Scientific Programming 19 (2011) 27–43
DOI 10.3233/SPR-2011-0317
IOS Press

PetaShare: A reliable, efficient and transparent distributed storage management system

Tevfik Kosar [a,b,c,*], Ismail Akturk [d], Mehmet Balman [e] and Xinqi Wang [a,b]

[a] *Department of Computer Science and Engineering, State University of New York, Buffalo, NY, USA*
[b] *Department of Computer Science, Louisiana State University, Baton Rouge, LA, USA*
[c] *Center for Computation and Technology, Louisiana State University, Baton Rouge, LA, USA*
[d] *Department of Computer Engineering, Bilkent University, Ankara, Turkey*
[e] *Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA, USA*

Abstract. Modern collaborative science has placed increasing burden on data management infrastructure to handle the increasingly large data archives generated. Beside functionality, reliability and availability are also key factors in delivering a data management system that can efficiently and effectively meet the challenges posed and compounded by the unbounded increase in the size of data generated by scientific applications. We have developed a reliable and efficient distributed data storage system, PetaShare, which spans multiple institutions across the state of Louisiana. At the back-end, PetaShare provides a unified name space and efficient data movement across geographically distributed storage sites. At the front-end, it provides light-weight clients the enable easy, transparent and scalable access. In PetaShare, we have designed and implemented an asynchronously replicated multi-master metadata system for enhanced reliability and availability, and an advanced buffering system for improved data transfer performance. In this paper, we present the details of our design and implementation, show performance results, and describe our experience in developing a reliable and efficient distributed data management system for data-intensive science.

Keywords: Distributed data storage, metadata management, asynchronous replication, advanced buffer, reliability, performance, data-intensive science, PetaShare

1. Introduction

A data-intensive cyber-infrastructure has become increasingly important in interdisciplinary research projects that are generating ever larger data archives and requiring ever more sophisticated data management services to handle these archives. Simply purchasing high-capacity, high-performance storage systems and adding them to the existing infrastructure of the collaborating institutions does not solve the underlying and highly challenging data management problems. Scientists are compelled to spend a great amount of time and energy on solving basic data-handling issues, such as how to find out the physical location of data, how to access it, and/or how to move it to visualization and/or compute resources for further analysis.

There are two main components in a distributed data management architecture: a data server which coordinates physical access (i.e., writing/reading data sets to/from disks) to the storage resources, and a metadata server which provides global name space to ensure location transparency of data as well as storage resources, and keeps all related information regarding the system. Along with other design issues and system components, metadata server layout has impact on the following system features: availability, scalability, load balancing and performance. The metadata server is generally implemented as a single central entity which makes it a performance bottleneck as well as a single point of failure. Obviously, replication of the metadata server is necessary to ensure high availability as well as increased local performance. On the other hand, a replicated multi-metadata server architecture comes with some challenges such as synchronization of these servers, data coherency and overhead of synchronization.

[*]Corresponding author: Tevfik Kosar, Department of Computer Science and Engineering, State University of New York, Buffalo, NY, USA. E-mail: tkosar@buffalo.edu.

Metadata servers can be synchronized either synchronously or asynchronously. In synchronous replication, incoming request that requires metadata update is propagated to all metadata servers before it gets committed. Metadata information is updated if and only if, all metadata servers agree to commit the incoming request. Propagating update messages to all replicating metadata servers and receiving corresponding confirmations takes time which degrades the metadata access performance. To eliminate the overhead of synchronization of metadata servers in synchronous replication, we exploit asynchronous replication. Asynchronous replication allows a metadata server to process the incoming request by itself without propagating the request to all replicating servers immediately. Metadata servers are updated asynchronously in the background. This dramatically increases performance for metadata access, especially for write operations. One of the challenges in asynchronous metadata replication is providing metadata consistency across all sites.

There have been many efforts in parallel and distributed data management systems to provide large I/O bandwidth [4,16,23]. However, metadata management is still a challenging problem in widely distributed large-scale storage systems. Scalability in file metadata operations for parallel filesystems has been studied in [5]. In [12], I/O operations are delegated to a set of nodes to overcome I/O contention. GPFS [15] handles metadata and data management separately and it uses shared lock mechanism to enable simultaneous updates to file metadata from multiple clients. Collective communication patterns are proposed between storage servers to simplify consistency controls. In [20, 21], metadata workload has been distributed among multiple servers for performance and scalability. With its dynamic subtree partitioning, Ceph [20] provides adaptability and failure detection for changing environment conditions. It also performs lazy commits in metadata operations to improve performance. I/O bottleneck is one of the bottlenecks in parallel scientific computation [9]. Adaptability, reliability, latency between resources and utilization of available capacity are some of the challenges in distributed data management to provide an efficient infrastructure [2,3].

In this paper, we present the design and implementation of a reliable and efficient distributed data storage system that we have developed, PetaShare, which spans multiple institutions across the state of Louisiana. In Section 2, we will give a brief introduction to PetaShare and its underlying technologies; Section 3 will present the design and implementation of the asynchronous multi-master metadata replication in PetaShare; Section 4 will discuss our work on advanced buffer to improve data transfer performance; and Section 5 will conclude the paper.

2. PetaShare

The NSF funded PetaShare project aims to enable transparent handling of underlying data sharing, archival, and retrieval mechanisms, and make data available to scientists for analysis and visualization on demand. The goal is to enable scientists to focus on their primary research problems, assured that the underlying infrastructure will manage the low-level data-handling issues. Our initial implementation and deployment of PetaShare involves five state universities and two health sciences centers in Louisiana. These institutions are Louisiana State University, Tulane University, University of New Orleans, University of Louisiana at Lafayette, Louisiana Tech University, Louisiana State University, Shreveport and Louisiana State University–Health Sciences Center in New Orleans. PetaShare manages 250 terabytes of disk storage distributed across these sites as well as 400 terabytes of tape storage centrally located nearby LSU campus. 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 [1,10]. The internal network connection of LONI resources and the distribution of the PetaShare resources among the LONI sites are shown in Fig. 1.

PetaShare provides scientists with simple uniform interfaces to store, access and process heterogeneous distributed data sources. The archived data is well cataloged to enable easy access to the desired files or segments of files, which can then be returned to the requester in a chosen format or resolution. Multiple copies of high priority information can be stored at different physical locations to increase reliability and also enable easier retrieval by scientists in different geographical locations. The data is also indexed to enable easy and efficient access to the desired data. The requested data is moved from the source or archival sites to the computation sites for processing as required, and the results then sent back to the interested parties for further analysis or back to the long term storage sites for archival. To improve data transfer performance, we introduced advanced buffer to our system, we will elaborate in Section 4.

The back-end of PetaShare, as illustrated in Fig. 2 is based on enhanced version of iRODS and Stork

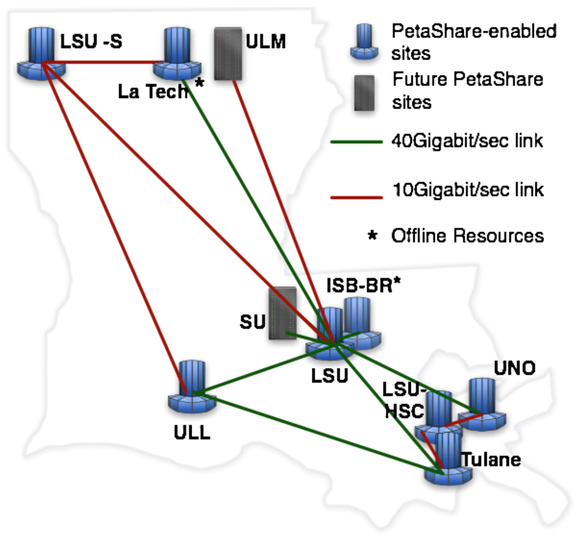

Fig. 1. Louisiana Optical Network Initiative and PetaShare sites. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0317.)

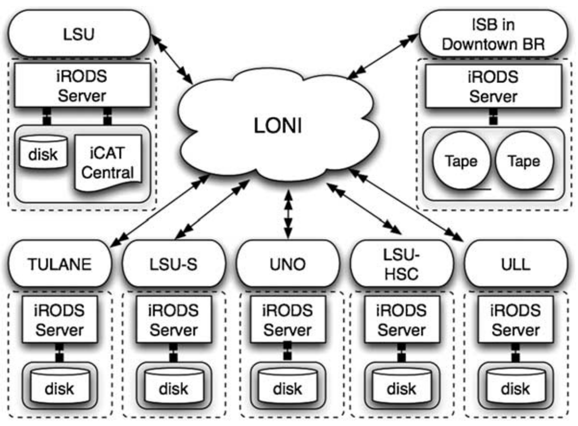

Fig. 2. PetaShare architecture.

technologies which provide a unified name space and efficient data movement across geographically distributed storage sites. iRODS stores all the system information, as well as user-defined rules in centralized database, which is called iCAT. iCAT contains the information of the distributed storage resources, directories, files, accounts, metadata for files and system/user rules. iRODS provides a generic client API to control the iCAT. We try to utilize and enhance this API for achieving better performance and reliability for PetaShare. iRODS is based on client/server architecture [22]. A sample deployment contains an iCAT server along with multiple iRODS servers on each site. These iRODS servers manage the accesses to the physical data in the resources. They interact with iCAT server to control the accesses to the resources. As it can be easily noticed, the existence of central iCAT server is a single point of failure. Since the iCAT server is the only authority to provide unified namespace and all system information, the overall system becomes unavailable whenever iCAT server fails. To overcome this problem, we introduced asynchronous replication into iCAT, we will discuss it in Section 3.

2.1. *Client tools*

PetaShare provides three different client tools for its users: petashell, petafs and pcommands. Petashell is an interactive shell interface that catches system I/O calls made by an application and maps them to the relevant iRODS I/O calls. Petafs is a userspace virtual filesystem that enables users to mount PetaShare resources to the local machines. Pcommands are a set of UNIX-like commands that are specialized for interacting with PetaShare. These tools differ from each other in terms of their capabilities and usage. For example, petafs can be used in systems that their kernels have FUSE [6] support. Contrary, petashell and pcommands do not require any kernel level support. Further discussions below make it clear why there are three different client tools.

Pcommands allow users to access PetaShare resources, and provide fundamental data access utilities, such as listing, copying, moving, editing and deleting. However, shortcoming of Pcommands is that data on PetaShare cannot be transparently accessed by the user applications that are run locally. Pcommands can work to stage in/out the data between PetaShare resources and local machine. However, in most cases the sizes of input and output data of applications are exceeding the storage limits of the machines on where application runs. Furthermore, it is usually impractical to deploy the application into where the data resides due to reasons such as incompatibility of machines, copyrights, licenses and security. This is why transparent remote access to data is important for the user applications.

This is the case where petashell and petafs come into the picture. They make it possible to run applications without staging in/out the data to/from local machines, or deploy the application to the machines where the data reside. The motivation behind petashell and petafs is that users should be able to run their own applications in their machines while the required and produced data stay on remote resources. Furthermore, users should be unaware of the details of connecting to remote resources, interacting applications

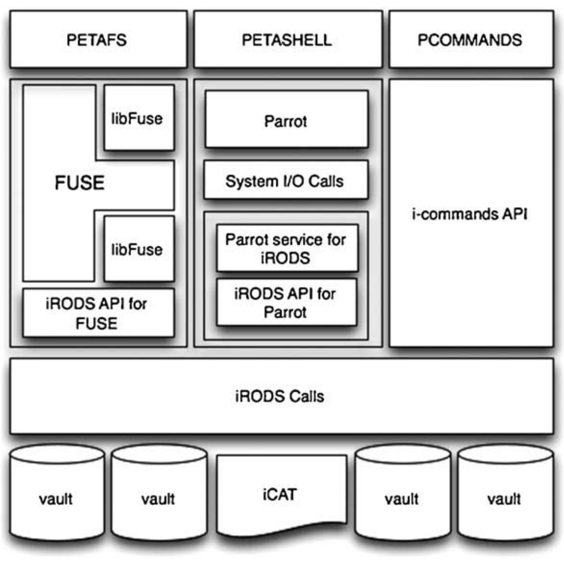

Fig. 3. Layered view of PetaShare client tools.

with the remote data and locating the data physically. All these operations should be transparent to the users. In PetaShare, applications interacts with remote data through petashell and petafs that translate the I/O requests made by user application into the respective iRODS I/O calls. These calls provide the required data or store the produced data in PetaShare system and update iCAT database. The oveview of PetaShare client tools and their interactions with low-level iRODS components are shown in Fig. 3.

As it can be seen in Fig. 3, the client tools use iRODS API to call relevant iRODS operations at the lowest level to handle I/O requests of an application. Petashell uses an existing open-source software, Parrot [19], to catch system I/O calls of an application and to match them with the respective iRODS I/O calls. On the other hand, petafs pretends as a filesystem to handle system I/O calls and maps filesystem calls to respective the iRODS calls through a special interface called FUSE. Pcommands use iRODS API to access the data on PetaShare.

2.1.1. *Petashell*

Petashell is an interactive shell interface that allows users to run their applications on their machines while data resides on remote PetaShare resources. That means, there is no need to move the data to the machine where application runs, or port the application to where data resides. Petashell attaches application and data together while both are physically separated.

Petashell is based on *Parrot* which is a tool for attaching running programs to remote I/O systems

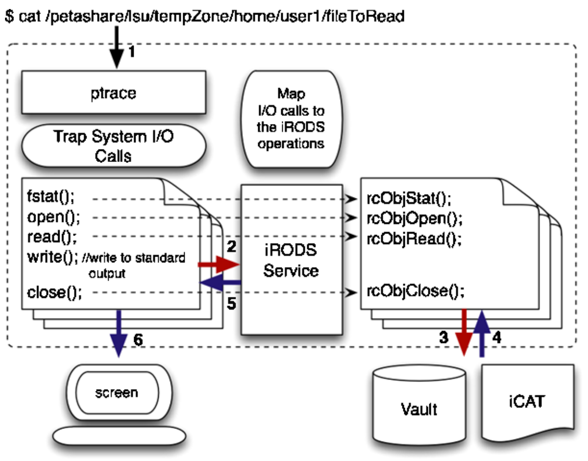

Fig. 4. Mapping system I/O calls to iRODS calls via petashell. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0317.)

through the filesystem interface [18]. The main idea behind Parrot is to catch system I/O calls and translate these calls into the corresponding I/O operations of remote system. Basically, the system calls of an application are trapped through the *ptrace* debugging tool, and corresponding remote I/O operations are sent to the remote system. Currently, Parrot offers service for various remote systems, such as http, ftp, gridftp, glite and iRODS [17]. In our case, iRODS service libraries are used to implement petashell where system I/O calls of application are translated into the respective iRODS I/O routines. For example, if a user runs *cat* utility to read a file on PetaShare resource, then Parrot captures the I/O calls (i.e., fstat, open, read, write, close) made by *cat*, and maps these I/O calls to corresponding iRODS calls (i.e., rcObjStat, rcDataObjOpen, rcDataObjRead, rcDataObjClose). This is illustrated in Fig. 4. Here, *cat* is run in petashell interface where */petashare* specifies the service that is being used in Parrot (basically PetaShare service uses iRODS service libraries of Parrot), */lsu* specifies the name of PetaShare site that is going to be accessed, and */tempZone* states the name of current zone in PetaShare. Petashell installation requires no kernel modification, so unprivileged users can run it without worrying about low-level system and permission issues [13].

2.1.2. *Petafs*

Petafs is a virtual filesystem that allows users to access PetaShare resources as a local filesystem after being mounted to their machines. By using petafs, PetaShare resources can be seen in the directory hierarchy of an existing filesystem and can be accessed in the same way as an existing filesystem.

Petafs is based on FUSE (Filesystem in Userspace) which is a simple interface to export a virtual filesystem to the Linux kernel in userspace [6]. FUSE interacts with the existing filesystem at the kernel level and maps virtual filesystem calls to the existing filesystem calls. Petafs provides a FUSE module for iRODS that matches iRODS calls with FUSE calls in the FUSE library. In the kernel, FUSE incorporates with the real filesystem and maps these FUSE calls to the actual filesystem calls. This is done through FUSE library in a similar way to matching iRODS calls with FUSE calls.

The communication between kernel module and FUSE library is established by specifying a file descriptor which is obtained from */dev/fuse* device file. This file descriptor is passed to the *mount* system call to match up the file descriptor with the mounted petafs virtual filesystem [6]. Installation of petafs requires kernel support for FUSE, so it is appropriate for privileged users. However, unprivileged users can still use petafs if their kernels support FUSE already.

2.1.3. Pcommands

Pcommands are command-line utilities to access PetaShare. They evoking the basic UNIX-like commands, such as pls, pcp, pmkdir where UNIX counterparts are ls, cp, mkdir, respectively. These commands interact with PetaShare directly by using iRODS API.

Pcommands are based on iRODS i-commands. The iRODS i-commands are command-line utilities of iRODS that interface to an iRODS system [14]. Pcommands differ from i-commands by providing transparent interface for multiple iRODS servers. In our case, PetaShare is composed of several iRODS servers where we want to make users unaware of the details of each of these iRODS servers. Pcommands provide an interface where users only need to know the name of the PetaShare site that they want to access. After providing the name of the PetaShare site, pcommands automatically adjust the iRODS environment files to access to the desired PetaShare site.

Pcommands also provide some additional utilities such as *pchangeuser* that enables users to switch between their existing PetaShare accounts (note that PetaShare accounts are created for project groups, not for individuals, so a user may have permission to access multiple PetaShare accounts if the user is involved in different research groups). Pcommands enable users to access PetaShare storage resources who are using various types of operating systems; such as Linux, Solaris, MacOS and AIX.

3. Asynchronous multi-master metadata replication

As discussed in the previous chapter, PetaShare based on iRODS and iCAT has a major weakness since there is a single point of failure. The failure of iCAT server that keeps complete system information makes the overall system failed unavailable, since there is no authority that provides globally unified namespace to access the resources. To solve this issue, we attempted to replicate iCAT server, the replication of iCAT server should be transparent, but not affect the performance of the system. We cloned the PetaShare system on the testbed and replicated iCAT servers synchronously. Unfortunately, we obtained high latency and performance degradation on data transfers while each transfer is committed after iCAT servers complete replicating themselves. As a result, we decided to develop an asynchronous replication system.

3.1. Conflict detection and resolution

The biggest problem of asynchronous multi-master replication is that conflicts occur if two sites update their databases within the same replication cycle. For this reason, the proposed multi-master replication method should detect and resolve possible conflicts. However, it is well known that detecting and resolving conflicts require complex algorithms. Fortunately, we have built a conceptual conflict resolver that handles such conflicts efficiently. Common conflict types are: (i) *uniqueness conflicts*: occur if two or more sites try to insert the records with the same primary key; (ii) *update conflicts*: occur if two or more sites try to update the same record within the same replication cycle; (iii) *delete conflicts*: occur if one site deletes a record from database while another site tries to update this record.

Typically, conflict resolution mechanisms are based on timestamps. However, we also introduce intelligent database design to eliminate some of these conflicts. For example, in the iCAT database, each object is given a unique ID within increasing sequence of *big int* type. If two separate sites insert new records into the database within the same replication cycle, then these two records will have the same ID in which case a uniqueness conflict occurs. To eliminate uniqueness conflicts, we divide the range of *big int* type into the certain-sized intervals (i.e., relative to the number of sites). Then, these intervals are assigned to different sites. Dividing ID space into the intervals allows each site to assign an ID to an inserted record from a disjoint sequence of IDs. For example, we assigned an interval of 10,000 and 8,000,000 to the first iCAT server, 8,000,001 and 16,000,000 to the second iCAT server, and so on. Thus, if two sites insert new records within the same replication cycle, we are ensured that these records will have different IDs and uniqueness conflicts never occur in iCAT database. On the other hand, update conflicts occur if two or more sites try to update the same record within the same replication cycle. Using timestamps is the most famous technique to resolve such conflicts. There are two ways of using timestamps. The first one is to use latest timestamp value [11]. This is the simplest technique since updates are run sequentially where the latest update (i.e., the latest timestamp value) overwrites all the previous updates. The second one is to use earliest timestamp value [11]. This is more complex than the first technique where all conflicting updates should be identified and only the first one (i.e., earliest timestamp value) should be processed.

We identified two different update types occurred in the ICAT database and developed different conflict resolution methods for them. The first type of updates target only the tables of iCAT database, such as updating accounts and permissions. The second type of updates target also the data stored in the resources, such as writing a file.

The update conflicts of the first type are resolved by negotiation. For example, an update conflict occurs if two sites update the password of the same account within the same replication cycle. Then, both sites are informed that an update conflict has occurred and negotiation process is started. They have to decide which update should be accepted and replicated. During the process of negotiation, the latest timestamp value is used to resolve the conflict temporarily. If there is no agreement within a certain amount of time (i.e., 24 h), the latest timestamp value becomes concrete and conflict is considered as resolved.

The update conflicts of the second type are resolved in the following way. The update request for data can be done in any site regardless the physical location of the data. So, if an update request is made for the data that reside in a remote resource, then the iRODS server should send the update request along with the changes (i.e., changed bytes) to the corresponding iRODS server. This iRODS server accepts the update requests from only one site for the same data within a replication cycle. Other sites that try to update the same data within the same replication cycle receive `update-rejected` message from the corresponding iRODS server.

Another case in which the update conflicts of the second type can occur is that if two iRODS servers try to add new files with the same file name to the same PetaShare directory (although files are located in different resources). The update conflict occurs since both files have the same name; although, they have unique IDs. This type of conflicts are also resolved by negotiation. The first concern of us is the data safety. We ensure that the written files are in safe in the resources. However, the conflict appears in the unified namespace has to be resolved. For this reason, whenever such a conflict is detected, our proposed conflict resolver automatically changes the names of the conflicting files. For example, if two files have the name of *fileA*, then one of the file name becomes *fileA_1*. Afterwards, the conflict resolver acknowledges both iRODS server that conflict has been occurred and file name has been changed. A user has a flexibility to accept the changed file, or rename it. These updates for the file name are also replicated among all sites.

The delete conflicts are treated in the same way of update conflicts. However, there is an additional control on delete operations. If a file is deleted, it is first moved to a special directory called `trash`. To delete this file permanently, a special command has to be used. However, the usage of this command is not allowed within the same replication cycle with the deletion of that file. This is done to prevent undesired circumstances. For example, a delete conflict occurs if a site requests an update for a file that is deleted by another site within the same replication cycle. However, if user does not agree on delete operation, then file can be rescued and updated since it is kept in the trash during the replication cycle. On the other hand, file can be deleted permanently if no delete conflict occurs within a replication cycle.

3.2. Implementation of asynchronous multi-master metadata 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 metadata 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 Fig. 5.

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 [8], we synchronize all metadata servers in every 30 s 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 Fig. 6.

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 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 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 cannot 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. 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

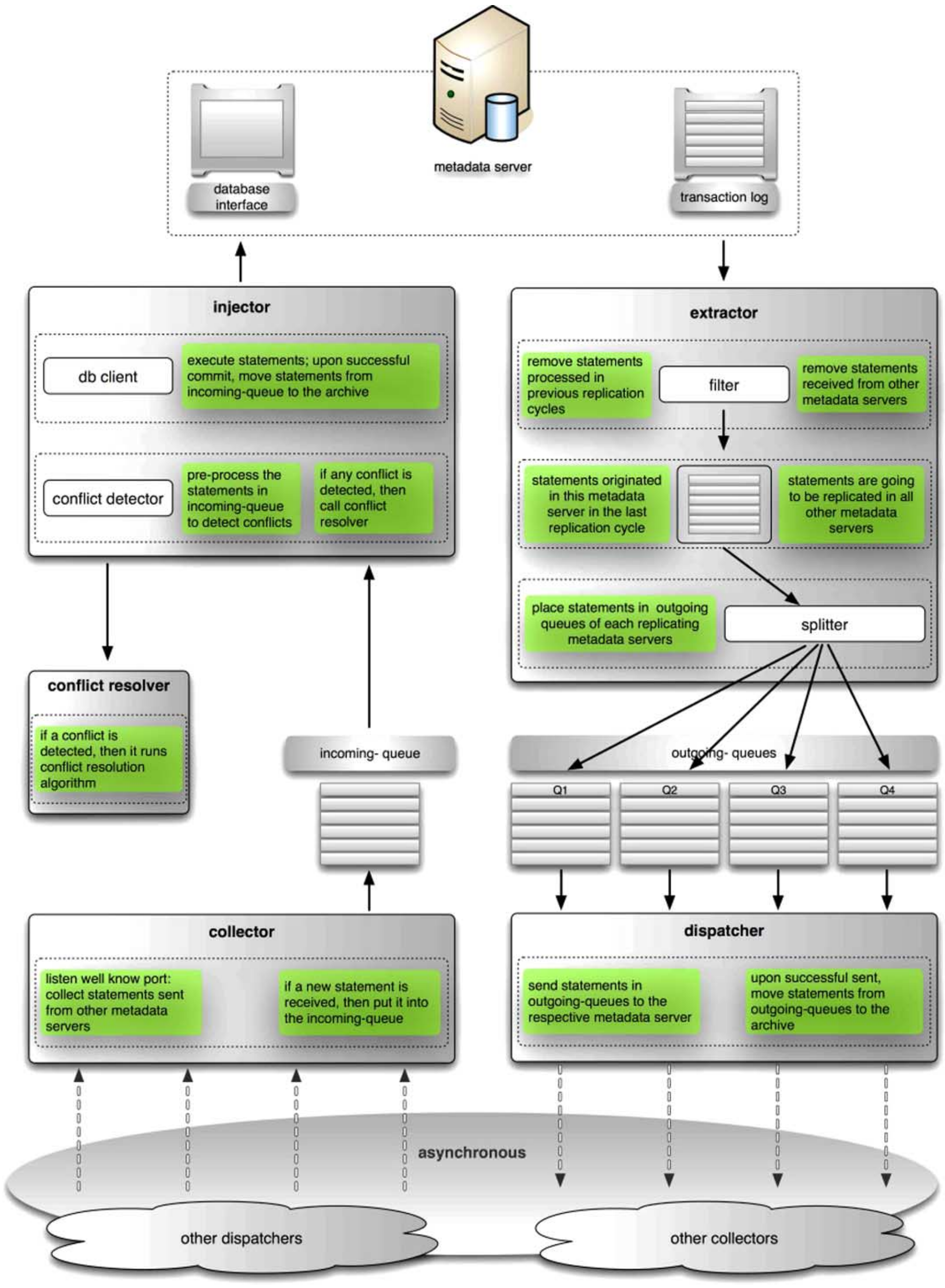
metadata server
database interface
transaction log
injector
db client
execute statements; upon successful commit, move statements from incoming-queue to the archive
conflict detector
pre-process the statements in incoming-queue to detect conflicts
if any conflict is detected, then call conflict resolver
conflict resolver
if a conflict is detected, then it runs conflict resolution algorithm
incoming- queue
extractor
remove statements processed in previous replication cycles
filter
remove statements received from other metadata servers
statements originated in this metadata server in the last replication cycle
statements are going to be replicated in all other metadata servers
place statements in outgoing queues of each replicating metadata servers
splitter
outgoing- queues
Q1
Q2
Q3
Q4
collector
listen well know port: collect statements sent from other metadata servers
if a new statement is received, then put it into the incoming-queue
dispatcher
send statements in outgoing-queues to the respective metadata server
upon successful sent, move statements from outgoing-queues to the archive
asynchronous
other dispatchers
other collectors

Fig. 5. Components of asynchronous multi-master replication tool. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0317.)

and conflict resolver deals with the conflict as discussed in previous section. The flowchart of the asynchronous multi-master replication procedure is shown in Fig. 7.

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

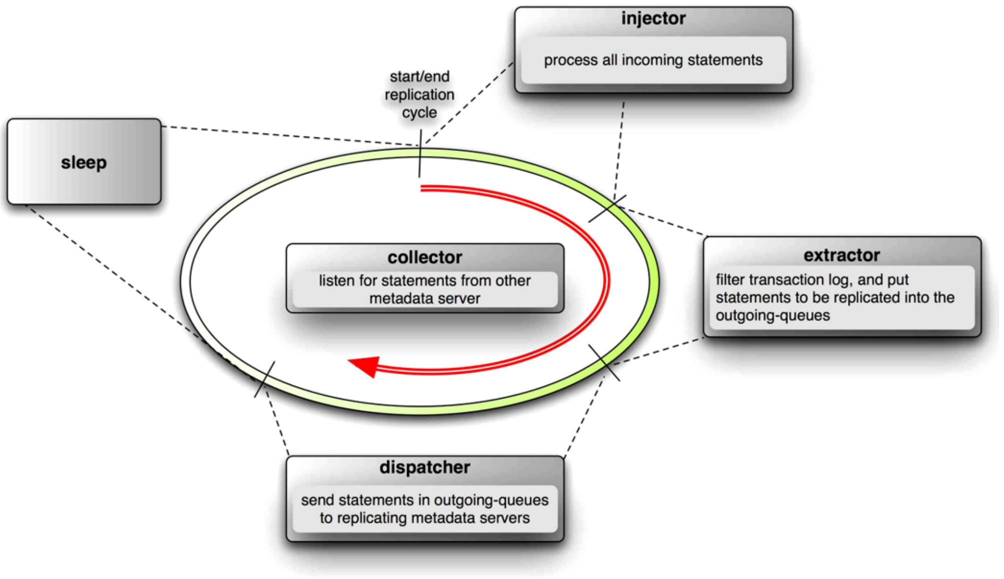
injector
process all incoming statements
start/end replication cycle
sleep
collector
listen for statements from other metadata server
extractor
filter transaction log, and put statements to be replicated into the outgoing-queues
dispatcher
send statements in outgoing-queues to replicating metadata servers

Fig. 6. Replication cycle. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0317.)

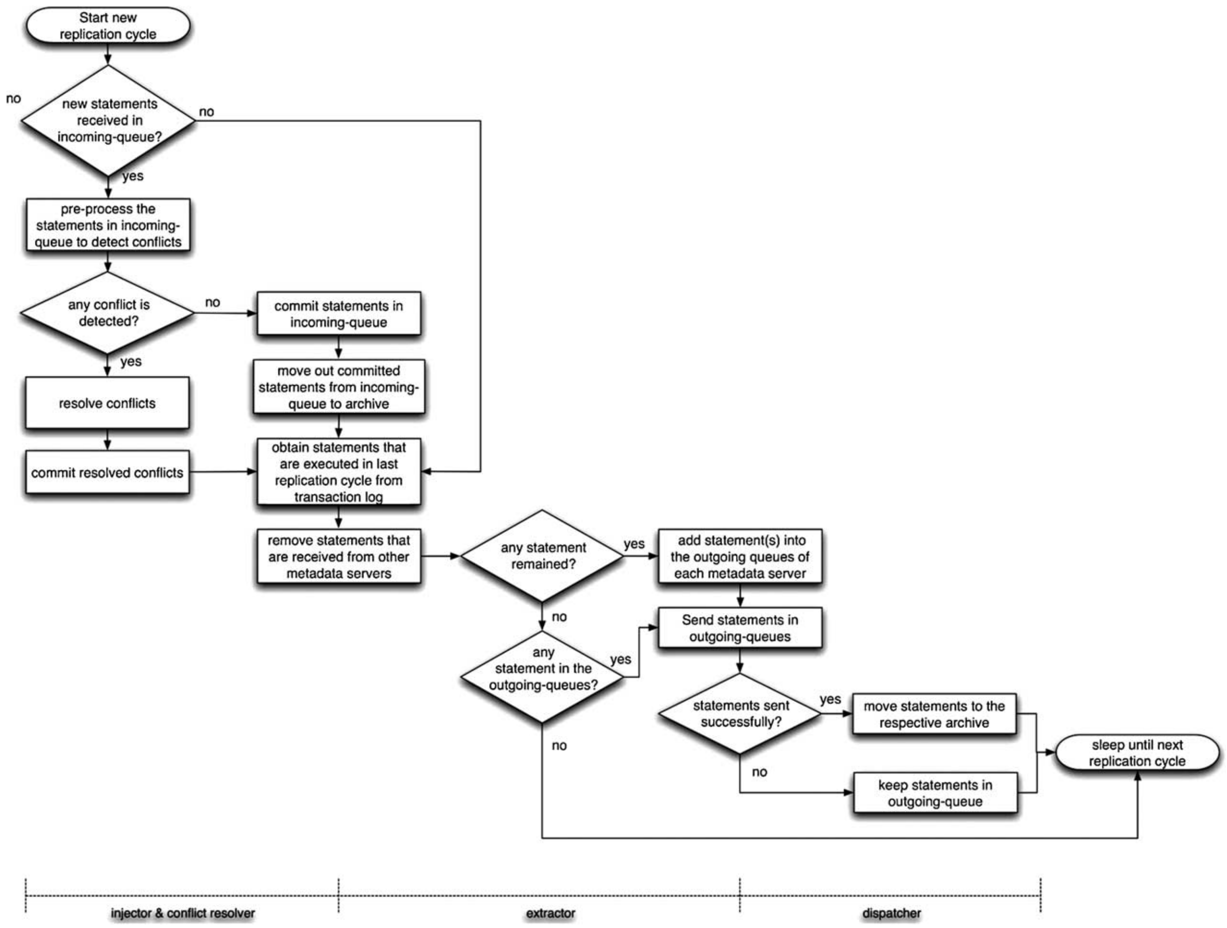
Start new replication cycle
new statements received in incoming-queue?
no
no
yes
pre-process the statements in incoming-queue to detect conflicts
any conflict is detected?
no
yes
commit statements in incoming-queue
move out committed statements from incoming-queue to archive
resolve conflicts
commit resolved conflicts
obtain statements that are executed in last replication cycle from transaction log
remove statements that are received from other metadata servers
any statement remained?
yes
add statement(s) into the outgoing queues of each metadata server
no
Send statements in outgoing-queues
any statement in the outgoing-queues?
yes
no
statements sent successfully?
yes
move statements to the respective archive
no
keep statements in outgoing-queue
sleep until next replication cycle
injector & conflict resolver
extractor
dispatcher

Fig. 7. Flowchart of replication process.

as network, space allocated for queues, and frequency of updating metadata servers. The asynchronous replication method that we have developed does not only improve reliability of the PetaShare system, but also leads us to achieve lower network overhead compared to the synchronous replication. It also outperforms the central iCAT model since the operations are made on the local iCAT server and replicated asynchronously. The quantitative assessment of asynchronous replication is given in Section 3.3.

3.3. Results

The replicated iCAT servers make PetaShare system more reliable. They also reduce the latency since the requests can be handled by local iCAT servers. However, the replication method can influence the operation time dramatically. For example, synchronous replication method increases the latency, since each database operation should be approved by all iCAT servers before getting committed. On the other hand, asynchronous replication method eliminates such latency. Of course, asynchronous replication has its own challenges and threats; however, we discussed them in Section 3. In this section, we compare the replication methods, and show the positive affects of asynchronous replication method on the system performance.

We have performed our tests on a testbed in LONI, on the same environment that real PetaShare system runs. We use Pcommands as a client interface in these tests. We have performed tests for writing to remote site, writing to local site, reading from remote site and reading from local site. These different categories of tests let us to see the characteristics of replication models and to draw a better conclusion regarding the performance of the models. In each test, we used 1000 files of the same size and a single file that has the size of 1000-file-bytes. By doing this, we can see the effect of the number of files on overall time. On the database, the number of operations increases while the number of files increases. This introduce database overhead along file open/close overhead. We repeated this strategy for different size of files to see the correlation between the file size and database and file open/close overheads. We use the data sets of 10 KB × 1000 files and a 10 MB file, 100 KB × 1000 files and a 100 MB file, and 1 MB × 1000 files and a 1 GB file.

The most expensive operation is to write a file in a remote resource. In central iCAT server model, write request should be sent to the iCAT server while file is sent to the remote resource. The requirement of these two distinct remote connection increases the latency and network traffic. In synchronous replication method, the latency and network traffic increases dramatically, since a write request should be forwarded to the all iCAT servers. Note that this is done for the sake of increasing the reliability of the system. On the other hand, a write request can be handled in local iCAT server in asynchronous replication method. The overhead of getting unified namespace information and updating database is minimized, since there is a local iCAT server. The main source of the latency is to send a file to the remote resource. Table 1 shows the average time of writing different data sets to the remote resource for all three replication methods.

Table 1
Average duration of writing to remote PetaShare resources for three replication scenarios

	Writing to remote PetaShare resources		
	No replication	Synchronous replication	Asynchronous replication
10K × 1000 files	75.94	240.32	38.81
10M (single file)	1.38	1.48	1.34
100K × 1000 files	83.53	247.43	47.76
100M (single file)	2.51	2.96	2.83
1M × 1000 files	156.61	320.96	144.53
1G (single file)	9.91	11.68	10.31

As we can see from Table 1, time required to write 1000 files is much bigger than writing a single 1000-file-sized file. This is the case because writing 1000 files requires 1000 writing request on iCAT server, as well as 1000 file open/close calls at both ends, opposed to single writing request on iCAT and single file open/close operation of a 1000-file-sized file. The effects of replication methods can be seen in respective columns. All the replication methods have similar values for single files, since time is dominated by sending data instead of iCAT operation. However, when the number of file is increased to 1000, each method can be identified easily. It is obvious that synchronous replication method is the worst one since it replicates 1000 write requests to the other iCAT servers. Central iCAT gives better performance than synchronous replication method since it requires to send write request to only one iCAT server. The asynchronous method outperforms the other two methods since write requests are handled by local iCAT server.

It is worth to note time does not increase linearly with the size of files. The ratio of time to write 1000 file and 1000-file-sized file becomes smaller when the file size increases. This is simply because the time spent to send the data to the resource hides the overhead of database and file open/close operations (i.e., these

Table 2

Average duration of writing to local PetaShare resource for three replication scenarios

	Writing to local PetaShare resource		
	No replication	Synchronous replication	Asynchronous replication
10K × 1000 files	63.51	221.33	19.92
10M (single file)	0.25	0.46	0.15
100K × 1000 files	64.42	223.06	21.38
100M (single file)	0.58	1.02	0.47
1M × 1000 files	75.02	232.48	36.75
1G (single file)	5.16	9.18	8.37

overheads are independent from the file size). In the last row of Table 1, central iCAT model has smaller value than asynchronous replication method. However, this was because temporary network congestion occurred that increases the average of asynchronous replication method when 1 GB file has been sent. We believe that it is not essential to draw a general picture.

Table 2 shows the average time of writing different data sets to the local resource for all three replication methods. Writing to local resource is less expensive than writing to remote resource, since there is no network latency to send the data to the remote resource. This situation lets us to evaluate the performance of replication methods, because the contribution of data transfer to the latency is minimized besides the contribution of the database and file open/close operations is fixed. Although, it takes less time to write files to local resources than remote resources, the conclusions derived from the tests are very similar. For all data sets the asynchronous replication method outperforms the others, since both write and database operations are done locally. Similar to the first case, the central iCAT model gives better results than synchronous replication.

Table 3 shows the average time of reading different data sets from the remote resources for all three replication methods. The biggest difference of reading than writing is that it requires less database transaction on iCAT server. A writing a file requires insert and update operations on different tables while reading a file requires a select (i.e., to learn physical address of data) operation on database. This alleviates the pressure on replication methods, especially synchronous replication method. It is worth to note in Table 3 that synchronous replication method performs better than central iCAT. This is because reading request can be replicated faster among iCAT servers since there is no need to negotiate on it (i.e., it is a select operation). However, synchronous replication method stays behind the asynchronous replication method. Table 4 shows the average time of reading different data sets from the local resources for all three replication methods. The asynchronous replication method performs the best while synchronous replication method and central iCAT model draw close performance for smaller files. The performance of synchronous replication method starts to draw away from the central iCAT model whenever the file size starts to increase. However, this was not the case in the tests of reading from remote resources. This can be explained as file is read from remote resource where network latency has a bigger contribution to the spent time. The contribution of network latency increases that hides the overhead of the synchronous replication as the file size increases. However, the contribution of network latency disappears if file is read from local resource. For this reason, the overhead of the synchronous replication method becomes obvious.

Table 3

Average duration of reading from remote PetaShare resources for three replication scenarios

	Reading from remote PetaShare resources		
	No replication	Synchronous replication	Asynchronous replication
10K × 1000 files	46	40.75	26.51
10M (single file)	1.58	1.74	1.47
100K × 1000 files	55	48.92	34.13
100M (single file)	3.44	3.54	3.29
1M × 1000 files	166.98	157.43	144.98
1G (single file)	14.47	11.55	9.57

Table 4

Average duration of reading from local PetaShare resource for three replication scenarios

	Reading from local PetaShare resource		
	No replication	Synchronous replication	Asynchronous replication
10K × 1000 files	24.33	23.63	17.8
10M (single file)	0.2	0.36	0.14
100K × 1000 files	25.57	25.23	19.43
100M (single file)	0.42	1.4	0.43
1M × 1000 files	31.94	38.08	25.43
1G (single file)	7.34	14.06	3.25

The results of the tests allow us to conclude that asynchronous replication outperforms central iCAT model and synchronous replication method. The synchronous replication method introduces high latency that degrades overall system performance. On the other hand, central iCAT server model gives reasonable results as opposed to synchronous replication method; however, it threats reliability of the system since it is a single point of failure. However, these results validate that asynchronous replication method satisfies the performance requirements while it improves the reliability of the system.

3.4. *Scalability and performance benchmarking*

With improved reliability and availability of metadata server, quality of service of PetaShare has been greatly improved. However, with the exponential growth of the amount of scientific data as well as the increasingly interdisciplinary nature of modern science, scientific data management must also address the issues of cross-domain data access in a heavily data-intensive environment. In this section, we present scalability and performance benchmarking results we conducted in preparation for the development of metadata management system that can support efficient and scalable cross-domain data access.

For scalability, four rounds of tests were conducted. First round consists of attaching data object metadata to 100,000 data objects; second round consists of attaching data object metadata to 200,000 data objects; third and fourth rounds each consists of attaching data object metadata to 300,000 and 400,000 data objects, respectively. Each individual data object will be attached with an set of ten triples data object metadata, as shown in Table 5. So together, after 4 rounds of expansion, the system now contains ten millions metadata triples in total. A vastly improved scalability benchmark than our previous experiments registered [7].

For performance benchmarking, Fig. 8 contains benchmarks of five rounds of performance tests on four key metadata operations respectively. Each rounds of test consists of tests ranging in size from 1 to 10,000 data objects, since metadata attached to each data object in our benchmarking tests consist of 10 triples, the maximum number of triples benchmarked in these tests is 100,000. As illustrated in Fig. 8(a)–(c), performance of insertion, deletion and modification of data object metadata shows strong linear positive correlation to the number of triples involved. Absolute performance wise, the results indicate that it is far more costly in time to insert and modify data object metadata than delete data object metadata. It takes days to insert and modify 100,000 triples into data object metadata store while it only takes hours to delete same number of triples from data object metadata store. The performance of insertion, comparing to our previous work [7], is improved considering the size of triples in data object metadata store increases ten times while the time taken to insert similar number of triples only doubles. Performance of modification, however, significantly deteriorates comparing to previous work [7], even after considering the much more data-intensive environment, cause of performance degradation is not clear at this stage, we plan to conduct more tests in the future to understand and improve performance of modification of data object metadata.

On the other hand, performance of query of data object metadata largely remains constant as the number of triples involved increases, as shown in Fig. 8(d). In terms of absolute performance, however, query of data object metadata does not perform as well as hoped as time taken to finish a query that returns relatively small number of data objects still reaches several minutes, the relatively unsatisfactory performance of data object query is related to the size of the data set, namely, data set contains up to 1 million data objects and metadata store has up to 10 millions triples stored, in a less data-intensive environment, performance of query operation should conceivably improve, our previous work [7] indicates that query performance in a less data-intensive environment is vastly improved, nonetheless, more tests are needed to definitively prove the hypothesis.

Table 5

Selective data-object metadata

Location	DateOfcreation	Filetype	Size	Name
Institution	Creator	Resolution	Department	Project

4. Advance buffer to improve performance of data transfers

In this section, we give details of our study on performance of PetaShare client tools. As mentioned in Section 2, although, petashell and petafs client interfaces provide a convenient way for accessing to PetaShare resources, they come with extra cost in terms of performance due to overhead of I/O forwarding. Every read/write request is converted to an iRODS request and sent over the network to the server. In addition to the network latency, there is also overhead at server side in receiving and processing each I/O operation. We have observed that transferring small blocks, especially over a low-latency network, results in poor application performance. Therefore, fetching large amount of data at once has great impact on performance of data transfers. For that reason, we have

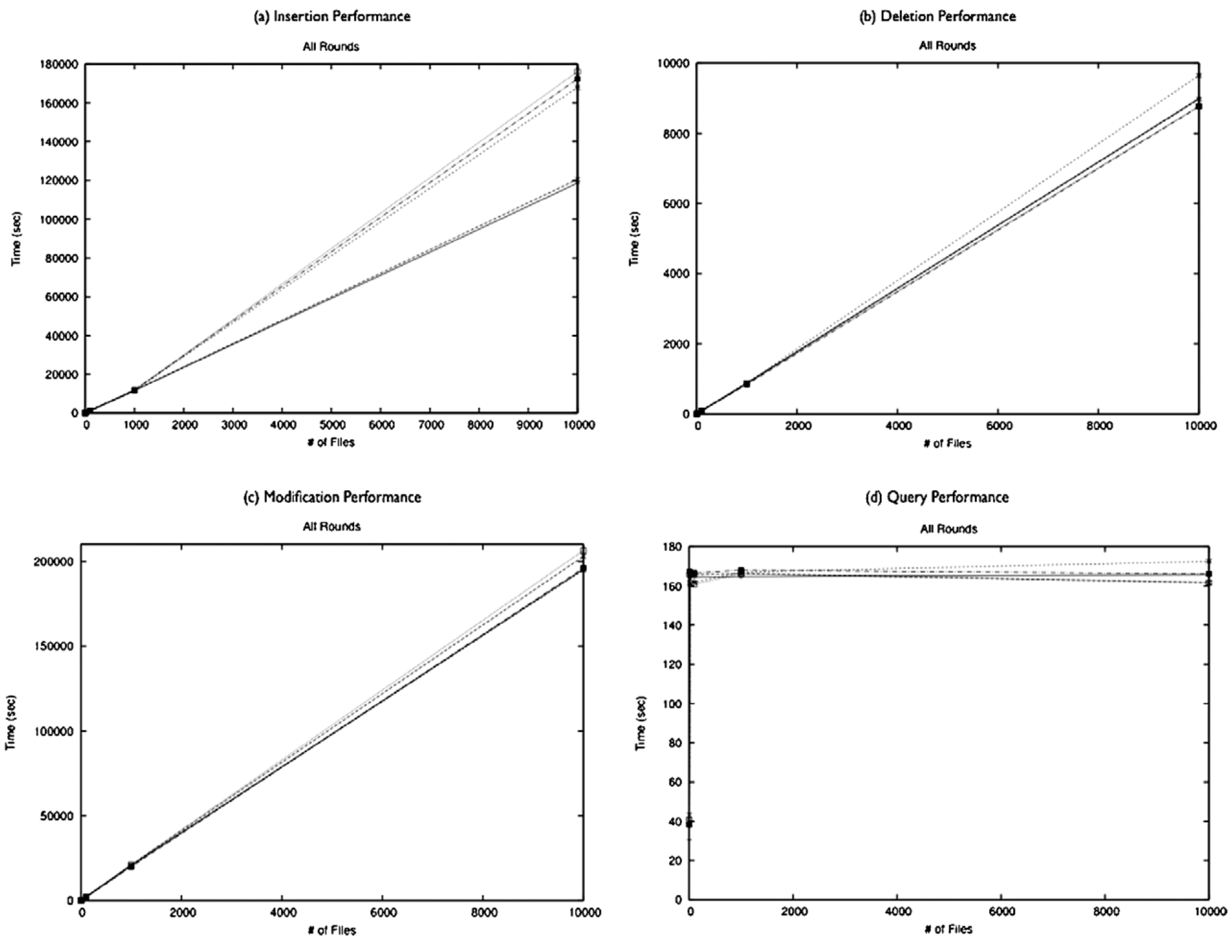

Fig. 8. Performance benchmarks for: (a) insertion; (b) deletion; (c) modification; (d) query.

optimized petafs and petashell clients by aggregating I/O requests to minimize the number of network messages.

Petafs and petashell write/read library functions accepts I/O calls for the file with offset and size as arguments. However, default block size for petafs using Unix commands is 4K for writes and 8K for reads. Thus, users would be reading and writing in small chunks of data from the server. For petashell client, writes and reads are in 64K blocks. On the other hand, pcommands use 32 MB blocks for transferring data. In Table 6, we show average results both from a client machine outside the LONI network (dsl-turtle), and a machine within the LONI network (quenbee). Those inefficient results of petafs and petashell led us to enhance the PetaShare clients for better performance in data transfers.

One option is to force users to read and write large blocks of data inside their applications. It is known that minimizing number of messages sent over the network by increasing the data size in each message will improve the overall throughput. However, standard Unix commands are highly preferred by our user community, and we would like to provide a transparent access to data resources while achieving desirable performance. Therefore, we have implemented prefetching for read operations, and caching for write operations by delaying I/O requests. There is no need to force user programs to write or read large chunks of data. Since it is done automatically, standard Unix commands such as *cp* benefit from advance buffer inside petafs and petashell clients.

The advance buffers in petafs and petashell act as a prefetch cache. Without any optimization, the offset and the size of data sent to the server, and the requested block is read and passed to the user. We keep a prefetch buffer in the client tool for each file. Instead of reading a small data chunk, we request and receive a larger block which includes the requested data and subsequent blocks of the file. Data read from the server

Table 6
Read and write performance comparison of Pcommands (pput & pget), petashell and petafs

	Read performance (Mb/s) – read to dsl-turtle from:			
	lsu	tulane	ull	uno
pget	91.61	89.43	90.14	90.85
petashell	45.31	13.63	22.02	13.99
petafs	22.35	5.44	9.06	5.33
	Write performance (Mb/s) – write from dsl-turtle to:			
	lsu	tulane	ull	uno
pput	23.66	20.62	25.29	26.55
petashell	14.74	9.46	11.55	9.04
petafs	6.48	3.60	6.22	3.61
	Read performance (Mb/s) – read to queenbee from:			
	lsu	tulane	ull	uno
pget	740.17	648.61	723.16	683.86
petashell	208.33	16.79	34.64	16.09
	Write performance (Mb/s) – write from queenbee to:			
	lsu	tulane	ull	uno
pput	460.91	645.28	658.79	653.66
petashell	253.18	24.88	50.60	23.66

is stored in a buffer assuming that subsequent calls will fall into the buffer. Therefore, there is no need to access the server for every incoming call. We process subsequent blocks from this prefetch buffer instead of requesting from the server.

Implementation details are described as follows. After receiving a read request, the client tool checks whether any prefetch buffer has been created for the file. The prefetch buffer is a contiguous sequence of data, such that we keep the beginning and the end offset of that data inside the cache. If the requested block falls into the prefetch cache, the client processes and copies data from the cache instead of requesting from the server. Otherwise, a new data chunk with the size of the buffer is read from the server starting from the beginning offset specified in the I/O call. The requested block in the I/O call is copied from the cache and returned to the user, and the rest of the data inside the buffer is kept for further requests.

The technique described above works well for sequential file accesses. If requested block size is larger than the buffer size, prefetching process is bypassed and data is directly requested from the server. For random reads and writes, advance buffer implementation might put unnecessary cost. If write calls are not coming in a contiguous order, the data inside the buffer is synchronized to the server and the advance buffer is initialized to zero. Same condition might happen for read operations, such that subsequent blocks might not fall into the buffer. However, our main focus is to enhance the sequential operations which is a common case in our system. Therefore, advance buffer implementation work over the existing mechanism for performance optimization by aggregating I/O requests.

Dealing with write operations is more complex. The client tool stores incoming write blocks instead of sending each write request to the server, such that we delay I/O calls and combine requests to minimize the number of messages over sent over the network. For each file, we keep a write buffer which is separate from the prefetch buffer used for read operations. There can be only one active buffer at a time; the client uses advance buffer either for sequential read operations or sequential write operations. For an incoming write I/O call, the client first checks if there is any active buffer. If so, the block to be written is appended and information about the buffer is updated. If there is no space in the buffer, data inside the buffer is first sent to the server as a write request. Later, we form a new cache buffer and store the blocks provided by the following I/O calls. Before delaying any write operation and storing data inside the buffer, we ensure that it is a subsequent request by controlling the offset and the size. If not, the process starts from beginning and the buffer is flushed to the server. Besides, if we receive a read request, we make sure the client writes data from the buffer to the server and deactivate the write buffer. Same happens when the file is closed, such that the buffer is synchronized before closing the file.

We emphasize that advance buffer implementation is basically for sequential reads and writes. One important problem is data consistency that may happen due to delaying I/O operations. We try to ensure data consistency by controlling beginning offsets before fetching data from the buffer. First, we make sure that read requests are coming in a sequential order. Later, we maintain two separate read and write modes for the buffers. As an example, the prefetch buffer will be deactivated when the client receives a write request. So, the following I/O calls will be forced to make another request from the server. Same situation happens for write operations. The advance buffer for write operations will be deactivated and synchronized to the server whenever a read request is received. We ensure that incoming write calls and also read calls are coming in a sequential order.

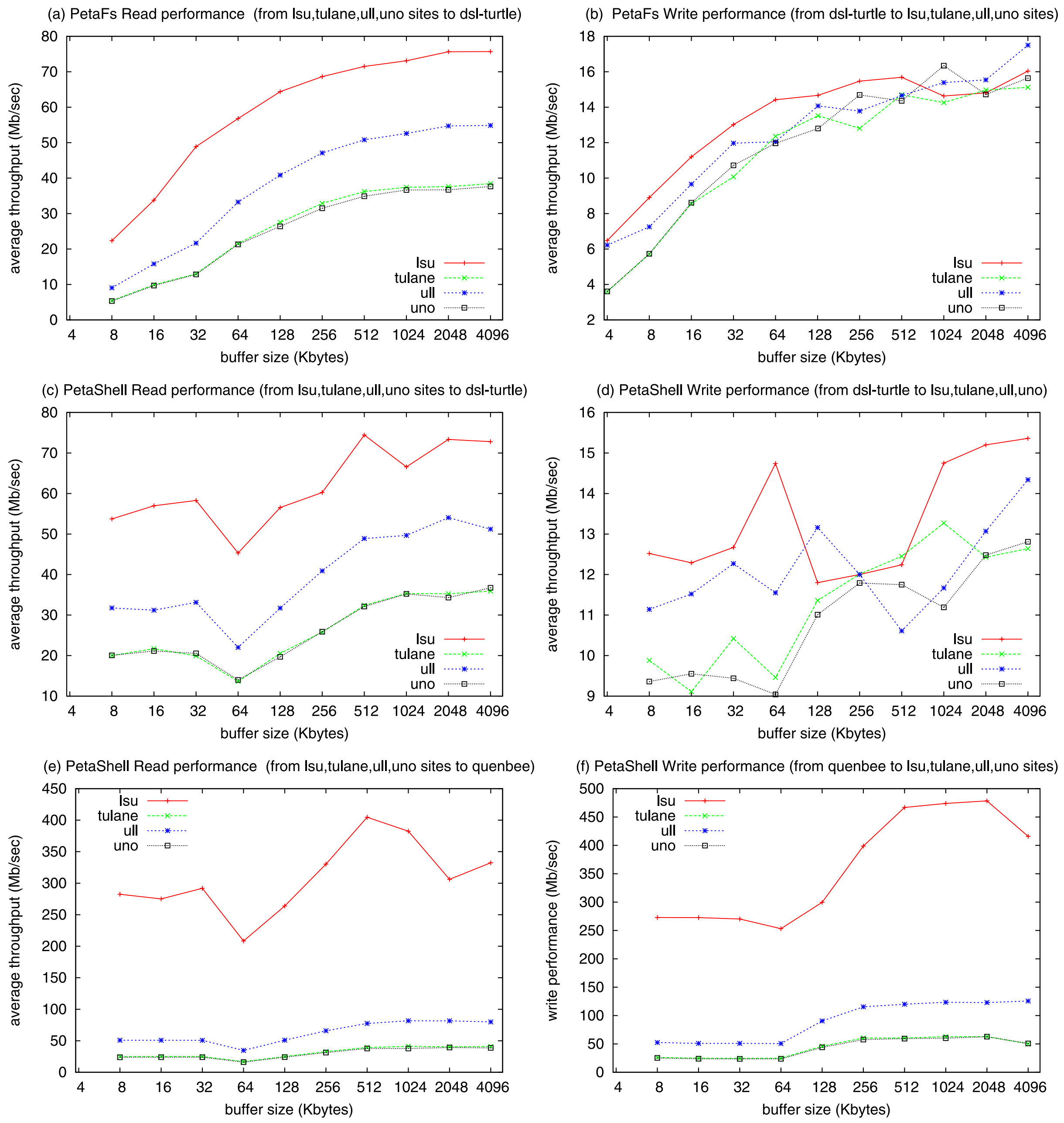

Fig. 9. Performance of petafs and petashell with advanced buffer. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0317.)

In our experiment testbed, there are 4 major remote sites and the metadata database is on *lsu* site. We have experimented data transfer performance from two different client machines with different access patterns to PetaShare sites. *Dsl-turtle* is outside of the LONI network and it has slow access to 4 PetaShare sites. *Queenbee* is inside the LONI network and it has much faster access to all of those 4 sites. Results are average values of 3–5 separate runs. We have used *cp* command and collected average throughput of 3–5 separate runs for copying 1, 10 and 100 MB files. The x-axis (buffer size) in Fig. 9 is in log scale. We forced client tools to use a fixed data chunk size for each network I/O call, such that I/O requests to the server are rearranged to fit into the buffer size. As can be seen in Fig. 9, we use large buffer size, we minimize the number of network I/O calls to the server, and thus increase the performance. Especially, we see better improvement in read operation as in Fig. 9(a) and (c). Petashell puts extra costs by making many systems calls for tracing I/O calls. Thus, we see a lot of fluctuation in Fig. 9(d). On the other hand, petashell client makes extra connection caching which was not available in petafs at the time when we performed those experiments.

The advance buffer simply aggregates I/O calls using a simple logic to improve performance of sequential operations. The size of the advance buffer both in petafs and petashell can be set by user as a command argument to PetaShare client tools.

5. Conclusion

In this paper, we have presented the design and implementation of a reliable and efficient distributed data storage system, PetaShare, which spans multiple institutions across the state of Louisiana. PetaShare provides an asynchronously replicated multi-master metadata system for enhanced reliability and availability, and an advanced buffering system for improved data transfer performance. Our results show that our asynchronous multi-master replication method can achieve both high performance, reliability and availability at the same time. We gave a brief overview of the benchmarking tests we did for key metadata operations. We have also presented the design and implementation of the advanced buffer system for improved data transfer performance. For future work, we plan to improve conflict resolver in asynchronous replication to ensure stability of our production PetaShare system. We also plan to enhance the advance buffer implementation by making buffer size dynamic, such that buffer size will increased by adapting to the frequency of incoming I/O operations.

Acknowledgements

This project is in part sponsored by NSF under award numbers CNS-0619843 (PetaShare), CNS-0846052 (CAREER) and OCI-0926701 (Stork). This work benefits from the use of resources of the Louisiana Optical Network Initiative (LONI). We also would like to thank iRODS and Parrot teams for their contributions.

References

[1] G. Allen, C. MacMahon, E. Seidel and T. Tierney, Loni: Louisiana optical network initiative, 2003, available at: http://www.cct.lsu.edu/~gallen/Reports/LONIConceptPaper.pdf.

[2] M. Balman and T. Kosar, Dynamic adaptation of parallelism level in data transfer scheduling, in: *International Workshop on Adaptive Systems in Heterogeneous Environments (ASHEs 2009)*, Fukuoka, Japan, 2009.

[3] M. Balman and T. Kosar, Early error detection and classification in data transfer scheduling, in: *Proceedings of International Workshop on P2P, Parallel, Grid and Internet Computing (3PGIC-2009)*, Fukuoka, Japan, 2009.

[4] P.H. Carns, W.B. Ligon III, R.B. Ross and R. Thakur, PVFS: a parallel file system for linux clusters, in: *ALS'00: Proceedings of the 4th Annual Linux Showcase and Conference*, USENIX Association, Berkeley, CA, USA, 2000, p. 28, available at: http://portal.acm.org/citation.cfm? id=1268407.

[5] P.H. Carns, B.W. Settlemyer, I. Walter and B. Ligon, Using server-to-server communication in parallel file systems to simplify consistency and improve performance, in: *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, IEEE Press, Piscataway, NJ, USA, 2008, pp. 1–8.

[6] Fuse: Filesystem in userspace, available at: http://fuse.sourceforge.net.

[7] D. Huang, X. Wang, G. Allen and T. Kosar, Semantic enabled metadata framework for data grids, in: *CISIS'08: Proceedings of the 2008 International Conference on Complex, Intelligent and Software Intensive Systems*, IEEE Computer Society, Washington, DC, USA, 2008, pp. 381–386.

[8] B. Keating, Challenges involved in multimaster replication, 2001, available at: http://www.dbspecialists.com/files/presentations/mm_replication.html.

[9] T. Kosar and M. Balman, A new paradigm: Data-aware scheduling in grid computing, *Future Generation Computer Systems* **25**(4) (2009), 406–413.

[10] Loni: Louisiana optical network initiative, http://www.loni.org.

[11] S.K. Madria, Timestamp-based approach for the detection and resolution of mutual conflicts in distributed systems, in: *DEXA'97: Proceedings of the 8th International Workshop on Database and Expert Systems Applications*, IEEE Computer Society, Washington, DC, USA, 1997, p. 692.

[12] A. Nisar, W.K. Liao and A. Choudhary, Scaling parallel I/O performance through I/O delegate and caching system, in: *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, IEEE Press, Piscataway, NJ, USA, 2008, pp. 1–12, available at: http://dx.doi.org/10.1145/1413370.1413380.

[13] Petashare, http://www.petashare.org.

[14] A. Rajasekar, M. Wan, R. Moore and W. Schroeder, A prototype rule-based distributed data management system, in: *HPDC Workshop on Next Generation Distributed Data Management*, Paris, France, May 2006.

[15] F. Schmuck and R. 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*, USENIX Association, Berkeley, CA, USA, 2002, available at: http://portal.acm.org/citation.cfm?id=1083349.

[16] H. Tang, A. Gulbeden, J. Zhou, W. Strathearn, T. Yang and L. Chu, The panasas activescale storage cluster – deliver-

ing scalable high bandwidth storage, in: *Proceedings of the ACM/IEEE SC2004 Conference*, 2004, available at: http://dx.doi.org/10.1109/SC.2004.57.

[17] D. Thain and M. Livny, Parrot: Transparent user-level middleware for data-intensive computing, in: *Workshop on Adaptive Grid Middleware*, 2003.

[18] D. Thain and M. Livny, Parrot: an application environment for dataintensive computing, *Scalable Computing: Practice and Experience* **6**(3) (2005), 9–18.

[19] D. Thain and C. Moretti, Efficient access to many small files in a filesystem for grid computing, in: *2007 8th IEEE/ACM International Conference on Grid Computing*, September 2007, pp. 243–250.

[20] S. Weil, S.A. Brandt, E.L. Miller, D.D.E. Long and C. Maltzahn, Ceph: A scalable, high-performance distributed file system, in: *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI'06)*, Vol. 7, November 2006, USENIX, available at: http://www.ssrc.ucsc.edu/proj/ceph.html.

[21] S.A. Weil, K.T. Pollack, S.A. Brandt and E.L. Miller, Dynamic metadata management for petabyte-scale file systems, in: *SC'04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, IEEE Computer Society, Washington, DC, USA, 2004, p. 4.

[22] A. Weise, M. Wan, W. Schroeder and A. 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, Springer-Verlag, Berlin/Heidelberg, 2008, pp. 321–330.

[23] W. Yu, R. Noronha, S. Liang and D.K. Panda, Benefits of high speed interconnects to cluster file systems: a case study with lustre, in: *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006, available at: http://dx.doi.org/10.1109/IPDPS.2006.1639564.