2009

Asynchronous replication of metadata across multi-master servers in distributed data storage systems

Ismail Akturk
Louisiana State University and Agricultural and Mechanical College

ASYNCHRONOUS REPLICATION OF METADATA ACROSS MULTI-MASTER SERVERS IN DISTRIBUTED DATA STORAGE SYSTEMS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

in

The Department of Electrical and Computer Engineering

by
Ismail Akturk
B.S., Dogus University, 2007
December, 2009

All praise is due to God alone, the Sustainer of all the worlds,

the Most Gracious, the Dispenser of Grace

Acknowledgments

I would like to express my deepest appreciation to my adviser Dr. Kosar for his support, invaluable advices, suggestions and steadfast guidance during my study at Louisiana State University. He is the one who made it possible to achieve graduate experience, work on the state-of-art projects, meet with scholars in different fields, and get a master's degree. No word, no sentence can fully express my gratitude to him.

Also, I would like to thank Dr. Vaidyanathan, co-chair of my thesis committee, for his guidance to obtain engineering approach and intuition. I am privileged to meet with him and take courses that have changed my perspective and give me a clear idea regarding how to solve engineering problems and be a better engineer.

Special thanks to Dr. Trahan for his grammatical suggestions while revising thesis, and Dr. Ramanujam for his time to be a part of my thesis committee. I would like to thank Distributed Systems Lab members at Computer Science Department, especially Mehmet Balman who was always excited to discuss and give positive comments, as well as the faculty and staff members of Center for Computation and Technology(CCT) for supporting me through out my study.

Finally, I would like to express my heartfelt gratitude to my family for their patience towards my absence during their hard times.

Table of Contents

List of Tables

List of Figures

Abstract

In recent years, scientific applications have become increasingly data intensive. The increase in the size of data generated by scientific applications necessitates collaboration and sharing data among the nation's education and research institutions. To address this, distributed storage systems spanning multiple institutions over wide area networks have been developed. One of the important features of distributed storage systems is providing global unified name space across all participating institutions, which enables easy data sharing without the knowledge of actual physical location of data. This feature depends on the "location metadata" of all data sets in the system being available to all participating institutions. This introduces new challenges. In this thesis, we study different metadata server layouts in terms of high availability, scalability and performance. A central metadata server is a single point of failure leading to low availability. Ensuring high availability requires replication of metadata servers. A synchronously replicated metadata servers layout introduces synchronization overhead which degrades the performance of data operations. We propose an asynchronously replicated multi-master metadata servers layout which ensures high availability, scalability and provides better performance. We discuss the implications of asynchronously replicated multi-master metadata servers on metadata consistency and conflict resolution. Further, we design and implement our own asynchronous multi-master replication tool, deploy it in the state-wide distributed data storage system called PetaShare, and compare performance of all three metadata server layouts: central metadata server, synchronously replicated multi-master metadata servers and asynchronously replicated multi-master metadata servers.

Chapter 1

Introduction

1.1 Distributed Data Storage Systems

Today, we are in the post-"Moore's Law"-era in which computational power doubles in every 14 months rather than every 18 months [43], as a result of the advances in multi-core technology. Furthermore, having shared wide area high speed networks brings the potential of building more powerful supercomputers in the form of distributed clusters. As of now, 82% of the supercomputers listed in the TOP500 are the composition of clusters that provide more than 13 Peta-FLOPS of computational power [12].

These advancements in hardware and network technologies make the number of computation-intensive scientific disciplines and applications increasing. The data volume commensurate with the increase of computational power exceeds Petabyte boundary for a number of scientific disciplines and applications, such as high energy physics [21, 42], computational biology [50, 49], coastal modeling [22, 56], computational fluid-dynamics [30], numerical relativity [14], and astrophysics [13, 62]. Such scientific applications require allocating computational and storage resources, as well as organizing, moving, visualizing, analyzing and storing massive amounts of data at multiple sites around the nation and the world [61]. Having stringent performance requirements, large volume of data sets, and geographically distributed human, computational and storage resources makes existing data management infrastructures insufficient [23]. Simply purchasing high-capacity, high-performance storage systems and adding them to the existing computing infrastructure of the collaborating institutions do not solve the underlying and highly challenging data handling problems. Scientists are compelled to spend a great amount of time and energy on solving basic data-handling issues, such as how to find physical location of data, how to access it, and/or how to move it to visualization and/or compute resources for further analysis.

There is a wide variety of distributed file systems developed to alleviate data management challenges in clusters, such as AFS [34], NFS [5], Lustre [2], PVFS [37], GFS [11], GPFS [54] and Panasas [41]. These file systems are sufficient and widely used in LANs as a cluster file system. However, when the volume of generated data sets increases and data sets are distributed over the clusters through WANs, it becomes very

expensive to maintain a unified shared file system running across distributed clusters. This is due to the constraints of WAN, heterogeneity of distributed resources and environments, and authorization/authentication policies of different administration domains. To address the challenges of data handling issues in geographically distributed and heterogeneous environments, distributed data storage systems have been proposed and implemented.

Distributed data storage systems provide flexible mechanisms for controlling, organizing, sharing, accessing and manipulating data sets over widely distributed resources that are under the control of different administration domains. One of the important features of distributed storage systems is providing global unified name space across distributed resources, which enables easy data sharing and accessing without the knowledge of actual physical location of data. This is known as 'location transparency'. The location transparency of distributed data sets is provided efficiently by distributed data storage systems. Distributed storage systems enable scalable, efficient and transparent access to the distributed resources including replicated data sets in different resources to enable fast access while ensuring data coherency. They can issue fine-grained authentication and authorization policies over shared and distributed resources as well as data sets.

The distributed data management challenge has also been discussed in the grid environments [36], known as data grids [23]. Current distributed data management solutions in data grids are largely based on replica catalogs [24]. However, they require to use customized APIs to access replica catalogs which brings burden of modifying existing applications according to the particular API. This is inconvenient for scientist and makes their applications restricted by certain APIs which degrades the portability of the applications [38]. A promising approach is to collect distributed data storage resources together under the unified logical global name space and provide a posix compatible interface which allows applications to access distributed resources and data sets transparently. This approach is taken while designing and implementing most of the distributed data storage systems [18, 57, 60, 65].

In the context of distributed data storage systems, the unified global name space and the information regarding data, such as file name, file size and physical location of the file, constitute the metadata(a.k.a "data about data"). In general, the data sets stored in distributed resources are accessed through data servers, while metadata is managed separately by metadata servers in distributed data storage systems for efficiency.

1.2 Metadata Management in Distributed Data Storage Systems

In the context of distributed data storage systems, metadata consists of the relevant information of data objects and system components (i.e. storage and network resources, users), such as physical location of resources, logical hierarchy of unified global name space, file name and size, physical location of file, file permission, and user information. Basically, metadata describes what is where, how and who can access it. In general, data and metadata are handled separately for making system design and management easier in distributed data storage systems. For this reason, physical data access is controlled by data servers while metadata is stored in and managed by metadata server(s). Topologically, we have the following server layouts for metadata [19]:

- Centralized Metadata server
- Distributed Metadata servers
- Federated or Hybrid Metadata servers

The centralized metadata server layout is the easiest one to implement and widely used in distributed data storage systems. A central metadata server enables convenient and easy access to metadata since it is located in a well known location. Also, it makes system management easier since it is the only administrative point in the system. However, a centralized metadata server is a single point of failure in the system leading to low availability. In the case of failure of the central metadata server, there is no way to retrieve metadata information regarding data objects which enable to locate the data sets stored in distributed storage resources. Another problem with the centralized metadata server is that all the requests have to be processed by the single metadata server which slows down the data operations and become a performance bottleneck in the system. Furthermore, the scalability of the overall system suffers as the number of storage resources increases which increases the load on the metadata server. Although the overhead of accessing metadata information is generally neglected compared to the time spent on data movement, it is noticed that 50% to 80% of all file system accesses are to metadata information [44]. This implies that having efficient metadata information retrieval is critical. A sample central metadata server deployment is illustrated in Figure 1.1.

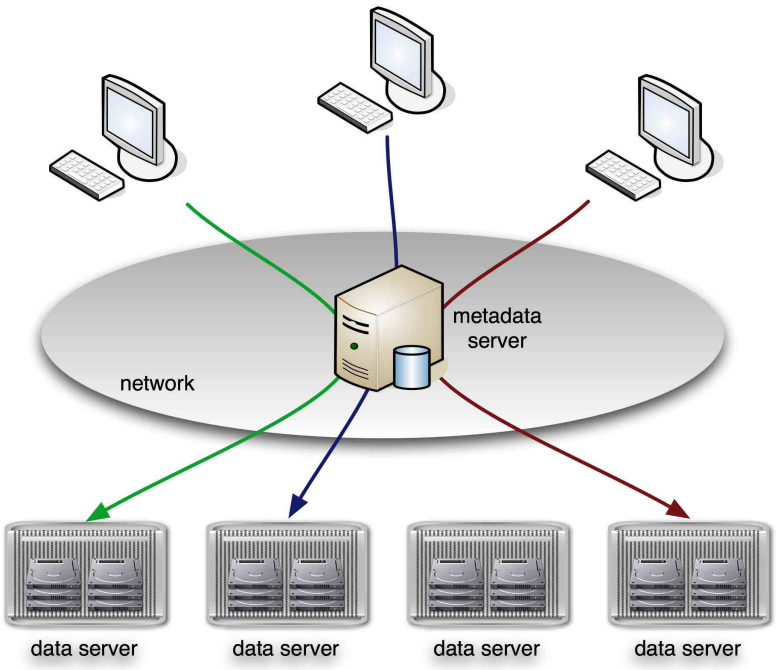


Figure 1.1: Central Metadata Server

In this thesis, we assume that each metadata server contains complete metadata information in the distributed metadata servers layout. A distributed metadata server layout provides high availability and avoids the limitation on scalability of the system as opposed to centralized metadata server layout. The load can be distributed among existing metadata servers. Distributed metadata servers enable faster metadata retrieval since metadata is often stored physically close to the data sets to which it refers. However, distributed metadata servers must be synchronized somehow to ensure metadata consistency. The way of synchronizing metadata servers has an impact on the metadata consistency as well as performance and scalability of the system. This makes it challenging to find a balance between metadata consistency, performance and scalability. A sample distributed metadata server layout is shown in Figure 1.2.

A federated metadata servers layout consists of servers which contain separate portions of the overall metadata information. The metadata information is partitioned among available metadata servers. There are two main approaches to partition the metadata among metadata servers. The first one is directory sub-tree partitioning which partitions the logical global name space according to logical directory sub-trees [67, 52]. The major disadvantage of directory sub-tree partitioning is that the workload may not be evenly distributed among metadata servers. This happens if one of the sub-tree is accessed heavily while other sub-trees are

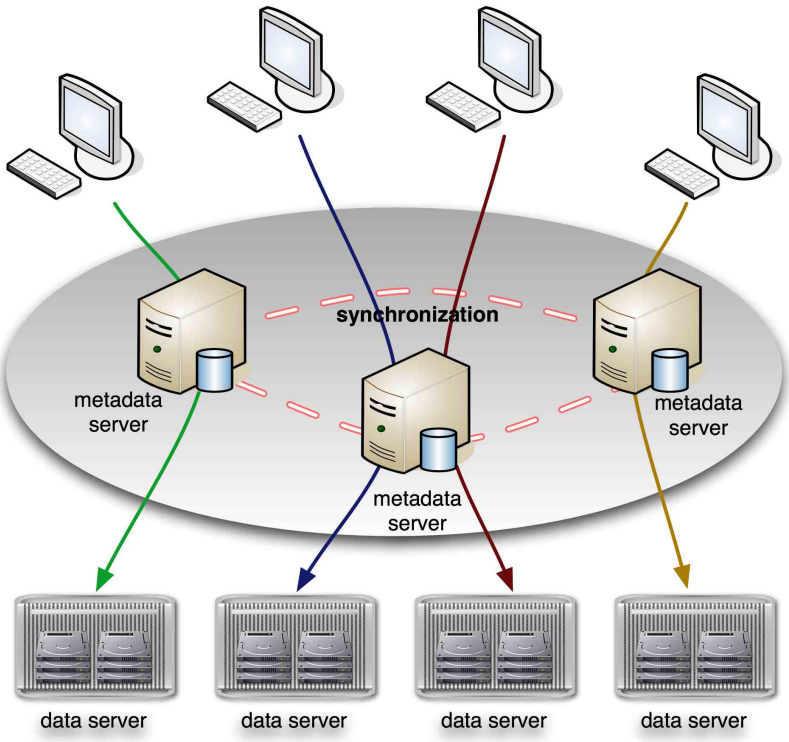


Figure 1.2: Distributed Metadata Servers

accessed rarely. The second approach is hash-based partitioning which splits metadata among metadata servers according to the hash values generated by using file name, file size or other related information [27]. Although it randomizes the distribution of workload among metadata servers, it requires to compute hash values of data objects which takes time. There are also dynamic and hybrid partitioning approaches which try to avoid the shortcomings of directory sub-tree partitioning and hash-based partitioning to achieve a desirable load distribution across metadata servers and faster processing [20, 29]. A sample layout of federated metadata servers based on sub-tree partitioning is shown in Figure 1.3. Federated metadata servers do not need to be synchronized as opposed to the distributed metadata servers since each metadata server contains mutually exclusive part of the overall metadata information. The federated metadata servers layout seems to be appropriate to distribute the workload and to scale; however, it comes with an extra cost of requiring to traverse multiple metadata servers, if relevant metadata is not stored in the current metadata server, which degrades the system performance. Also, federated metadata servers layout suffers from availability similar to the centralized metadata server layout. There is no way to retrieve metadata which is stored in the failed

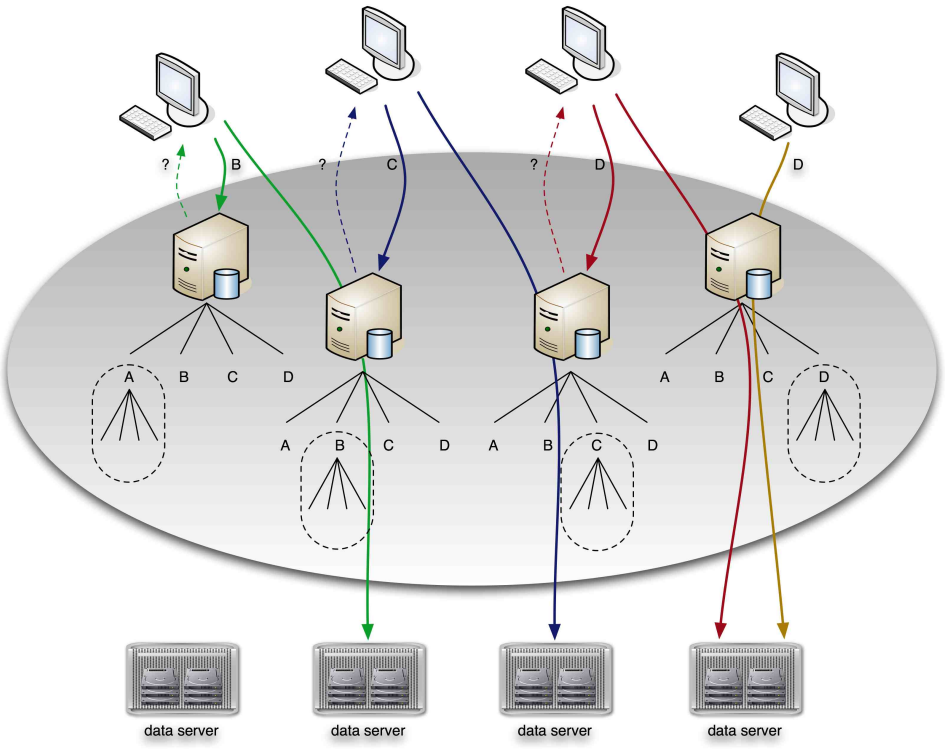


Figure 1.3: Federated Metadata Servers

server. However, it is not as severe as it is in the centralized metadata server layout. Only the data sets that are represented by the failed metadata servers become unavailable.

Due to the above mentioned limitations of central metadata server layout and federated metadata servers layout, we focus on distributed metadata servers in which complete metadata has been stored and replicated. This ensures high availability of complete metadata and provides load distribution across metadata servers. We discuss the implications of the synchronization methods on scalability, performance and load distribution as well as metadata consistency and conflict resolution in the following chapters.

1.3 Problem Definition

There are two main components in distributed data storage systems; one is *data server* which coordinates physical accesses (i.e. writing/reading data sets to/from disks) to the storage resources, and the other one is *metadata server* which provides global name space to ensure location transparency of data sets and keeps all kinds of related information regarding data objects. As well as other design issues and system com-

ponents, metadata server layout has impact on the following system metrics: *availability*, *scalability*, and *performance*. An efficient metadata server layout should maximize these system metrics. In this thesis, we address the pros and cons of different metadata server layouts and propose the asynchronously replicated multi-master metadata server layout which maximizes the metrics mentioned above in distributed data storage systems.

Centralized metadata server layout is widely used approach to provide global name space in distributed data storage systems. We have three evident problems in this layout: *i)* the centralized metadata server becomes a single point of failure leading to low availability; *ii)* all read/write requests need to go through this metadata server first decreasing the overall performance; and *iii)* the scalability suffers as the number of participating sites increases due to increased load on the metadata server.

Replication of metadata is necessary to ensure high availability in distributed data storage systems. To achieve better availability, a metadata server should be deployed along with a data server on each storage site. This enables a data server to have the advantage of cooperating with a local metadata server which minimizes the cost of metadata retrieval. If a metadata server fails in one of the storage sites, there are available metadata servers located in the other storage sites which ensure high availability.

The existence of multiple metadata servers requires to define the role of each of the metadata servers. A metadata server can be either a master which has the capability of updating metadata, or a slave which can only read metadata. In distributed metadata servers layout, we have two main approaches: either all metadata servers are masters called multi-master, or only one metadata server is master while others are slaves called master-slave. In the one master option, any operation that requires updating metadata, such as writing a data set, must be processed by the master server which is similar to the central metadata server layout. The main benefit of master-slave replication is to have an alternative metadata server to use in the case of failure in the master metadata server which alleviates the availability problem of the system. If failure occurs in the master metadata server, one of the slaves temporarily becomes master. Although, master-slave replication provides better availability, it does not scale well since the write operations can only be processed by the master server. Master server becomes overwhelmed when the volume of write operations increases and drifts to low performance. For this reason, we focus on multi-master replication instead.

Having a master metadata server in each storage site relieves the burden on the single master metadata server that is the case in central and master-slave metadata server layouts. However, it is challenging to find the balance between performance, scalability and consistency while updating multi-master metadata servers. Metadata servers can be updated either synchronously or asynchronously. In synchronous replication, an incoming request that requires metadata update is propagated to all metadata servers before it is committed. Metadata information is updated if and only if all metadata servers agree to commit the incoming request. Propagating incoming requests to all metadata servers and receiving corresponding acknowledgments take time which degrades the performance. We exploit asynchronous replication to eliminate the overhead of synchronous replication while updating metadata servers. Asynchronous replication allows a metadata server to process incoming request on its own without propagating the request to others immediately. Metadata servers are updated asynchronously in the background by delaying update messages. This increases performance, especially for write operations since immediate synchronization is not enforced. A challenge of asynchronous multi-master metadata server replication is that it yields to have inconsistent metadata unless all metadata servers are synchronized. We further discuss the implications of asynchronous replication in multi-master layout on metadata consistency and conflict resolution in this thesis.

1.4 Thesis Organization

The rest of this thesis is organized as follows: in Chapter 2, we give background information about existing replication models used in the database community. Further, we look into metadata server replication in distributed data storage systems in Section 2.5. Then, we move into the details of asynchronous multi-master replication in Chapter 3. We discuss the challenges of asynchronous replication, such as conflict detection and resolution, in Section 3.1. In Section 3.3, we examine the characteristics of the distributed data storage systems and demonstrate the feasibility of deploying asynchronously replicated multi-master metadata servers in such systems. In Chapter 4, we introduce the state-wide distributed data archival infrastructure called PetaShare, in which we have deployed asynchronously replicated multi-master metadata servers. We compare three replication layouts in PetaShare and provide performance results in Chapter 5. In Chapter 6, we compare our work with the most relevant related work. We summarize and conclude our thesis in Chapter 7.

Chapter 2

Replication Schemes and Types

In general, metadata is stored in a relational database, so when we refer to a metadata server we basically mean a database server that contains metadata. For this reason, replication schemes and types discussed for metadata servers here can be used interchangeably with database replication. However, the distributed storage system type of applications are different than traditional database applications which require special system requirement and characteristics analysis to fully exploit the potential of the replication and overcome the constraints of different replication scenarios. We discuss system requirements and characteristics of distributed storage systems in database replication point of view in Section 2.5; thus, here we discuss replication schemes and types in general domain.

Metadata server replication is used to sustain high availability and enhance system performance. Accessing metadata through local metadata server improves performance, meanwhile distributing workload among metadata servers leads system to balance the load on metadata servers. Having replicated metadata servers ensures metadata to remain available through replicated metadata server in the case of failure in particular metadata server. However, most of the replication settings used for high availability often lack of performance and scalability which makes it challenging to exploit replicated metadata servers [25].

The word "replication" implies the existence of more than one server. Having more than one server requires to define the capabilities and responsibilities of each server. Basically, *replication scheme* indicates the capabilities as well as responsibilities of each metadata server to be replicated. There are two replication schemes: *master-slave* and *multi-master* replication scheme. In broad sense, there is only one master server that is designated to process all the requests and has capability to update database by itself in master-slave replication scheme, while slaves can only replicate the updates that have been committed by master server. Contrary, all the replicating servers have capability of processing the requests and updating database in the multi-master replication scheme.

In general, replication should ensure that there is more than one copy of the database at any point in time. No matter what replication scheme is used, the need of having copies of database requires synchronization among replicating servers. The replication type specifies the synchronization policy that is employed to

replicate servers. The replicating servers can be updated either *synchronously* or *asynchronously* [26, 31]. Synchronous replication enforces immediate synchronization among replicating servers. On the other hand, asynchronous replication does not enforce immediate synchronization; instead, it delays the propagation of update messages committed by metadata server which have to be replicated in all metadata servers in order to synchronize metadata servers. Table 2.1 shows possible combinations of replication schemes and types that can be deployed as a layout.

Table 2.1: Combination of Replication Schemes and Types

Synchronous Master-Slave	Asynchronous Master-Slave
Synchronous Multi-Master	Asynchronous Multi-Master

2.1 Synchronous Replication

In synchronous replication, incoming requests are propagated to and processed by all replicating servers immediately. The benefit of synchronous replication is to guarantee that all ACID (i.e. atomicity, consistency, isolation, durability) [32] requirements are fulfilled.

While propagating requests and synchronizing servers, two-phase commit protocol is used [55]. When a request comes in a server, the same request is also forwarded immediately to all replicating servers. All servers have to process incoming request to see if it is OK to be committed, and have to inform propagating server in this regard. If and only if all replicating servers inform that request can be committed, then second message is propagated to commit the request in all replicating servers. If any replicating server complains about the request, then abort message is propagated and all servers have to disregard the request. An illustration of two-phase commit protocol is shown in Figure 2.1.

Although it ensures that replicating servers are synchronized immediately when a request is committed and prevents consistencies may occur otherwise, it generates huge network traffic due to high number of sends and receives to decide to commit or abort. It increases processing latency which degrades operation performance since operation has to wait until all replicating servers have been synchronized. Scalability also

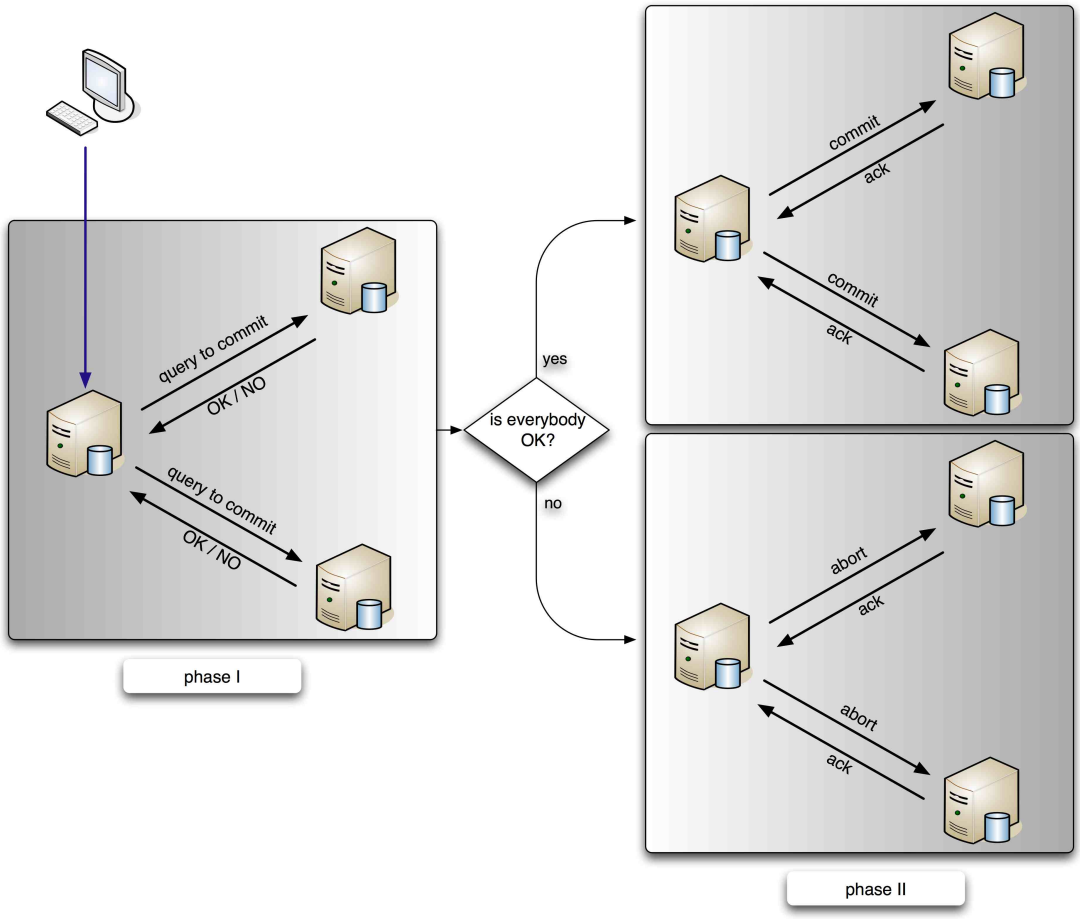


Figure 2.1: Two-phase Commit Protocol in Synchronous Replication

suffers from increasing number of replicating metadata servers that tends to create exponentially growing network traffic and processing latency that ends up with longer response time.

2.2 Asynchronous Replication

In asynchronous replication, an incoming request is processed and get committed on the receiving server without propagating it to other replicating servers simultaneously. Instead, committed requests are deferred and sent to all other replicating servers asynchronously. Once replicating servers receive these deferred requests, they process them and make themselves synchronized. Contrast to synchronous replication, asynchronous replication may suffer from inconsistencies due to possibility of simultaneous updates occurred in different replicating servers for the same metadata. Although there are ways to detect and resolve the conflicts, deployment of asynchronously replicated servers is extremely application dependent. To deploy

asynchronously replicated servers, the application must have tolerance to the inconsistencies at least for a certain amount time (i.e. until all servers are synchronized. Thus, asynchronous replication requires meticulous analysis of characteristics of targeted application domain.

Although, asynchronous replication has consistency and conflict challenges to deal with, it utilizes network resources intelligently, creates less traffic, and provides higher performance. Deferring multiple requests and propagating them all as a big chunk of requests is much more efficient rather than to propagate each of them separately [35]. Operation latency is reduced as opposed to synchronous replication because a server can go ahead and process a request without need to talk with other servers to commit it. It also provides better scalability since response time of a server is independent from the number of replicating servers, and generated network traffic is proportional to the number of replicating servers. Moreover, network latency introduced due to the geographical distance between replicating servers can be tolerated and hidden since requests are deferred and propagated asynchronously.

2.3 Master-Slave Replication

In master-slave replication, there is only one server in the system which is capable of updating database called master server. All other replicating servers are called slaves and can only accept read-only requests. Any operation that requires to update database must go through and processed by master server. The updates performed by master server can be propagated on slave servers either synchronously or asynchronously.

In master-slave replication, the master server becomes overwhelmed and system suffers from scalability and drifts to low performance for the write intensive applications. However, the load for read operations can be distributed among the slave servers. For this reason, master-slave replication is mostly used in read-intensive applications for load distribution purposes. Moreover, master-slave replication provides high availability and easy fail-over mechanisms. If a master fails, one of the slaves can be promoted as a master and starts to accept requests that need database update. When the master server is back, it can stay as a slave, or master role can be given back to it after being synchronized and temporary master becomes slave again. An illustration of master-slave replication is shown in Figure 2.2.

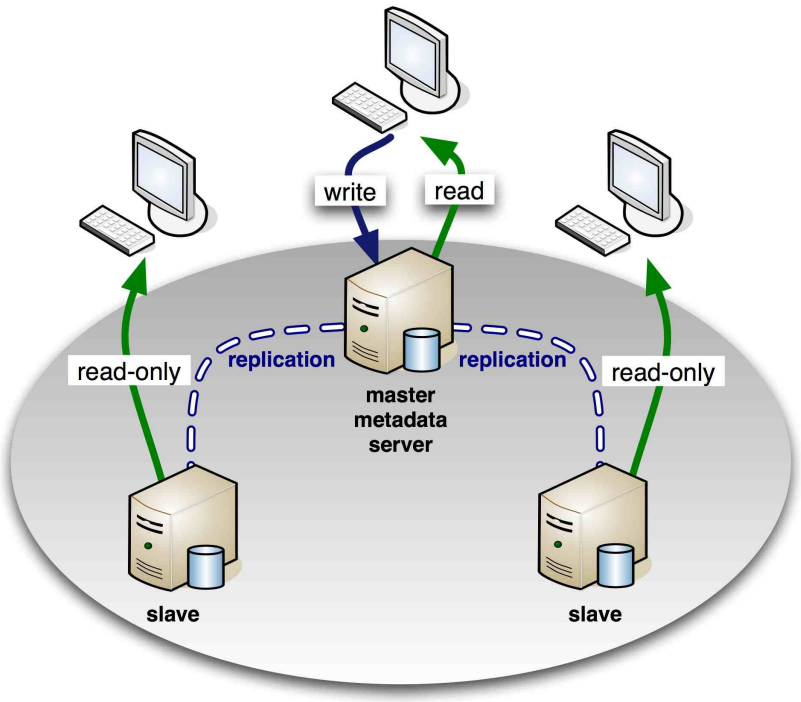


Figure 2.2: Master-Slave Replication

2.4 Multi-Master Replication

In multi-master replication, all the servers run as master servers, thus can update database by their own. This makes system highly flexible in a way that any operation can be processed in any server which enables better load balancing. However, capability of updating database of multiple servers brings the challenge of keeping servers consistent. A conflict occurs if more than one server tries to update the same object in the database simultaneously. Detecting and resolving such conflicts require advance resolution schemes to be designed and deployed that is the most challenging issue in multi-master replication. Most widely used strategy for avoiding conflicts in multi-master layout is to use synchronous replication. Since each server has to agree to commit the given request, it is possible to avoid the request that cause a conflict in synchronous replication. Although, this approach is straightforward and efficient in many cases, it introduces high latency and network overhead due to synchronous replication which slow down all the system. An illustration of multi-master replication is shown in Figure 2.3.

Multi-master replication can be fully exploited if the consistency requirement can be relaxed for the application. However, relaxing consistency requirement is extremely application dependent and it is not

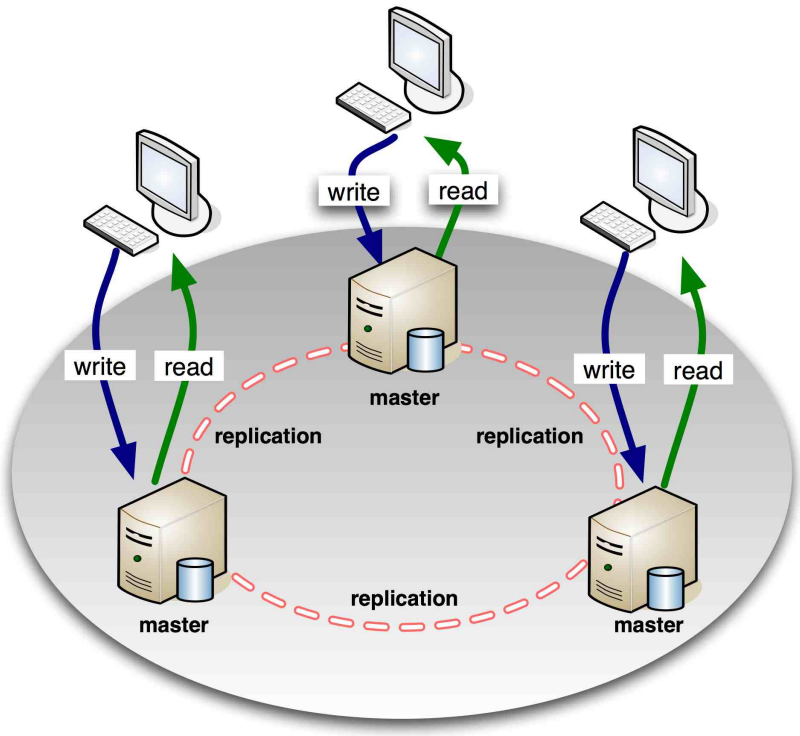


Figure 2.3: Multi-master Replication

always possible. In some applications relaxing consistency requirement may cause catastrophic failures. However, for applications that can relax consistency requirement and survive with inconsistencies for a while, it is possible use asynchronous replication to obtain high performance and make system highly scalable. In such applications, synchronization can be done asynchronously which eliminates the bottlenecks of synchronous replication.

It is worth to mention that asynchronous multi-master replication has its own merit to provide high availability, scalability and performance; however, it is not possible to deploy this replication method in all kind of applications since it relaxes the consistency requirement. For example, banking applications can not tolerate inconsistencies which might cause unintended results. In this thesis, we investigate on and show feasibility of asynchronous multi-master metadata replication in distributed data storage systems to achieve high availability, scalability and increased performance while keeping system away from catastrophic failures caused by possible inconsistencies.

2.5 Metadata Server Replication in Distributed Data Storage Systems

As mentioned in Section 1.2 before, central metadata server is most widely used approach while designing and deploying a distributed data storage system. This is because it is easy to configure, control and manipulate system settings from a single point of administration. Also, it is convenient to implement applications for developers because they do not need to write complex algorithms to locate metadata server (i.e. it is located in well known location), as opposed to ones who need to write complex algorithms to locate relevant metadata server in federated metadata servers layout.

Although central metadata server provides convenience to system administrators and application developers, it makes system vulnerable to the failures. In the case of failure in metadata server for some reason, such as power outage, network disconnection and hardware failure, overall system becomes offline. Even-though data servers are up and running on distributed resources, there is no way to access and manipulate data sets stored in storage resources, because no one can translate logical path given in global name space into the physical location of storage resources and data sets. Being single point of failure in the system, central metadata server makes overall system suffer from low availability.

Increasing number of data storage resources distributed among geographical locations, as well as operations made in the system overwhelms the central metadata server. It has to keep track of metadata information of all storage resources and data sets associated with them, and it has to process all incoming requests made to access distributed resources and data sets. The scalability of the system suffers as the number of storage resources increases (i.e. it also implies increased number of requests) in the central metadata server layout.

As it is observed over the several years in high-performance computing community, write performance bottleneck is more than read performance bottleneck [43]. A metadata server has to be updated at the end of each write operation which introduces metadata access and processing latency in distributed data storage management. This latency is correlated with the distance between client and metadata server, as well as the workload metadata server has. In most cases, central metadata server is located in remote site and overwhelmed by high volume of requests. Thus, write performance may suffer even more due to increased

latency in central metadata server layout. It is not only the write performance, but read performance may also suffer in central metadata server layout because of the same reasons.

A federated metadata servers layout can be used to relieve the burden on central metadata server. In this model, global name space maintained in metadata server is partitioned among metadata servers in a certain manner, and each metadata server becomes responsible for only part of the global name space. Global name space can be partitioned by directory sub-tree partitioning or hash-based partitioning techniques. The directory sub-tree partitioning splits the logical global name space according to logical directory hierarchy into the sub-trees [52, 67]. Then, each sub-tree is assigned to particular metadata server. On the other hand, hash-based partitioning uses hash values generated by using file name, file size or other related metadata to split global name space and other metadata information of the system over federated metadata servers. Hash-based partitioning is used to distribute the workload among metadata servers as arbitrary as possible for balancing the workload in a much fair manner. Metadata servers may suffer from imbalanced workload in directory sub-tree partitioning, if any particular directory sub-tree becomes a hot-spot (i.e. accessed heavily) in the global name space.

Although federated metadata servers layout provides better scalability and load balancing than central metadata server layout, it introduces inconveniences for clients who are accessing data sets in distributed storage resources. This is because they have to look for corresponding metadata server in which relevant metadata information of requested data set is stored. This may require to traverse more than one metadata server in the case of metadata of requested data set is not stored in current metadata server. Traversing federated metadata servers until finding out respective metadata introduces high latency which degrades the performance of read operations. Similarly, write operations may slow down because of latency introduced while deciding the metadata server that will store the corresponding metadata of data set that is being written.

Federated metadata servers layout does not fulfill the high availability requirement of the system either. If a failure occurs in a particular metadata server, the portion of the global name space that is maintained by failed metadata server becomes inaccessible. Any data set that is mapped under that portion of global name space becomes inaccessible during the failure of corresponding metadata server. The failure in federated metadata server layout is less severe compared to the central metadata server layout in which a failure causes

all system to be unavailable.

On the other hand, distributed metadata servers (i.e. replicated metadata servers) have a potential to alleviate the challenges of availability, scalability, load balancing and performance in distributed data storage systems. As opposed to federated metadata servers, all replicating metadata servers have complete metadata information of the system. Having multiple copies of metadata information on multiple metadata servers that are geographically apart from each other ensures high availability as well as faster metadata access. If any metadata server fails, there is always at least one metadata server that contains complete information of global name space which ensures high availability. Moreover, it is no longer needed to traverse multiple metadata servers to find out metadata of requested data set which is the case in the federated metadata servers layout if metadata of requested data set is not stored in the current metadata server.

Although replicated metadata servers ensure high-availability in any setting, scalability, load balancing and overall performance are extremely bound to the scheme and type of replication used in the system. In general, master-slave replication settings suffer from scalability due to imbalanced workload. All the operations that require metadata update, such as writing a data set into the storage, have to be processed by master metadata server. Although slave metadata servers can help to balance the load for read operations (i.e. do not require metadata update), the volume of write operations overwhelm the master server, and scalability may start to suffer from the increased number of write operations. Moreover, processing all write operations in master metadata server that is running on remote site introduce processing latency as well as network latency which degrades the performance. Moreover, using synchronous replication in master-slave layout introduces synchronization overhead which makes this setting extremely inefficient. We do not consider to deploy any master-slave replication setting due to the shortcomings mentioned above.

On the other hand, multi-master replication releives the burden on single master server and allows replicating metadata servers process both read and write operations. The capability of processing read and write operations on any replicating metadata servers brings the flexibility of distributing loads among metadata servers fairly.

Similar to the master-slave replication, the scalability and performance of the system are bound with the type of replication used. Synchronous multi-master replication introduces synchronization overhead as well as network traffic. It is worse than synchronous master-slave replication since each of the replicating

metadata server is a master and capable of updating metadata information. This synchronization overhead increases exponentially with respect to the number of replicating metadata servers which limits the system to scale. Especially, write operations suffer from the synchronization overhead introduced by synchronous replication. Although synchronous multi-master replication ensures consistency among metadata servers and provides high availability, we look forward to achieve a better performance and scalability by employing asynchronous multi-master replication in distributed data storage system.

Asynchronous replication eliminates the replication overhead due to immediate synchronization by deferring the propagation of the updates messages to all replicating metadata servers. Allowing metadata server to commit all operations on its own and send updates at a later time allows data operations to be processed without suffering from synchronization overhead. Since there is a metadata server along with data server in a storage site, write and read operations can be processed faster due to local access to metadata server which increases the performance. Keeping the consistency issue of asynchronous replication in mind, we argue that asynchronous multi-master metadata server replication satisfy all of the requirements of **high-availability**, **scalability**, **load-balancing** and **performance** of the system. A comparison of metadata server layouts in different replication settings is summarized in Table 2.2.

Table 2.2: Comparison of Metadata Server Replication Settings

	Central Server	Sync. Master-Slave	Async. Master-Slave	Sync. Multi-Master	Async. Multi-Master
Availability	low	high	high	high	high
Scalability	low	low	mid	low	high
Write Perf.	low	low	low	low	high
Read Perf.	low	mid	high	low	high
Inconsistency	none	none	may occur	none	may occur

In the following sections, we discuss consistency issues of asynchronous multi-master metadata replication as well as conflict detection/resolution schemes, analyze the characteristics of distributed data storage systems, and explain feasibility of asynchronous multi-master metadata replication in distributed data storage systems in practice.

Chapter 3

Asynchronous Multi-Master Metadata Server Replication

3.1 Challenges Associated With Asynchronous Multi-Master Metadata Server Replication

Allowing each metadata server to update metadata information on its own and deferring the propagation of update messages to other metadata servers brings the issue of inconsistency between replicating metadata servers. As opposed to the synchronous replication, asynchronous replication does not enforce immediate update of metadata servers. A metadata server does not propagate update requests to the replicating metadata servers and does not wait to receive confirmation from all of them before committing the update request. Instead, it processes and commits an update request on its own, and propagates the committed updates to other metadata servers asynchronously. If a metadata server updates metadata information of an object, such as file, that has been already updated by another metadata server, but respective update request has not been received yet, then conflict occurs on replicating metadata servers. They receive two update requests for the same object with different values, so they need to know which one is correct and must be committed.

Basically, there are three types of conflicts that may occur in replicating metadata servers [6]: *update conflicts*, *uniqueness conflicts*, *delete conflicts*.

An *update conflict* occurs when a metadata server updates metadata information of an object that has been updated recently by another metadata server, but respective update request has not been propagated to all metadata servers yet.

A *uniqueness conflict* occurs when different metadata servers create data objects with the same metadata information which have to be unique. For example, all the files in the same logical directory must have unique logical file names.

A *delete conflict* occurs when a metadata server updates metadata information of an object that has been deleted by another metadata server already, but delete request has not been propagated to all metadata

servers yet.

Some of the conflicts that are mentioned above can be avoided at the time of designing internal structure of metadata servers; however, some of them not. For the conflicts that can not be avoided in asynchronous multi-master metadata server replication, there are ways to detect and resolve them later. However, metadata servers will stay inconsistent until all updates requests have been propagated and all the conflicts have been resolved.

In distributed data storage systems, it can be tolerated to have inconsistent metadata servers until all metadata servers synchronize themselves to achieve better performance. However, it is unpleasing to have metadata servers that can not update themselves accordingly and remain inconsistent forever due to conflicts occurred while propagating update requests asynchronously. For this reason, any conflict that makes replicating metadata servers remain inconsistent must be detected and resolved, or avoided if possible.

In the next section, we discuss the ways of avoiding some of the uniqueness conflicts at the time of designing metadata servers in a distributed data storage system. Further, we discuss the resolution of update and delete conflicts without causing any catastrophic failures in the system, as well as creating any inconvenience for the users.

3.1.1 Conflict Avoidance

Each object in the system, such as files, directories, physical storage resources and users, is associated with a unique ID which is a value obtained from monotonically increasing sequence of numbers. If an object is assigned an ID that has a value x, then the next object will be assigned with an ID has a value x+1.

In asynchronous replicated multi-master metadata server layout, if two metadata servers create new objects in their sites before receiving metadata create request of each other's, then both metadata servers will assign the same ID for created objects. In this case, uniqueness conflict occurs when they propagate create requests to all metadata servers, because there are two requests which assign same ID for different objects. This scenario is illustrated in Figure 3.1.

Uniqueness conflicts can be avoided at the time of designing internal structure of metadata servers. As mentioned above, ID is a value obtained from monotonically increasing sequence of numbers. We call this sequence of numbers as *ID space*. The ID space can be partitioned into the mutually exclusive sub-spaces

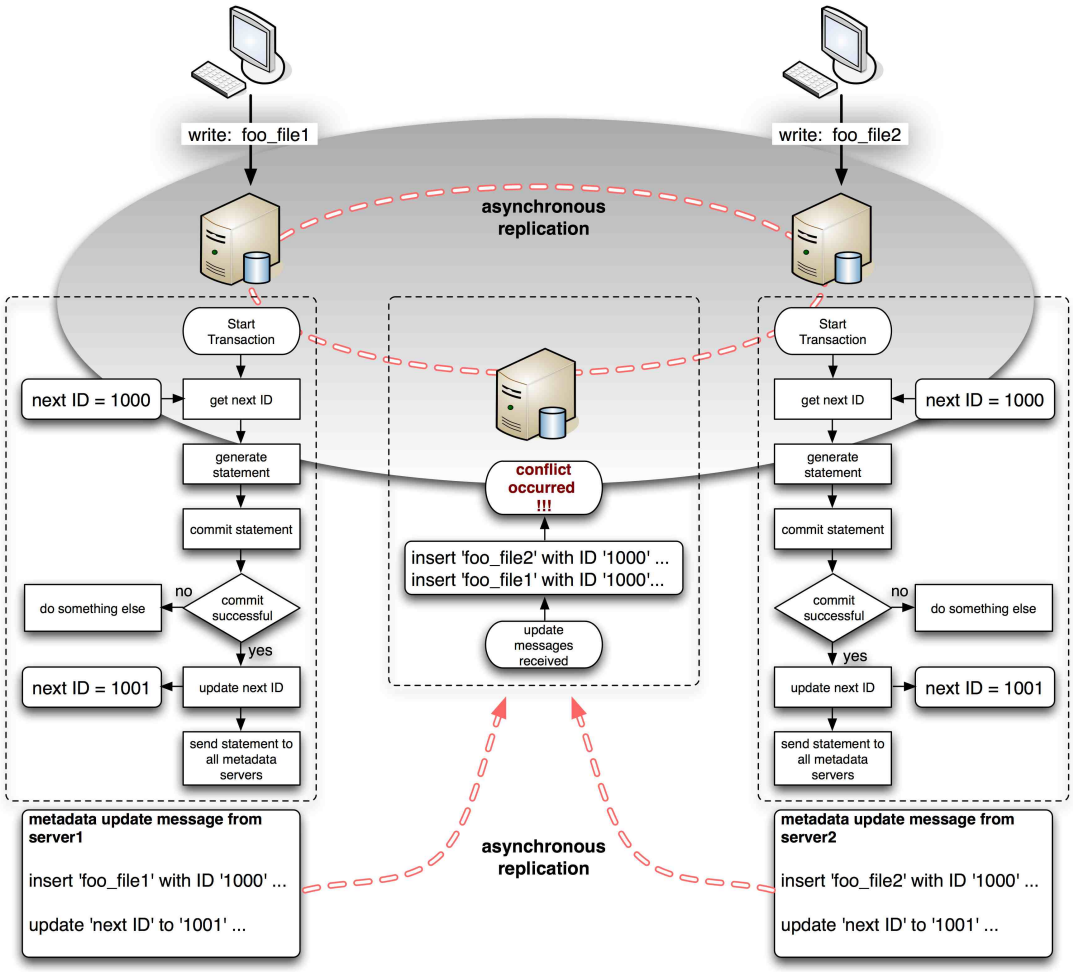


Figure 3.1: Uniqueness Conflict in Asynchronous Multi-Master Replication

and each sub-space can be associated with one of the metadata servers. Each metadata server is restricted to assign an ID to a new object from its ID sub-space. This ensures that metadata servers assign different IDs when they create new objects; although, they have not received metadata create request one from the other. It is worth to mention that although metadata servers are associated with mutually exclusive ID sub-spaces (i.e. they can only assign an ID from their ID sub-spaces), they still have capability of updating metadata information of objects created in other metadata servers. An illustration of ID space partitioning and conflict avoidance is shown in Figure 3.2.

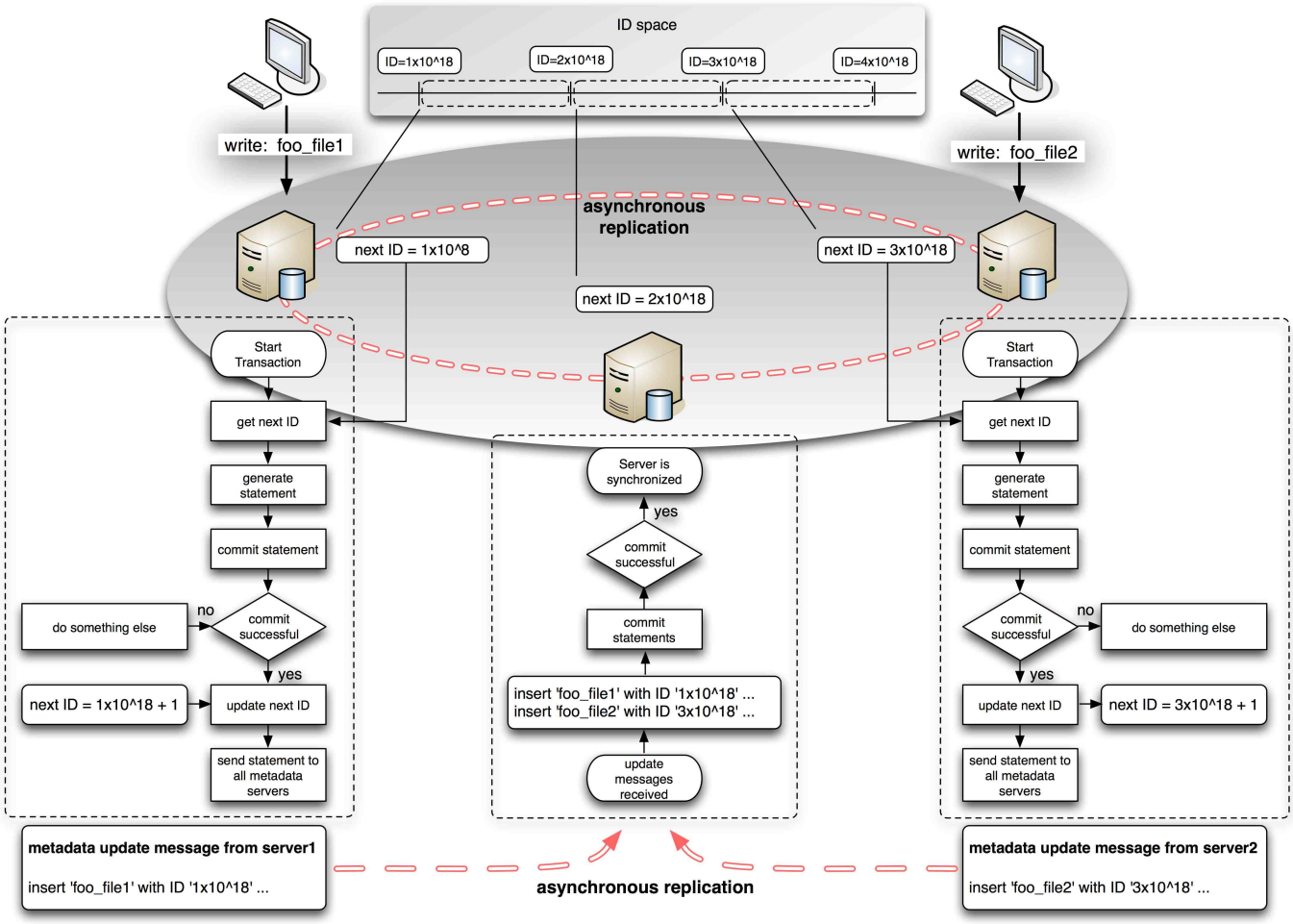


Figure 3.2: ID Space Partitioning to Avoid Uniqueness Conflict in Asynchronous Multi-Master Replication

3.1.2 Conflict Detection and Resolution

It is not possible to avoid all types of conflicts in asynchronously replicated multi-master metadata server layout. Especially update and delete conflicts are challenging to avoid unless the capabilities of metadata servers are restricted; however, we insist on exploiting multi-master replication without introducing any restriction on any of the replicating metadata servers. Although we can not design conflict-free asynchronously replicated multi-master metadata servers layout for our application domain, we can detect and resolve the conflicts efficiently. All replicating metadata servers become consistent again, after all the conflicts have been resolved.

Update conflicts occur only when more than one metadata server update the same metadata information before any one of the update requests has been received by all metadata servers. For example, a metadata

server X updates the file size of file F1 and commits this update. It defers to propagate this update request to the other metadata servers. Assume that metadata server Y updates the file size of the same file F1 before metadata server X propagates the update request. When both metadata server X and Y propagate their update requests, other metadata servers will have two update requests for the same file. It is not known which one is correct, thus causing a conflict. This conflict can not be avoided because all metadata servers are masters and can update metadata information of any object in the system. However, it is straightforward to resolve such conflicts by using *timestamps*. Metadata servers need to check the timestamps of both update requests when a conflict is detected. Then, they must process the update request that has the most recent time stamp and discard the other one.

There are update conflicts that timestamps are not sufficient to resolve them. Data servers in storage sites do not allow simultaneous updates for the same file, so timestamps of update requests will be different. For this reason, using timestamps is sufficient for metadata update requests which are generated due to physical access to the data, such as editing a file. However, there are operations that do not involve physical access to the data, but update metadata information of an object, such as changing permission of the file. For such operations, it is possible to update metadata information of the same object from different metadata servers simultaneously, which results in having exactly the same time stamp for both of the metadata update requests for the same object which are propagated from different metadata servers. Update conflict occurs when metadata servers receive these update requests with the same time stamp. A solution is to use *Site-priority* to resolve this kind of conflicts which is a unique numeric value assigned to all metadata servers. The metadata update request is accepted of whose site-priority is highest and the others are discarded.

There is a possibility of having uniqueness conflict if more than one metadata server create a new file in the same global directory with the same file name in different storage resources; although the files have different IDs (i.e. ID conflict is avoided by ID space partitioning). These conflicts are resolved by *renaming* the conflicted files.

For example, metadata server X creates a file called F1 in storage resource S1 under logical directory called DIR. Similarly, metadata server Y creates a file called F1 in storage site S2 under logical directory DIR before it receives metadata create request from metadata server X. When both metadata servers X and Y propagate metadata create requests, all metadata servers receive conflicting requests. Two different files

have been mapped by the same file name under the same logical directory. It is undesirable to discard any one of the requests. If any one of the requests is discarded, then there would be no way to access the file which has been already stored in storage resources since metadata information is missing for the file. Thus, to make both files accessible and to resolve the conflict, renaming is used. In the given example, file created in S1 is renamed as F1.X.S1 (i.e. metadata server name and storage name are appended), and the file created in S2 is renamed as F1.Y.S2. This renaming are also done in originating metadata servers. Although renaming of files seems unpleasant for users, it is more important to resolve the conflict that would cause serious inconveniences otherwise.

Delete conflict occurs when a file is deleted in one metadata server while it is updated in another metadata server. A temporary location is maintained for deleted files before moving them out from the system permanently. Deleted files are kept in temporary location unless all the metadata servers are synchronized and any one of the metadata servers ask for it. If there is a request made for a deleted file, then it is possible to roll-back the delete request, and file can be restored.

3.2 Implementation of Asynchronous Multi-Master Metadata Server Replication

Metadata information is kept in a relational database and managed by metadata server. Thus, metadata server replication and database replication can be used interchangeably. Implementing replication logic in database itself is complicated and creates extra work for database. For this reason, we design and implement our own replication tool called MASREP (Multi-master ASynchronous REPlication) which is maintained separately from the database. MASREP runs on the background and lets metadata server to run on its own. This allows database not deal with replication, and makes all replication related issues transparent to the database and users. Moreover, it provides flexibility of changing replication settings without interrupt or stop metadata servers.

Our replication strategy is based on transaction logs generated by databases. The databases to be replicated are configured to log only Data Manipulation Language (DML) statements (i.e. insert, update, delete) in their transaction logs. All statements in the transaction log correspond to one of the metadata update made in that metadata server. For this reason, these statements have to be replicated among all other meta-

data servers to make them all consistent and synchronized. Other operations, such as read operations, are handled by running *select* statements on metadata server. Since select statements do not change metadata information of any object, they are not needed to be replicated; thus, we avoid them to be logged in transaction log.

MASREP is responsible for processing transaction logs and sending/receiving them to/from its counterparts in all other replicating metadata servers. MASREP acts as a database client when it processes requests received from other metadata servers. It consists of five main components which are coordinating replication and synchronization related operations in the system. These components are *extractor*, *dispatcher*, *collector*, *injector* and *conflict resolver*. Along with these components, MASREP maintains two types of statement queues: *outgoing-queues* and *incoming-queue*. Outgoing-queues are used to store the statements that must be propagated to metadata servers to make them synchronized. There are separate outgoing-queues for each replicating metadata server. On the other hand, incoming queue is used to store the statements that have been received from other metadata servers. The components of asynchronous replication tool and interaction among them are shown in Figure 3.3.

We define a *replication cycle* that identifies the sequence of actions have to be made to replicate and synchronize metadata servers. Basically, it is a duration of time in which all replication related functions have been completed. Replication cycle starts with executing statements stored in incoming-queue (i.e. by injector), and goes on with extracting statements from transaction log of metadata server, and filling them into the respective outgoing-queues (i.e. by extractor). Then, statements in outgoing-queues are sent to the respective metadata servers (i.e. by dispatcher). After this step, there is pre-defined waiting (i.e. sleeping) period. A replication cycle finishes when waiting period is over, and a new replication cycle starts. It is expected that all metadata servers become synchronized at the end of the replication cycle. Although, it is said that synchronizing metadata servers once in a minute is sufficient [35]), we synchronize all metadata servers in every 30 seconds to reduce the duration of inconsistencies. We observed that all metadata servers become synchronized before any request comes for updated data object through other metadata servers. A typical replication cycle is shown in Figure 3.4.

In MASREP, *extractor* component process transaction log of replicating metadata server to find the statements that have been committed within last replication cycle. Also, it is responsible for eliminating

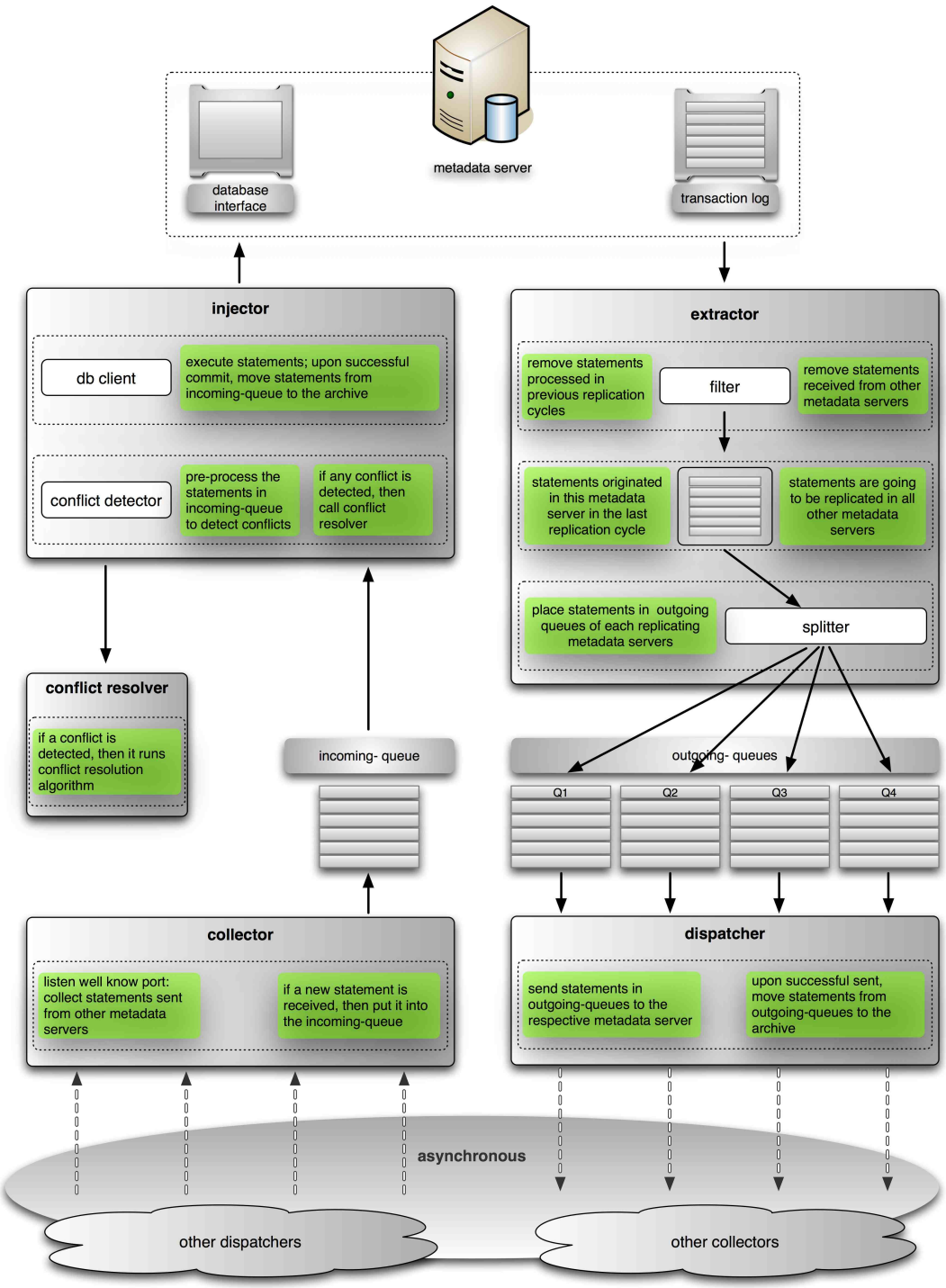


Figure 3.3: Components of Asynchronous Multi-Master Replication Tool

the statements that have been received from other metadata servers. It is worth to mention that transaction log contains both the statements that are originated in actual metadata server, and also the statements that are received from other metadata servers. If the statements that are received from other metadata servers

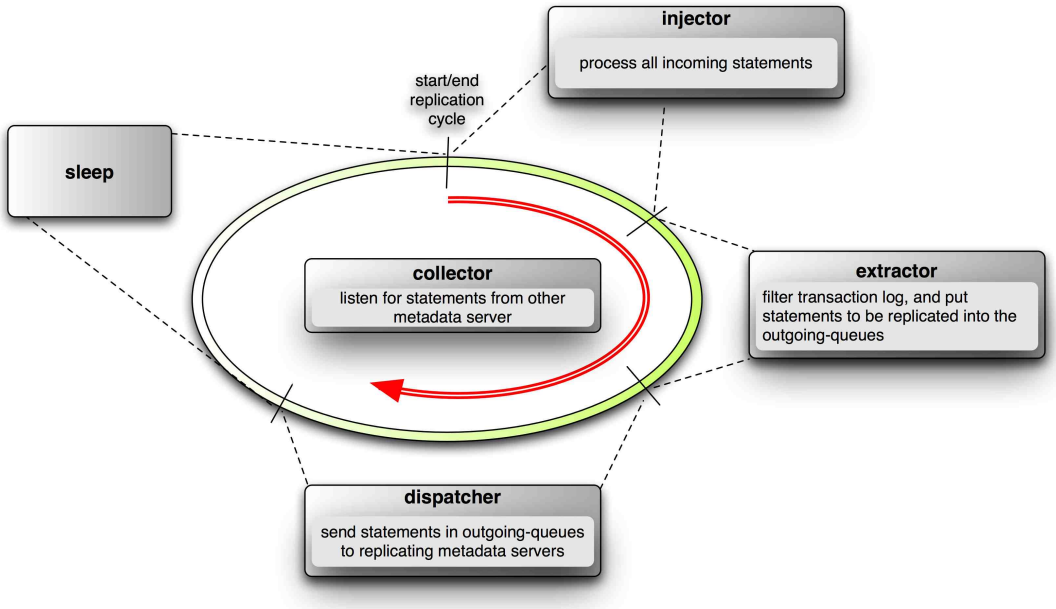


Figure 3.4: Replication Cycle

are propagated again to other metadata servers, metadata servers will receive the statements that they have executed before and they will re-run and re-send the these statements which creates infinite loop. For this reason, extractor is responsible for extracting the statements that are originated in the actual metadata server. Only these statements should be propagated to other metadata servers to make them synchronized. Extractor makes copies and moves these statements (if any) that have to be replicated into the outgoing-queue of each metadata server.

Dispatcher is responsible for propagating the statements (i.e. statements that have been filled by extractor) in the outgoing-queues to the respective metadata server. If dispatcher sends these deferred statements to the respective metadata server successfully, then these statements are removed from respective outgoing-queue. If dispatcher can not send statements in particular outgoing-queue of a metadata server, then these statements are kept in the outgoing-queue of respective metadata server. Dispatcher retries to send these statements in the next replication cycle.

Collector is responsible to collect the statements that are propagated from other metadata servers. When a statement has been received, collector stores it in the incoming queue. Collector consistently listens to receive statements.

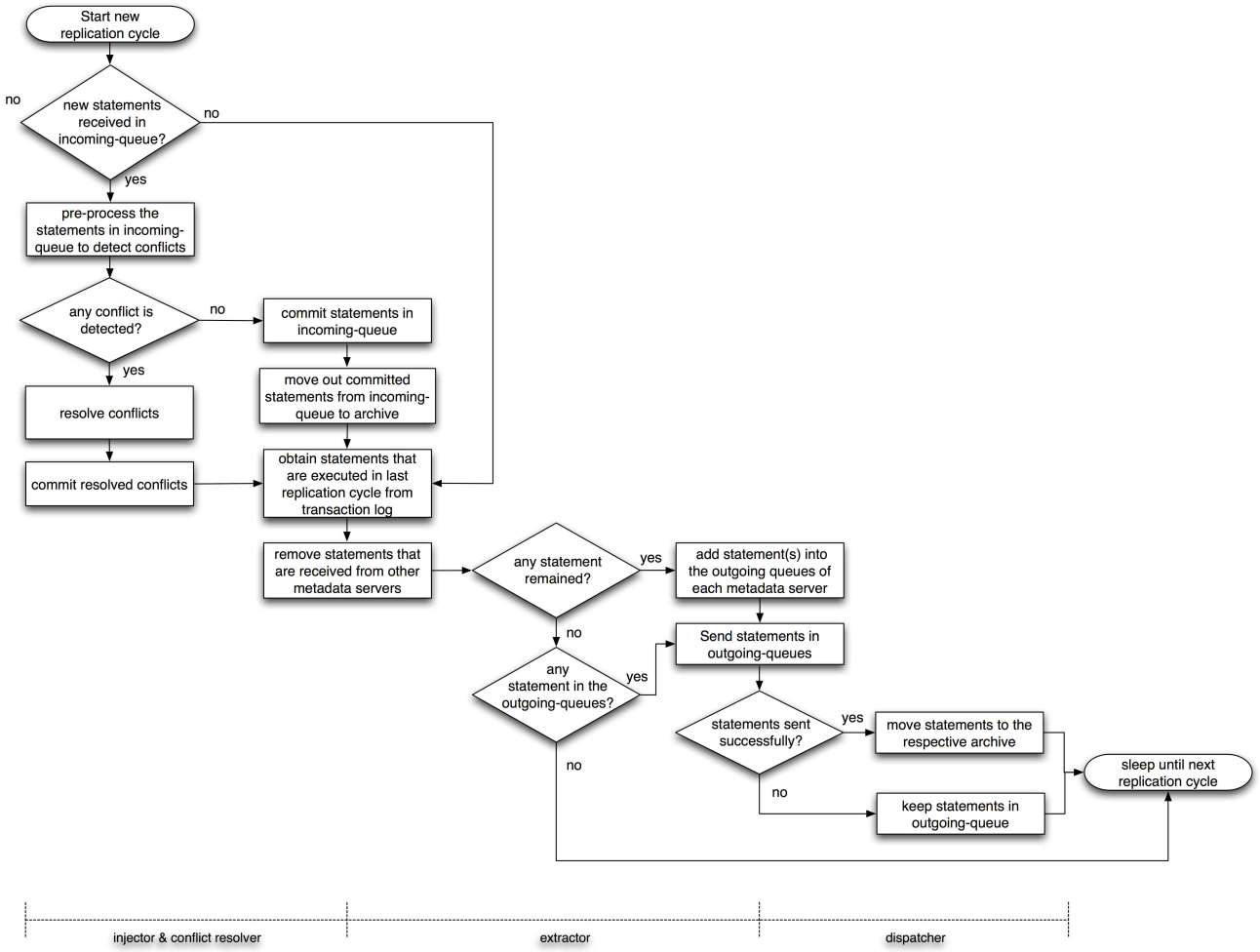


Figure 3.5: Flowchart of Replication Process

Injector acts as a database client. Basically, it asks database to process the statements stored in incoming-queue. These statements have to be executed in order to make metadata server synchronized with others. If a statement in incoming queue is successfully executed by metadata server, then it is moved out from the incoming-queue and stored in archive. If any error or conflict occurs, then conflict resolver is called and conflict resolver deals with the conflict as discussed in previous section. The flowchart of the asynchronous multi-master replication procedure is shown in Figure 3.5.

Defining the duration of waiting period in replication cycle is highly dependent on how long an application can survive or tolerate inconsistent metadata servers in the system. There are also other factors such as network, space allocated for queues, and frequency of updating metadata servers.

3.3 Feasibility of Asynchronous Multi-Master Metadata Server Replication in Distributed Data Storage System

Along with high-availability, another feature of the system that makes sense for end-users is performance. There are many factors that may affect the performance; however, we investigate on the effect of metadata replication setting that is used in distributed data storage system. We mention the benefits of asynchronous multi-master metadata replication from high-availability, scalability, load balancing and performance point of view. However, there is a challenge to be addressed in asynchronous multi-master metadata replication which is keeping metadata servers consistent. No matter your system is well designed and implemented there is always possibility of having inconsistencies in the system even though it may be only within a small fraction of time. The existence of inconsistent multi-master metadata servers may cause a catastrophic failures for some applications; however, some applications can survive with inconsistencies for a while and do not cause a catastrophic failure. In this thesis, we work on asynchronously replicated multi-master metadata servers in distributed data storage systems that can tolerate the existence of inconsistencies in metadata servers for a while. Although, it is said that synchronizing metadata servers once in a minute is sufficient [35]), we synchronize all metadata servers in every 30 seconds to reduce the duration of inconsistencies. We observed that all metadata servers become synchronized before any request comes for updated data object through other metadata servers.

In general, distributed data storage systems are dealing with processing huge volume of data sets that enforces to have stringent performance requirements. In such systems, the basic access pattern to the storage resources is ***"write once, read many, edit rare"***. This pattern basically tells that a data set is not likely to change so often; however, it is likely to be accessed many times after it has been written into the storage resource. What this access pattern implies is that the probability of having update and delete conflicts in asynchronously replicated multi-master metadata server layout is extremely low, because of editing a data set during its life time is rare. Our observation of data access pattern is demonstrated in Figure 3.6. Addition to the low probability of editing data set, it has even less probability of editing a data set from more than one metadata server within the same replication cycle which may cause an update conflict. As well as there are ways to resolve the conflicts as mentioned previously, having low probability of occurring conflicts

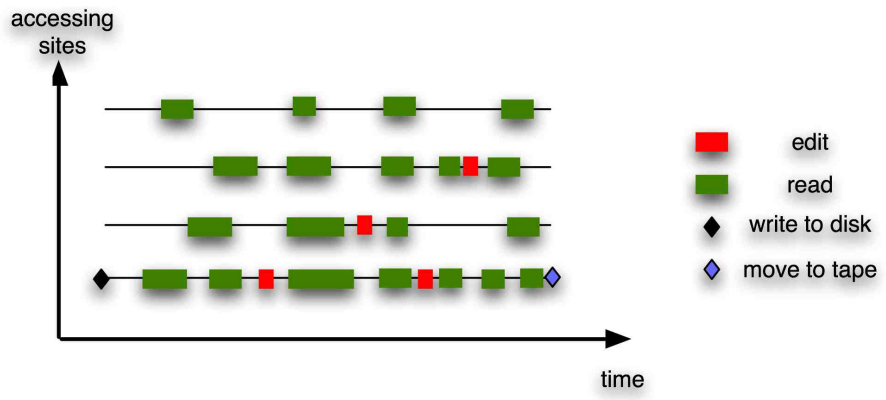


Figure 3.6: Access Pattern of Sample Data Set During Its Lifetime

makes us eager to defend asynchronously replicated multi-master metadata server setting for distributed data storage systems.

Having conflicts and resolving them are not only challenges of asynchronously replicated multi-master metadata server setting. Although the probability of editing a data set from more than one metadata server within the same replication cycle is low, adding a new file from more than one metadata server is more likely. This does not imply any conflict, but implies the existence of inconsistencies between metadata servers unless all metadata servers are synchronized at the end of next replication cycle. If replication cycle is too long that keeps metadata servers inconsistent for a long time, then inconveniences may occur for the users. For example, a metadata server creates a metadata information for a data set that is written into the storage resource, but does not propagate it until replication cycle is over. If a user asks to another metadata server for this data set before related metadata information has been propagated, then metadata server can not locate it. User has to wait until the end of replication cycle to be able to access this data set through different metadata server. Fortunately, we do not have too long replication cycles that may create such inconvenience. Moreover, looking for a data set which has been already written from other metadata servers takes time (e.g. user needs to be informed from his/her collaborating partners that data set has been written). Immediate access through other metadata servers is unexpected. It is observed that replication cycle is much smaller than the time passed until data set is asked through other metadata servers. This is also true for editing data sets, too. Immediate access of edited data set through other metadata servers is not expected. This observation is demonstrated in Figure 3.7.

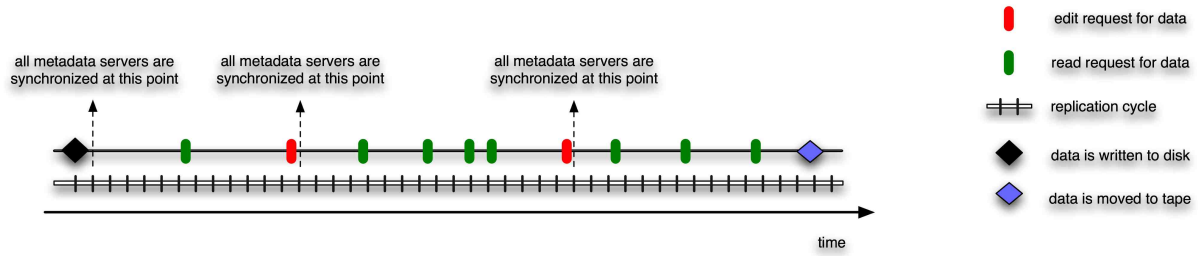


Figure 3.7: Relationship Between Access Pattern and Replication Cycle

These observations make us confident about using asynchronous replication in multi-master metadata server layout in distributed data storage system. Overall, we can say that by keeping replication cycle small enough, applications such as distributed data storage systems do not feel the inconvenience of inconsistencies and conflicts occurred in the system while synchronizing metadata servers. Instead, they enjoy the convenience of having high availability as well as increased performance of the system through multi-master metadata servers.

Chapter 4

Use Case: PetaShare

The NSF funded PetaShare project is a distributed data archival, analysis and visualization cyberinfrastructure effort for data-intensive collaborative research and targets the challenge of data storage, access and sharing by providing a transparent interface to complex distributed storage infrastructures [7]. The goal of PetaShare is to enable domain scientists to focus on their primary research problems, assured that the underlying infrastructure manages low-level data handling issues. In the design and implementation of PetaShare, we employ a novel approach to solve the distributed data sharing and storage management problems. Unlike existing approaches, PetaShare treats data storage resources and the tasks related to data access as *first class entities* just like computational resources and compute tasks, and not simply the side effect of computation [63].

4.1 PetaShare Overview

Current implementation and deployment of PetaShare involves five state universities and two health sciences centers in Louisiana. The participant institutions are Louisiana State University, Tulane University, University of New Orleans, University of Louisiana at Lafayette, Louisiana Tech University, Louisiana State University-Shreveport, Louisiana State University-Health Sciences Center in New Orleans.

PetaShare manages 300 Terabytes of disk storage distributed among these sites as well as 400 Terabytes of tape storage centrally located in downtown Baton Rouge. For connecting all of the participating sites together, PetaShare leverages 40 Gbps high bandwidth and low-latency optical network: LONI, the Louisiana Optical Network Initiative [15]. The internal network connection of LONI resources and the distribution of the PetaShare resources among the LONI sites are shown in Figure 4.1.

The back-end of PetaShare is based on enhanced version of iRODS to provide a global name space and efficient data access among geographically distributed storage sites. iRODS system is the descendant of SRB (Storage Resource Broker [39]) technology that aims to provide convenient interface for managing storage resources in data grids, digital libraries, persistent archives, and real-time data systems [65]. iRODS

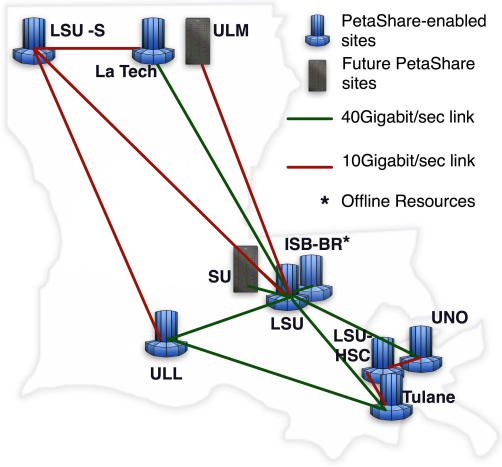


Figure 4.1: Louisiana Optical Network Initiative and PetaShare Sites

consists of two main components: data server called irodsServer to manage physical accesses to the storage resources, and metadata server called iCAT to provide global name space and keep all system related metadata information [51].

PetaShare exploits provided global name space by iCAT that enables transparent access to the distributed storage resources as well as data sets without locating storage and data set explicitly by stating resource host, connection port and path to directory.

As a management policy in PetaShare, accounts are assigned to research groups, instead of individuals. That means, all members of the particular research group can access to the common group account even if group members are affiliated with different institutions. This makes it easy to share data between users and helps to exploit resources available in a distributed environment. Users do not need to stage in/out data to their local machines, in fact, they can utilize high-speed optical network by attaching remote data to their local applications through provided infrastructure.

The transparent access to distributed storage infrastructure provided by PetaShare is illustrated in Figure 4.2. For this example, we assumed that the research group called *'GRP1'* has members at LSU, ULL, UNO and LSUS. Research group *GRP1* has home directory as *\tempZone\home\GRP1* in the global namespace. Group member at LSU uploads a data set called *file1* into PetaShare. The global directory information of *GRP1* is updated in metadata server such that *file1* is added under global directory

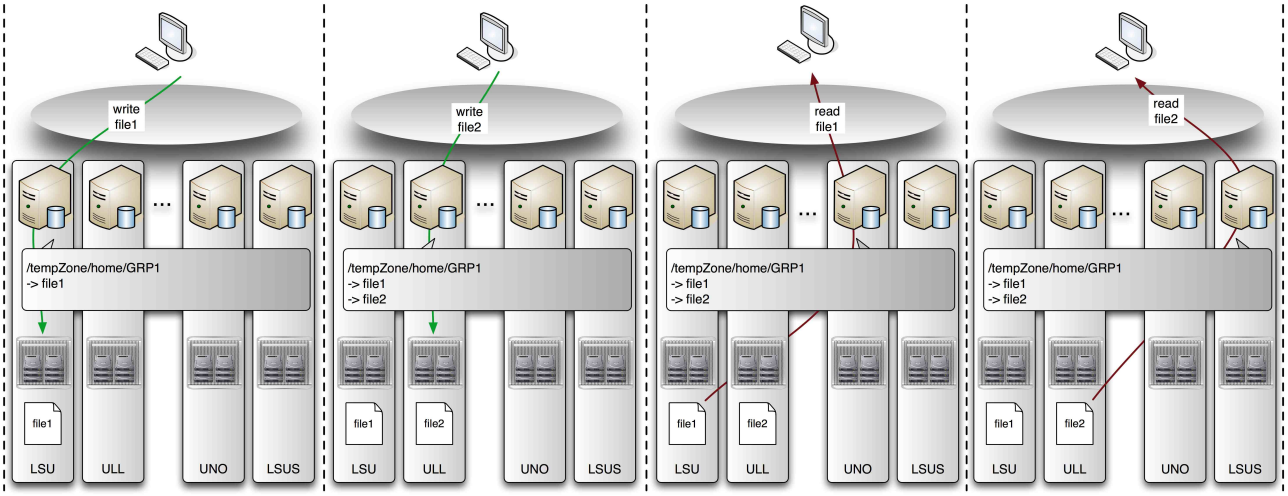


Figure 4.2: Global name space and transparent access

of *tempZone**home**GRP1*. This information is replicated to all other metadata servers in other resource hosts, so every metadata server knows that *file1* under *tempZone**home**GRP1* has been stored in resource of LSU. Similarly, group member at ULL uploads a data set called *file2* into PetaShare. The same procedure is followed and *file2* is added under global directory of *tempZone**home**GRP1*, and this information is replicated to all other metadata servers. Right now, global directory *tempZone**home**GRP1* has two files: *file1* and *file2*.

On the other hand, group member at UNO wants to download data set *file1*. After being asked for *file1*, data server at UNO cooperates with metadata server and figures out that *file1* under *tempZone**home**GRP1* is physically located at LSU. Then, it communicates with its counterpart at LSU, retrieves *file1* and returns it to the user. User could be able to receive desired data set through UNO; although, it is not physically located at there. Similarly, group member at LSUS can obtain data set *file2* through LSUS just by using global name space provided by PetaShare where all underlying path resolution and communication are transparent to the user.

4.2 PetaShare Clients

PetaShare provides different clients to fulfill the variety of user requirements. It is widely heterogeneous environment that users have variety of machines and run variety of applications on different kind of operating

systems. Also, some of the applications may require to access data sequentially; others may need to stage in/out data to/from compute resources for post-processing; and some of them may require to manipulate data sets with local applications from personal computers or laptops as convenient as manipulating data set in local disk. For the sake of supporting board range of applications and access patterns through highly diverse systems, PetaShare provides very light weight clients which enable easy, transparent, and scalable access at the user level.

Petashell is an interactive shell interface that virtualizes global name space provided by PetaShare and attaches data sets stored in PetaShare to the running applications in its shell. *Petafs* is a user space virtual file system that enables users to mount PetaShare resources to their machines. This gives ability to access PetaShare resources in a way of accessing local disk through existing file system. *Pcommands* are a set of unix-like commands that are specialized for accessing to and managing resources and data sets in PetaShare. In addition to these clients, a web portal is provided which enables accessing to the resources of PetaShare through a web browser, and provides a search utility.

These clients differ from each other in terms of capabilities, usage and portability. For example, petafs can be used in linux-based systems of which kernel has FUSE (File system in user space) support. Contrary, petashell and pcommands do not require additional kernel level module. The web portal can be accessed literally from any system, since the accesses are performed via a web browser.

All the clients use iRODS API to call corresponding remote system operations at the lowest level to handle I/O requests of an application. Petashell uses an existing open-source software, Parrot, to catch system I/O calls of an application and match them with respective remote system I/O calls. On the other hand, petafs pretends as a file system to handle system I/O calls and maps file system calls to respective remote system calls through a special interface called libFuse [4].

Petashell

Petashell is an interactive shell interface that allows users to access global name space of PetaShare and run local applications on their machines while input/output data resides on remote PetaShare resources. This eliminates the necessity of moving data to machine where application is running, or porting the application to where data resides. Petashell attaches application and data together while both are physically separated.

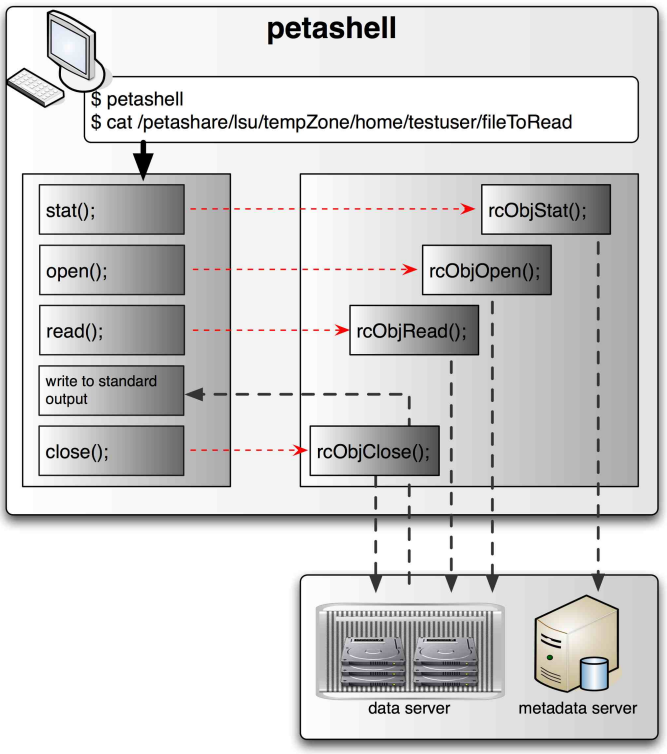


Figure 4.3: Mapping System I/O Calls to Remote I/O Calls via Petashell

Petashell is based on Parrot which is a tool for attaching running programs to remote I/O systems through the file system interface [59]. The main idea is to catch local system I/O calls and translate these calls into the corresponding remote I/O calls. This idea is shown in Figure 4.3.

Petashell can be used in linux-based systems.

Petafs

Petafs is a virtual file system that allows users to access PetaShare resources as a convenient as accessing local storage resources through file system. It enables to mount PetaShare resources into the users' machine. Although mounting requires root privileges, Petafs allows non-root users to mount PetaShare resources.

Petafs is based on FUSE which is a simple interface to export a virtual file system to the Linux kernel in user space [4]. Whenever system I/O calls are made towards mounted PetaShare resource, FUSE captures these I/O calls in kernel and forwards them to user space library called libFuse. This library maps local system I/O calls into remote I/O calls. The interaction between FUSE and petafs is shown in Figure 4.4.

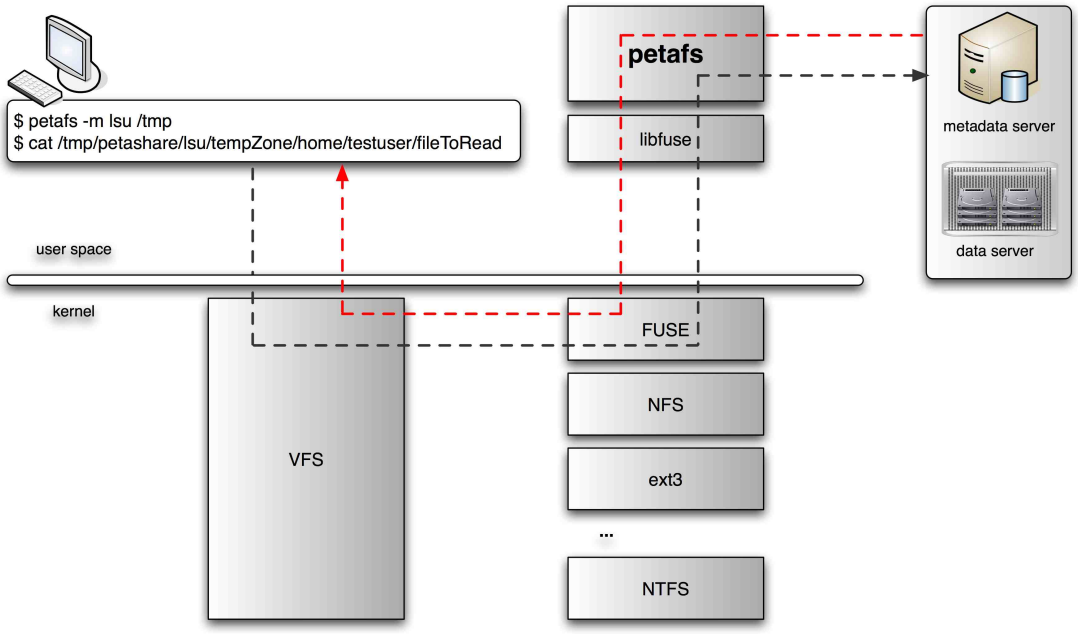


Figure 4.4: Interaction between PetaFs and FUSE kernel module

Petafs can be used in linux-based systems and MacOS if FUSE support is provided in the kernel.

Pcommands

Pcommands are a set of command line utilities that enable accessing and manipulating data sets stored in PetaShare resources. They evoke basic unix commands, such as pls, pmkdir, prm where unix counterparts are ls, mkdir, rmdir respectively. Pcommands provides advance utilities to manage data set and storage resources, such as replication of particular data set in more than one storage resource, physical movement of data between storage resources and inserting user-defined metadata for data sets. There are two basic commands which are used heavily: *pput* and *pget*. Users can store their data sets in PetaShare resources by using *pput* command, and retrieve data sets from PetaShare resources by using *pget* command.

Pcommands are a revised version of built-in command line utilities of iRODS for PetaShare and can be used in various types of operating systems; such as Linux, Solaris, MacOS, and AIX.

Web Portal

Another way to access data sets stored in PetaShare resources is using Web Portal. It provides seamless access to PetaShare resources through any modern web browser. Basic data operations such as storing and

retrieving data sets can be done through Web Portal, also. The portal provides additional features such as search functionality to explore directory hierarchy conveniently, editing user-defined metadata of data sets, and creating HTML links to data sets that can be shared with other users.

4.3 Asynchronously Replicated Multi-Master Metadata Servers in PetaShare

In the very first deployment of PetaShare, along with data servers in each storage site to manage storage resources there was a single central metadata server that provided global name space to assure location transparency of data sets. The very first deployment of PetaShare is illustrated in Figure 4.5. This deployment suffered from four system parameters: availability, scalability, load balancing and performance.

There has been service outages in cluster where central metadata server has been deployed that cause all PetaShare system to go offline. Although, all other data servers were running, there was no way to access them through global name space due to absence of metadata server. Generally, it is frustrating for those who stored their data sets in PetaShare resource that is in a cluster which is up and running; however, they can not access data sets due to a failure in remote cluster in which metadata server runs. It is hard to convince users in such cases. Basically, existence of single metadata server in the system leads to low availability and causes inconveniences which should be mitigated.

The increase in the scale of applications and research projects let data volume as well as number of collaborating parties to grow. Thus, it is very likely to add more storage resources to the existing system for the sake of alleviating the storage demand. Eventually, adding more resources into the system means having increased number of requests (i.e. both number of participating parties and volume of data sets are increasing) overwhelm the central metadata server. Overwhelmed metadata server limits the system to scale due to increased processing and response time caused by the volume of the requests.

Another system parameter is performance which is extremely essential for the satisfaction of users. Central metadata server suffered from low performance in PetaShare due to increased processing latency as well as network latency. Processing latency increases, as mentioned above, because single metadata server has to process all incoming requests, so it may take time to process all requests. On the other hand, clients had to connect metadata server which was remote server to be able to access storage resources and data sets.

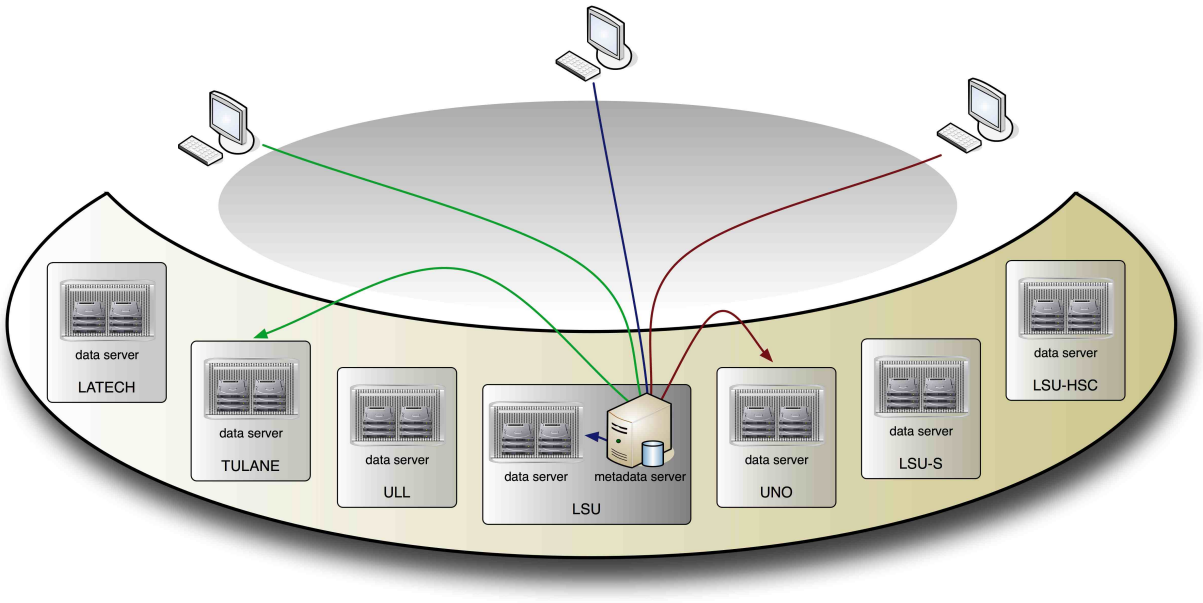


Figure 4.5: PetaShare Deployment with Central Metadata Server

Connecting to remote metadata server introduces network latency which degrades the performance.

Due to drawbacks of central metadata server mentioned above, we came up with the solution of replicating metadata server. We do not consider master-slave replication because it also suffers from existence of single master metadata server which becomes overwhelmed. For this reason, we looked multi-master replication solutions to achieve better load-balancing and performance. First, we deployed synchronously replicated multi-master metadata servers in PetaShare. Although it fulfills the requirements of high availability and load balancing, we observed increased processing time, as well as network traffic that lead us getting worse performance especially for write operations. An asynchronous replication was an option to increase the write performance of the multi-master metadata servers; however, we were hesitating to deploy asynchronous multi-master replication layout due to inconsistencies occur in metadata servers. However, after reviewing the system characteristics and access pattern of data sets stored in resources, we observed that the inconsistencies occur in asynchronous multi-master metadata server layout do not cause any catastrophic failure in the system, and metadata servers become consistent again before inconsistencies cause an inconvenience for the users. Eventually, we decided to deploy asynchronous multi-master metadata server

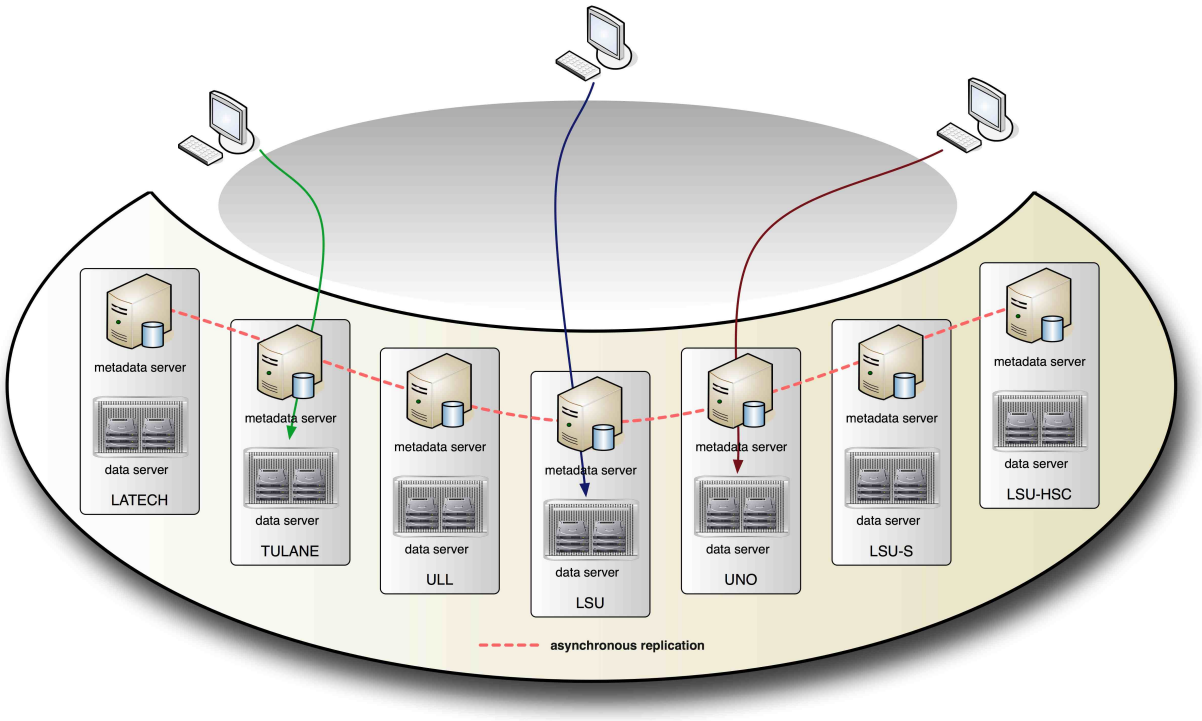


Figure 4.6: PetaShare Deployment with Replicated Metadata Servers

replication layout in PetaShare to achieve increased availability, scalability, load balancing and performance at the same time.

In the current deployment of PetaShare, we employ a metadata server and data server in each storage site. The motivation behind doing this is to maximize availability of metadata servers, and minimize the cost of communication with metadata server. A data server in storage site can cooperate with local metadata server which is faster than cooperating with remote metadata server. Furthermore, for clients, it is more likely to store their data sets in PetaShare resources in their site. Thus, if they find a metadata server nearby data server in their sites, this would reduce the communication overhead and give better performance both for read and write operations. Also, having metadata servers in each PetaShare site balances the load, since all incoming requests from respective site can be processed within that site. The current deployment of PetaShare is illustrated in Figure 4.6.

All metadata servers can process both read and write operations. Read operations do not change meta-

data information of any object in the system. However, write operations update metadata information. If a write request is made in a metadata server, then this request has to be replicated in all other metadata servers in order to synchronize all metadata servers. In asynchronous multi-master metadata server layout, these requests are deferred and propagated to other metadata servers asynchronously. To perform asynchronous replication and synchronization in PetaShare, we deploy our asynchronous replication tool called MASREP in each storage site along with metadata server. MASREP operates in the background and is transparent to the metadata server. It propagates the deferred requests to other metadata servers in order to synchronize them.

Chapter 5

Experimental Results and Evaluation

5.1 Deployment Details

We deployed three metadata server layouts (i.e. central metadata server, synchronously replicated multi-master metadata servers, and asynchronously replicated multi-master metadata servers) and evaluated their performance for read/write operations.

The database that we use in our metadata servers is postgresql which is broadly used in advanced database applications in open source community [10]. In central metadata server layout, we employed single instance of postgresql database that contains all metadata information. All metadata operations have conducted through central metadata server which was located in LSU cluster.

There is a widely used synchronous replication tool for postgresql database called Pgpool [9]. Basically, it works as a middleware between database and clients. This means that all metadata requests have to go through Pgpool before they actually arrive metadata server. It can be thought as a proxy which talks with database on behalf of the clients.

We used Pgpool for replicating multi-master metadata servers synchronously in PetaShare. We deployed pgpool peers along with metadata servers in all six PetaShare sites [1]. When a request made by client, pgpool captures it and initiates the synchronous replication process. It connects to pgpool peers in all replicating metadata servers, and starts two-phase commit protocol to synchronize all metadata servers. The pgpool counterparts inall metadata servers check corresponding metadata server if the request is OK for them or not and inform the initiating pgpool. If all pgpool peers send OK for the request, then initiating pgpool sends a message to all pgpool peers to execute the request and synchronize the metadata servers. If any one of the pgpool peers complains about the request, then it is discarded in all metadata servers. The interactions between pgpool peers and metadata servers is illustrated in Figure 5.1.

For asynchronous replication, we deployed our asynchronous multi-master metadata server replication tool called MASREP along with metadata servers in all PetaShare sites. As opposed to pgpool, MASREP

[1] At the time of conducting tests, there were six PetaShare sites online: LSU, TULANE, UNO, ULL, LSUS, LSUHSC

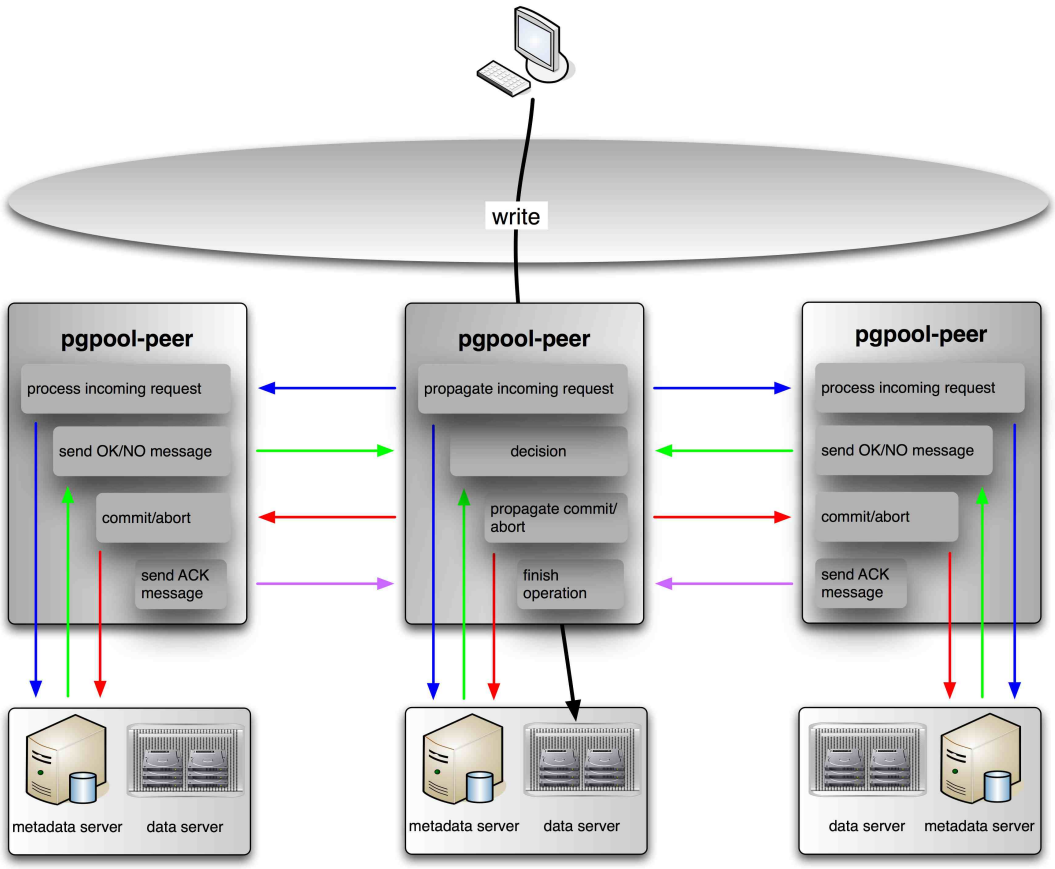


Figure 5.1: Replication of Metadata Servers with Pgpool

runs in the background, not in front of the metadata server, and do not interfere clients. This means that clients communicate with metadata server directly, which makes it faster. Since MASREP runs in the background and invisible by clients, all replication and synchronization processes are transparent to both metadata server and clients. When MASREP receives requests coming from other metadata servers, it acts as a client to process the requests in order to synchronize the metadata server. The interactions between MASREP instances and metadata servers are illustrated in Figure 5.2.

5.2 Test Cases

In this section we show results of read/write operations made in PetaShare for three replication scenarios: no replication (i.e. there is only one metadata server), synchronous replication and asynchronous replication

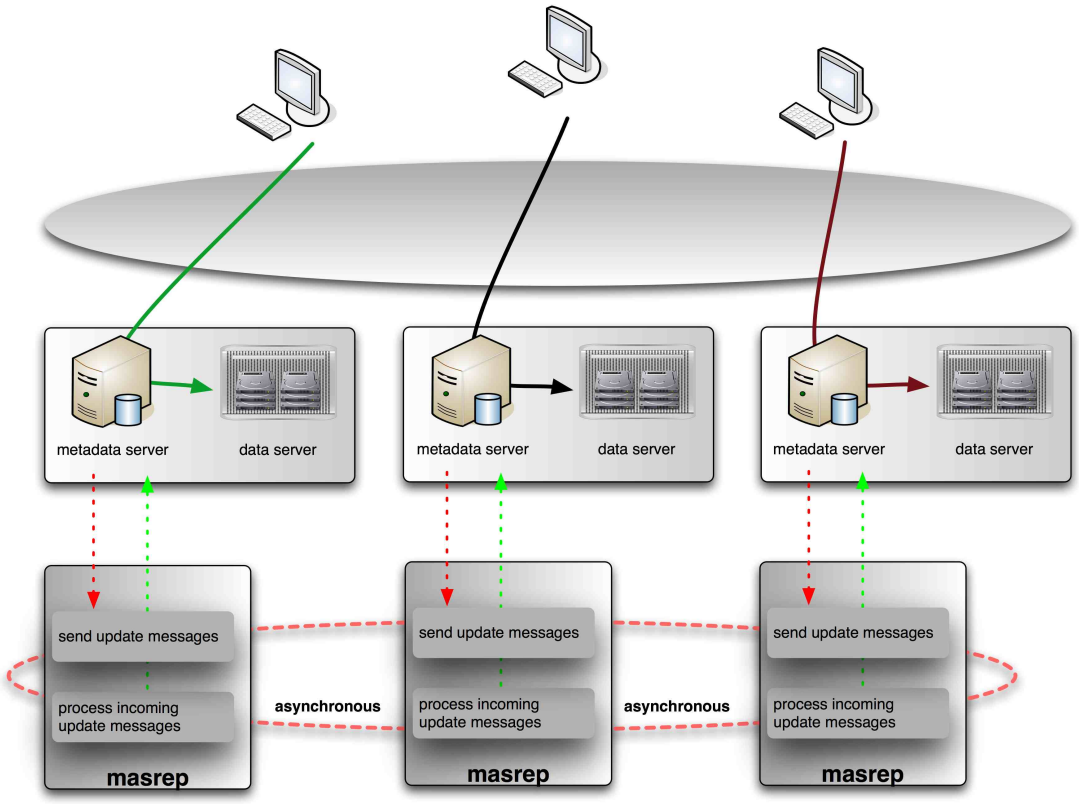


Figure 5.2: Replication of Metadata Servers Asynchronously

of the multi-master metadata servers.

During our experiments we have assumed that a client is in one of the PetaShare site and trying to do the one of the following operations:

- client is writing to remote PetaShare resource
- client is writing to local PetaShare resource
- client is reading from remote PetaShare resource
- client is reading from local PetaShare resource

We repeated the experiment for each scenarios given above in all PetaShare sites. Also, we performed read/write operations from/to all remote PetaShare resources in a given site. The values shown in graphs are the average of results obtained from all PetaShare resources.

We used two data sets in the experiments: a data set contains 1000 files, and a data set contains a single file. We had 10KB, 100KB and 1000KB files for the first data set, and 10MB, 100MB and 1000MB files for the second data set.

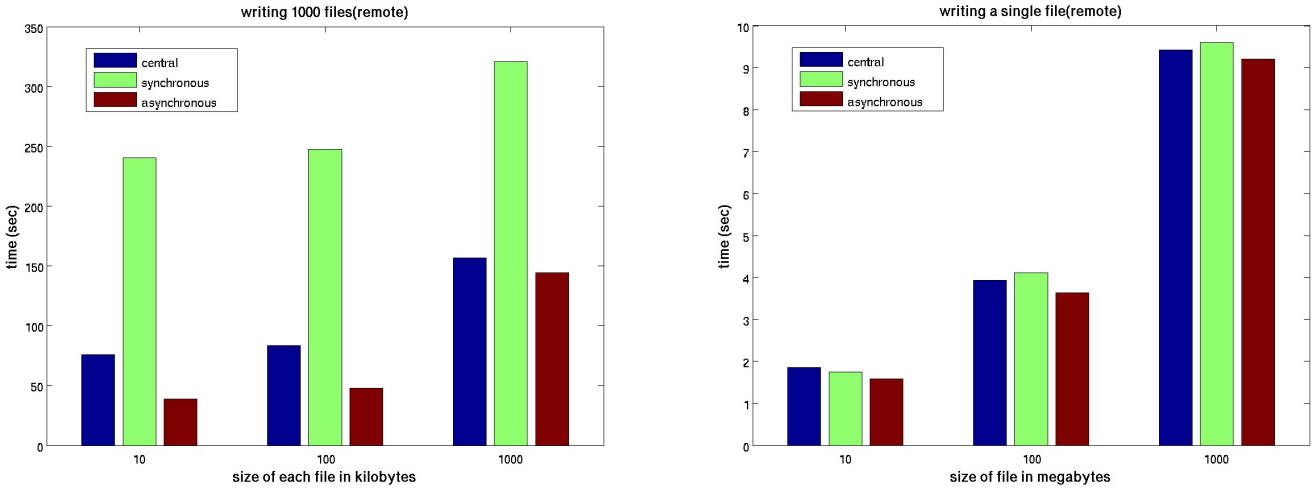
Comparing these two data sets enabled us to understand the effect of the number of files on overall operation performance under all replication scenarios. We observed that the performance is hurt by the number of files, even though the file size is small. This validates one of our assumptions: metadata operations become performance bottleneck in the system, so it should be done as fast as possible. Asynchronously replicated metadata servers gives the possibility of performing metadata operations faster compared to the others; thus gives better performance.

5.3 Evaluation

In general, test results show us synchronous replication suffers from the overhead of two-phase commit protocol which increases the time spent especially for write operations. Read operations are less time consuming since they do not change metadata information that avoids the necessity of synchronization of metadata servers. For this reason, synchronously replicated metadata servers and central metadata server give similar results for read operations. On the other hand, asynchronously replicated metadata servers outperform both synchronously replicated metadata servers and central metadata server for read and write operations.

In the first scenario, we look at the performance of writing to remote storage resources. Our client has two data sets. First data set contains 1000 files, and other data set contains a single file; however, total size of both data sets are equal. With the given data sets, our client writes data sets into the remote PetaShare resource.

In Figure 5.3(a) shows the average time of writing data set which contains 1000 files into the remote resource in three replication layouts. Similarly, the Figure 5.3(b) shows the average time of writing a data set that contains a single file. As it can be seen from these two figures, the time spent on metadata operation has a big impact on the performance. Although the same size of data sets are written into the remote resource, the data set which contains 1000 files takes longer time to write. This is because writing 1000 files need creating 1000 data objects and related metadata information on metadata server, as opposed to one data



(a) data set contains 1000 files

(b) data set contains a single file

Figure 5.3: Average Duration of Writing to Remote PetaShare Resources

object and corresponding metadata information in the case of single file.

Synchronous replication takes longer time for the first data set. This is because it needs to initiate a synchronization process and propagate a write request to all metadata servers and wait for their replies for each file. The overhead of synchronous replication is more than the cost of accessing to central metadata server as it is seen in Figure 5.3(a). On the other hand, asynchronously replicated metadata servers layout takes less time to finish than synchronously replicated metadata servers layout. This is because it does not have a synchronization overhead. It performs better than central metadata server because client is not required to connect remote metadata server; instead it connects to local metadata server which reduces accessing time.

In Figure 5.3(b), all replication scenarios have similar similar performance. This is because the overall time is dominated by sending file to remote resource rather than updating metadata server. There is a single write request that have to be processed by metadata server which takes less time compared to sending a file itself.

In the second scenario, we look at the performance of writing to local storage resource. Our client has the same data sets given above. With the given data sets, our client writes data sets into the local PetaShare resource.

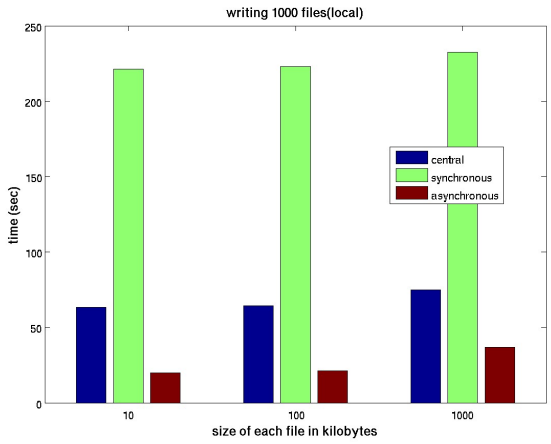


Figure 5.4: Average Duration of Writing to Local PetaShare Resource

Table 5.1: Time Table of Two Data Sets for Writing to Local PetaShare Resource

	No Replication	Synchronous Replication	Asynchronous Replication
10K * 1000	63.51	221.33	19.92
10M(single)	0.25	0.46	0.15
100K * 1000	64.42	223.06	21.38
100M(single)	0.58	1.02	0.47
1M * 1000	75.02	232.48	36.75
1G(single)	8.37	9.18	5.16

Similar to the Figure 5.3(a), Figure 5.4 makes it clear that write requests result low performance in synchronously replicated metadata servers. It is slightly better than the performance of writing into the remote storage resources, because data set is written into the local storage resource. This reduces the time spent to write the data set. In central metadata server layout, network is used only for metadata operations. Conversely, both data operation (i.e. writing files into the resource) and metadata operations are performed locally in asynchronously replicated metadata servers layout which minimizes the overall time of the operation. The average duration of writing two data sets into local PetaShare resources is given in Table 5.1.

In the third scenario, we look at the performance of reading from remote storage resource. Our client has the same data sets mentioned above. With the given data sets, our client reads data sets from the remote PetaShare resource.

In general, reading is less expensive request compared to writing from the metadata operation point of view. It does not require any metadata update, but just retrieving metadata information of the data object. For this reason, the overhead of two-phase commit protocol seen for write operations does not appear for read operations in synchronously replicated metadata servers in this case. Figure 5.5 shows that central metadata server layout takes more time than the others while reading the first data set. This is because metadata information of each file is retrieved from central metadata server which is located in remote site. Conversely, metadata information is retrieved from local metadata server in both synchronously and asynchronously

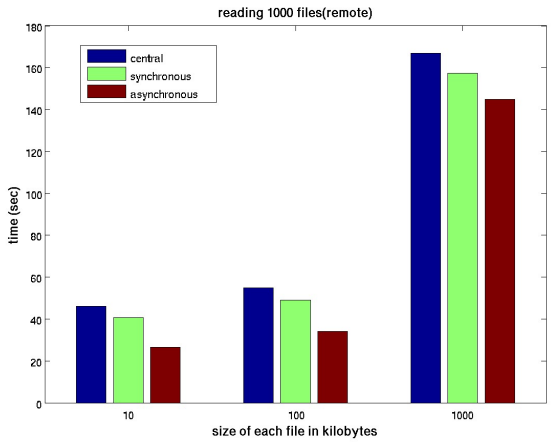


Figure 5.5: Average Duration of Reading from Remote PetaShare Resources

Table 5.2: Time Table of Two Data Sets for Reading from Remote PetaShare Resources

	No Replication	Synchronous Replication	Asynchronous Replication
10K * 1000	46	40.75	26.51
10M(single)	1.58	1.74	1.47
100K * 1000	55	48.92	34.13
100M(single)	3.44	3.54	3.29
1M * 1000	166.98	157.43	144.98
1G(single)	14.47	11.55	9.57

replicated metadata servers. This reduces the time of retrieving metadata information and gives better performance. An interesting point in Figure 5.5 is that although metadata information is stored in local metadata server in both synchronously replicated metadata servers and asynchronously replicated metadata servers, it takes more time in synchronously replicated metadata servers than asynchronously replicated metadata servers. This is because read requests are not received by metadata server directly, but they go through pgpool in synchronous replication. Pgpool has load balancing functionality. It can distribute the read requests among metadata servers if current server is overwhelmed. To do that gpool requires to identify the current load on metadata servers and distributes the read requests across metadata servers if necessary. This extra step takes time. In spite of the good will of pgpool, this functionality introduces latency which increases overall time of read operation. For this reason, synchronously replicated metadata servers gives worse performance than asynchronously replicated metadata servers. The average duration of reading two data sets from remote PetaShare resources is given in Table 5.2.

In the last scenario, we look at the performance of reading from local storage resource. Our client has the same data sets given above. With the given data sets, our client reads data sets from the local PetaShare resource.

Figure 5.6 shows that the overhead of synchronous replication is comparable to the overhead of retrieving metadata information from central metadata server while reading data set from local storage resource. On the other hand, asynchronously replicated metadata servers outperform central metadata server since

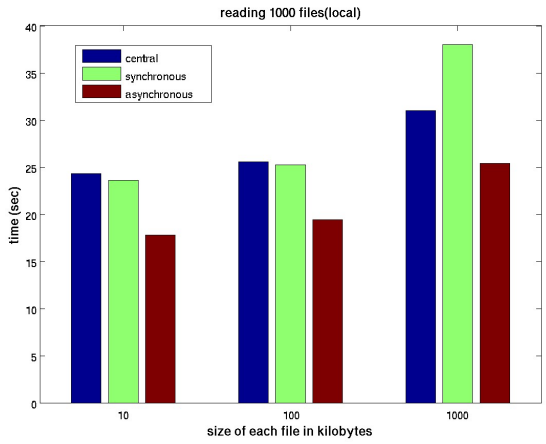


Figure 5.6: Average Duration of Reading from Local PetaShare Resource

Table 5.3: Time Table of Two Data Sets for Reading from Local PetaShare Resource

	No Replication	Synchronous Replication	Asynchronous Replication
10K * 1000	24.33	23.63	17.8
10M(single)	0.2	0.36	0.14
100K * 1000	25.57	25.23	19.43
100M(single)	0.42	1.4	0.43
1M * 1000	31.94	38.08	25.43
1G(single)	7.34	14.06	3.25

they retrieve metadata information locally; thus, they eliminate the overhead of retrieving metadata information from remote metadata server. Similarly, they outperform synchronously replicated metadata servers since they directly process the read requests. Synchronously replicated metadata servers spend more time to retrieve metadata information through pgpool compared to central metadata server and asynchronously replicated metadata servers. The average duration of reading two data sets from local PetaShare resource is given in Table 5.3.

The results shown above allow us to conclude that asynchronously replicated metadata servers layout outperforms central metadata server layout and synchronously replicated metadata servers layout for both write and read operations. The synchronously replicated metadata servers suffer from high synchronization overhead for write operations, and metadata information retrieval delay for read operations due to having middleware (i.e. pgpool) in front of the metadata servers. On the other hand, central metadata server is a single point of failure in the system and increases metadata update and retrieval time because of being a remote server.

Chapter 6
Related Work

As far as our knowledge, there is not much effort made on asynchronous replication of metadata across multi-master servers in distributed data storage systems, compared to the data replication in distributed data storage systems. Especially, data replication in distributed data storage systems and its implications on availability, consistency and performance have been widely studied [17, 33, 48, 53, 58, 64].

On the other hand, database replication has been extensively studies over the years. Pacitti et al. [46] proposed a preventive lazy multi-master database replication solution in a cluster system for application service provider management. The motivation behind replication is to provide availability and load balancing to increase performance. Our work differs from theirs in a way that they embedded replication mechanisms into the actual database as a layer. Any incoming request has to go through their replication layer before it is committed. The request may be forwarded to the other server in order to load the balance and then they schedule the propagation of incoming request to other replicating servers within this layer. This layer introduces a processing overhead which reduces performance. In our case, all replication related operations are hidden and done behind the server which eliminates such processing overhead.

An algorithm has been proposed to minimize the duration of inconsistencies between a master server and slaves in asynchronous replication [45]. They added four components to the regular database in order to implement their replication algorithm. Original database basically consists of transaction manager and query processor. The first component called replication module has three sub components: log monitor, propagator, receiver. Log monitor is used to extract the changes made in local database by reading the content of history log. Whenever it catches an update request, it places it in input log which is used by propagator. Propagator tries to send update requests stored in input log. Receiver captures the update requests coming from master server and store it in pending queue. Second component is called refresher. It implements refreshment strategy and processes the pending queue and adds update transactions into the running queue according to the refreshing algorithm. Third component is called deliverer which manages submission of update transactions stored in running queue to the local transaction manager. Local transaction manager executes the requests. Last component is network interface which is used to propagate and

receive the update messages. Although, similarities exist between their implementation and our tool, there are two main differences. First, their implementation was built in database. We do not change the original database. Second, they did not consider the multi-master replication. They focused on asynchronous single-master primary copy replication.

MADIS has been developed especially for enterprise applications, which supports different replication protocols which can be chosen dynamically according to the need of an application [16]. MADIS is a middleware which has two layers. First layer is implemented on database driver (JDBC in MADIS case) which handles incoming requests and runs replication related functions. Second layer is an extension of original database scheme. This extended scheme is called report table and used for transparent replication management. MADRIS upper layer uses report table while employing replication functions. The report table is migrated to the other servers to replicate changes. MADRIS allows using different replication protocols adaptively on the fly, such as asynchronous replication, depending on the application tolerance for inconsistency; however,the performance tends to be low since it is a middleware which introduce processing latency.

Moura et al. [40] proposed a metadata replication architecture in e-learning applications. They have autonomous metadata repositories in each site. They only need to update respective metadata only when a site wants work on the learning objects of others. For this reason, they do not enforce any replication cycle, but metadata repositories are updated only when replication process is started by a specific site. Their replication architecture has three modules: replication module, query builder and access module. Repletion module is responsible for controlling replicating sites and replication processes. The query builder generates queries to extract metadata from other sites and generates queries that allow inserting metadata into the other sites. The access module provides access to the metadata repositories in other sites. It accesses the replicating repositories that replication module points to and executes the queries that has been generated by query builder in order to replicate metadata of interest. Their application domain is totally different from ours, but proposed architecture has similarities to our replication tool.

In postgresql database community, a tool called bucardo [1] provides asynchronous multi-master replication; however, it maintains a central database to coordinate synchronization which is also single point of failure. On the other hand, there are many synchronous multi-master replication tools exist for post-

gresql, such as pgpool [9], postgres-r [66], pgcluster [8] and cybercluster [3]. Along with implementation of replication tools, another field of research is propagating update requests. Different update propagation strategies have been discussed to minimize the duration of inconsistencies [47].

There are variety of applications where asynchronous replication is used especially in federated layout. One of the application area is mobile wireless network where there is a limited network resources, and the consistency of servers is not strict as it is in other applications. Asynchronous replication is also used in applications which have to run on intermittent networks and bandwidth limited environments. TierStore is an example for this which provides a file system interface that can be accessed and updated from multiple nodes in the network [28]. Any modification made in the node against shared file system is asynchronously distributed among other nodes as update messages. Inconsistency exists in the system since update messages are sent asynchronously and some of the nodes may be unreachable due to intermittent network. They address this issue to some extend by proposing automatic conflict resolution method as we do; however, the main focus is operability of the overall system. To increase performance and efficiency of low-bandwidth network, TierStore partitions the name space into disjoint sub-trees called publications, and applies a subscription scheme for nodes such that they can subscribe to the publications of their interests only. In this way, total amount of update messages sent through network is reduced significantly which leads the system to achieve optimal resource utilization. In contrast, we employ fully replicated global name space.

Chapter 7
Conclusion

In this thesis, we propose asynchronous replication of metadata across multi-master servers in distributed data storage systems to maximize the following system metrics: availability, scalability and performance. Our study shows that asynchronously replicated metadata servers layout performs better compared to centralized metadata server and synchronously replicated metadata servers layouts.

Main functionality of metadata server in distributed data storage systems is to employ global name space to provide location transparency and keep all system related information. In general, metadata server is a centralized server that makes system vulnerable to failures and leads to low availability due to being single point of failure in the system. Moreover, central metadata server prohibits the system to scale and becomes overwhelmed with the increased number of operations which is correlated with the number of data storage resources in the system. Addition to that, central metadata server is located in remote site which introduces network latency to retrieve metadata information. This has an effect on both read and write operations and results poor performance for the increased number of data sets.

To alleviate the availability problem, metadata server must be replicated. Replicating metadata servers increases availability; however, the capabilities of replicated metadata servers have impact on scalability, load balancing as well as performance. Deploying master-slave replication for metadata servers may increase availability; however, suffers from scalability and performance due to having a single master metadata server which is the only one who can update metadata information. All operations that require metadata update have to be processed by metadata server, similar to the central metadata server. Slaves may only reduce the load of master metadata server for operations that require just metadata information retrieval, such as reading a data set from storage resources. However, write operations that require metadata update have stringent performance requirements and become bottleneck for most of the applications.

To achieve maximum availability and increased read/write performance, a replicated master metadata server is deployed along with a data server in all storage sites. Since there is a master metadata server in each storage site, all read and write operations can be processed locally. This seems to be a solution to increase performance; however, replicated metadata servers have to be synchronized somehow. If metadata servers

updated synchronously, then system suffers from increased processing time, due to synchronization overhead introduced while propagating incoming requests to all metadata servers and waiting them to process these requests. For this reason, synchronously replicated multi-master metadata servers layout degrades the performance.

On the other hand, asynchronously replicated metadata servers layout eliminates the synchronization overhead by delaying metadata update requests and send them to the replicating metadata servers in the background. The advantage of asynchronous replication is that it allows each metadata server to process incoming requests on its own without need to propagate these requests to other metadata servers immediately which reduces processing time. However, delaying metadata update requests and updating metadata servers asynchronously have implications on consistency of metadata servers. Applications may suffer from inconsistencies occurred in metadata servers before metadata servers update themselves asynchronously. We discuss the implications of inconsistencies occurred in metadata servers until they are synchronized for distributed data storage systems. We demonstrate that system can tolerate the duration in which metadata servers are inconsistent and does not cause any catastrophic failure. Further, we pay attention to conflict detection and resolution to minimize the inconveniences might be originated due to asynchronous multi-master metadata server replication. We discuss the characteristics of distributed data storage systems, as well as access pattern of data sets stored in distributed data storage systems. Basically, we observe that having *"write once; read many; edit rare"* pattern of data set accesses minimizes the possibility of occurring conflicts in asynchronously replicated multi-master metadata servers.

7.1 Contribution

In general, our work demonstrates the feasibility of asynchronous metadata replication across multi-master servers in distributed data storage system. Although asynchronous replication of metadata across multi-master servers is vulnerable to have inconsistencies between multi-master servers, we show that the duration of inconsistencies are extremely short and almost invisible to the users.

In particular, our work enables distributed data storage systems based on iRODS to have high availability of metadata and improved performance due to accessing metadata locally and distributing metadata operation load on storage sites.

We designed and implemented our own asynchronous multi-master replication tool called MASREP. We deployed multi-master metadata servers in the state-wide distributed data archival system called PetaShare and used MASREP in the background to update metadata servers asynchronously. This enhancement on PetaShare enables users to experience with faster data operations and to have less down times due to failures in the systems.

Bibliography

[1] Asynchronous postgresql replication system. http://bucardo.org/wiki/Bucardo.

[2] Cluster file system, inc. lustre: A scalable, high performance file system. http://www.Lustre.org/docs.html.

[3] Cybercluster. http://www.cybertec.at/english/pr_cybercluster_e.html.

[4] Fuse: Filesystem in userspace. http://fuse.sourceforge.net.

[5] Network file system(nfsv4). http://tools.ietf.org/html/rfc3530.

[6] Oracle8i replication release 2 (8.1.6): Conflict resolution concepts & architecture. http://www.mcs.csuhayward.edu/support/oracle/doc/8.1.7/server.817/a76959/conflict.htm.

[7] Petashare: A distributed data archival, analysis and visualization cyberinfrastructure for data-intensive collaborative research. http://www.petashare.org.

[8] Pgcluster: the synchronous replication system of the multi-master composition for postgresql. http://pgfoundry.org/projects/pgcluster/.

[9] Pgpoll-II. http://pgpool.projects.postgresql.org/.

[10] Postgresql: The world's most advanced open source database. http://www.postgresql.org/.

[11] Red hat global file system: The open source cluster file system for enterprise deployments. http://www.redhat.com/gfs/.

[12] Top 500 supercomputer sites. http://www.top500.org/stats/list/33/archtype.

[13] G. Aldering and S. N. A. P. Collaboration. Overview of the supernova/acceleration probe (snap), Sep 2002. http://arxiv.org/abs/astro-ph/0209550.

[14] G. Allen, T. Goodale, J. Massó, and E. Seidel. The cactus computational toolkit and using distributed computing to collide neutron stars. In *Proceedings of Eighth IEEE International Symposium on High Performance Distributed Computing, HPDC-8, Redondo Beach, 1999*. IEEE Computer Society, 1999.

[15] Gabrielle Allen, Charles MacMahon, Ed Seidel, and Tom Tierney. Loni: Louisiana optical network initiative. Technical report, Louisiana State University, Dec 2003. http://www.cct.lsu.edu/ãllen/Reports/LONI_ConceptPaper.pdf.

[16] José Enrique Armendáriz-Iñigo, Hendrik Decker, Francesc D. Muñoz-Escoí, Luis Irún-Briz, and Rubén de Juan-Marín. A middleware architecture for supporting adaptable replication of enterprise application data. In *Trends in Enterprise Application Architecture, VLDB Workshop*, pages 29–43, 2005.

[17] Mary Baker and John Ousterhout. Availability in the sprite distributed file system. In *EW 4: Proceedings of the 4th workshop on ACM SIGOPS European workshop*, pages 1–4, New York, NY, USA, 1990. ACM.

[18] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The sdsc storage resource broker. In *In Proceedings of CASCON*, 1998.

[19] Alex Berson and Larry Dubov. *Master Data Management and Customer Data Integration for a Global Enterprise*, chapter 6. McGraw-Hill, Inc., New York, NY, USA, 2007.

[20] Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Lan Xue. Efficient metadata management in large distributed storage systems. *IEEE Symposium on Mass Storage Systems*, 0:290, 2003.

[21] F Carena, W Carena, S Chapeland, R Divia, U Fuchs, I Makhlyueva, K Schossmaier, C Soos, and P V Vyvre. The alice daq online transient data storage system. *Journal of Physics: Conference Series*, 119(2):022016 (7pp), 2008.

[22] Qin Chen, Haihong Zhao, Kelin Hu, and Scott L. Douglass. Prediction of wind waves in a shallow estuary. *Journal of Waterway, Port, Coastal, and Ocean Engineering*, 131(4):137–148, 2005.

[23] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven 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, 1999.

[24] Ann L. Chervenak, Robert Schuler, Matei Ripeanu, Muhammad Ali Amer, Shishir Bharathi, Ian Foster, Adriana Iamnitchi, and Carl Kesselman. The globus replica location service: Design and experience. *IEEE Transactions on Parallel and Distributed Systems*, 20(9):1260–1272, 2009.

[25] Mabrouk Chouk. Master-slave replication, failover and distributed recovery in postgresql database. Master's thesis, School of Computer Science, McGill University, 2003.

[26] Parvathi Chundi, Daniel J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*, pages 469–476, Washington, DC, USA, 1996. IEEE Computer Society.

[27] Peter F. Corbett and Dror G. Feitelson. The vesta parallel file system. *ACM Trans. Comput. Syst.*, 14(3):225–264, 1996.

[28] Michael Demmer, Bowei Du, and Eric Brewer. Tierstore: a distributed filesystem for challenged networks in developing regions. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[29] Yinjin Fu, Nong Xiao, and Enqiang Zhou. A novel dynamic metadata management scheme for large distributed storage systems. *High Performance Computing and Communications, 10th IEEE International Conference on*, 0:987–992, 2008.

[30] Kelly Gaither. Visualization's role in analyzing computational fluid dynamics data. *IEEE Computer Graphics and Applications*, 24(3):13–15, 2004.

[31] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM.

[32] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[33] Lukas Hejtmanek and Ludeck Matyska. Nonblocking distributed replication of versioned files. In *In Journal of Software*, volume 2, pages 16–23, 2007.

[34] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[35] Brian Keating. Challenges involved in multimaster replication, 2001. http://www.dbspecialists.com/files/presentations/mm_replication.html.

[36] Carl Kesselman and Ian Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.

[37] W. B. Ligon and R. B. Ross. Implementation and performance of a parallel file system for high performance distributed applications. In *HPDC '96: Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, page 471, Washington, DC, USA, 1996. IEEE Computer Society.

[38] Soha Maad, Brian Coghlan, Geoff Quigley, John Ryan, Eamonn Kenny, and David O'Callaghan. Towards a complete grid filesystem functionality. *Future Generation Computer Systems*, 23(1):123–131, 2007.

[39] R.W. Moore, M. Wan, and A. Rajasekar. Storage resource broker; generic software infrastructure for managing globally distributed data. In *International Symposium on Mass Storage Systems and Technology*, volume 0, pages 65 – 69, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[40] Simone L. Moura, Rafael J. N. dos Santos, Sean W. M. Siqueira, Maria Helena Lima Baptista Braz, and Rubens N. Melo. Metadata replication in e-learning using web-services and ontologies. In *International Conference on Information Integration and Web-based Applications and Services*, 2004.

[41] David Nagle, Denis Serenyi, and Abbie Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 53, Washington, DC, USA, 2004. IEEE Computer Society.

[42] H. Newman. Data intensive grids and networks for high energy and nuclear physics. *Nuclear Physics B - Proceedings Supplements*, 120:109–112, June 2003.

[43] Paul Nowoczynski, Nathan Stone, Jared Yanovich, and Jason Sommerfield. Zest checkpoint storage system for large supercomputers. In *3rd Petascale Data Storage Workshop*, pages 1–5, Nov. 2008.

[44] John Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the unix 4.2 bsd file system. pages 15–24, 1985.

[45] Esther Pacitti, Pascale Minet, and Eric Simon. Replica consistency in lazy master replicated databases. *Distributed Parallel Databases*, 9(3):237–267, 2001.

[46] Esther Pacitti, M. Tamer Özsu, and Cedric Coulon. Preventive multi-master replication in a cluster of autonomous databases. In *In Euro-Par 2003 Parallel Processing*, volume 2790/2004, pages 318–327, 2003.

[47] Esther Pacitti and Eric Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *The VLDB Journal*, 8(3-4):305–318, 2000.

[48] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 288–301, New York, NY, USA, 1997. ACM.

[49] S. Y. Ponomarev, T. C. Bishop, and V. Putkaradze. Dna relaxation dynamics in 1id3 yeast nucleosome md simulation. *Biophysical Journal*, 96, February 2009.

[50] M. Qu Yang and J.Y. Yang. High-performance computing for drug design. In *Bioinformatics and Biomeidcine Workshops, 2008. BIBMW 2008. IEEE International Conference on*, pages 120–120, Nov. 2008.

[51] Arcot Rajasekar, Mike Wan, Reagan Moore, and Wayne Schroeder. A prototype rule-based distributed data management system. In *HPDC workshop on Next Generation Distributed Data Management*, Paris, France, May 2006.

[52] Mahadev Satyanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, 23(5):9–18, 20–21, 1990.

[53] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[54] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 19, Berkeley, CA, USA, 2002. USENIX Association.

[55] Dennis Shasha. Lessons from wall street: case studies in configuration, tuning, and distribution. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 498–501, New York, NY, USA, 1997. ACM.

[56] B.H. Stamey, V.Wang, and M. Koterba. Predicting the next storm surge flood. *Sea Technology*, pages p10–15, August, 2007.

[57] Danny Teaff, Dick Watson, and Bob Coyne. The architecture of the high performance storage system (hpss). In *In Proceedings of the Goddard Conference on Mass Storage and Technologies*, pages 28–30, 1995.

[58] Jeff Terrace and Michael J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *Proceedings of USENIX Annual Technical Conference*, San Diego, CA, Jun 2009.

[59] Douglas Thain and Miron Livny. Parrot: An Application Environment for Dataintensive Computing. *Scalable Computing: Practice and Experience, Volume 6, Number 3*, pages 9–18, 2005.

[60] Brian Tierney, Jason Lee, Ling Tony Chen, Hanan Herzog, Gary Hoo, Guojun Jin, and William E. Johnston. Distributed parallel data storage systems: a scalable approach to high speed image servers. In *MULTIMEDIA '94: Proceedings of the second ACM international conference on Multimedia*, pages 399–405, New York, NY, USA, 1994. ACM.

[61] Brian L. Tierney, Jason Lee, Brian Crowley, Mason Holding, Jeremy Hylton, and Fred L. Drake Jr. A network-aware distributed storage cache for data intensive environments. In *HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, page 33, Washington, DC, USA, 1999. IEEE Computer Society.

[62] J. A. Tyson. Large synoptic survey telescope: Overview. In *Survey and Other Telescope Technologies and Discoveries. Edited by Tyson, J. Anthony; Wolff, Sidney. Proceedings of the SPIE, Volume 4836, pp. 10-20 (2002).*, pages 10–20, dec 2002.

[63] Xinqi Wang, Dayong Huang, Ismail Akturk, Mehmet Balman, Gabrielle Allen, and Tevfik Kosar. Semantic enabled metadata management in petashare. In *International Journal of Grid and Utility Computing (IJGUC)*, volume 1, pages 275–286, 2009.

[64] Sage Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06)*, volume 7. USENIX, November 2006.

[65] Andrea Weise, Mike Wan, Wayne Schroeder, and Adil Hasan. Managing groups of files in a rule oriented data management system (irods). In *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part III*, pages 321–330, Berlin, Heidelberg, 2008. Springer-Verlag.

[66] Shuqing Wu and Bettina Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. *International Conference on Data Engineering*, pages 422–433, 2005.

[67] Gongye Zhou, Qiuju Lan, and Jincai Chen. A dynamic metadata equipotent subtree partition policy for mass storage system. *Frontier of Computer Science and Technology, Japan-China Joint Workshop on*, 0:29–34, 2007.

Vita

Ismail Akturk was born in 1984, in Trabzon, Turkey. He has spent his life in Samsun until he received his high school diploma from Samsun Anatolian High School in 2002. Then, he moved to his beloved city, Istanbul, for his higher education. He received a bachelor's degree in computer engineering at Dogus University in Istanbul, Turkey in 2007. He worked on mobile ad-hoc networks in his graduation thesis. In August 2007, he has joined Louisiana State University to pursue his master's degree in the Department of Electrical and Computer Engineering. He has worked in Center for Computation and Technology as a graduate research assistant and involved in state wide data archival infrastructure project called PetaShare in which he has focused on distributed storage and data management systems.