e. [1–2] and [1–3]). *Greedy Add* shows consistently better execution time than *Greedy Remove* for Zipf-0.85 except for the QoS constraint of 3. The average number of replicas that

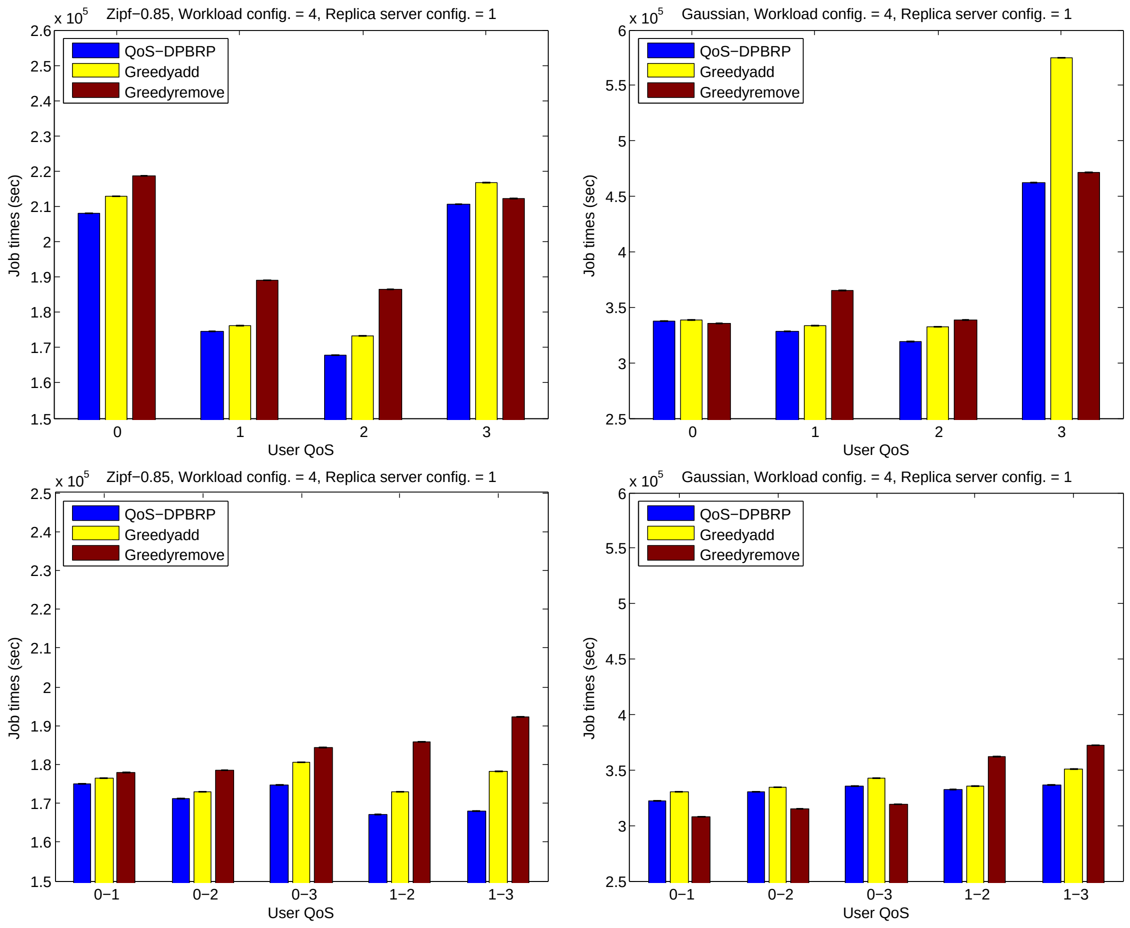


Figure 5.34: Run time, Zipf-0.85 and Gaussian access, Constant and uniform QoS constraints

Greedy Add creats in this case is significantly less (Figure 5.35) than *Greedy Remove* which in turn increases the job time. This is because the first stage of *Greedy Add* tries to find a feasible solution with less replicas and in most cases the second stage (i.e. reducing replication cost) never runs due to the specification of the least constrained QoS request. As for the Gaussian distribution, DPBRP performs better than the other algorithms for constant user QoS values. However, *Greedy Remove* shows shorter job times for relatively constrained ranges of QoS requests (i.e. [0–1],[0–2], and [0–3]).

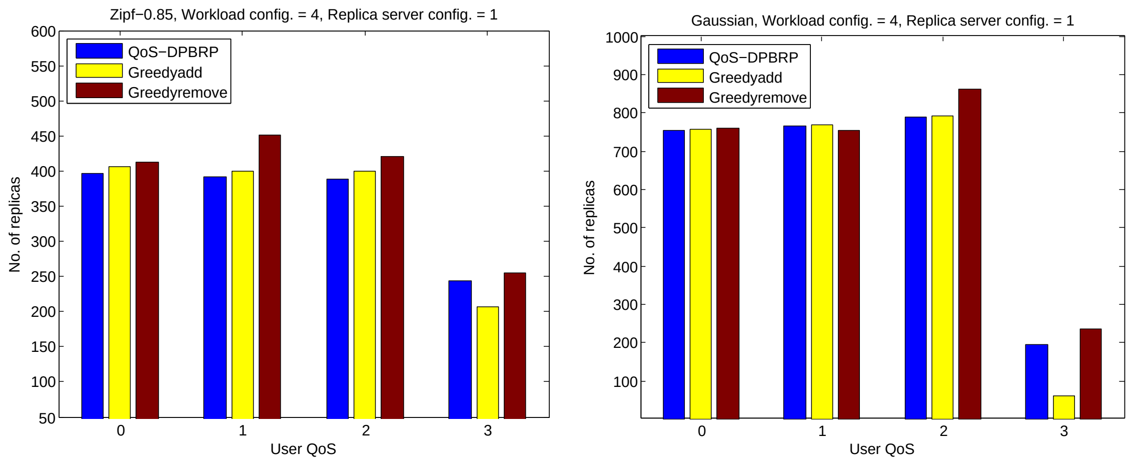
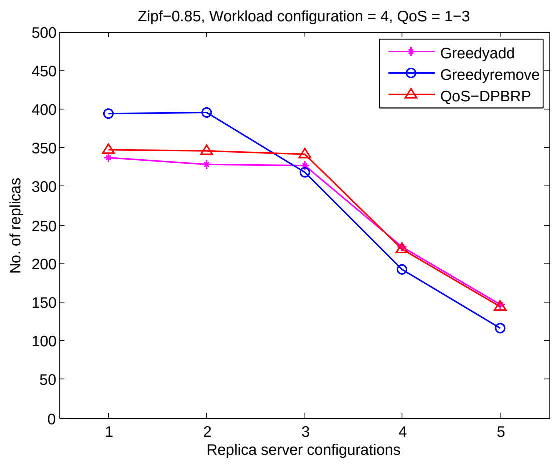


Figure 5.35: No. of replicas, Zipf-0.85 and Gaussian access, Constant QoS constraints

With decreasing storage capacity of replica servers, the execution times of all methods are increased but by different amounts. Figure 5.37 shows the execution times for the Zipf-0.85 and Gaussian access patterns for storage configuration four. DPBRP shows shorter run times for all the cases. However, the benefit of DPBRP over the other algorithms in terms of job time reduces when faced with more constrained storage configurations. Limited storage capacity also results in lower user QoS sat-



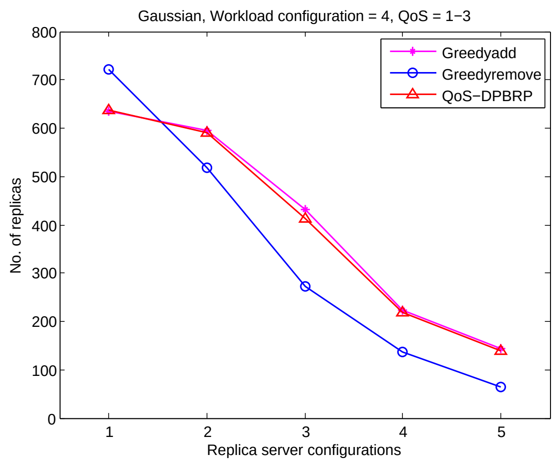


Figure 5.36: No. of replicas, Zipf-0.85 and Gaussian access, All storage configurations

isfaction rates. Figure 5.33 (right) shows the satisfaction rates of client requests for storage configuration four. Similarly, the number of replicas created decreases with the decrease of available storage capacity. Figure 5.36 shows the number of replicas created for all five storage configurations using workload configuration four.

Effects of link capacity constraints for constant access rate

Figure 5.38 compares job execution times using storage configuration one for uniform QoS values as the reserved link capacities for data transfer are varied from 10% reserved to 100%. With increasing reserved link capacity, the execution times are decreased for all the variants but by different amounts. DPBRP, however, shows exception to this trend for the Gaussian pattern. This is due to the significant drop in the number of replicas that are created (as shown in Figure 5.39). Figure 5.39 shows that the average number of replicas drops for DPBRP with the increase in link capacity while QoS-DPBRP does not show sigficant variation in the number of replicas created for all QoS values. The reason is that additional replicas are created

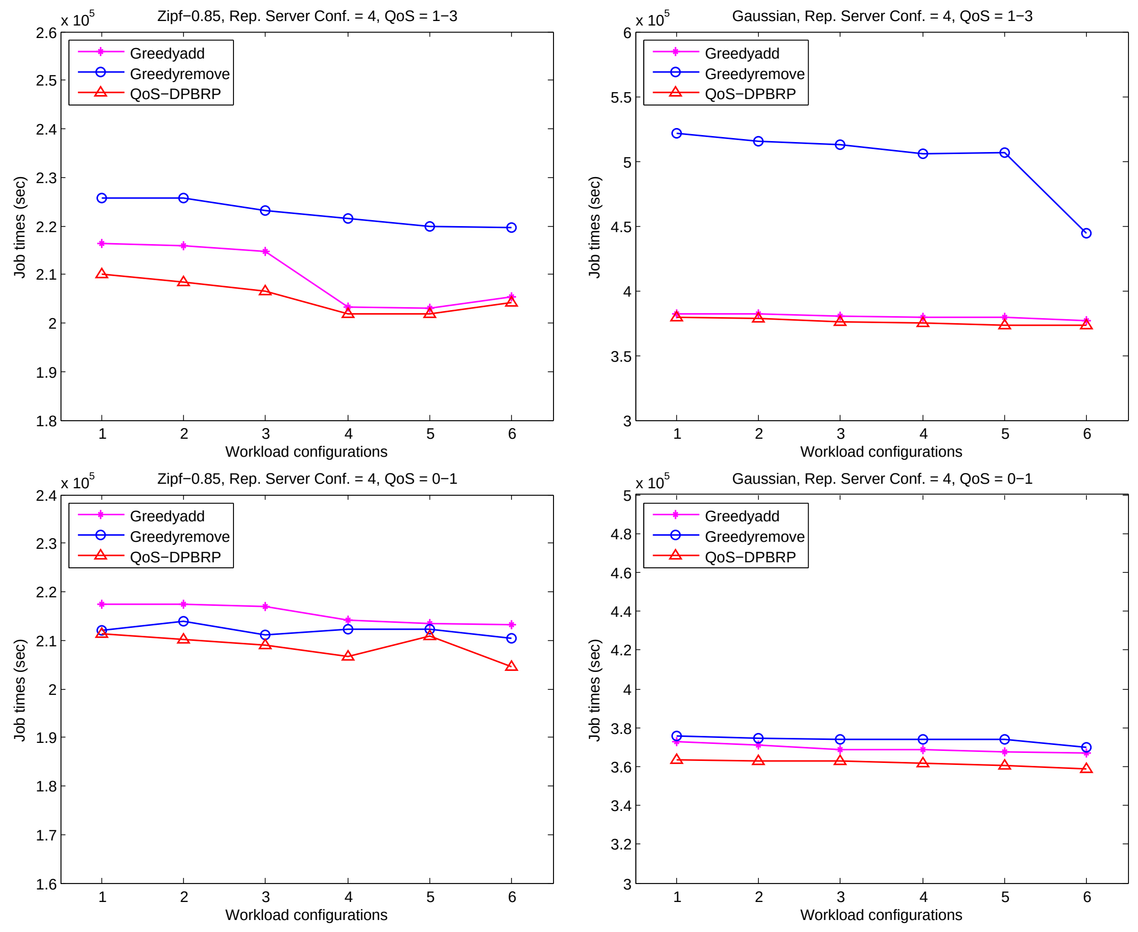


Figure 5.37: Run time, Zipf-0.85 and Gaussian access, Replica server configuration four

due to constrained user QoS which results in the creation of more replicas even with the increase of link capacity. As before, QoS-DPBRP shows shorter run times in most cases for QoS requests of [0–2].

I also tested the scenario when both workload and link constraints are considered. The resulting performance is largely unchanged from the performance when either workload or link capacity constraint is considered in addition to user QoS because this does not significantly affect the replicas that need to be created. Figure 5.40

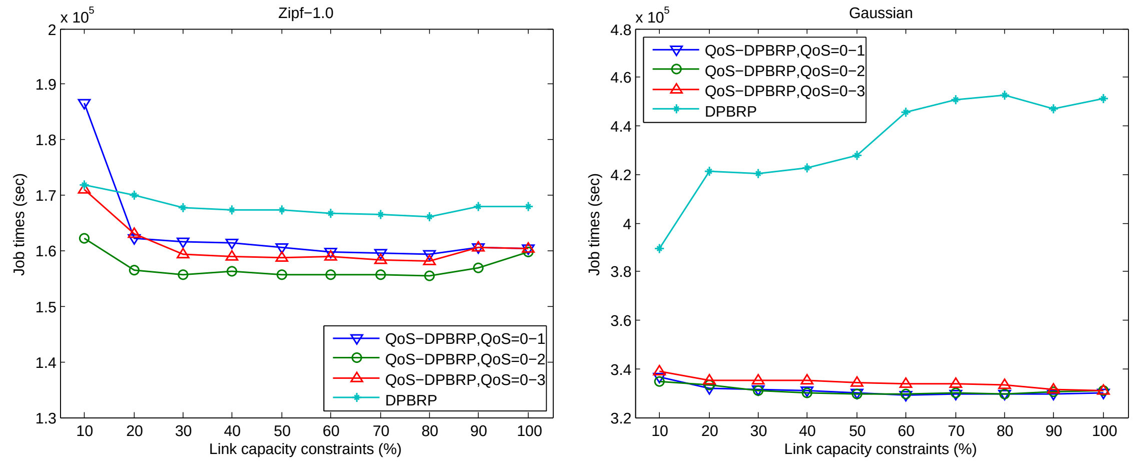


Figure 5.38: Run time with link constraints, Uniform QoS constraints

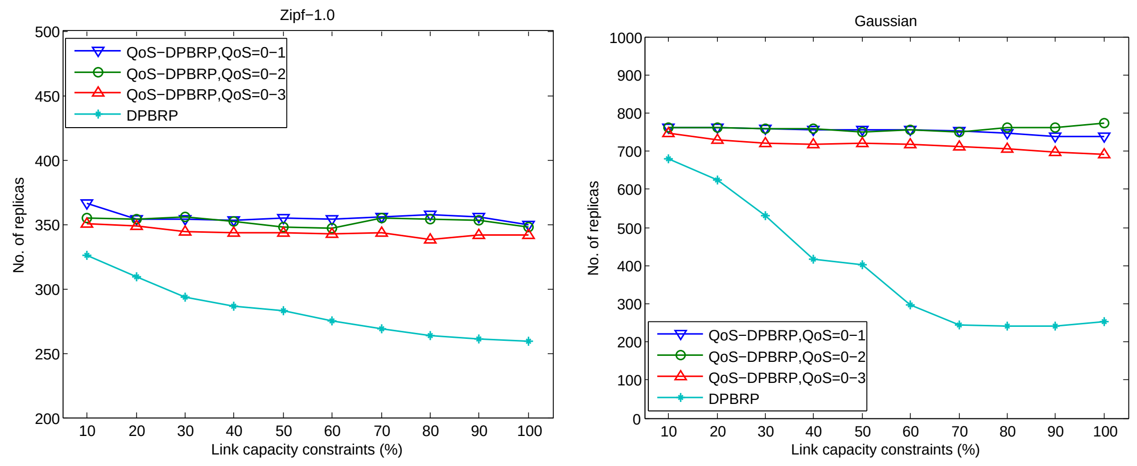


Figure 5.39: No. of replicas with link constraints, Uniform QoS constraints

shows this for all storage configurations for the Zipf-1.0 and Gaussian distributions.

5.5.2 Average Bandwidth Cost

This section analyzes the results on bandwidth consumption considering the effects of both user and systems QoS constraints. In almost all cases, DPBRP consumes somewhat less bandwidth than QoS-DPBRP but at the cost of longer job times and

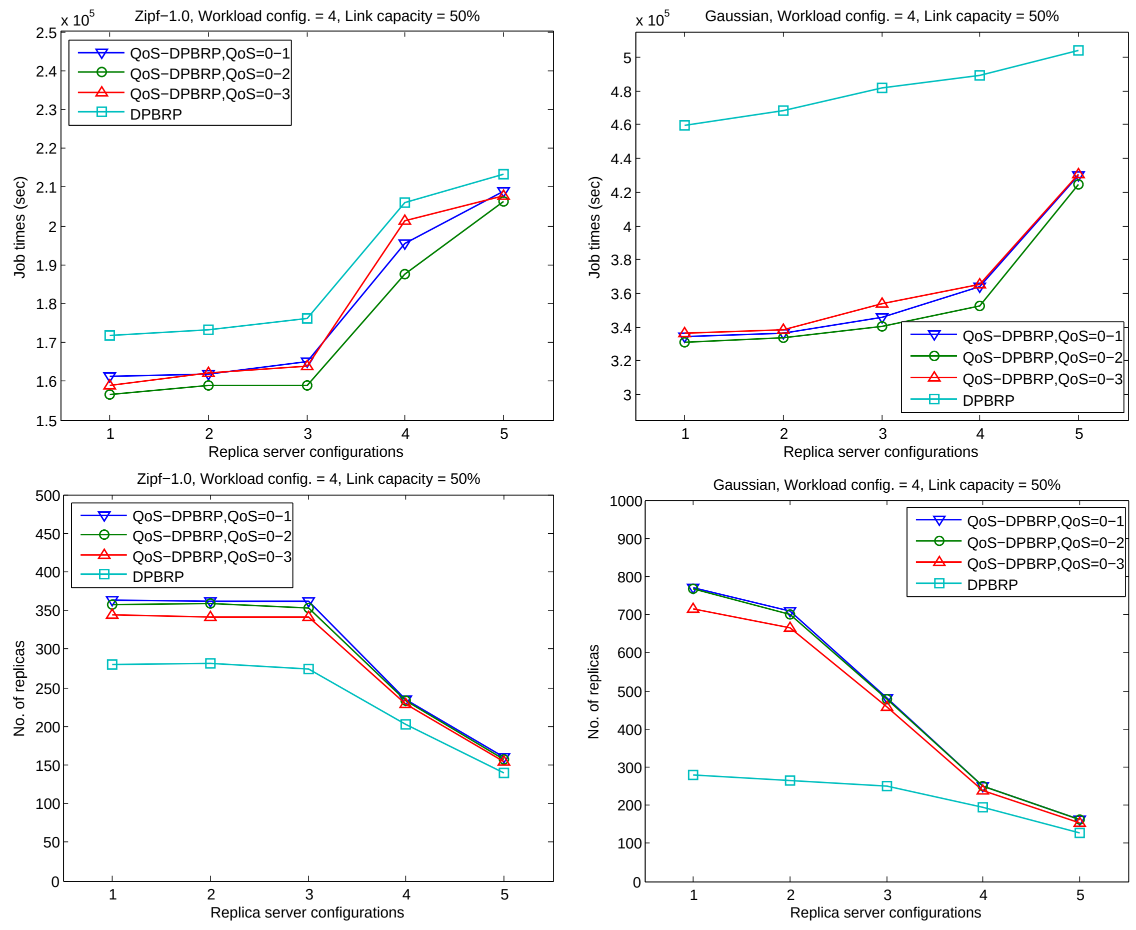


Figure 5.40: Run time with both link and workload constraints (left), No. of replicas (right), Uniform QoS constraints

no guarantee of user QoS satisfaction.

Effects on bandwidth consumption of user QoS for constant data access rate

Figure 5.41 shows the average bandwidth cost for configuration one. For all access patterns, the bandwidth cost of DPBRP is somewhat less than the QoS-DPBRP variants. This is due to the creation of fewer replicas resulting in less bandwidth for creating and updating the replicas which outweighs the increased bandwidth con-

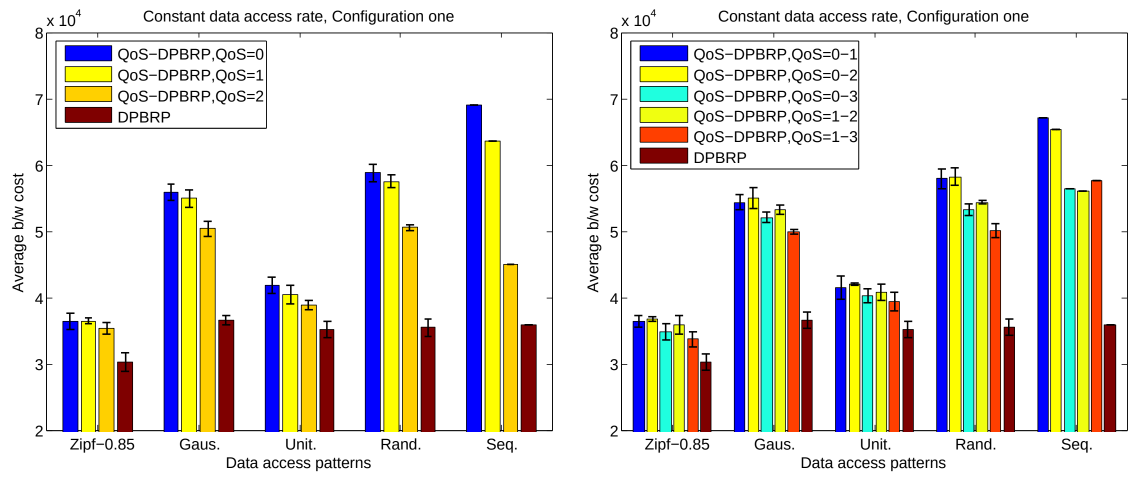


Figure 5.41: Average bandwidth, Configuration one, Constant and uniform QoS constraints

sumption for nearby data access.

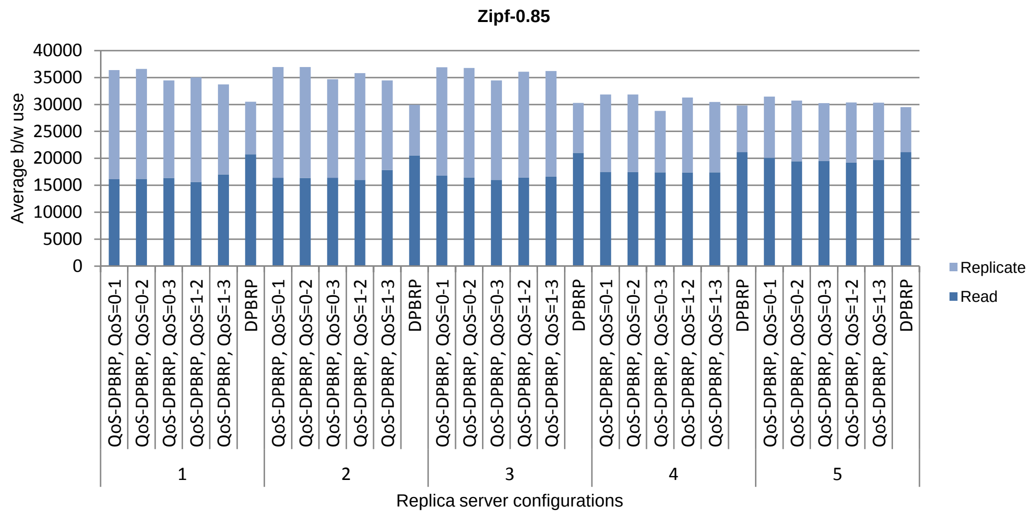


Figure 5.42: Read and replication cost for Zipf-0.85 for uniform QoS values

Figure 5.42 shows the replication (create and update) and read costs for the Zipf-0.85 access pattern when QoS values are taken from a uniform distribution. QoS-DPBRP shows less read cost for all configurations when a QoS range of [1–2] is used which results in faster execution, as described earlier. In this case, more client requests are served by better placed replicas in mid-tier nodes. As a result, the bandwidth used for data access is decreased. However, in this case, read cost is outweighed by replication cost. A QoS constraint of [1–2] shows moderate overall bandwidth cost for all storage configurations. The same is true for a constant QoS requirement of 2.

Figures 5.43 and 5.44 show the bandwidth cost for all storage configurations. DPBRP has less average bandwidth consumption for all configurations. Among the QoS-DPBRP variations, QoS of 2 and a range of [1–3] show less average bandwidth cost due to lower read cost and replication cost, respectively. With limited storage capacity the bandwidth costs of all strategies are significantly reduced due to the creation of fewer replicas.

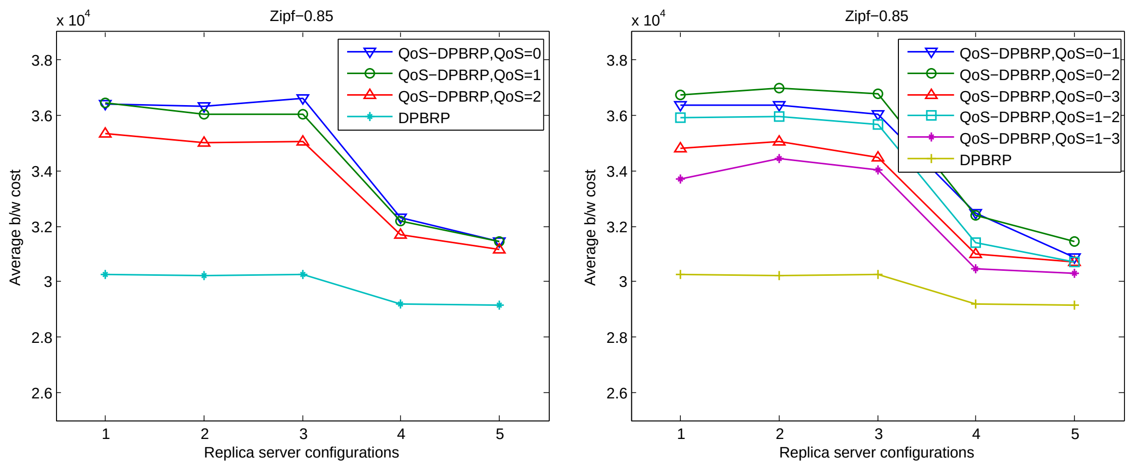


Figure 5.43: Average bandwidth, Zipf-0.85 access, Constant and uniform QoS constraints

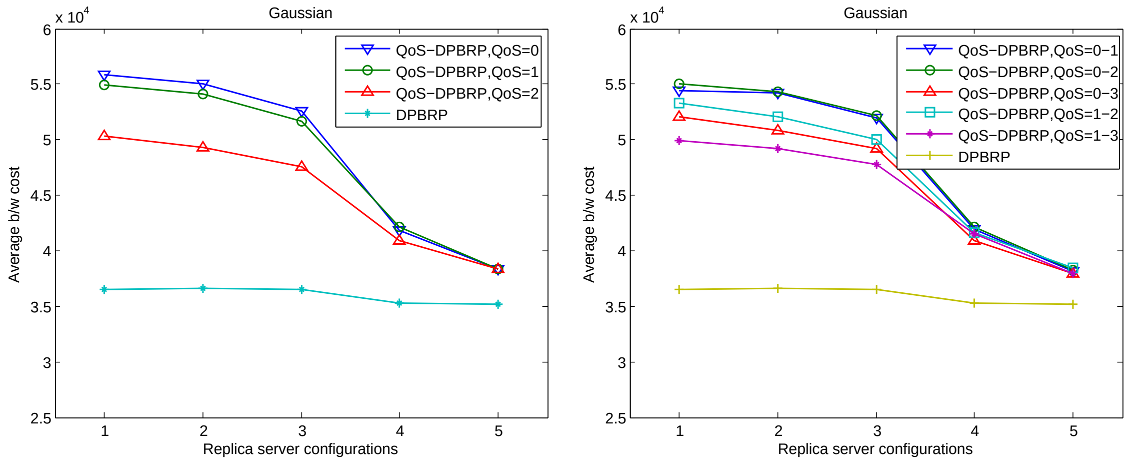


Figure 5.44: Average bandwidth, Gaussian access, Constant and uniform QoS constraints

Effects on bandwidth consumption of user QoS, varying data access rate

The average bandwidth cost for all storage configurations with fluctuating access request rates is shown in Figure 5.45. With decreasing storage size, bandwidth costs decrease for all algorithm variants but by different amounts. As before, DPBRP has somewhat less average bandwidth consumption for all storage configurations. The performance of the QoS-DPBRP variants are, again, consistent with DPBRP when the access rate is constant due to creation of a similar number of replicas. Figure 5.29 reflects this for all storage configurations for the Zipf-0.85 distribution.

Effects on bandwidth consumption of workload capacity constraints for constant access rate

Figure 5.46 shows the average bandwidth cost for storage configuration one when the workload capacities of replica servers are varied. For both access patterns, the bandwidth cost of DPBRP is moderate among the studied algorithms and in some

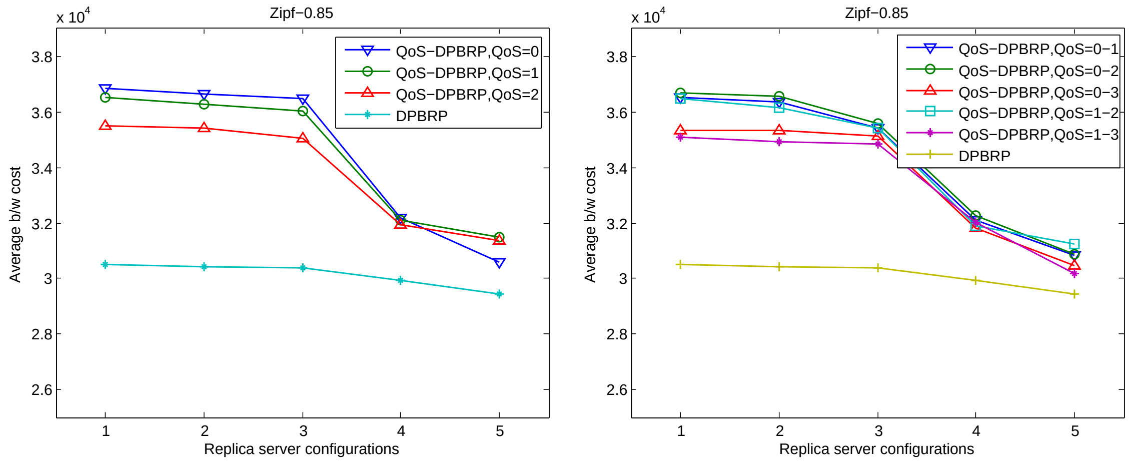


Figure 5.45: Average bandwidth, Varying Zipf-0.85 access, Constant and uniform QoS constraints

cases (for example, when the Zipf-0.85 distribution and QoS range of [0–1] are used) DPBRP shows the lowest bandwidth consumption. This is due to the creation of a moderate number of better placed replicas down the hierarchy resulting in less bandwidth consumption for data access. *Greedy Remove* shows the lowest bandwidth cost in most cases due to the creation of a large number of replicas up in the hierarchy which results in a low replication (create and update) cost outweighed by a higher read cost. When the workload capacity of replica servers decreases, the bandwidth cost increases in all cases due to the creation of an increased number of replicas.

Figure 5.47 shows the replication (create and update) and read costs for the Zipf-0.85 access pattern and user QoS values of [1–3]. DPBRP shows less read cost for all workload configurations which results in faster job execution, as mentioned before. In this case, more client requests are served by better placed replicas in mid-tier nodes. As a result, the bandwidth used for data access is decreased.

Figures 5.48 shows the bandwidth cost for storage configuration four for the Zipf-

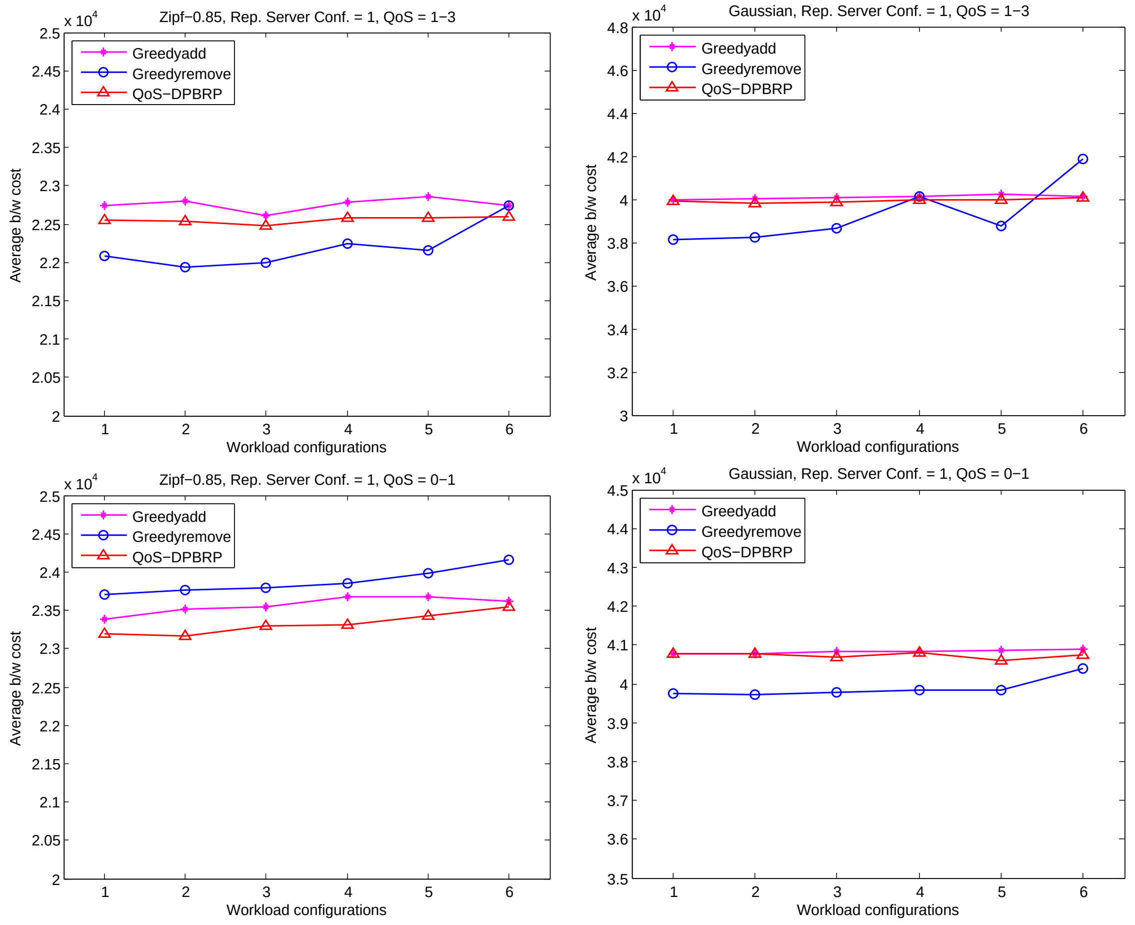


Figure 5.46: Average bandwidth cost, Zipf-0.85 and Gaussian access, Replica server configuration one

0.85 and Gaussian access patterns. DPBRP has moderate bandwidth consumption for all workload configurations. *Greedy Remove* shows less average bandwidth cost due to lower replication cost when the Gaussian distribution is used. *Greedy add*, however, performs better than the other algorithms for the Zipf-0.85 distribution under conditions of limited storage capacity. The bandwidth costs of all strategies are significantly reduced with limited storage capacity due to the creation of fewer replicas and user QoS variants of larger ranges show less overall bandwidth consumption.

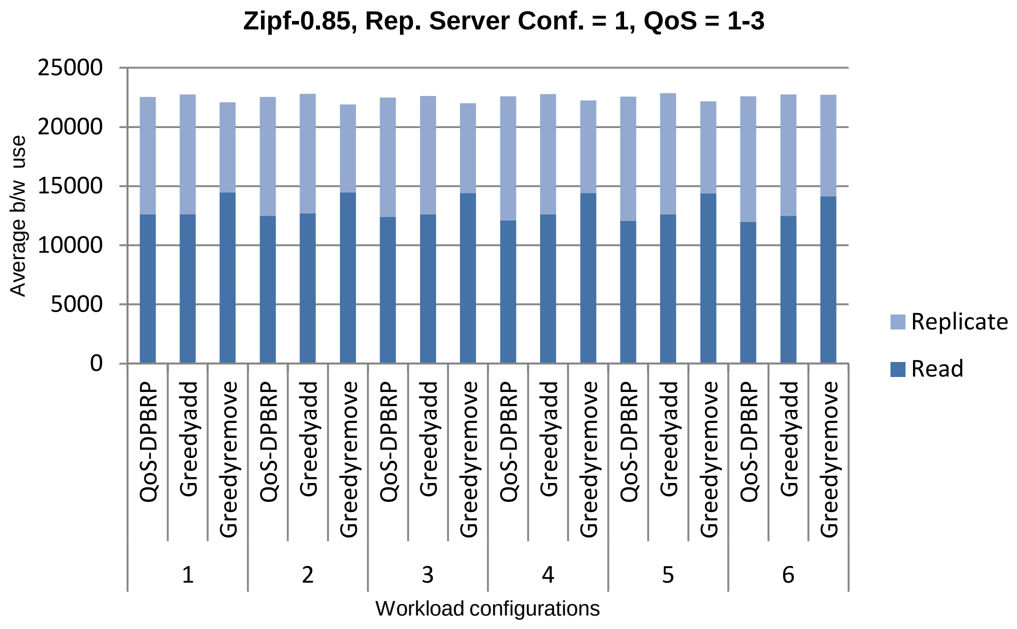


Figure 5.47: Read and replication cost for Zipf-0.85 for a QoS value of [1–3]

Effects on bandwidth consumption of link capacity constraints for constant access rate

Figure 5.49 compares bandwidth cost using storage configuration one for uniform QoS values as the reserved link capacities for data transfer are varied from 10% to 100% reservation. With increasing link capacity, bandwidth costs decrease for all algorithm variants but by different amounts. As before, DPBRP has less average bandwidth consumption for all cases. As before, among the QoS-DPBRP variants, aQoS range of [0–3] shows less average bandwidth cost due to lower replication cost. The performance of the QoS-DPBRP variants are, again, consistent with DPBRP for the scenario when both workload and link constraints are considered due to creation of a similar number of replicas. Figure 5.50 reflects this for all storage configurations for the Zipf-1.0 and Gaussian distributions.

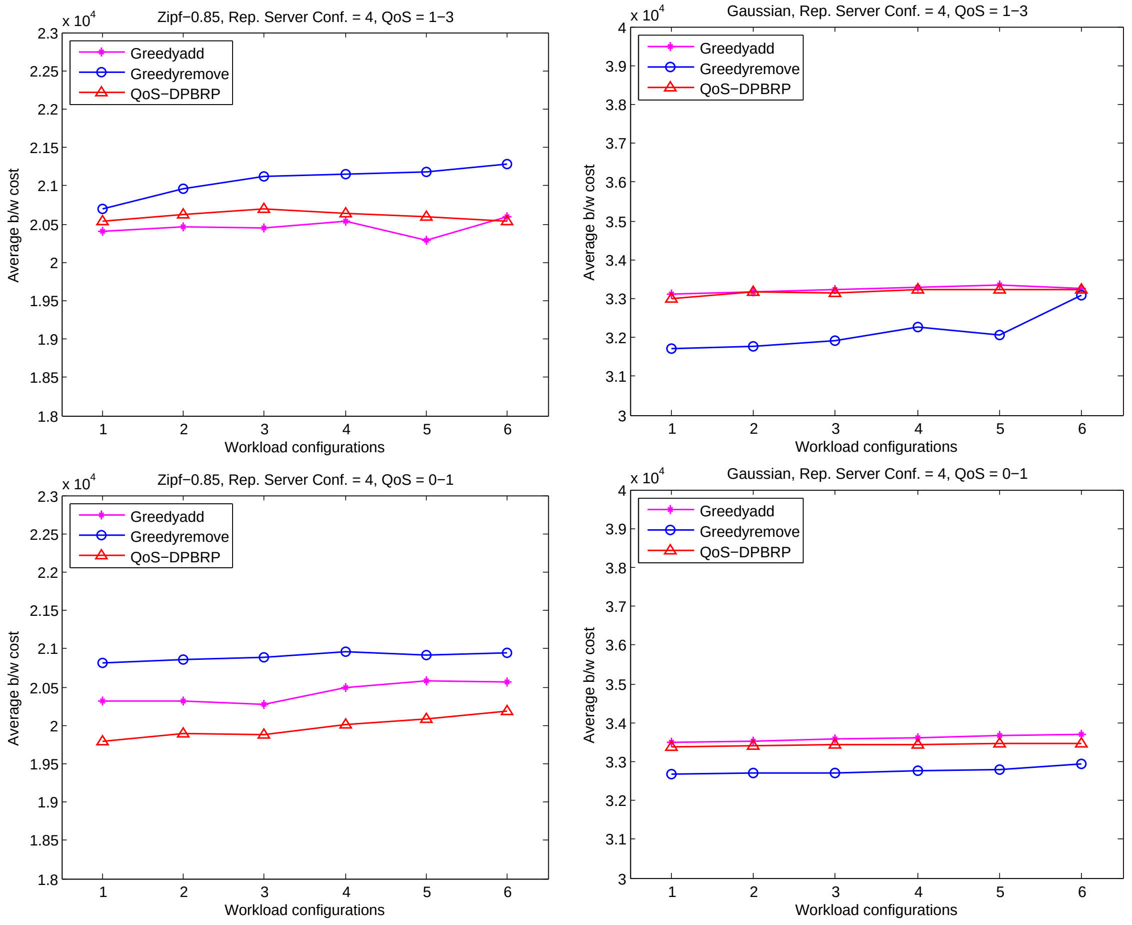


Figure 5.48: Average bandwidth cost, Zipf-0.85 and Gaussian access, Replica server configuration four

5.5.3 Storage Use

As mentioned, the available replica storage capacity greatly impacts the performance of the replication algorithms. The increased use of storage due to its relatively low cost can be traded-off for improvements in job execution time, network bandwidth used, and QoS satisfaction, as needed. The section describes how replica storage utilization is affected by the enforcement of user QoS, workload, and link capacity constraints.

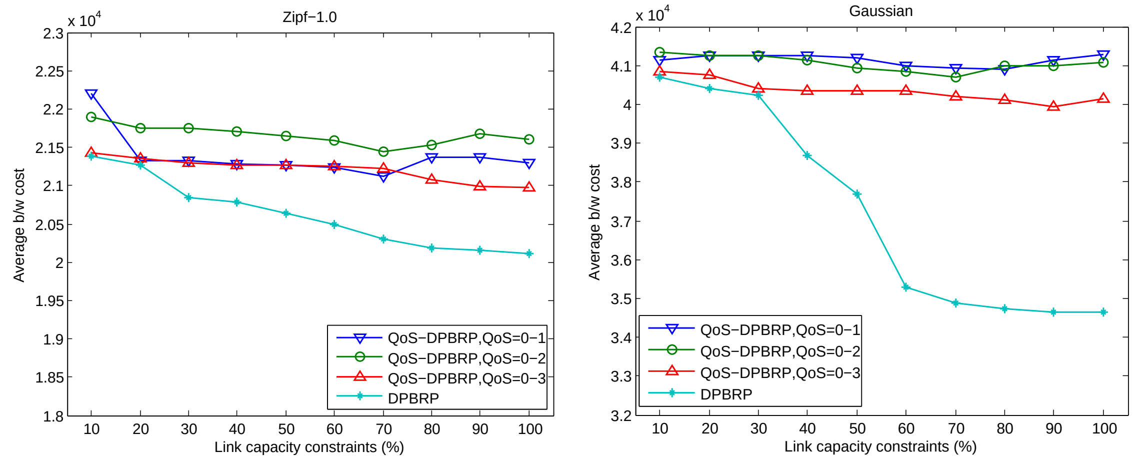


Figure 5.49: Average bandwidth with link constraints, Uniform QoS constraints

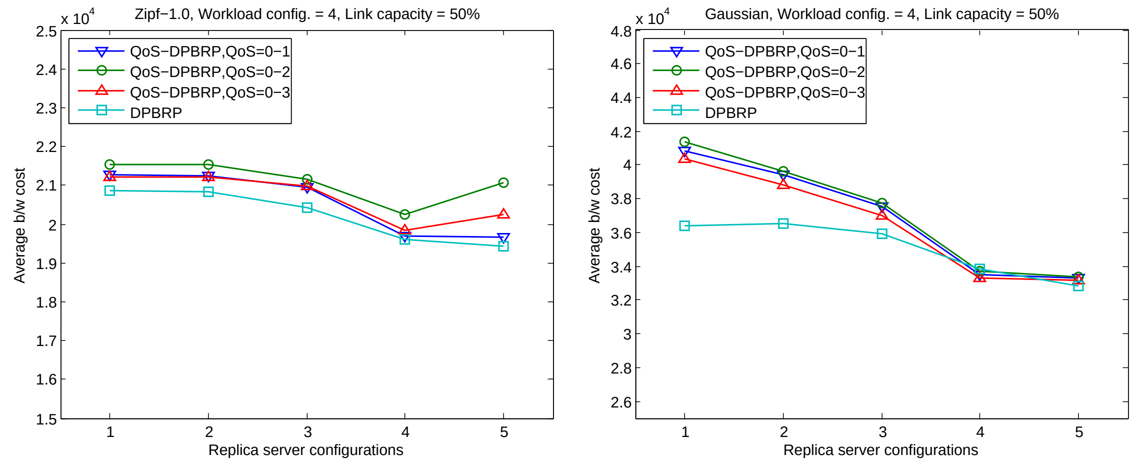
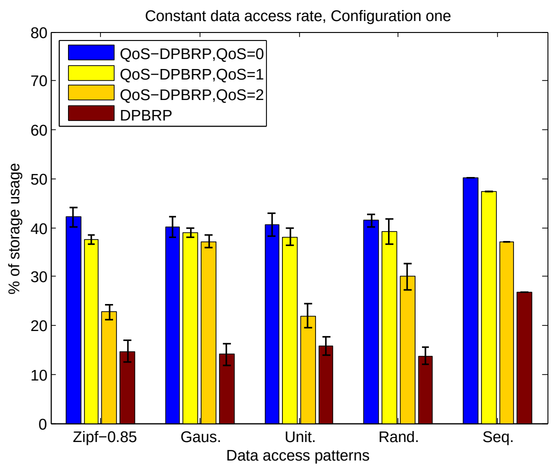


Figure 5.50: Average bandwidth with both link and workload constraints, Uniform QoS constraints

Effects on storage use of user QoS, constant data access rate

The storage used by the various placement schemes for configuration one are shown in Figure 5.51. For all access patterns, the storage cost of DPBRP is lower than QoS-DPBRP due to the fewer number of replicas created.



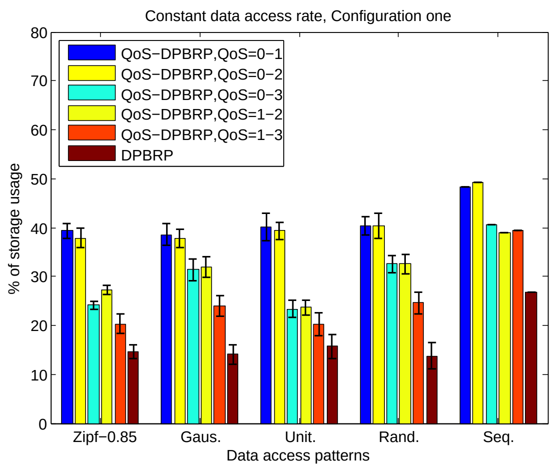
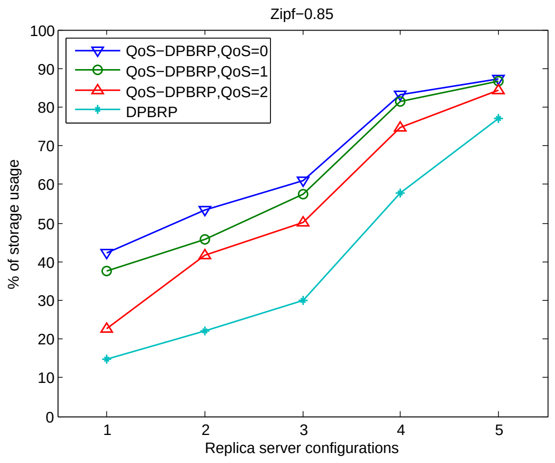


Figure 5.51: Storage cost, Configuration one, Constant and uniform QoS constraints



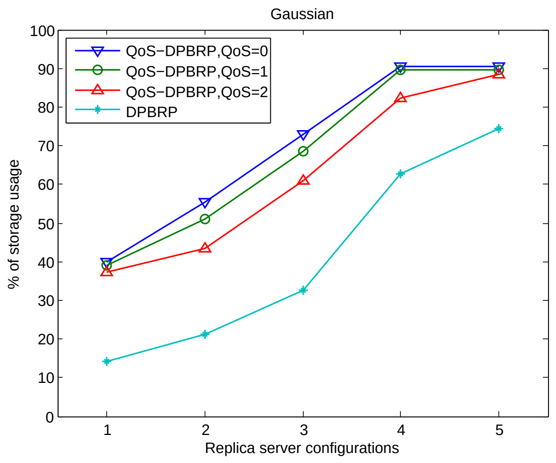


Figure 5.52: Storage cost, Zipf-0.85 and Gaussian access, Constant QoS constraints

The capacity of the replica servers has a clear and expected impact on the percentage of storage used by both DPBRP and QoS-DPBRP as shown in Figures 5.52 and 5.53. One might expect that the reduction of storage size should lead to 100% use of storage but some grid sites use their storage completely while others might not use any space at all, depending on data access patterns. The benefit of DPBRP in terms of storage used is offset by its higher job execution time. QoS-DPBRP effec-

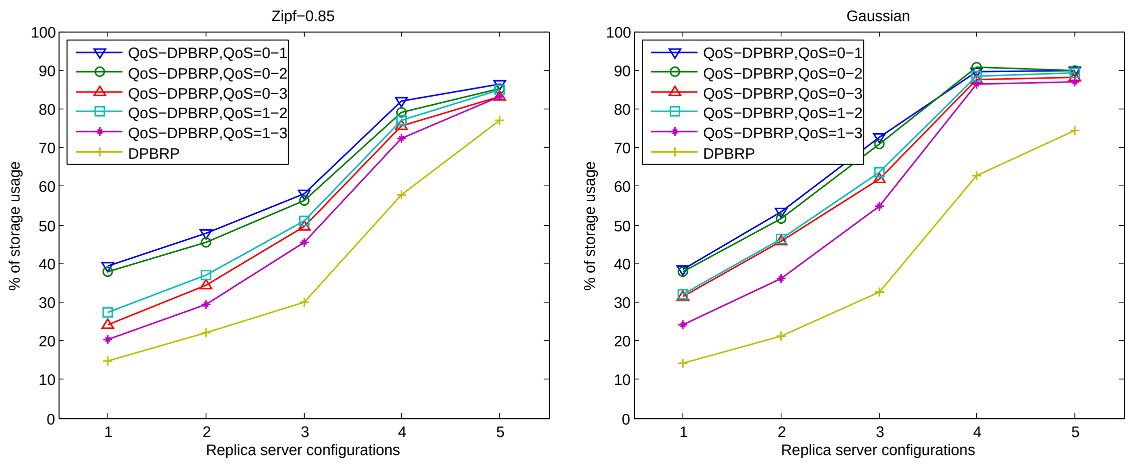


Figure 5.53: Storage cost, Zipf-0.85 and Gaussian access, Uniform QoS constraints

tively trades off additional storage used to achieve faster access times while consuming moderate bandwidth.

Effects on storage use of user QoS, varying data access rate

Figure 5.54 compares the storage use of my placement algorithms for all storage configurations when the request rate fluctuates. With decreasing storage availability, the percentage of storage used naturally increases but by different amounts. DPBRP shows less storage requirements for all storage configurations. In most cases, the cost of QoS-DPBRP for such demanding QoS constraints of 0 and [0–1] is higher due to its creation of an increased number of replicas.

Effects on storage use of workload capacity constraints, constant access rate

The storage used by the placement schemes for storage configuration one with varied workload capacities is shown in Figure 5.55. For both access patterns, the cost

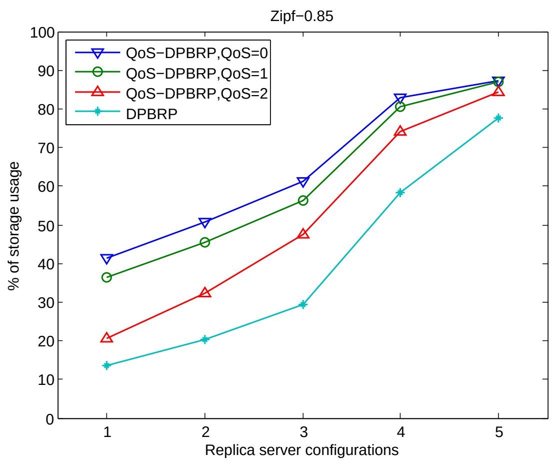


Figure 5.54: Storage cost, Varying Zipf-0.85 access, Constant and uniform QoS constraints

of DPBRP is moderate among the studied algorithms corresponding to a moderate number of replicas created. When the workload capacity of replica servers decreases, the storage overhead increases in all cases due to the creation of an increased number of replicas.

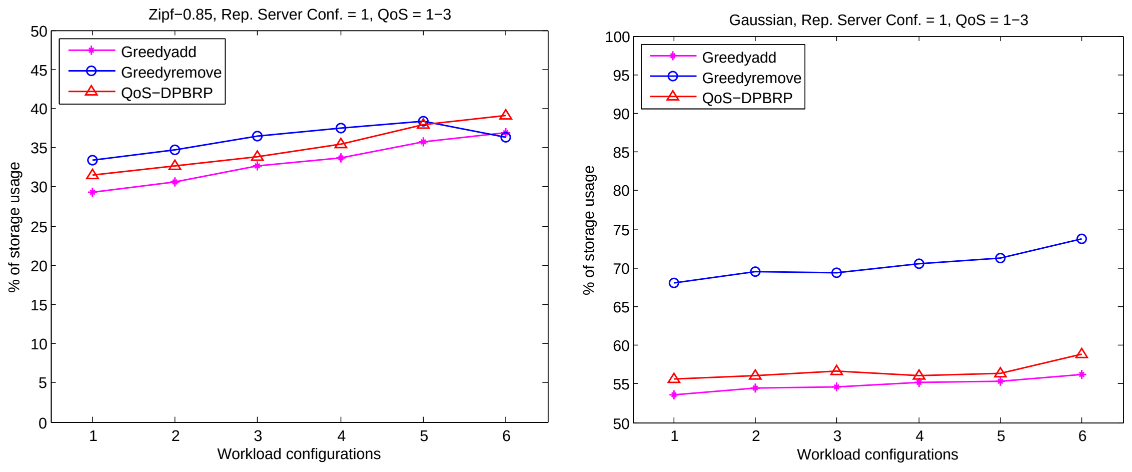


Figure 5.55: Storage cost, Zipf-0.85 and Gaussian access, Replica server configuration one

Effects on storage use of link capacity constraints, constant access rate

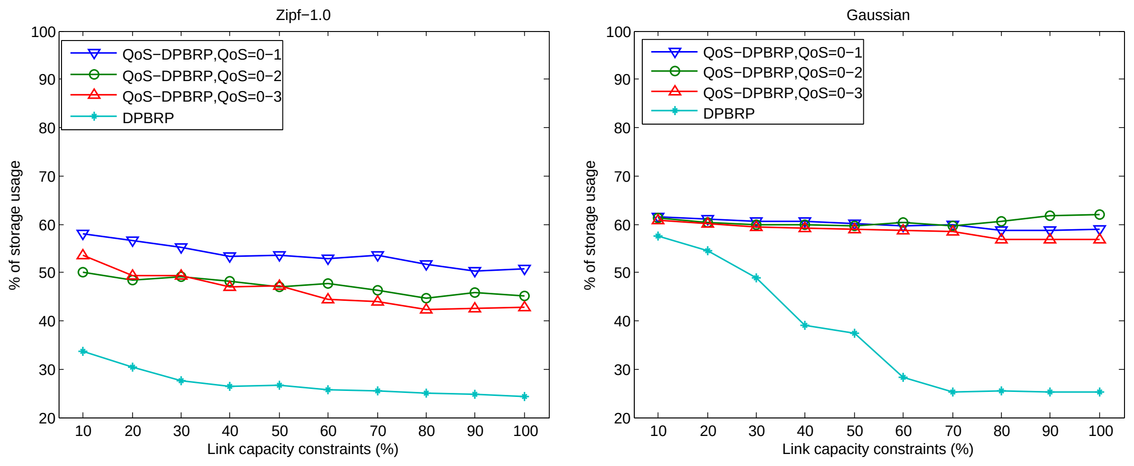
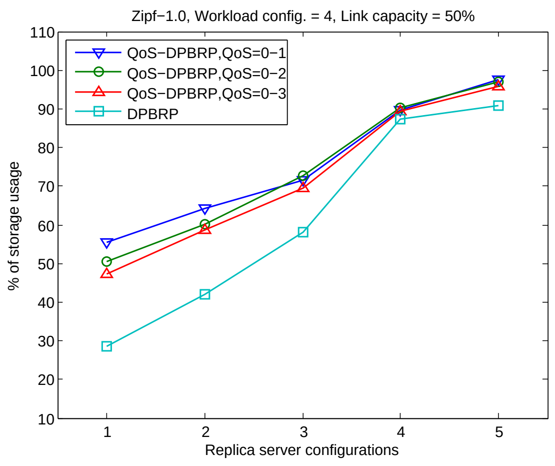


Figure 5.56: Storage cost with link constraints, Uniform QoS constraints

Figure 5.56 compares the storage use using replica server configuration one for uniform QoS values as the reserved link capacities for data transfer were varied from 10% to 100% reservation. With increasing link capacity, the percentage of storage used naturally increases but by different amounts. DPBRP shows less storage requirements for all cases. The cost of QoS-DPBRP for QoS values of [0–1] is higher due to its creation of an increased number of replicas. Figure 5.57 shows the storage use for all replica server configurations when both workload and link constraints are considered. As before, the percentage of storage used is increased with the decrease of available storage.

5.5.4 Discussion

The primary goal of my QoS-aware distributed dynamic replica placement algorithm is to reduce job execution time and to satisfy QoS constraints as much as



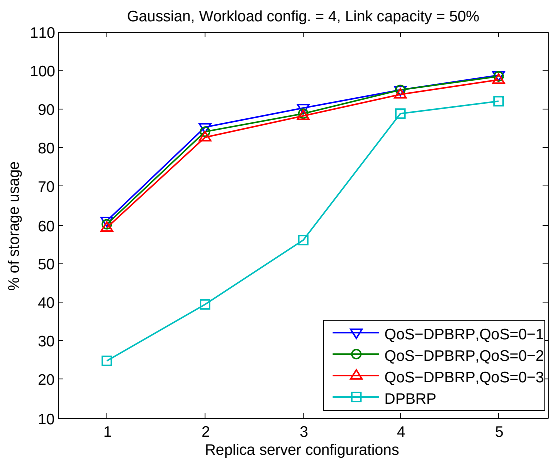


Figure 5.57: Storage cost, Zipf-1.0 and Gaussian access, Constant and uniform QoS constraints

possible. At the same time, from the perspective of the whole system, the performance metrics of bandwidth consumption and storage use need to be managed to ensure that QoS-DPBRP does not induce unduly heavy load on the system.

In most situations, QoS-DPBRP shortens job execution time, sometimes significantly, compared to its unconstrained counterpart at the cost of moderate to high bandwidth consumption. The benefits of QoS-DPBRP are achieved by creating a number of additional, well-placed replicas to meet user QoS requests. This, of course, incurs higher storage cost. Moreover, QoS-DPBRP shows improved job execution time compared to *Greedy Add* and *Greedy Remove* at the cost of moderate use of bandwidth and storage resources to meet user and system (workload) QoS requirements. The results suggest that our algorithm is successful in deciding which data files should be replicated and where the replicas should be placed to meet both types of QoS constraints. Adding workload and link constraints on replica servers gives performance benefit in terms of execution times. The available storage capacities of the replica servers naturally has a major impact on the performance of replication techniques. Increasing the replica server storage capacity leads to performance improvement in terms of job execution time and satisfaction rates of user QoS requests. QoS-DPBRP appears to be better able to exploit increases in available storage size than its non-QoS-aware counterpart. Overall, relaxed QoS constraints (i.e. wider acceptable ranges) lead to improved satisfaction of requests compared to strict QoS requirements.

QoS-DPBRP supports the optimization of multiple QoS parameters concurrently. Each user/client can specify its own acceptable QoS, in terms of the number of hops towards the root of the hierarchy to ensure timely retrieval of the requested data. The resulting replica locations due to the satisfaction of this user QoS constraints reduce the server load and bandwidth use up in the hierarchy. This, in turn, helps meeting the workload and link capacity constraints. Replicas are pushed down the hierarchy by QoS-DPBRP to meet any of the specified QoS requirements. This facilitates optimization of multiple QoS parameters concurrently and does not impose heavy load on the system. QoS-DPBRP thus offers the benefit of this flexibility and its potential usefulness for other applications than data grids.

My QoS-aware distributed dynamic replica placement algorithm (QoS-DPBRP) and a performance comparison of QoS-DPBRP to its unconstrained counterpart (DPBRP) have been published in [SGE11b]. Only user QoS requirement in terms of hop count (replica distance) is considered in this paper. In our subsequent paper [SGE11a], the QoS-aware replication problem is addressed from a system point of view where the workload capacity of replica servers and link capacities are bounded.

Chapter 6

Overall Assessment and Scope of Application

A data grid enables thousands of scientists sitting at geographically distributed universities and research centers to effectively share their data. The sheer volume of the data and computation calls for sophisticated data management and resource allocation. This thesis is one step towards better understanding the dynamics of such a system and the issues involved in increasing the overall efficiency of a data grid by intelligent replica creation and movement. This chapter presents an overall assessment of the studied replication algorithms based on the results obtained from simulation experiments and also analyses the benefits and challenges of deploying the placement strategies in real data grid applications and applications in other domains as well. The chapter starts with discussing the efficiency of my centralized and distributed popularity-driven replica placement algorithms. Then it analyses the aspect of incorporating QoS constraints into the base distributed algorithm and investigates to what extent these quality expectations are met given that the required resources (computation, communication, and storage) are typically limited. The goal of the assessment is to look for patterns between various performance metrics to be able to report on general trends that will tell us when my approach is likely to work well. As mentioned, there will be "hot" files in real applications which means some files will be more popular than others. This access behavior makes my results significant.

6.1 Replica Placement Using Centralized Algorithms

To address the replica placement problem in large-scale data grids, a base "popularity-driven" dynamic replica placement algorithm was introduced in this thesis for use in hierarchically structured data grids. I also presented an adaptive version of my basic replica placement algorithm which considers both data access arrival rates from the clients and the storage capacities of the replica servers to select the best candidate sites at which to place replicas. The performance of my placement algorithms was evaluated using a set of simulation experiments over a wide range of data access patterns and replica server capacities and was compared to a number of other existing replica placement algorithms.

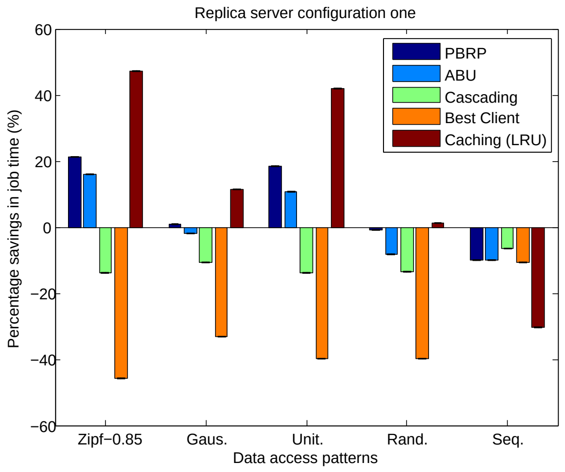


Figure 6.1: Percentage savings in job time for different replication methods compared to Fast spread using sufficient storage at the replica servers (i.e. storage configuration one)

Figures 6.1 and 6.2 show the performance results of the studied algorithms for different data access patterns and replica server configurations. All the algorithms are compared with Fast spread being the standard of comparison. Thus the graphs illustrate the savings achieved by the algorithms beyond those achieved by Fast spread. Caching's performance in terms of job execution time is better than that of other

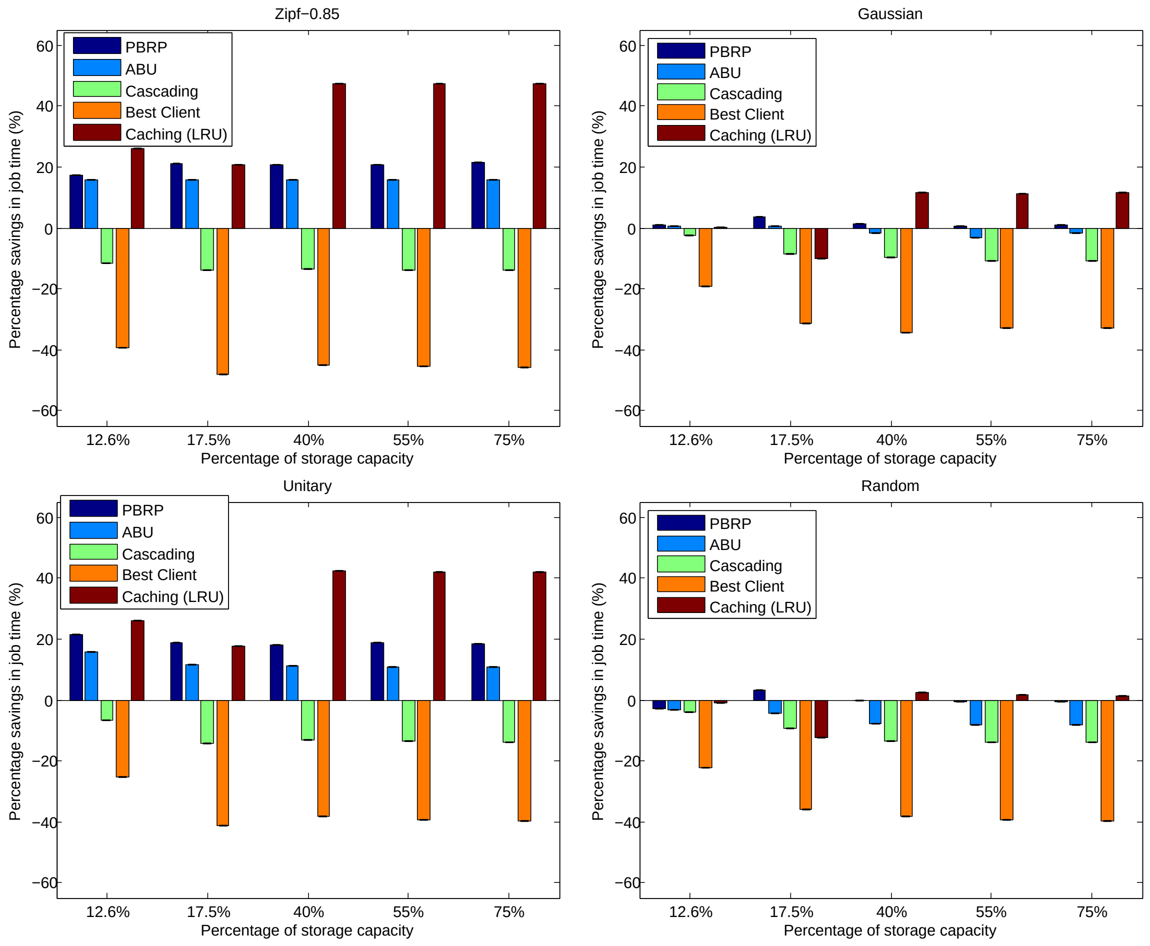


Figure 6.2: Percentage savings in job time as compared to Fast spread for different access patterns using various relative storage capacity (Section 4.4.3)

algorithms for access patterns containing both randomness and temporal locality provided client sites have sufficient storage. However, in cases of limited storage capacity of clients, caching files will get replaced quickly which in turn will increase access time causing an increase in job execution time (refer to Figure 6.2). Relatively longer simulation runs (as shown in Section 4.5.1) highlight the advantage of PBRP which consistently performs better than Caching in terms of job time for the random data access patterns (such as Gaussian and flat Random) where data requests from clients occur for a wider range of files. The performance improvement of PBRP in terms of job time is minimal compared to Caching when clients have sufficient storage (i.e. Configurations 1, 2 and 3) and data access patterns contain temporal locality (i.e. Zipf-0.85 and Zipf-1.0). In fact once the access patterns contain more locality Caching in some cases shows somewhat better job execution time than PBRP though the difference in this case is up to 6%. On the other hand, Best client consistently performs worse than other algorithms in most situations both in terms of job time and bandwidth consumption.

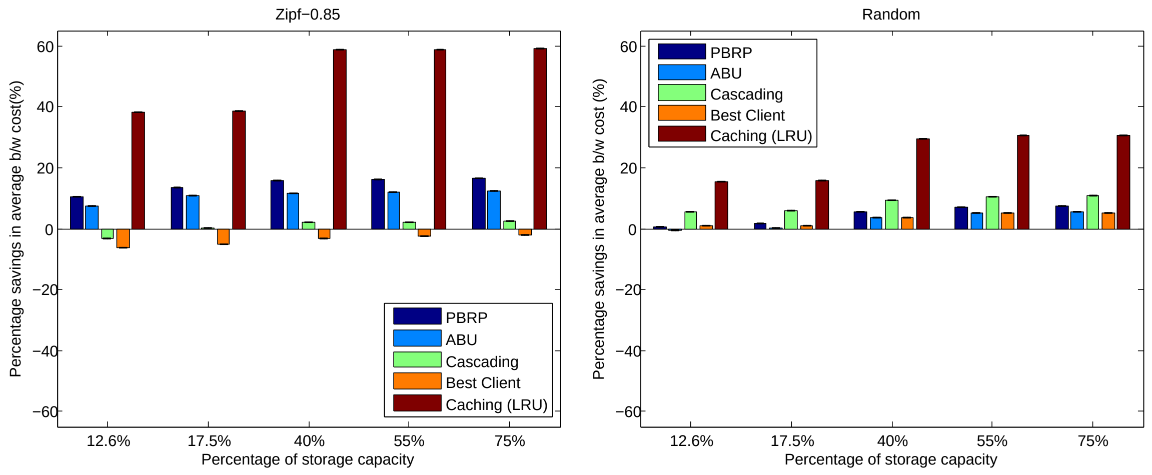


Figure 6.3: Performance savings in average bandwidth usage as compared to Fast spread using various storage configurations

Considering the remaining strategies Fast spread, Cascading, ABU, and PBRP, there is no sure best strategy for all scenarios. Fast spread consistently performs better than Cascading in terms of job execution time for all access patterns. The job time savings of Fast spread are up to 13% more than that of Cascading. In spite of the advantage Fast spread has, it's overhead in terms of high frequency of replication is obvious. It has high storage requirement (up to 36% more) and consumes more bandwidth (up to 12%) compared to Cascading. This is shown in Figures 6.3 and 6.4.

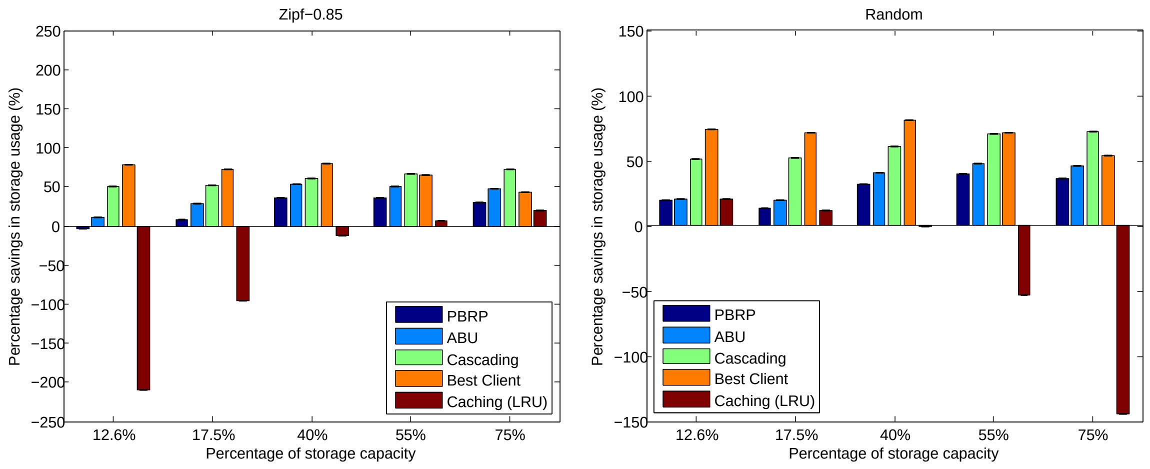


Figure 6.4: Performance savings in storage usage as compared to Fast spread using various storage configurations

PBRP consistently performs better in terms of job execution time and bandwidth consumption compared to ABU, Fast Spread, and Cascading for data access patterns that reflect temporal locality (i.e. Zipf-0.85 and Zipf-1.0). The advantage of PBRP over other algorithms increases as the access patterns contain more locality. The job time savings of PBRP are up to 6% and 25% more than that for ABU and Fast spread, respectively as shown in Figures 6.1 and 6.2. This is because when the data access patterns follow Zipf, some file requests occur frequently making them "popular" where many others occur rarely. Thus the clients focus on a smaller range of data files with higher frequencies compared to other access patterns. Consiquently, PBRP can identify the popular files effectively based on past data accesses. PBRP also considers the locations of replica servers and clients for determining the replication destinations, so the replica servers are properly utilized. As for the random data access patterns, the performance improvement of PBRP in terms of job time is not terribly significant compared to Fast spread and ABU and in fact for flat random distribution Fast spread shows somewhat better job time than PBRP though the difference in this case is marginal (about 2%). This can be attributed to the fact that the overhead in creating additional replicas in PBRP is not offset by the advantage of moving them closer to the clients. ABU does not differ significantly from PBRP in terms of bandwidth consumption. Fast spread on the other hand leads to the higher bandwidth usage (up to 8%) due to higher replication cost compared to PBRP (refer to Figure 6.4).

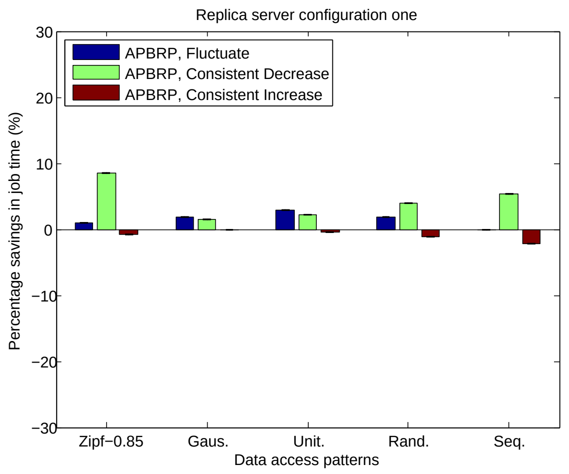


Figure 6.5: Percentage savings in job time for APBRP as compared to PBRP for all data access rates

Compared to the other algorithms PBRP shows moderate requirements for storage use. This moderate use of (relatively cheap) storage space by PBRP makes it a good candidate when data access performance and bandwidth use are of primary concern. Overall, the storage capacities of the replica servers have a major impact on the performance of replication techniques. Increasing the replica server capacity leads to performance improvement for job execution time and average bandwidth cost.

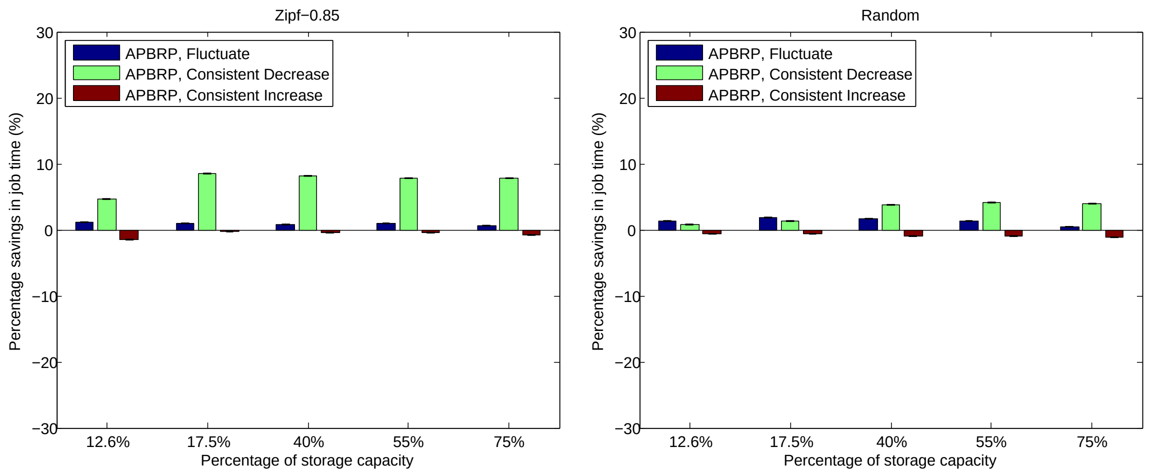


Figure 6.6: Percentage savings in job time for APBRP as compared to PBRP using various storage configurations

These results lead us to conclude that if there is sufficient temporal locality in the access patterns then the strategy that would work best is PBRP. With only a moderate increased amount of storage use, PBRP lowers job times significantly, while judiciously using network resources. If, however, grid users exhibit total randomness in accessing data, then depending on what is more important in the grid scenario, lower access times or lesser bandwidth consumption, a trade-off between PBRP and Fast spread can be made. If the chief aim is to elicit faster responses from the system, Fast spread might work better. On the other hand if conserving bandwidth is of top priority, PBRP is a better grid replication strategy.

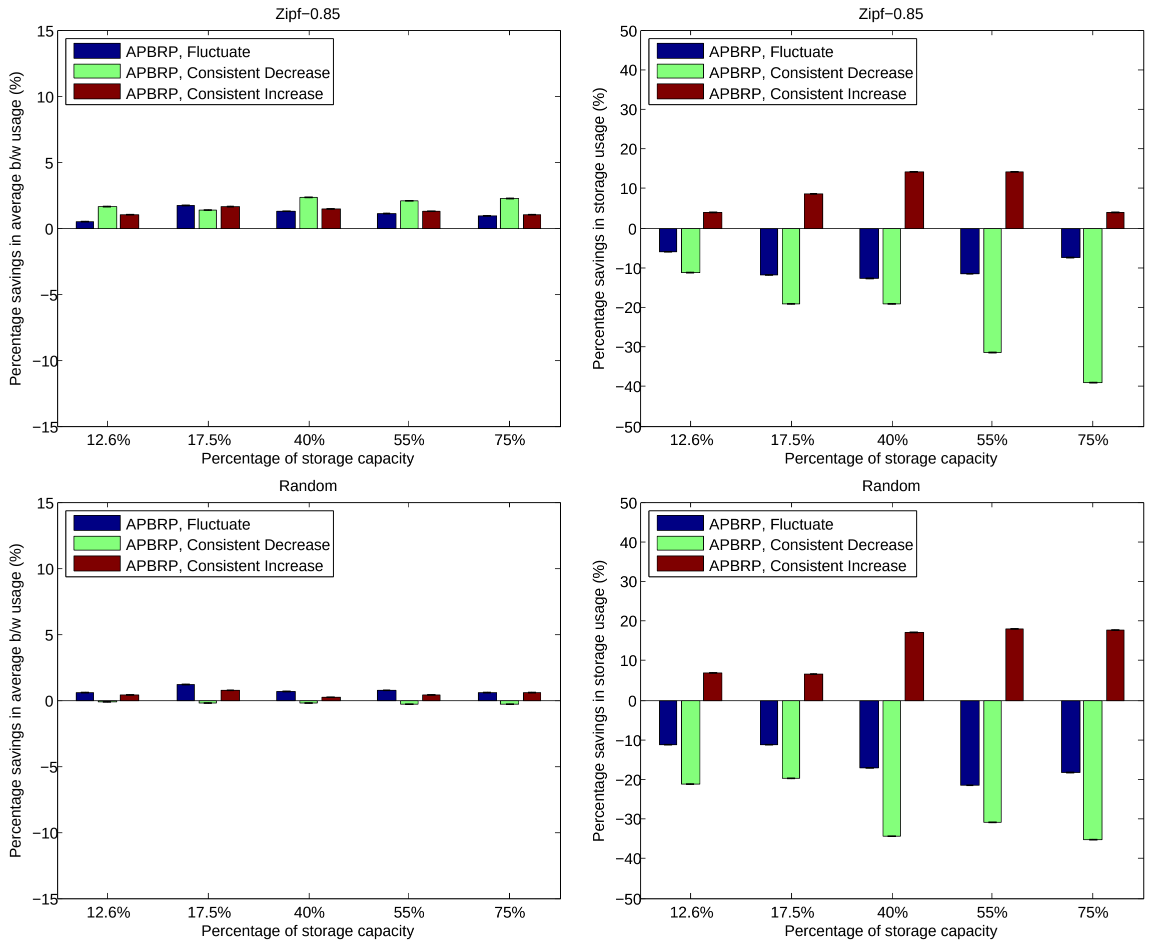


Figure 6.7: Performance savings in average bandwidth usage (left) and storage usage (right) as compared to PBRP using various storage configurations

In PBRP, the threshold value remains constant irrespective of variation in data access arrival rate and the available storage capacities of the replica servers. APBRP addresses this issue by dynamically changing the threshold value based on the data access rate and storage availability. Figures 6.5 to 6.7 compare the performance of PBRP and APBRP considering three scenarios: when the data access rate is consistently increasing, when it is consistently decreasing, and when it fluctuates.

Simulation results show that APBRP is able to further reduce the job execution time and bandwidth use in most cases when the data access rate regularly fluctuates and decreases at the expense of some additional storage consumption. PBRP performs as well as APBRP or somewhat better for all access patterns due to a significantly increased number of replicas when the replica servers have sufficient storage and the client request rate is consistently increasing.

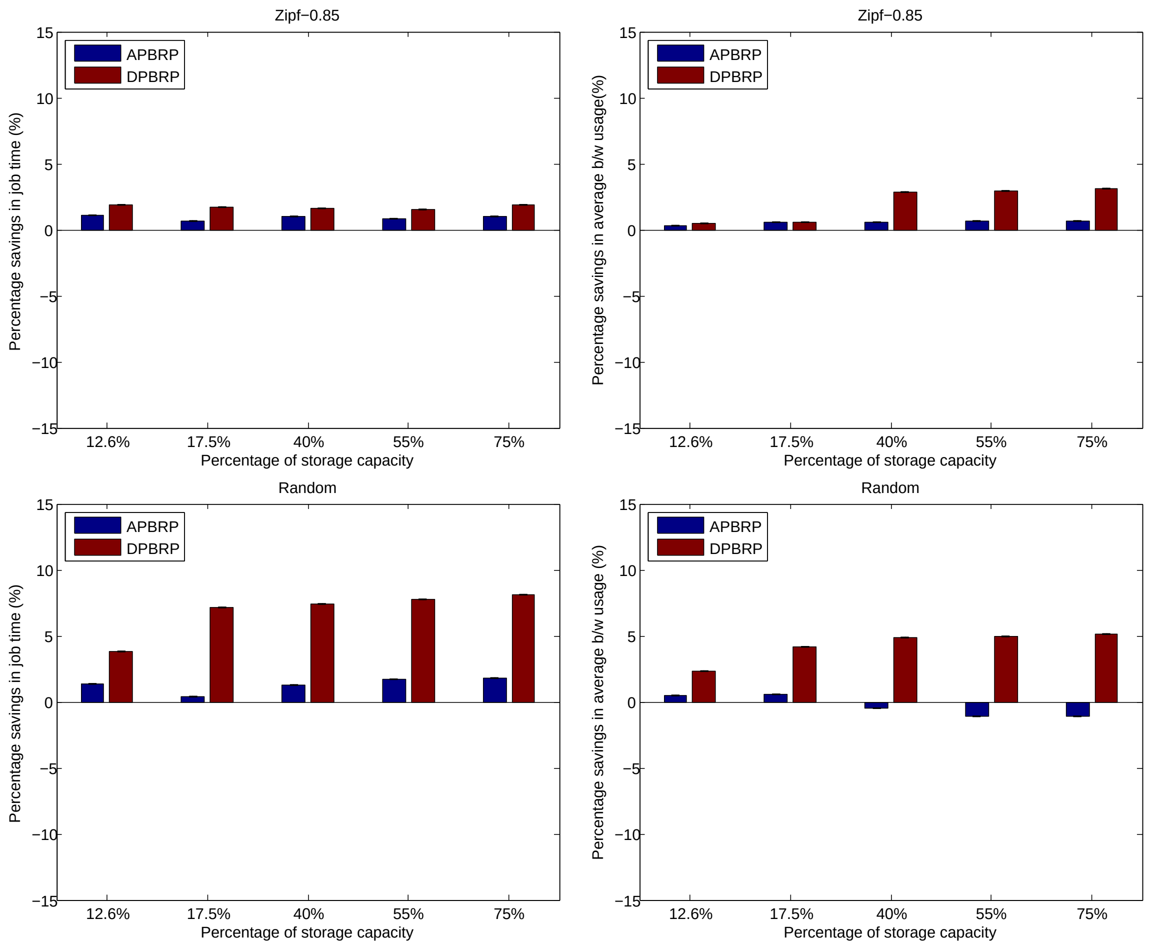


Figure 6.8: Percentage savings in job time (left) and average bandwidth usage (right) for DPBRP and APBRP as compared to PBRP using various storage configurations

6.2 Replica Placement Using Distributed Algorithms

A distributed solution to the replica placement problem was then presented that minimizes overall replication overhead (access and update costs) for a given traffic pattern. The primary goal of my basic distributed dynamic replication algorithm (DPBRP) is to reduce the job execution time experienced by the end-user (by decreasing data access latency). At the same time, from the perspective of the whole system, the performance metrics of bandwidth consumption and storage use need to be managed to ensure that DPBRP does not induce heavy load on the system.

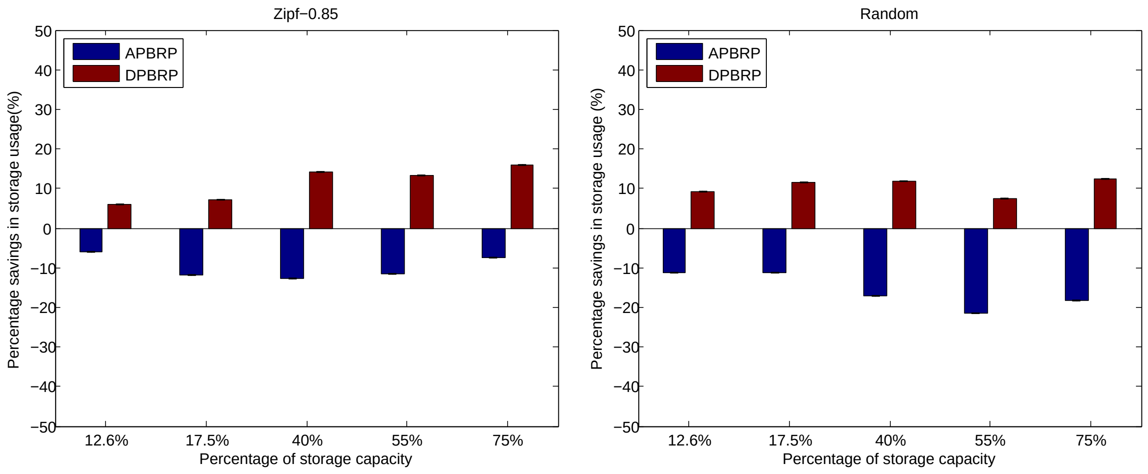


Figure 6.9: Performance savings in storage usage as compared to PBRP using various storage configurations

Figures 6.8 and 6.9 show the performance results for DPBRP, APBRP, and PBRP. The first two strategies are compared, with PBRP being the standard of comparison. In most situations, DPBRP shortens job execution time (up to 10%) and reduces bandwidth consumption compared to APBRP and PBRP while offering the benefits of a distributed algorithms (no single point of failure, distribution of algorithm overhead across many machines). Further, the storage cost incurred by DPBRP is less than

APBRP and PBRP in most cases. The benefits of DPBRP are achieved by creating

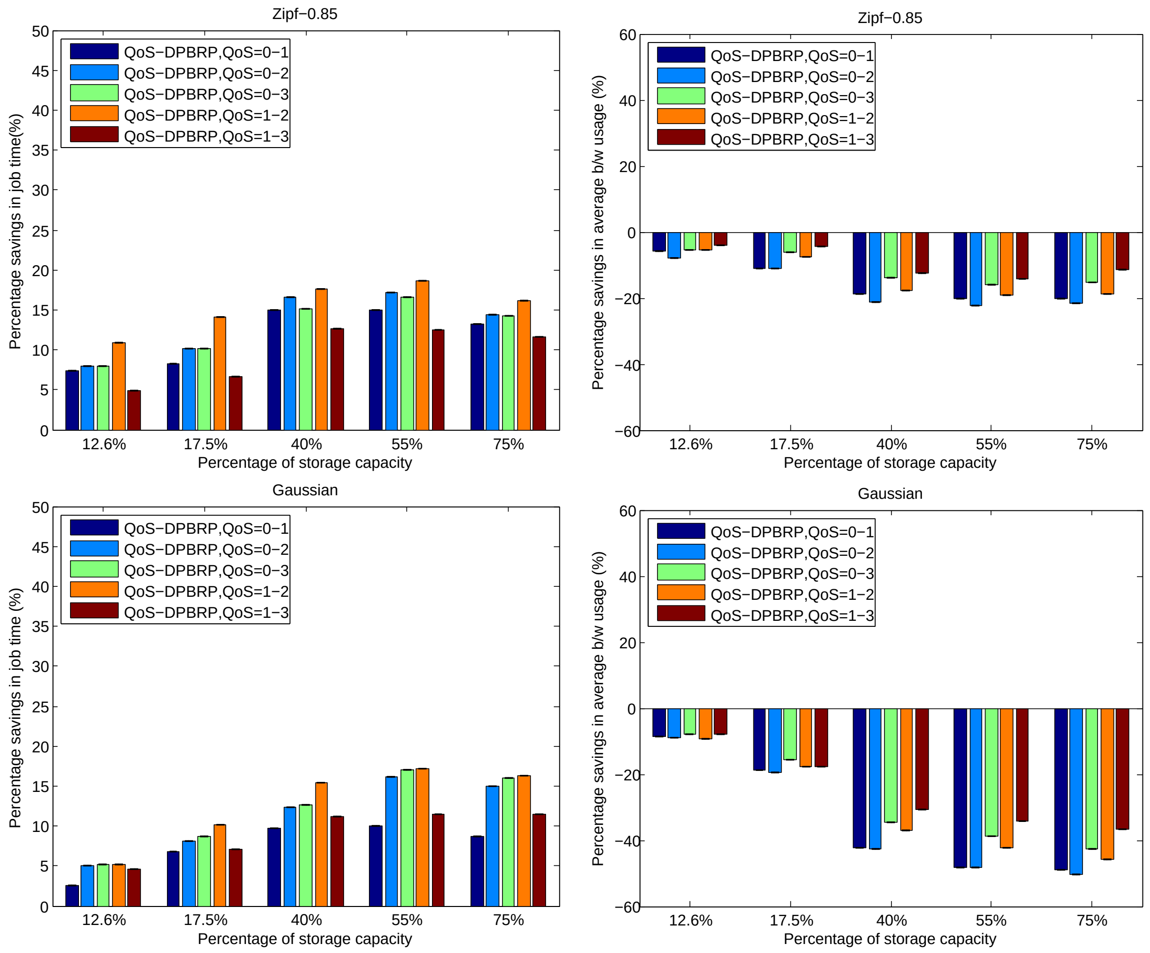


Figure 6.10: Percentage savings in job time (left) and average bandwidth usage (right) for QoS-DPBRP as compared to DPBRP using various storage configurations

an "appropriate" number of well-placed replicas. Performance results suggest that the DPBRP algorithm is successful in deciding which data files should be replicated and where the replicas should be placed. The available storage capacities of the replica servers naturally has a major impact on the performance of all replication techniques. Increasing the replica server capacity leads to performance improvement in terms of job execution time and average bandwidth cost. DPBRP also appears to be better able to exploit increases in available capacity than my other algorithms.

My earlier results show that PBRP can shorten job execution time significantly and reduce bandwidth consumption compared to other dynamic methods including ABU, Fast Spread and Cascading placement. Thus, transitively, DPBRP also performs better than these other non-adaptive dynamic replication methods.

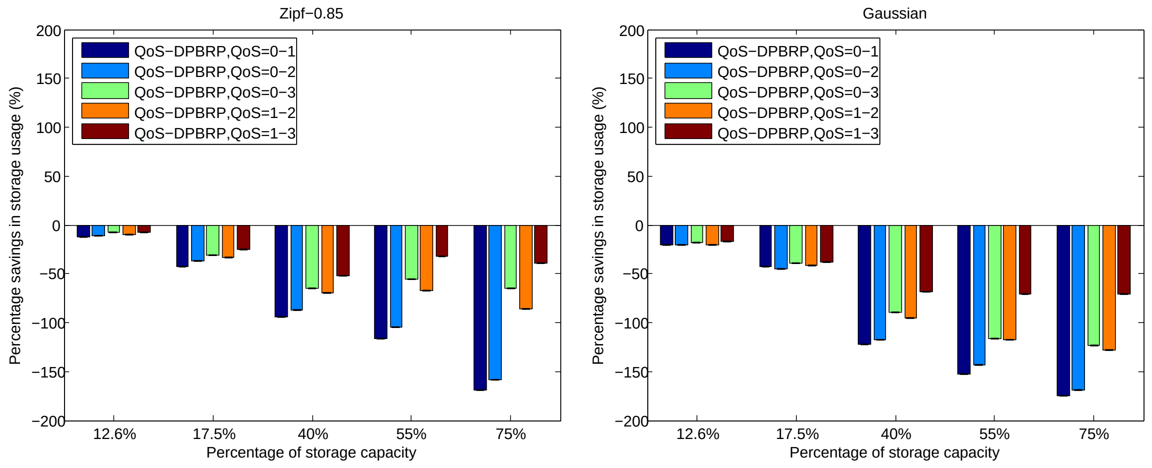


Figure 6.11: Performance savings in storage usage for QoS-DPBRP as compared to DPBRP using various storage configurations

My modified, QoS-aware, replica placement algorithm, QoS-DPBRP aims simultaneously to improve system performance and satisfy the quality requirements of users. In most situations, QoS-DPBRP shortens job execution time, sometimes significantly (up to 20% for a QoS request range of [1–2]), compared to its unconstrained counterpart at the cost of moderate to high bandwidth consumption. This is shown in Figure 6.10. The benefits of QoS-DPBRP are achieved by creating a number of additional, well-placed replicas to better meet user QoS requests. This, of course, incurs higher storage cost (refer to Figure 6.11). As before, the available storage capacities of the replica servers has a major impact on the performance of replication techniques. Increasing the replica server storage capacity leads to performance improvement in terms of job execution time and also satisfaction rates of user QoS requests.

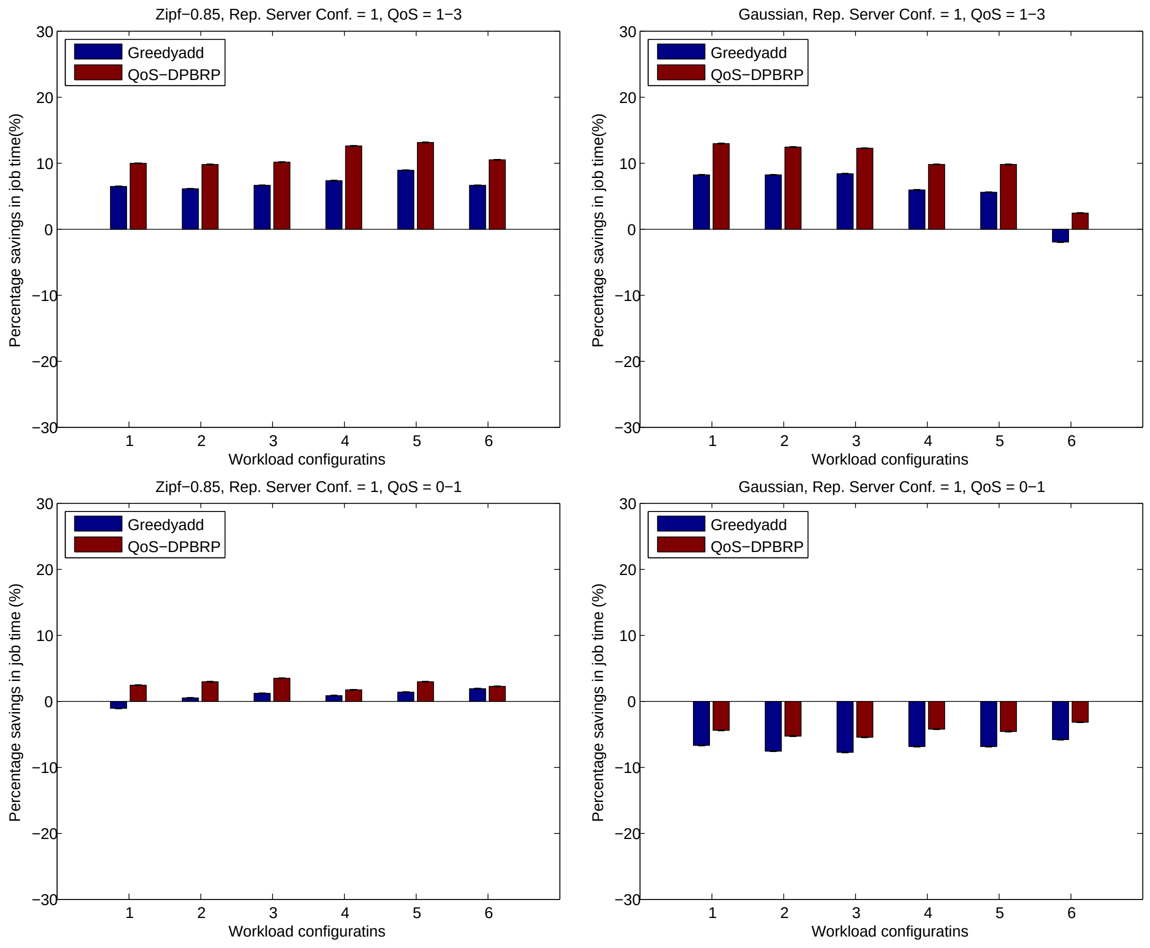


Figure 6.12: Percentage savings in job time as compared to *Greedy Remove* for different user QoS using various workload configurations

QoS-DPBRP appears to be better able to exploit increases in available storage size than its non-QoS-aware counterpart. Overall, relaxed user QoS constraints (i.e. wider acceptable ranges) lead to improved satisfaction of requests compared to strict QoS requirements.

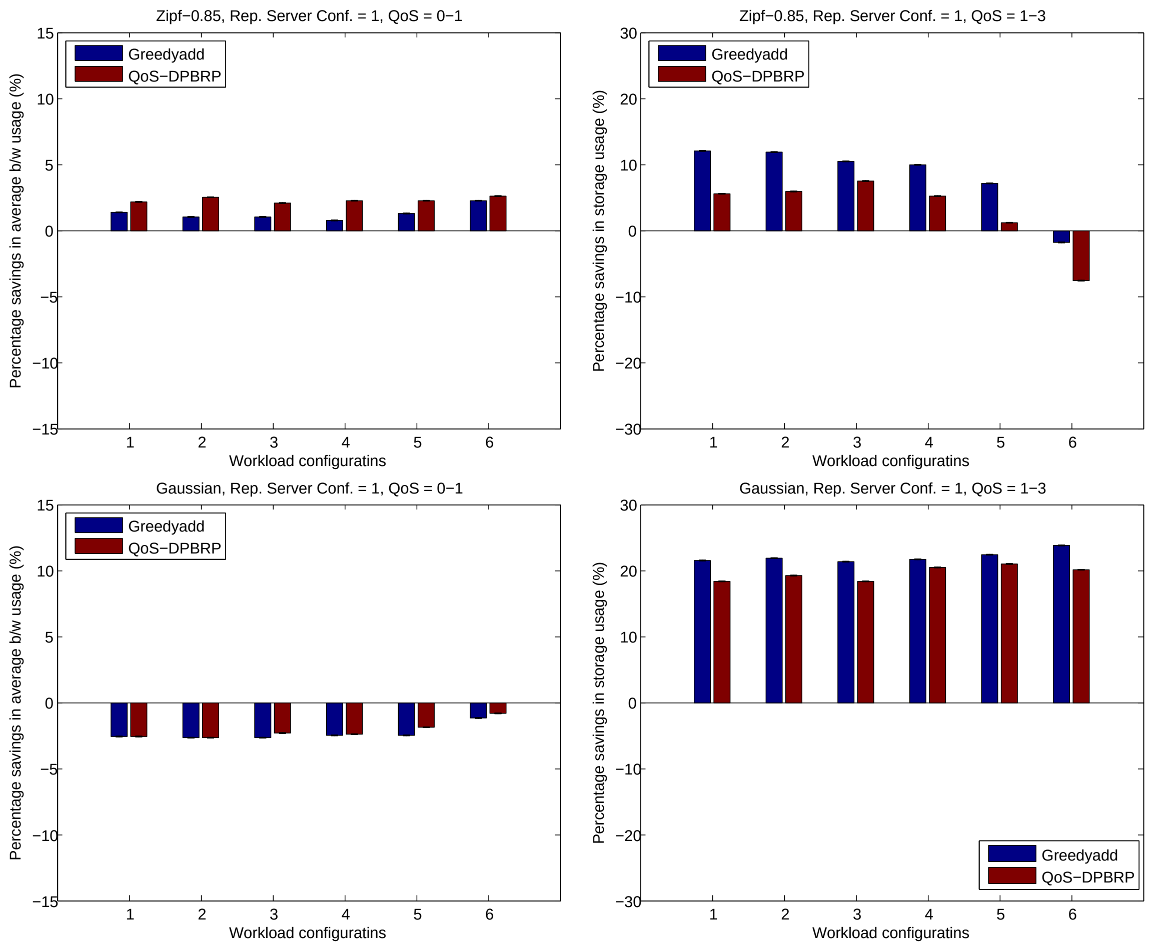


Figure 6.13: Performance savings in average bandwidth usage (left) and storage usage (right) as compared to *Greedy Remove* using various workload configurations

QoS-DPBRP shows improved job execution time in most cases compared to the *Greedy Add* and *Greedy Remove* algorithms by Cheng et al. [CWL09] at the cost of low to moderate increases in use of bandwidth and storage resources to meet user and system (i.e. workload) QoS requirements as shown in Figures 6.12 and 6.13. Using a Gaussian data request distribution and user QoS of [0–1] shows an exception to this in which case *Greedy Remove* performs better than the other algorithms. In this case, the replica locations determined by *Greedy Remove* is relatively more scattered in the lower tiers of the hierarchy due to the constrained QoS requirements. Consequently, the randomness in data access pattern results in reduced job execution times. Overall, adding workload constraints on replica servers gives performance benefit in terms of execution times.

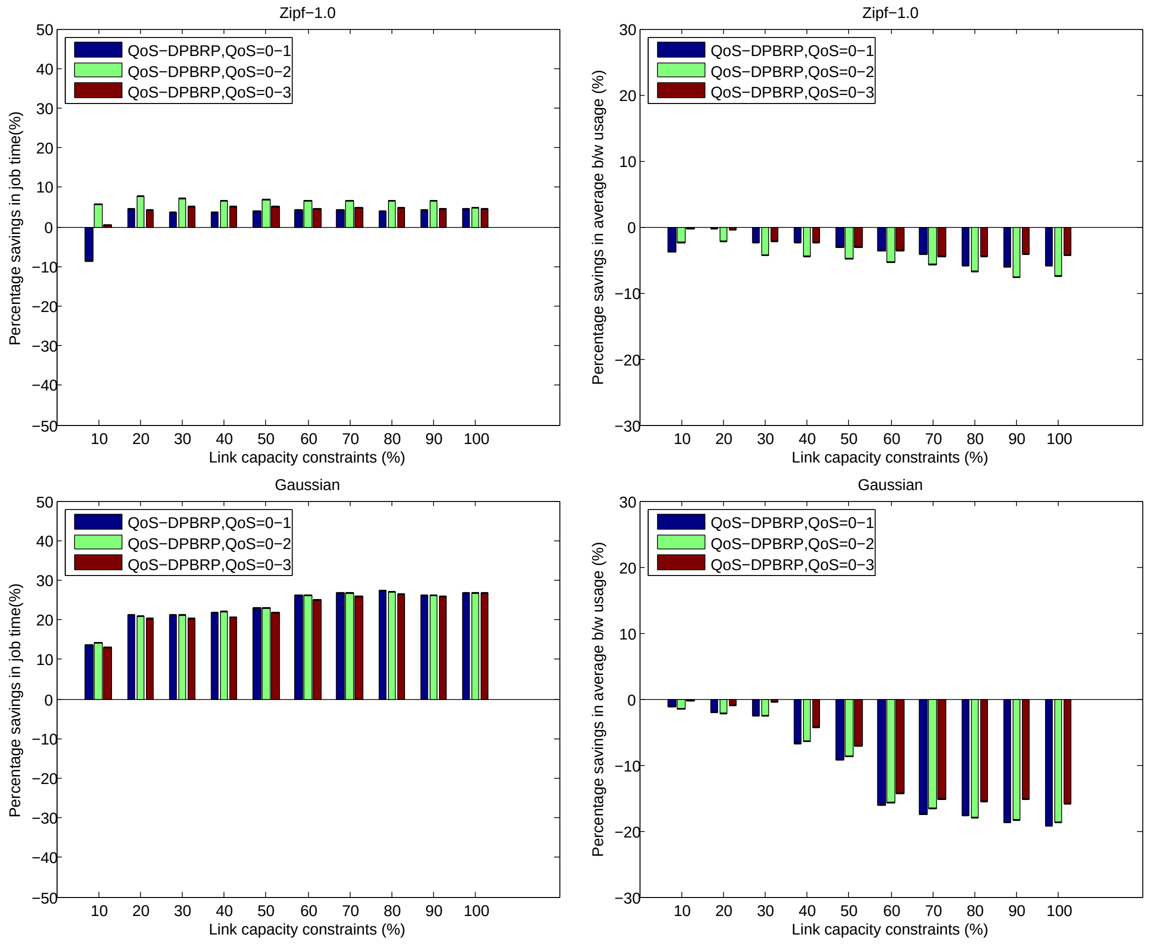


Figure 6.14: Percentage savings in job time (left) and average bandwidth usage (right) for QoS-DPBRP as compared to DPBRP for varied link capacities

Support for link capacity constraints were also added to QoS-DPBRP and assessed by varying the available link capacities (as shown in Figures 6.14 and 6.15). With increasing link capacity, the execution time decreased for all the variants of QoS-

DPBRP compared to DPBRP but by different amounts. The performance savings in job time when compared to DPBRP for the Gaussian distribution is more pronounced than for Zipf-1.0. This is due to the significant drop in the number of replicas created by DPBRP for the Gaussian distribution. With increasing link capacity, bandwidth costs decrease for all algorithm variants but, again, by different amounts. As before, DPBRP has less average bandwidth consumption for all cases. Finally, the percentage of storage used naturally increases with increasing link capacity. DPBRP shows less storage requirements for all cases. The resulting performance due to the addition of both workload and link constraints is largely unchanged from the performance when either the server workload or link capacity constraint is considered in addition to user QoS because this does not significantly affect the replicas that need to be created.

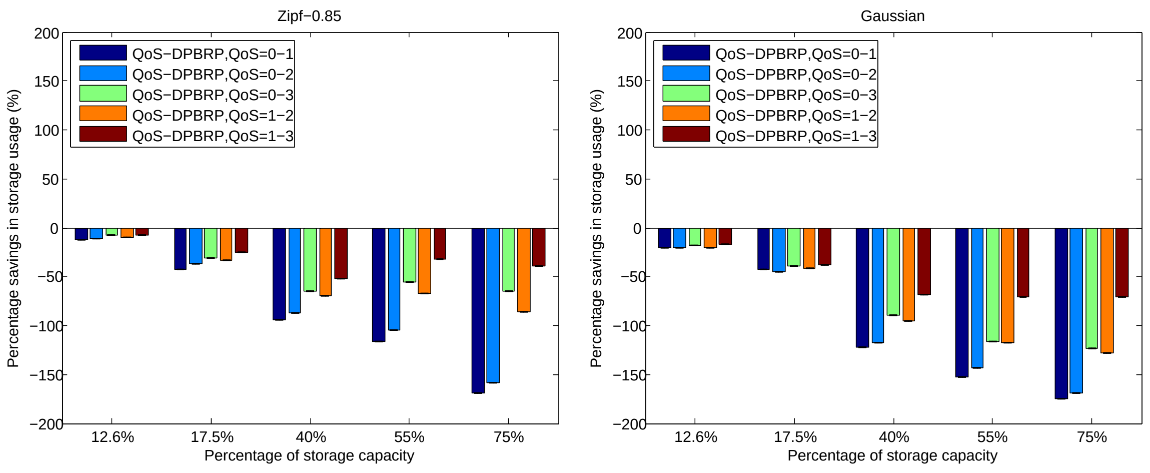


Figure 6.15: Performance savings in storage usage for QoS-DPBRP as compared to DPBRP for varied link capacities

6.3 Applicability

In this section, I briefly consider the applicability of my various algorithms for data grid applications. Collaborating scientists in data grids can form Virtual Organization (VO) [FKT01], which enables them to access different resources over Wide Area Networks (WANs) without regard to their own organizational and administrative domains. As mentioned earlier, the size of the data that needs to be accessed in these VOs is on the order of petabytes today and is fast growing [Hol01; gri01a]. In high energy physics applications such as the ones considered in this thesis, the data distribution can follow a hierarchy. In this scenario, data is collected at a single location, where the detector is located, for example CERN, and then shared by geographically distributed collaborators. Not all potential applications fit this sort of data model. In some cases, data can be collected at multiple sites, replicated to other locations, and then shared among collaborators. This type of replication scenario can be found, for example, still within the E-science domain in the gravitational wave community, where there are multiple detectors (two in the US and two in Europe) [gri01a]. This replication scheme can be fit into a peer-to-peer organization model as other non-E-science applications (e.g. media content distribution).

My centralized popularity-driven base replica placement algorithm and its adaptive version aims to balance the storage utilization and access latency trade-off by determining the frequency and degree of replication given changing grid conditions. The simplicity of the approach makes it easy to deploy on a large number of nodes. The data in this scenario is read-only and so there are no consistency issues. This assumption holds at least reasonably for grid applications where data updates are expected to be infrequent. However, it is necessary to guarantee that the updates will be eventually propagated and that the users will have access to consistent copies of the data. This aspect of replication is taken care of by my distributed replica placement model. My algorithms can handle limited storage capacities at the replica servers and cope with different file sizes as the file sizes are likely to vary across various grid and other applications.

In my algorithms, I assume that recently popular files will tend to be accessed more frequently than others in the near future. Violation of this assumption will lead to decreased performance until the next sampling period commences so that popularity information can be updated. Furthermore, scientific collections of data may comprise tens to hundreds of millions of files. The time to aggregate access information about files cached at multiple hierarchical levels in real scenarios might significantly increase the run time of the algorithms.

My distributed replica placement approach is capable of performing automatic and "best possible" data replication based on the user and application demands. It has many attractive features that make it suitable for grid applications. First, to address the data placement policy I use a cost model to evaluate the overhead (access and update) of replicating data before deciding when and where to create and place new replicas. The cost model is formulated as a dynamic programming problem and its solution is obtained for large-scale hierarchical data grids in a distributed fashion where different performance metrics are evaluated against the different optimization goals (e.g. minimizing replication overhead) and both user and system quality of service requirements are supported. Each grid node is able to calculate the cost of creating a local replica or the cost of transferring data from a remote replica server up in the hierarchy, an important feature that enables the algorithm to scale well. Second, an important aspect of replication-based systems is the protocol used to maintain consistency among replicas. The main issue in such systems is maintaining scalability with large numbers of replicas distributed over the grid while maintaining the same view of all replicas (i.e. consistency). The updates are delivered from the original copy to all replicas via application-level multicast, in which each server receives the updates from its parent and is responsible for further distributing the updates to its children. It is assumed that in grid applications, updates to the data are infrequent and that the consistency can be more relaxed than in, for example, high-performance commercial databases. Given this, I have used an approach that achieves greater scalability while making modest compromises in terms of update propagation and replica synchronization.

My replica placement algorithms can also adapt to various distributed environments. Since the algorithms in this thesis are designed for hierarchical data grids, however, they cannot directly be applied to environments requiring other network topologies. Some grid-like complex and other parallel and distributed applications cannot always be organized and controlled in a hierarchical manner. Any central directory service would inevitably become a performance bottleneck and a single point of failure. One open question for replica placement in such environments is how to determine replica locations when the network topology is a general graph, instead of a hierarchy. It will be challenging to consider the properties of such networks and adapt my algorithms for use with them. One possible solution to this problem might be to embed a tree in the available network topology and use the solution for the resulting embedded tree topology.

Let the network model be represented by an undirected graph G = (V,E), where V is the set of servers, and $E \subseteq V \times V$ denotes the set of network links among the servers. Each link $(u, v) \in E$ will be associated with a cost $d(u, v)$ that will denote the "communication cost" of the link as discussed in the previous section. Assuming that the graph is connected, one server can connect to any other server via some path. Again, define the communication cost of a path as the sum of the communication cost of the links along the path and define $d(u, v)$ between two servers u, v to be the communication cost of the shortest path between them. I could then define a special server called the *origin server*, in the network. Without loss of generality, assume that this origin server constitutes the root of the tree that is going to be embedded. We can calculate the all-pairs shortest paths and build a shortest path tree rooted at the origin server.

Another issue would be to decide whether the embedded tree should be static or dynamic. A static tree would definitely be easier to maintain while a dynamic tree would be more appropriate to capture the dynamics of the Grid at the cost of extra maintenance (i.e. restructuring) overhead. Embedded trees would certainly be less balanced than those I have worked with to this point and could have a significant impact on the performance of my algorithms.

Further, in certain types of applications, such as image, video, and map servers, it is necessary to place with each copy of an object replica a copy of an appropriate software system, for example a DBMS or a GIS system, for servicing read and write requests. In many cases, however, such systems impose restrictions on the number of concurrent users. This kind of situation can be modeled by using loads and capacity constraints for the nodes.

The issue of maintaining consistency among object replicas has been addressed in previous distributed file systems, databases, content distribution networks, and web applications by the use of optimistic consistency protocols [SL00; EM02]. However, these consistency issues have not been addressed in a similar fashion by my algorithms on the scale of a grid environment that crosses multiple organizational domains. Additionally, some of the requirements for such systems are different from those of the applications targeted in this thesis (e.g. updates are only generated from the root). The sizes of the data stored in data grids are often much bigger than those supported by existing distributed file systems, and the replica granularity is much higher. Hence, a modified replica consistency mechanism might be needed for effective adaptation of my algorithms to these environments.

Chapter 7

Conclusions and Future Work

In this thesis, I have addressed the problem of replica placement in large-scale hierarchical data grids to improve the performance of data access while ensuring efficient use of both bandwidth and storage resources. To this end, I have proposed a family of efficient algorithms for dynamic replica placement in a hierarchical data grid structures as is common in current data grid systems [BCCS$^+$03; LCG01; RF01b; RF01a]. My basic popularity-driven dynamic replica placement strategy, Popularity Based Replica Placement (PBRP), underpins the family and aims to increase data access performance from the perspective of the clients by dynamically creating replicas for "popular" files. I have also proposed an adaptive version of this algorithm which considers data request arrival rates and available storage capacities at the replica servers when doing placement resulting in faster access and efficient use of bandwidth and storage in spite of changing access patterns and loading conditions. These two algorithms are centralized in nature due to the fact that replication decisions are made by a single entity in the data grid system which invokes the algorithms at regular intervals.

I have also developed a *distributed* popularity based replica placement (DPBRP) algorithm the goal of which is to determine appropriate locations for replicas to minimize overall replication cost/overhead (access and update) for a given traffic pattern (i.e. a recurring pattern of access frequencies from clients for different files). To satisfy QoS requirements from both the user (imposed by data requests) and system perspectives, I have provided a QoS-DPBRP algorithm to determine the locations of the replicas to improve system performance while satisfying the quality requirements simultaneously.

The idea behind PBRP is to create replicas as close as possible to those clients that frequently request the corresponding files or more specifically to the clients that request the files with access rates exceeding a threshold value. While the file access count as a measure of popularity is not new PBRP uniquely balances the space utilization and access latency trade-off by selectively replicating files. In hierarchical data grids, every node accesses replicas only from its ancestor nodes. Thus, PBRP exploits the relationship among the access records of clients that are siblings to determine the effective utilization of various replicas. All the replication algorithms from the literature except ABU process the records in the access history individually for a client and do not study the relations among these records.

Simulation results show that PBRP consistently performs better in terms of job execution time and bandwidth consumption compared to ABU, Fast Spread, and Cascading for data access patterns of interest (those that contain a degree of temporal locality). The advantage of PBRP over other algorithms increases as the access patterns contain more locality. The job time savings of PBRP are up to 6% and 25% more than that for ABU and Fast spread, respectively. The performance improvement of PBRP in terms of job time is minimal compared to Caching when clients have sufficient storage (i.e. Configurations 1, 2 and 3) and data access patterns contain temporal locality (i.e. Zipf-0.85 and Zipf-1.0). In fact once the access patterns contain more locality Caching in some cases shows somewhat better job execution time than PBRP though the difference in this case is up to 6%. For flat random distribution Fast spread shows somewhat better job time than PBRP though the difference in this case is marginal (about 2%). This can be attributed to the fact that the overhead in creating additional replicas in PBRP is not offset by the advantage of moving them closer to the clients. However, PBRP consistently performs better than Caching in terms of job time for the random data access patterns (such as Gaussian and flat Random) where data requests from clients occur for a wider range of files. This causes an increased cache misses thereby resulting in high job execution times and bandwidth consumption for Caching compared to PBRP. ABU does not differ significantly from PBRP in terms of bandwidth consumption. Fast spread, on the other hand, leads to the higher bandwidth use (up to 8%) due to higher replication cost compared to PBRP. Compared to the other algorithms PBRP shows a moderate increase in storage utilization. This moderate use of (relatively cheap) storage space by PBRP makes it a good candidate when data access performance and bandwidth use are of primary concern. Naturally, the storage capacities of the replica servers have a major impact on the performance of all replication techniques. Increasing the replica server capacity leads to performance improvement in job execution time and average bandwidth cost.

These results lead us to conclude that if there is sufficient temporal locality in the file access patterns then PBRP is to be preferred. With moderate storage utilization, PBRP lowers job times significantly, while judiciously using network resources. If the chief aim is to elicit faster responses from the system at all costs, Fast spread might work better. On the other hand if conserving bandwidth is also of priority, PBRP is a better grid replication strategy.

In PBRP, the threshold value remains constant irrespective of variation in data access request rate and the available storage capacities of the replica servers resulting in performance degradation. APBRP addresses this issue by dynamically changing the threshold value based on the data request rate and storage availability. Simulation results show that APBRP is able to further reduce the job execution time and bandwidth use in most cases when the data access rate regularly fluctuates and decreases at the expense of some additional storage cost. PBRP performs as well as APBRP or somewhat better for all access patterns due to creation of a significantly increased number of replicas when the replica servers have sufficient unallocated storage and the client request rate is consistently increasing. Naturally, this quickly fills the servers with replicas that may or may not be useful in the future.

The performance of my base distributed algorithm (DPBRP) was compared to its centralized counterparts. In most situations, DPBRP shortens job execution time, sometimes significantly, and reduces bandwidth consumption compared to my centralized algorithms and thus to those described in [RF01a]. Despite this, the storage cost incurred by DPBRP is less than the other algorithms in most cases. The benefits of DPBRP are achieved by creating an appropriate number of well-placed replicas.

I also analyzed the performance of DPBRP once QoS requirements from users were incorporated into the algorithm (QoS-DPBRP). QoS-DPBRP further shortens job execution time compared to its unconstrained counterpart while effectively satisfying user QoS requirements at the cost of moderate to high bandwidth consumption. A number of additional replicas are created by QoS-DPBRP to meet user QoS requests. This, of course, also incurs higher storage consumption. Moreover, QoS-DPBRP shows improved job execution time compared to the QoS-aware *Greedy Add* and *Greedy Remove* algorithms at the cost of moderate use of bandwidth and storage resources to meet both user and system (workload) QoS requirements. The results suggest that my algorithm is successful in deciding which data files should be replicated and where the replicas should be placed to meet both types of QoS constraints. Further, adding workload and link constraints on replica servers gives performance benefit in terms of execution times while providing the ability to manage system costs. Increasing the replica server storage capacity leads to performance improvement in terms of job execution time and satisfaction rates of user QoS requests. QoS-DPBRP appears to be better able to exploit increases in available storage size than its non-QoS-aware counterpart. Relaxed QoS constraints (i.e. wider acceptable ranges) lead to improved overall satisfaction of requests compared to strict QoS requirements.

This thesis contributes to our understanding of replica placement in large-scale data grid environments and advances the state-of-the-art through its contributions. The work done in this thesis and the contributions made have led to new challenges that need to be addressed. I plan to explore several directions for future research. Some are natural continuations of my work, while others deal with more general and outstanding issues in data replication.

An obvious extension to my work would be to modify my replica placement algorithms to determine replica locations when the data grid topology represents a general graph, instead of a hierarchy (as discussed in the previous chapter). This would broaden the scope of applicability of my algorithms across various parallel and distributed and other complex grid environments that require non-hierarchical network structures. Additionally, an extended replication model should take into consideration the replica consistency issue to meet different applications' requirements without imposing undue overheads on those not requiring strict consistency.

Another extension of my work would be to modify PBRP to determine replica locations by considering LRU Caching at the lowest tier, with PBRP and other algorithms at the higher tiers. It would be also interesting to see how LRU Caching performs relative to other methods when caching is done in all tiers.

In my work, the replication strategies I discussed exploited temporal locality in the request patterns. I put off considering the spatial locality of the requests. Once the relationship among various files in a scientific data set is better understood, some amount of anticipatory pre-fetching would also be possible. To accomplish this I therefore plan to investigate the data access patterns of real scientific and engineering computing applications that run in grid-like distributed environments.

The number of sites is fixed in my simulations and the connection between sites is assumed to be reliable throughout the simulations. As future research, I want to explore modest dynamism of sites such that sites can add join and quit the grid and the replica placement algorithm is capable of taking care of the resulting dropped connections between sites. Also, at present, I consider a single resource broker that submits jobs to different sites and makes decisions about replica placement. I plan to consider multiple resource brokers submitting jobs across the grid sites and making replica placement decisions.

Moreover, I am planning to explore ways to handle peak bandwidth use and its relation to capacity constraints on network links. Due to the dynamic nature of the grid, candidate sites that hold currently replicas may not be the best sites to fetch replicas from in the future due to network load changes. Thus, I plan to explore dynamic replica maintenance issues. Taking these factors into account will be important for implementing my replica placement techniques in real data grids including those operating without leased network lines offering bandwidth guarantees.

Another area for further research is to study the movement of code towards data instead of vice versa. In this thesis, I assumed that clients ask for data files and locally run the data through their code to analyze the data. Thus, I have only considered moving data towards code. Another option might be to move code towards where data resides and communicate only the result of the computation back to the client. This could be a feasible option considering that in data grid scenarios the data may be tens of thousands or more times larger than either the code or the result.

Before deploying replica placement algorithms in a real grid environment, the algorithms must be tested thoroughly. I evaluated the performance and feasibility of my replication algorithms by using the OptorSim simulation toolkit. As future research, I also want to validate my model by deploying the replication algorithms in a real grid environment such as WestGrid or EUGrid. This will include integrating my algorithms with a Replica Location Service (e.g. Giggle framework [Che02]).

Bibliography

[AAR89] The Historical Charges for AARNet Services. http://www.aarnet.edu.au/services.aspx, 1989. Last accessed: Mar 26, 2006.

[Aba04] J.H. Abawajy. Placement of file replicas in data grid environments. In *Proceedings of International Conference on Computational Science*, volume 3038, pages 66–73, Jun 2004.

[ABB$^+$01] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnal, and S. Tuecke. Secure, efficient data transport and replica management for high-performance data-intensive computing. In *Proceedings of the 18th IEEE Symposium on Mass Storage Systems and Technologies (MSS'01)*, pages 13–28, Apr 2001.

[ABB$^+$02] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnal, and S. Tuecke. Data management and transfer in high performance computational grid environments. *Parallel Computing Journal*, 28(3):749–771, May 2002.

[ACK$^+$02] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer.

SETI@home: An experiment in public resource computing. *Communications of the ACM*, 45:56–61, Nov 2002.

[AFN$^+$01] W. Allcock, I. Foster, V. Nefedova, A. Chervenak, E. Deelman, C. Kesselman, J. Lee, A. Sim, A. Shoshani, B. Drach, and D. Williams. High-performance remote access to climate simulation data: A challenge problem for data grid technologies. In *Proceedings of the Supercomputing*, pages 20–35, Nov 2001.

[AGK00] D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with Nimrod-G: Killer application for the global grid. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, pages 520–528, May 2000.

[aka07] Akamai. http://www.akamai.com, 2007. Last accessed: Mar 21, 2009.

[BAD05] Australian Belle Analysis Data Grid. http://epp.ph.unimelb.edu.au/epp/grid/badg/, 2005. Last accessed: Oct 20, 2011.

[BAG00] R. Buyya, D. Abramson, and J. Giddy. Nimrod-G: An architecture for a resource management and scheduling system in a global computational grid. In *Proceedings of the 4th International Conference and Exhibition on High Performance Computing in Asia-Pacific Region*, pages 283–289, May 2000.

[BBL06] M. Baker, R. Buyya, and D. Laforenza. Grids and grid technologies for wide-area distributed computing. *Software: Practice and Experience (SPE)*, 32:1437–1466, Dec 2006.

[BCC$^+$02] W. Bell, D. Cameron, L. Capozza, A. Millar, K. Stockinger, and F. Zini. Simulation of dynamic grid replication strategies in optorsim. In *Proceedings of the 3rd International IEEE Workshop on Grid Computing (GRID'02)*, 2002.

[BCC$^+$03] W. Bell, D. Cameron, L. Capozza, P. Millar, K. Stockinger, and F. Zini. Optorsim - A grid simulator for studying dynamic data replication strategies. *International Journal of High Performance Computing Applications*, 17:403–416, Nov 2003.

[BCCS$^+$03] W. Bell, D. Cameron, R. Carvajal-Schiaffino, A. Millar, K. Stockinger, and F. Zini. Evaluation of an economy-based file replication strategy for a data grid. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 661–668, May 2003.

[BCF03] D. Bosio, J. Casey, and A. Frohner. Next generation EU DataGrid data management services. In *Proceedings of Computing in High Energy and Nuclear Physics (CHEP'03)*, pages 1–8, Mar 2003.

[Bio02] BioGRID. http://www.thebiogrid.org/, 2002. General Repository for Interaction Datasets. Last accessed: Mar 5, 2009.

[bir05] Biomedical Informatics Research Network (BIRN). http://www.nbirn.net/, 2005. Last accessed: Sep 20, 2007.

[BMRW98] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In *Proceedings of the Centre for Advanced Studies Conference(CASCON'98), IBM*, pages 5–16, Nov 1998.

[BMT$^+$98] R. Bagrodia, R. Meyer, M. Takai, Y. an Chen, X. Zeng, J. Martin, and H. Y. Song. Parsec: A parallel simulation environment for complex systems. *Computer*, 31(10):77–85, Oct 1998.

[BO00] G. Barish and K. Obraczka. World wide web caching, trends and techniques. *EEE Communications, Internet Technology Series*, 38:178–184, May 2000.

[BV01] R. Buyya and S. Vazhkudai. Compute power market: Towards a market-oriented grid. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid (CCGRID'01)*, pages 574–581, May 2001.

[BV04] R. Buyya and S. Venugopal. The Gridbus toolkit for service oriented grid and utility computing: An overview and status report. In *Proceedings of the 1st International Workshop on Grid Economics and Business Models (GECON'04)*, pages 19–66, Apr 2004.

[Cah98] R. Cahn. *Wide Area Network Design: Concepts and Tools for Optimization.* Elsevier Science, 1998.

[Cao00] G. Cao. A scalable low-latency cache invalidation strategy for mobile environments. In *Proceedings of the 6th annual International Conference on Mobile computing and Networking*, pages 200–209, Aug 2000.

[CBS$^+$03] W. Cirne, F. Brasileiro, J. Sauve, N. Andrade, D. Paranhos, E. Santos-Neto, and R. Medeiros. Grid computing for bag of tasks applications. In *Proceedings of the 3rd IFIP Conference on E-Commerce, E-Business and E-Government*, Sep 2003.

[CDF$^+$98] R. Caceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web proxy caching: the devil is in the details. *ACM Performance Evaluation Review*, 26:11–15, Dec 1998.

[CDK01] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems, Concepts and Designs.* Addison Wesley, 2001.

[CDO$^+$00] L. Childers, T. Disz, R. Olson, M. E. Papka, R. Stevens, and T. Udeshi. Access grid: Immersive group-to-group collaborative visualization. In *Proceedings of the 4th International Immersive Projection Technology Workshop*, May 2000.

[CFK$^+$00] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The Data Grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23:187–200, Jul 2000.

[CH76] K. Chandy and J.E. Hewes. File allocation in distributed systems. In

Proceedings of the International Symposium on Computer Performance Modeling, Measurement, and Evaluation, pages 10–13, Mar 1976.

[Che02] A. Chervenak. Giggle: A framework for constructing scalable replica location services. In *Proceedings of the IEEE Supercomputing*, pages 1–17, Nov 2002.

[CHNB05] D. Choon-Hoong, S. Nutanong, and R. Buyya. *Peer-to-Peer Computing: Evolution of a Disruptive Technology*, chapter Peer-to-Peer Networks for Content Sharing, pages 28–65. Idea Group Publishers, Hershey, PA, USA, 2005.

[Chu73] W. Chu. *Computer-Communication Systems*, chapter Optimal File Allocation in a Computer Network, pages 577–587. N Abramson and FF Kuo, Eds., Prentice-Hall, 1973.

[CLQ08] H. Casanova, A. Legrand, and M. Quinson. SimGrid: A generic framework for large-scale distributed experimentations. In *Proceedings of the 10th International Conference on Computer Modeling and Simulation*, pages 126–131, Apr 2008.

[CP85] S. Ceri and G. Pelagatti. *Distributed Databases- Principles and Systems.* McGraw-Hill, 1985.

[CSK$^+$05] A. Chervenak, R. Schuler, C. Kesselman, S. Koranda, and B. Moe. Wide area data replication for scientific collaborations. In *Proceedings of the 6th International Workshop on Grid Computing*, pages 1–8, Nov 2005.

[CT03] M. Cannataro and D. Talia. The knowledge grid. *Communications of the ACM*, 46:89–93, Jan 2003.

[CWL09] C. Cheng, J. Wu, and P. Liu. QoS-aware, access-efficient, and storage-efficient replica placement in grid environments. *Journal of Supercomputing*, 49:42–63, Jul 2009.

[DAS04] M. Deris, J. Abawajy, and H. Suzuri. An efficient replicated data access approach for large-scale distributed systems. In *Proceedings of the First IEEE/ACM International Conference on Cluster Computing and the Grid (CCGRID'04)*, pages 588–594, Apr 2004.

[DDP$^+$04] A. Domenici, F. Donno, G. Pucciani, H. Stockinger, and K. Stockinger. Replica consistency in a Data Grid. *Nuclear Instruments and Methods in Physics Research*, 534:24–28, Nov 2004.

[DDW08] A. Devulapalli, D. Dalessandro, and P. Wyckoff. Data structure consistency using atomic operations in storage devices. In *In Proceedings of the 5th IEEE International Workshop on Storage Network Architecture and Parallel I/Os*, pages 65–73, Sep 2008.

[DF05] C. Dumitrescu and I. Foster. GangSim: A simulator for grid scheduling studies. In *Proceedings of the 5th International Symposium on Cluster Computing and the Grid (CCGRID'05)*, pages 1151–1158, May 2005.

[DH95] M. Dunham and A. Helal. Mobile computing and databases: Anything new? *SIGMOD Record*, 24:5–9, Dec 1995.

[dil07] Digital island. http://www.digitalisland.co.nz/, 2007. Last accessed: Jul 25, 2008.

[DMP+02] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6:50–58, Sep 2002.

[DPM00] D.Li, P.Cao, and M.Dahlin. WCIP: Web cache invalidation protocol. *IETF Internet Draft*, May 2000.

[ea00] M. Aderholz et al. Monarc project phase 2 report. Technical report, Technical Report, CERN, 2000.

[edi01] eDiaMoND. http://www.ediamond.ox.ac.uk/, 2001. Diagnostic Mammography National Database Project. Last accessed: Apr 5, 2011.

[EM02] J. A. Elias and L. N. Moldes. A demand based algorithm for rapid updating of replicas. In *Proceedings of IEEE Workshop on Resource Sharing in Massively Distributed Systems*, pages 686– 691, Jul 2002.

[EN03] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison Wesely, fourth edition, 2003.

[ESG00] Earth System Grid. http://www.earthsystemgrid.org/, 2000. Climate Modeling and Simulation. Last accessed: Mar 5, 2009.

[EU:08] EU Data Mining Grid. http://www.dataminінggrid.org/, 2008. Last accessed: Jan 21, 2009.

[eur01] The european data grid project, the datagrid architecture. http://eu-datagrid.web.cern.ch/eu-datagrid/, 2001. Last accessed: Jun 21, 2011.

[FAC$^+$01] I. Foster, E. Alpert, A. Chervenak, B. Drach, C. Kesselman, V. Nefedova, D. Middleton, A. Shoshani, A. Sim, , and D. Williams. The earth system grid: Turning climate datasets into community resources. In *Proceedings of the American Meteorological Society Conference*, May 2001.

[FCAZ00] L. Fan, P. Cao, J. Almeida, and A. Z.Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8:281–293, Jun 2000.

[FKT01] I. Foster, C. Kesselman, and S. Tukcke. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, Aug 2001.

[Fos06] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *Proceedings of the International Conference on Network and Parallel Computing*, pages 2–13, Oct 2006.

[GHaDO96] G. Gray, P. Helland, and a. D. O'Neil. The dangers of replication and a solution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 173–182, Jun 1996.

[GKL$^+$02] L. Guy, P. Kunszt, E. Laure, H. Stockinger, and K. Stockinger. Replica management in data grids. In *Global Grid Forum 5*, Apr 2002.

[GPS03] S. Graupner, J. Pruyne, and S. Singhal. Making the utility data center a power station for the enterprise grid. Technical report, Technical Report HPL-2003-53, HP Labs, Palo Alto, USA,, 2003.

[GR93] J. Gray and A. Reuter. *Transaction processing : concepts and techniques*. Morgan Kaufmann Publishers, 1993.

[gri01a] Grid Physics Network (GriPhyN). http://www.griphyn.org/, 2001. Last accessed: Mar 10, 2007.

[Gri01b] GridPP- UK Computing for Particle Physics. http://www.gridpp.ac.uk/, 2001. Last accessed: Mar 21, 2009.

[gri08] GridSim. http://www.gridbus.org/gridsim, 2008. Last accessed: Feb 1, 2010.

[Hak64] S. Hakami. Optimum location of switching centers and the absolute centers and medians of a graph. *Operations Research*, 12:450–459, 1964.

[Har01] T. Hara. Effective replica allocation in ad hoc networks for improving data accessibility. In *Proceedings of the IEEE INFOCOM*, pages 1568–1576, Apr 2001.

[Har03] T. Hara. Replica allocation methods in ad hoc networks with data update. *Mobile Networks and Applications*, 8:343–354, Aug 2003.

[HJMS$^+$00] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger. Data management in an international data grid project. In *Proceedings of the first IEEE/ACM International Workshop on Grid Computing*, pages 77–90, Dec 2000.

[HMN04] T. Hara, N. Murakami, and S. Nishio. Replica allocation for correlated data items in ad hoc sensor networks. *SIGMOD Record*, 33, Mar 2004.

[Hol01] K. Holtman. CMS Data grid system overview and requirements, 2001. CMS Experiment Note 2001/037, CERN.

[icp97] Application of Internet cache protocol (ICP), version 2. Internet Draft IETF, Jul 1997.

[jab05] Jabber project. http://www.jabber.org/protocol/, 2005. Last accessed: Mar 19, 2008.

[JGN06] W. Jeon, I. Gupta, and K. Nahrstedt. QoS-aware object replication in overlay networks. In *Proceedings of IEEE GLOBECOM*, pages 42–63, Jul 2006.

[KDW01] K. Kalpakis, K. Dasgupta, and O. Wolfson. Optimal placement of replicas in trees with read, write, and storage costs. *IEEE Transactions on Parallel and Distributed Systems*, 12:628–637, Jun 2001.

[KF98] C. Kesselman and I. Foster. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, November 1998.

[KL01] H. Kang and S. Lim. Bandwidth-conserving cache validation schemes in a mobile database system. In *Proceedings of the Mobile Data Management Conference*, pages 121–130, Jan 2001.

[KLM97] T. M. Kroeger, D. D. E. Long, and J. C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *Proceedings* *of the Usenix Symposium on Internet Technologies and Systems*, pages 13–22, Dec 1997.

[KM02] M. Karlsson and M. Mahalingam. Do we need replica placement algorithms in content delivery networks? In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution (WCW)*, Aug 2002.

[KRR02] J. Kangasharju, J. Roberts, and K. Ross. Object replication strategies in content distribution networks. *Computer Communications*, 25:367–383, Mar 2002.

[KRW01] C. Krick, H. Racke, and M. Westermann. Approximation algorithms for data management in networks. In *Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA'01)*, pages 237–246, Jul 2001.

[LA94] A. Luotonen and K. Altis. World wide web proxies. *Computer Networks and ISDN Systems*, 28:147–154, Apr 1994.

[LCG01] Worldwide LHC computing grid. http://lcg.web.cern.ch/lcg/, 2001. Distributed Production Environment for Physics Data Processing. Last accessed: Jan 20, 2010.

[LLW06] Y. Lin, P. Liu, and J. Wu. Optimal placement of replicas in data grid environments with locality assurance. *In Proceedings of the 12th Inter-*

national Conference on Parallel and Distributed Systems(ICPADS'06), 01:465–474, Jul 2006.

[LS97] H. Leong and A. Si. On adaptive caching in mobile databases. In *Proceedings of ACM Symposium of Applied Computing*, pages 302–309, Feb 1997.

[LSsD02] H. Lamehamedi, B. Szymanski, Z. shentu, and E. Deelman. Data replication strategies in grid environments. In *Proceedings of the 5th International Conference on Algorithms and Architectures for Parallel Processing*, pages 378–383, Oct 2002.

[LSSD03] H. Lamehamedi, B. Szymanski, Z. Shentu, and E. Deelman. Simulation of dynamic data replication strategies in data grids. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 10–20, Apr 2003.

[LV06] M. Lei and S. V. Vrbsky. A data replication strategy to increase data availability in data grids. In *Proceedings of the International Conference on Grid Computing and Applications*, pages 221–227, Jun 2006.

[MJR$^+$04] R. Moore, A. Jagatheesan, A. Rajasekar, M. Wan, and W. Schroeder. Data grid management systems. In *Proceedings of the 21st IEEE Conference on Mass Storage Systems and Technologies*, Apr 2004.

[MKL$^+$02] D.S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne,

B. Richard, S. Rollins, and Z. Xu. Peer-to-peer computing. Technical report, Technical Report HPL-2002-57, HP Labs, 2002.

[nee01] NEESgrid. http://www.neesgrid.org/, 2001. Building the National Virtual Collaboratory for Earthquake Engineering, Last accessed: Mar 5, 2010.

[ns208] NS-2 network simulator. http://www.isi.edu/nsnam/ns, 2008. Last accessed: May 15, 2010.

[Ora01] A. Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* O'Reilly and Associates, Inc., 2001.

[OV99] M. Ozsu and P. Valduriez. *Principles of distributed database systems.* Prentice-Hall, Inc., 2nd edition, 1999.

[PKG+04] L. Pearlman, C. Kesselman, S. Gullapalli, Jr. B. F. Spencer, J. Futrelle, K. Ricker, I. Foster, P. Hubbard, and C. Severance. Distributed hybrid earthquake engineering experiments: Experiences with a ground-shaking grid application. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 14–23, Jun 2004.

[PKKY03] S. Park, J. Kim, Y. Ko, and W. Yoon. Dynamic data grid replication strategy based on Internet hierarchy. In *Proceedings of the Second International Workshop on Grid and Cooperative Computing(GCC'03)*, pages 838–846, Dec 2003.

[PPD03] Particle Physics Data Grid (PPDG). http://ppdg.net/, 2003. Last accessed: Jan 21, 2009.

[PS00] E. Pacitti and E. Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *The VLDB Journal*, 8:305–318, Feb 2000.

[PvST02] G. Pierre, M. van Steen, and A. Tanenbaum. Dynamically selecting optimal distribution strategies for web documents. *IEEE Transactions on Computers*, 51:637–651, Jun 2002.

[RAD$^+$02] M. Russel, G. Allen, G. Daues, I. Foster, E. Seidel, J. Novotny, J. Shalf, and G. von Laszewski. The astrophysics simulation collaboratory: A science portal enabling community software development. *Cluster Computing*, 5(3):297–304, Aug 2002.

[RBA05a] R. M. Rahman, K. Barker, and R. Alhajj. Replica placement in data grid: A multi-objective approach. In *Proceedings of the International Conference on Grid And Cooperative Computing*, pages 645–656, Nov 2005.

[RBA05b] R. M. Rahman, K. Barker, and R. Alhajj. Replica placement in data grid: Considering utility and risk. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)*, volume 1, pages 354–359, Apr 2005.

[RF01a] K. Ranganathan and I. Foster. Design and evaluation of dynamic replication strategies for a high performance data grid. In *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics*, Sep 2001.

[RF01b] K. Ranganathan and I. T. Foster. Identifying dynamic replication strategies for a high-performance data grid. In *Proceedings of the International Workshop on Grid Computing (GRID'01)*, pages 75–86, Nov 2001.

[RIF02] K. Ranganathan, A. Iamnitchi, and I. Foster. Improving data availability through dynamic model-driven replication in large peer-to-peer communities. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*, pages 376–381, May 2002.

[RKA05] R. M. Rahman, K.Barker, and R. Alhajj. Replica selection in grid environment: A data-mining approach. In *Proceedings of the ACM symposium on Applied computing*, pages 695–700, Mar 2005.

[RSB01] P. Rodriguez, C. Spanner, and E. W. Biersack. Analysis of web caching architectures: Hierarchical and distributed caching. *IEEE/ACM Transactions on Networking*, 9:404–418, Aug 2001.

[RWMS04] A. Rajasekar, M. Wan, R. Moore, and W. Schroeder. Data grid federation. In *Proceedings of the 11th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 541–546, Jun 2004.

[sds00] Sloan digital sky survey (sdss). http://www.sdss.org/, 2000. Last accessed: Mar 10, 2008.

[SGE08] M. Shorfuzzaman, P. Graham, and R. Eskicioglu. Popularity-driven dynamic replica placement in hierarchical data grids. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'08), Workshop on HPDataGrid*, pages 524–531, Dec 2008.

[SGE09] M. Shorfuzzaman, P. Graham, and R. Eskicioglu. Adaptive popularity-driven replica placement in hierarchical data grids. *Journal of Supercomputing*, 51(3):374–392, 2009.

[SGE10a] M. Shorfuzzaman, P. Graham, and R. Eskicioglu. Adaptive placement of replicas in hierarchical data grids. In *Proceedings of the High Performance Computing Symposium (HPCS'10)*, Jun 2010.

[SGE10b] M. Shorfuzzaman, P. Graham, and R. Eskicioglu. Distributed popularity based replica placement in data grid environments. In *Proceedings of the 11th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'10)*, pages 66–77, Dec 2010.

[SGE11a] M. Shorfuzzaman, P. Graham, and R. Eskicioglu. Distributed placement of replicas in hierarchical data grids with user and system qos constraints. In *Proceedings of the the Sixth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC'11)*, pages 177–186, Oct 2011.

[SGE11b] M. Shorfuzzaman, P. Graham, and R. Eskicioglu. Qos-aware distributed replica placement in hierarchical data grids. In *Proceedings of the 25th IEEE International Conference on Advanced Information Networking and Applications (AINA'11)*, pages 291–299, Mar 2011.

[Sim02] C. Simatos. Making SimJava count. Technical report, The University of Edinburgh (UK), Master's Thesis, 2002.

[Sim08] Simscript: A simulation language for building large-scale, complex simulation models. http://www.simscript.org, 2008. Last accessed: Feb 10, 2011.

[SL90] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22:183–236, Sep 1990.

[SL00] Y. Saito and H. M. Levy. Optimistic replication for Internet data services. In *Proceedings of International Symposium on Distributed Computing*, pages 297–314, Oct 2000.

[SN02] V. J. Sosa and L. Navarro. Influence of the document validation/replication methods on cooperative web proxy caching architectures. In *Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation*, pages 238–245, Jan 2002.

[squ00] Squid Internet object cache. http://squid.nlanr.net/, 2000. Last accessed: Dec 20, 2009.

[SR03] F. Schintke and A. Reinefeld. Modeling replica availability in large data grids. *Journal of Grid Computing*, 1(2):219–227, Sep 2003.

[SSA$^+$02] H. Stockinger, A. Samar, B. Allcock, I. Foster, K. Holtman, and B. Tierney. File and object replication in data grids. *Cluster Computing*, 5:305–314, Jul 2002.

[SWH98] A. Sistla, O. Wolfson, and Y. Huang. Minimization of communication cost through caching in mobile environments. *IEEE Transactions on Parallel and Distributed Systems*, 9:378–390, Apr 1998.

[SYAD05] K. Seymour, A. YarKhan, S. Agrawal, and J. Dongarra. Netsolve: Grid enabling scientific computing environments. *Grid Computing and New Frontiers of High Performance Processing*, 14:33–51, 2005.

[Tho79] R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4:180–209, Jun 1979.

[TLM$^+$05] M. Tu, P. Li, Q. Ma, I. Yen, and F. Bastani. On the optimal placement of secure data objects over internet. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, pages 237–246, Apr 2005.

[TLYT05] M. Tang, B. Lee, C. Yeo, and X. Tang. Dynamic replication algorithms for the multi-tier data grid. *Future Generation Computing System*, 21(5):775–790, May 2005.

[TLYT06] M. Tang, B. Lee, C. Yeo, and X. Tang. The impact of data replication on job scheduling performance in the data grid. *Future Generation Computing System*, 22(3):254–268, Feb 2006.

[TS02] A. Tanenbaum and M. Steen. *Distributed Systems Principles and Paradigms*. Prentice Hall, 2002.

[TTGD04] R. Tuchinda, S. Thakkar, Y. Gil, and E. Deelman. Artemis: Integrating scientific data on the grid. In *Proceedings of the 16th Innovative Applications of Artificial Intelligence*, pages 892–899, Jul 2004.

[TX05] X. Tang and J. Xu. Qos-aware replica placement for content distribution. *IEEE Transaction on Parallel and Distributed System*, 16(10):921–932, Oct 2005.

[UC04] O. Unger and I. Cidon. Optimal content location in multicast based overlay networks with content updates. *World Wide Web*, 7:315–336, Sep 2004.

[VBR06] S. Venugopal, R. Buyya, and K. Ramamohanarao. A taxonomy of data grids for distributed data sharing, management, and processing. *ACM Computing Surveys*, 1:1–53, Jun 2006.

[VR07] V. Valloppillil and K. W. Ross. Cache array routing protocol v1.0. Internet Draft, 2007.

[WHCW06] C. Wang, C. Hsu, H. Chen, and J. Wu. Efficient multi-source data transfer in data grids. In *Proceedings of the Sixth IEEE International*

Symposium on Cluster Computing and the Grid (CCGRID'06), pages 421–424, May 2006.

[WL88] C. T. Wilkes and R. J. Jr LeBlanc. Distributed locking: A mechanism for constructing highly available objects. In *Proceedings of the Seventh Symposium on Reliable Distributed Systems*, pages 194–203, Oct 1988.

[WLL08] J. Wu, Y. Lin, and P. Liu. Optimal placement of replicas in data grid environments with locality assurance. *Journal of Parallel and Distributed Computing*, 68(12):1517–1538, Dec 2008.

[WLW06] H. Wang, P. Liu, and J. Wu. A QoS-aware heuristic algorithm for replica placement. In *Proceedings of the 7th ACM/IEEE International Conference on Grid Computing*, pages 96–103, Sep 2006.

[WM91] O. Wolfson and A. Milo. The multicast policy and its relationship to replicated data placement. *ACM Transactions on Database Systems*, 16(1):181–205, Mar 1991.

[YBdA$^+$07] C. S. Yeo, R. Buyya, M. D. de Assuncao, J. Yu, A. Sulistio, S. Venugopal, and M. Placek. *Utility Computing and Global Grids.* The Handbook of Computer Networks. John Wiley and Sons, 2007.