A Distributed Persistent Object Store for Scalable Service

Chao Jin, Weimin Zheng, Feng Zhou, Yinghui Wu
jinchao99@mails.tsinghua.edu.cn

Institute of High Performance Computing Technology
Dept. of Computer Science and Technology, Tsinghua University
100084, Beijing, PR China

Abstract

This paper presents a distributed persistent object store designed to simplify scalable service in cluster environment. This distributed object store, called TODS (Tsinghua Object Data Store), presents a single-imaged, transparent persistent and object-oriented view of the storage devices of the whole cluster. TODS is designed to be incremental scalable and efficient, and also has the properties of the high concurrency, high throughput and availability which are necessary for scalable service. TODS supports distributed ACID transactions within the cluster, which qualifies its use in the building of complex transactional services. And the user interface of TODS is fitter for building service than that of file system, and significantly easier to use than that of RDBMS. TODS is a reusable platform for scalable service in cluster by forming many general data management functions into one independent layer. This paper gives the motivation, principle and architecture of TODS. Some technique details are also discussed. In our performance experiments, the system scales smoothly to a 36-node server cluster and achieves 11,160 In-memory reads/sec and 396 transactions/sec.

1. Introduction

Because of its scalability, high availability and cost-effective, clusters have been thought to be natural-platform for network service [1}. Storage system solution used in cluster environment can mainly be divided into two categories: file system and RDBMS. However for scalable service build on cluster environment, these two methods respectively have their disadvantages. File system provides directly supports for stream interface, and this is very inconvenient for structured or semi-structured data structure of network service, especially Internet service. And file system lacks import facilities like transaction, data recovery and query etc. Many distributed file system can liberate the programmers from complex distributed data management, but the prevailing distributed file system such as NFS and AFS, were designed for WAN and could not sufficiently utilize the full feature of modern clusters. While RDBMS provides transaction and complete ACID supports, its high reliability and durability and the overhead of SQL make it not fit for network services which needs high performance and high availability at first. Although distributed or parallel database possess scalability in some degree, its expensive cost make them not acceptable by general customs.

As illustrated above, up to now there is no storage systems fit for distributed persistent management needed by scalable service in cluster environment. Many network services have to implement these properties besides their service logic, for example Porcupine [3]. Although this method can solve the problem, it also aggravates the burden of programmers and results in the code hard to maintain and evolve. This paper presents a distributed persistent object store ——TODS (Tsinghua Object Data Store) to simplify scalable network service. It is a reusable separate platform to abstract the distributed data management from the service logic. The goal of TODS is to simplify the construction of Internet service in cluster.

First, TODS provides transparent persistent object access support, this object-based interface is higher-level than file system, and fitter for network service development. Is behaves as a scalable and fault-tolerance object store with transaction support. And the whole storage space of cluster is single image to users. The transparent persistent interface in Java language is compatible with JDO [4]. User applications written in Java can transparently access objects stored in a TODS system, which means that application developers are completely freed from writing code to make his data objects persistent. Objects are automatically fetched from store when they are accessed, and modified objects are automatically written back.

Second, TODS utilizes cluster architecture to provide increment scalability. Distributed persistent data management, fault-tolerance and crash recovery are maintained by TODS. As a light weighted scalable object store, the design of TODS keeps the high concurrency, scalable throughput and availability in mind. The design of TODS also considers the properties of clusters such as high speed, low latency interconnection and incremental scalability. Service built on TODS naturally inherits these properties and becomes scalable too. Compared with relational table of RDBMS, the granularity of object makes TODS higher concurrency; without SQL interface makes TODS more parallel than RDBMS.

The remaining of the paper is organized as follows; Section 2 of this paper presents the overview of TODS. Section 3 presents design principles, architecture and implementation of TODS. Section 4 presents performance results. Section 5 describes the related work and section 6 concludes this paper.

2. TODS Overview

A TODS system is a self-contained data management layer running on a server cluster to handle storage requests of network services running on the same cluster. TODS provides a repository of persistent objects. Each object in the repository has a unique identity, a type and value. The information of type includes each field of this class, which is maintained by TODS. TODSLib is the interface of TODS, which is a binary library. Service processes use this library to map the API calls to messages sent to TODS server side, which is a collection of cooperating server. The architecture of TODS server is peer to peer. Each of the server sites provide the clients same image of data stored in TODS. Currently a Java version of TODSLib is implemented, which is compatible with JDO [4], the emerging standard of SUN. Below the server side is the storage brick, which are single-node transactional data stores. They manage local data in a key-value fashion, and provide the function of data filtering to assist query functionalities of server side.

The interface of TODS is very convenient to the users, they only need to link the TODSLib and call the responding API to make the object persistent, transaction etc. Binary code of the user class is modified before loading to implement the mechanism which works with other parts of TODSLib to track the state of persistent objects. In current Java prototype of TODS, this mechanism is a specific interface——`javax.jdo.PersistenceCapable`, which defined by JDO.

Objects managed by TODS are put into name spaces known as **Object Spaces**. Each object space has its own set of class hierarchy and objects. An Object Space is analogous to a database or a table space in RDBMS, or a directory in file systems. The list of all Object Spaces, class meta-data and permission rules of each Object Space are all maintained by the Meta-Server. Data of an Object Space are stored on a subset of all the Bricks, whose list is managed by the Meta-Server. However, a Brick may be well shared by multiple Object Spaces.

3. Architecture and Implementation of TODS

3.1 Architecture of TODS

Figure 1 illustrates the process architecture of TODS. TODS executes as two group of communicating process: peer-server processed and store-brick process. As the assumptions of cluster environment, problems about security are not taken into account.

Each peer-server plays several roles. First, it acts as an agent for application processes which can be local or from local network. When an application needs object, TODSLib automatic fetch it through socket request from peer-server, which fetches the object from other peer-server site or from the store-brick server. Second, it is an object-cache manager. The granularity of the cache is object, and all the peer-servers cooperate to provide a global object-cache. Third, the peer-server is responsible for concurrency control and crash recovery. The peer-server is also responsible for distributed transaction. Considering the low latency of LAN, we use the two-phase commit protocol to implement the distributed transaction. Fourth, the distributed query on object data is done on peer-servers, which utilize the pattern match function of brick to implement this work.

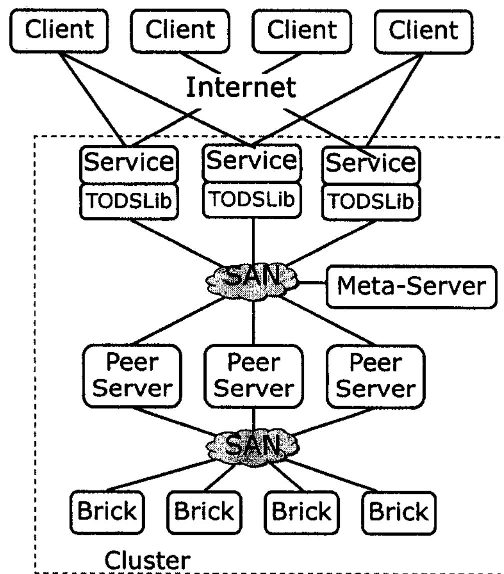


Figure 1 TODS Architecture

Bricks are single-node transactional data stores. They manage local data in a key-value fashion, and provide local transaction and primitive query functionalities.

The meta-server is responsible for meta-data management and system configuration. Meta-server is duplicated for high availability. System configuration includes location (IP, port) and parameters of all components such as Peer Servers and Store-Bricks. This ensures centralized management of the whole system. Meta-data includes information about name space structure, users and group information, persistent class list and detail field information for each user class.

This process architecture provides a great deal of flexibility. Using more than one peer server can prevent Peer-Server from becoming performance bottlenecks, which is often the case in classic client-server systems. Also, this method is able to increase availability. The client library can transparently switch to another Peer-Server in case one fails. If the service instance and Peer-Server are running on the same node, which is often true, they can communicate with some relative faster means than network, such as a shared memory block.

3.2 TODS Software Components

TODSLib

When the application attempts to reference objects which are not present in the client object-cache, TODSLib sends "fetch object" request by socket message to connected server, which fetches the necessary objects from other peer server or from store-brick (the requested object not in the global server cache). When a transient object is made persistent, all objects reachable from it are also made persistent (persistence by reachability).

TODS provides several ways to retrieve objects from the store. The simplest way is to access referenced objects from existing ones. Queries cab be used to get the first group of objects. TODS implements the simple query interface defined by JDO. A query is defined by a query filter against a collection of candidate objects of a certain class, with other elements such as parameters and unbound variables.

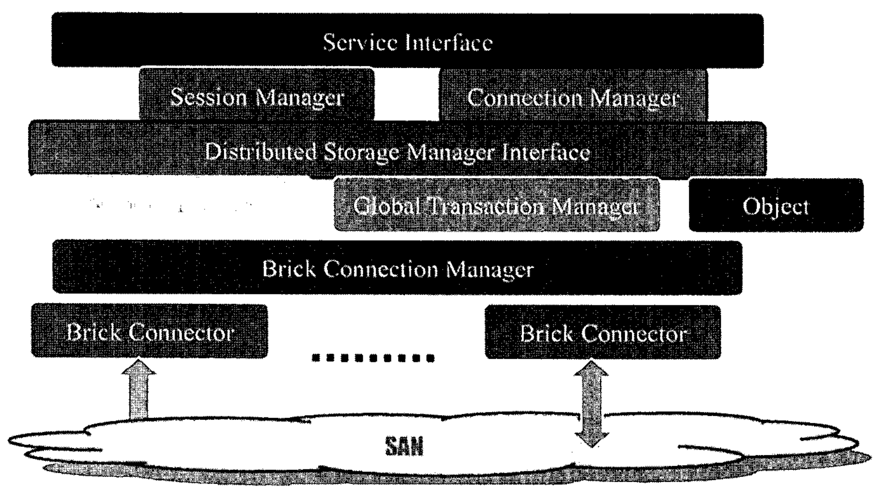


Figure2 Peer-Server Components

Peer-Server

Figure 2 illustrates the internal structure of the TODS peer-server in more detail. It is divided into two main components: a Server Interface which communicates with the applications, and the Distributed Store Manager which manages the persistent object data. The Server Interface is an agent responsible to meet the requests from the clients. When an application connects with the peer-server, the server associates session state (User information and Object Space information) with this connection. User information is checked against unauthorized access.

Below the Server Interface is the Distributed Store Manager. As shown in Figure 2, the Store Manager is responsible for global-cache manager, distributed transaction and query request. It manipulates all the Store-Brick into a single image system.

Bricks

Brick is currently implemented upon the Berkeley DB [10] library. With advanced features such as XA transaction support and replication, and a long evolving history, Berkeley DB provides a very stable foundation for our work. Peer-Servers and Bricks also manage directory-based caches to improve non-transactional data access performance. Peer-Servers cache objects accessed frequently in memory. Each Brick maintains a directory about which Peer-Server currently cache which objects, and invalidates data on corresponding Peer-Server when certain objects are updated.

3.3 Data Model

Objects managed by TODS are put into name spaces known as **Object Spaces**. Object (or persistent object) is the granularity of most operations in TODS. Every persistent object is associated with a globally unique id (OID). An object has a number of value fields and can reference other objects. Figure 3 illustrates an example of object references. **A** and **B** are active persistent objects, with A referring to an inactive one, **C**. **D** and **E** are transient objects not currently managed by TODSLib. However, they will become persistent by a *make persistent* call to TODSLib.

When a transient object is made persistent, all objects reachable from it are also made persistent (persistence by reachability). Persistent objects are long-lived and independent of life-cycles of the service instances or TODS runtime. Any modification to the object will be written to the store implicitly at some time (e.g., when transaction is committed). Persistent objects are loaded into memory automatically when needed.

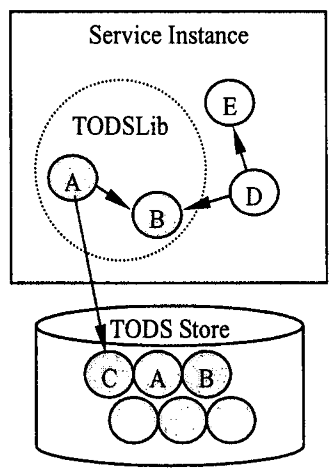


Figure 3 Objects references

There are several ways to retrieve objects from the store. The simplest way is to access referenced objects from existing ones, e.g., accessing employee objects from the containing department object. It works as it should. Queries or class extents are used to get the first group of objects. TODS implements the simple query interface defined by JDO. A query is defined by a query filter against a collection of candidate objects of a certain class, with other elements such as parameters and unbound variables. Class extent is actually a degenerated query, which lists all objects of a certain class. Apparently it provides better performance for enumerating operations compared with query.

An Object ID is a 128bit integer, whose structure is shown in figure 4. Object Space ID (OSID) indicates which Object Space this object belongs to. Class ID (CLID) references to class definition in the Meta-Server, it is assigned when first object of its class is inserted into the system. Node ID (NID) denotes the location of the object, while Serial Number is the local ID of the object.

One thing to notice is that the OID is a *physical* ID, in the sense that it indicates on which node the object is located. The system can directly find the object just by the OID. This contrasts to the alternative approach of using a logical object ID or "path" and thus needs to look up the real location of objects before accessing them, which introduces more overhead and the problem of effectively and coherently caching the lookups. Logical ID or text path are often introduced for user friendliness and location transparency. The former is *not* a problem in TODS because TODSLib completely hides from users the details of fetching and storing persistence objects. OIDs are not even seen by them. The latter reason is most justified for wide-area distributed systems, where nodes and network failures and changes are common. As TODS is designed for well managed cluster environment, it is found that an OID with more information greatly simplifies system design and improves performance.

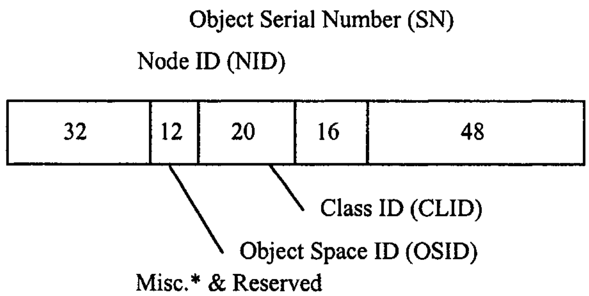


Figure 5 128bit OID format

* E.g. One bit used to indicate temporary OIDs

3.4 Object State Maintaining

Every persistent object is stored within Bricks as a key-value pair, with its OID as the key and content as the value. The value part is divided into two parts, a header and field values. The header is an index to every field, which is of variable length, for fast lookup. Primitive field types such as numbers, strings and dates are embedded in the data block. Certain collection types such as lists, sets and hash tables which are supported by standard Java library are also embedded. Other user-defined persistent objects referenced by this object are represented by its OID.

Class definitions and hierarchy of all user classes in each Object Space are maintained by the Meta-Server. The record of each class contains the names and types of all persistent fields, plus the Class ID of the parent class, if there is one. The class hierarchy is built from all the records at start-up time of the Meta-Server and maintained in memory. It is used mostly to support class extents (enumeration of all objects of the same class), which should sometimes return all objects of the class and its descendent classes.

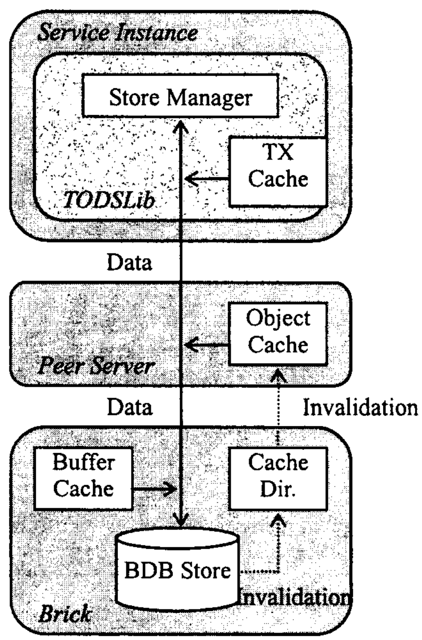
During operations like object storing and fetching, the composing and decomposing of data blocks are done by TODSLib, where Peer Servers and Bricks do not care about the structure of data blocks. This makes the system behave like a scalable distributed hash table [5]. For other operations like queries, Peer Servers and Bricks access the internals of the data block. Bricks only do simple and fast filtering of objects with the help of the header. Peer Servers access fields with the class meta-data they get from the Meta-Server. We found this layered data access approach a good trade-off between flexibility and efficiency. Every component in the system gets reasonable knowledge of structures of data and does no redundant work like repetitively packaging and un-packaging an object.

Inside user service processes, read and write to any persistent data object should be tracked. As mentioned in 4.2, these are done by online addition to the binary code of user objects. Although there are lots of details under the scene, the general idea is clear. Binary code of the user class is modified before loading to implement a specific interface `javax.jdo.PersistenceCapable`, which works with other parts of TODSLib to track the state of persistent objects. Many of user operations finally translate into command messages being sent to Peer Servers and result messages back from them, all without a single line of hand-written code by user.

TODSLib maintains *soft state* with the Peer Server it connects to. Knowledge about meta-data and persistent objects are often cached by TODSLib. But if the Peer Server fails, TODSLib simply fails all the active transactions and connect to another Peer Server. The service will continue to run, although some users may need to retry the failed transactions.

3.5 Cache

Caches are maintained by components of TODS to improve performance. Figure 5 shows how they interact.



The Transaction Cache (*TX Cache* in the figure) caches objects during the process of a transaction. It improves transactional access performance by avoiding repetitive fetching the same object within a single transaction. The cache is emptied when the transaction is finished. Subsequent read requests will go directly to the Bricks.

The Object Cache on each Peer Server serves to improve non-transactional performance. Transactional operations just ignore this cache. It is essentially a hash table with size constraint and LRU replacement policy, mapping OIDs to object data blocks. A non-transactional *read* will first look at this cache. If a matching record is found, a network message exchange with the Brick and a disk read will be saved. Due to the fact that service instances and their Peer Servers are often on the same node, the overhead of a *read*

master-slave structure. Each update to its data is synchronous distributed from the master to all the slaves. In case that the master fails, a new master will be elected immediately to take the place. And the failed Meta-Server simply joins the replication group again after it is repaired. Its data store is then updated to current state before put into work again.

TODS managed to recover distributed transactions mainly with the help of persistent transactions state Peer Servers maintain. It contains the states of each distributed transaction and its corresponding local transactions. When one brick server fails, during its restarting process, it will detect whether there are unresolved distributed transactions. If there is any, it will notify the related Peer Servers with broadcasting. These Peer Servers in turn send *commit* or *rollback* commands to the Brick regarding these transactions and finally put the Brick back to normal operations.

Another simpler case is Peer Server failures. When a failed Peer Server restarts, it checks its transaction state store and re-commits all distributed transactions in *committing* state and rolls back all other transactions before that state.

3.7 Query

The simplest form of a query is accessing a class extent. Every persistent object in a Brick has its CLID as a secondary index in the underlying Berkeley DB table. So it is straightforward to iterate over all objects of a certain class by look them up via the CLID of the class. This is done by using a cursor supported by Berkeley DB.

A real query contains a candidate extent or collection, a Boolean filter string, parameters and unbound variables. The query interface is more programmatic where users call methods to define each of the above elements, rather than declarative where user use a string to specify all these elements which is the case with SQL. This apparently saves some parsing overhead. The query is first translated into a `TODSQuery` object. If the query is simply against a collection of in-memory objects, it is immediately done by TODSLib, using a simple iteration method, within the service process. For most cases the query will be against a class extent residing on the Peer Server. Then the `TODSQuery` object is sent to the Peer Server for execution. At the Peer Server, the query is decomposed into two parts, one is simple filters like *the second field equals "Mary"*, and the other is the remaining more complex filters such as navigations (filters regarding referenced objects). The former is sent to the related Bricks, which in turn return satisfying objects. The later set of filters is then applied on them by the Peer Server to get the final result set.

Currently Bricks do not maintain indexes. Thus all queries are executed with linear scanning. Introduction of indexing support is rather straightforward and will probably be done in later versions of TODS.

4 Performance

Performance experiment results are presented in this section. Our test environment is a 36-node server cluster. Each node is equipped with 4 Intel Pentium III Xeon processors at 750 Mhz, 1 GB of RAM and a 36 GB 10000 RPM SCSI disk. The network is 100M fast Ethernet. All nodes run Redhat Linux 7.2 with stock 2.4.7 enterprise kernel. TODS and test programs were run with Sun JDK 1.4.0-b92 for x86 Linux. All tests had a warm-up period of 1 minute and test period of 5 minutes. Each test was run 3 times and averages were taken.

In-Memory Reads

In this test, a special version of Brick is used, which keep all data in a simple in-memory hash table, instead of Berkeley DB tables. The test is designed to measure the communication overhead of the system and maximum scalability without considering disk I/O. We ran one copy of Brick, Peer Server and the test program on each of the nodes. All test programs connect to local Peer Servers and each Peer Server connect to *all* Bricks by round-robin. The object size is about 1KB. Results are shown in figure 9. The results show that the system is nearly linear scalable, with max throughput for our 36 cluster as 11,160 reads/sec.

Another test is performed to measure performance of reading objects of different sizes (1KB – 128KB). All 36 nodes were used in this test. The results are shown in figure 10 and 11. Briefly, for 1KB objects 11,160 reads can be done a second with payload bandwidth of 11MB/s, and for 128KB objects, 1,332 reads with payload bandwidth of 170MB/s. This ideal performance is adequate for any conceivable Internet services apart from media and file services.

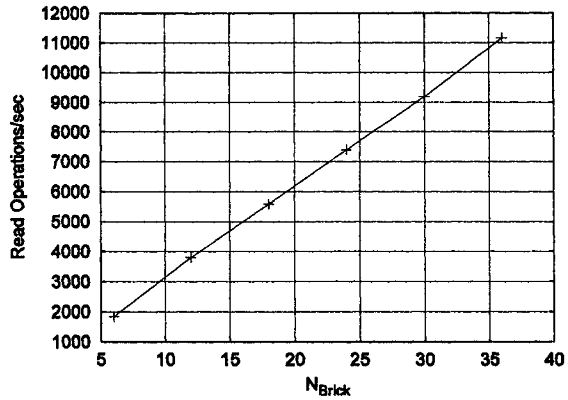


Figure 6 In-memory Read Scalability

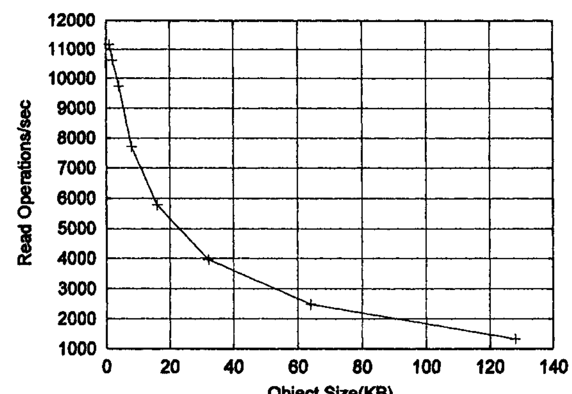


Figure 7 In-memory Read Throughput

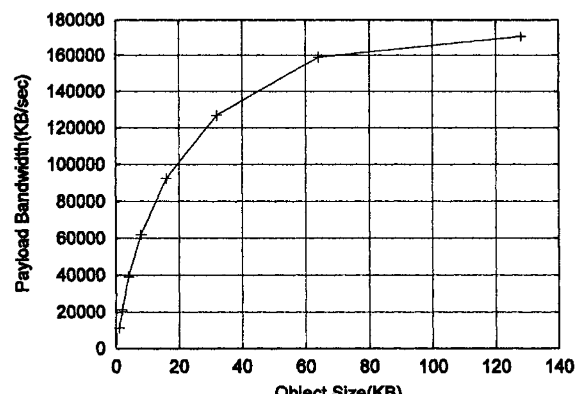


Figure 8 In-memory Read Payload

On-Disk Reads

This test is closer to actual operational environment than the first one. To approximate real-world workload, we first populated each of the Bricks with 5000 objects of the object length being tested. Then we access these objects randomly by Object ID we gather when inserting them. Although random access is not a good "real-world" pattern, it effectively shows the bottom-line performance we should expect. The Object Caches in Peer Servers are turned off to show raw Brick read performance. In figure

12 and 13, the system generate throughput at about 1/3 of the In-Memory throughput. It completes as many as 2740 reads of 1KB object in a second. As object size increases, payload bandwidth increases quickly, to 46MB/s when object size is 128KB. This throughput result is satisfying. Since actual work-load usually has good locality, the efficiency of the Buffer Cache will be much better, thus overall throughput higher.

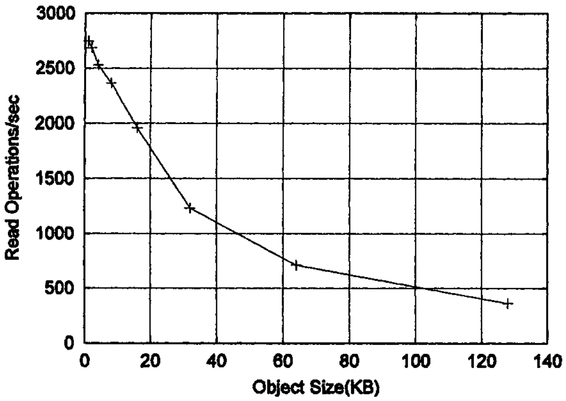


Figure 9 On-disk Read Throughput

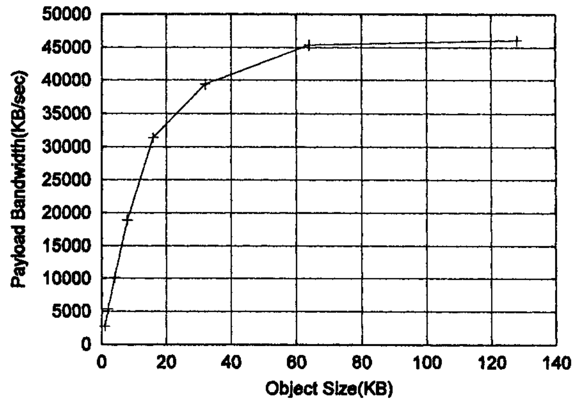


Figure 10 On-disk Read Payload

Cache-Hit Reads

To test the effectiveness of Object Caches in Peer Servers, we modified the Peer Server to report all read requests as hitting the cache. With the same configuration as the previous tests, we get the results shown in figure 14 and 15. For 1KB objects 13,392 reads per second with payload bandwidth of 13MB/s, and for 128KB objects, 1,566 reads with payload bandwidth of 200MB/s. These figures are about 20% more than those in the in-memory test and 4 times of those in the on-disk test. This shows that hitting the Object Cache gains substantial throughput for reduced network overhead and disk I/O.

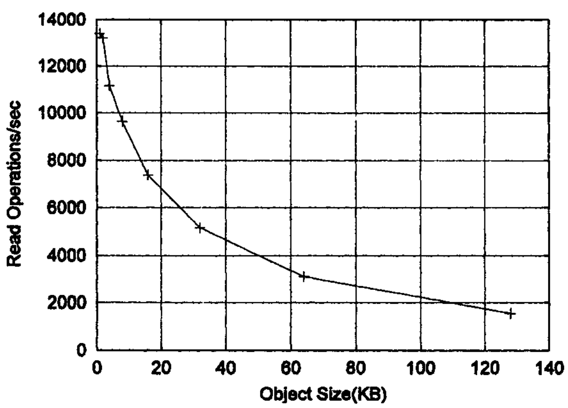


Figure 11 Cache-hit Read Throughput

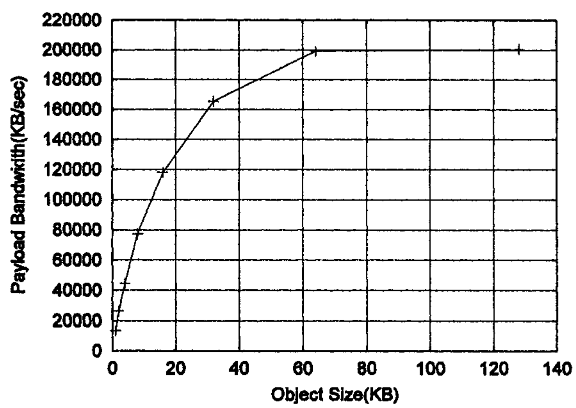


Figure 12 Cache-hit Read Payload

Transactional Writes

Transaction performance is directly tied to disk write performance because they include synchronous writes to the log file. Here we test inserting objects into Bricks by transactions. In each transaction, we insert four objects that are about 2K in size totally. These transactions are done locally on Bricks because as we mentioned above, TODS prefers to do local transactions whenever possible. From the results shown in figure 15, we can see that the transaction performance grows linearly with Brick number, just as we expected. When all 36 Bricks participate in, 396 transactions can be done in a second.

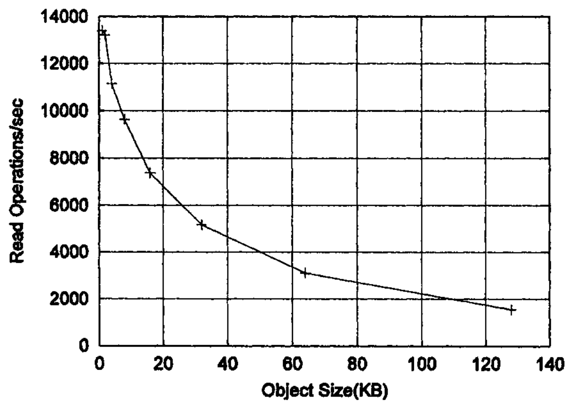


Figure 11 Cache-hit Read Throughput

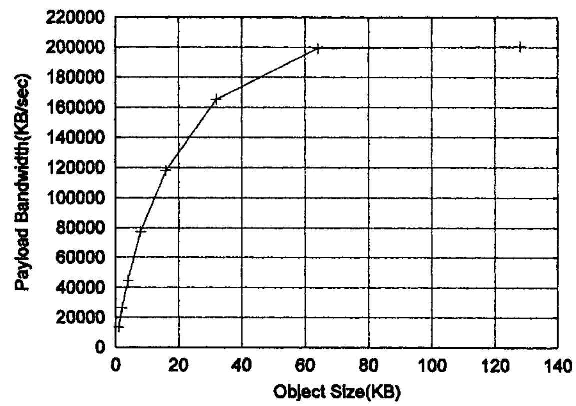


Figure 12 Cache-hit Read Payload

Distributed Transactions

Distributed transactions incur additional overhead of network communication and persistent transaction state maintaining. The Peer Server is modified to put subsequent inserted objects into different Bricks, thus all transaction inserting more than one object becomes distributed. We rerun the last test with a different configuration – as Bricks are added in, a Peer Server is also run on the nodes, and multiple copies of the test program is run simultaneously. This avoids that Peer Servers maintaining transaction state become bottlenecks. The results are shown in figure 16. The performance is roughly 1/3 of that of local transactions.

5. Related works

TODS is a novel distributed persistent object system that inherits many ideas from previous research. This section compares TODS with these related works.

DDS (Distributed Data Structure) [5] is persistent data management layer in cluster environment. DDS presents a single site data structure interface to clients, but partitions and replications the data across a cluster. The DDS design focuses on availability, performance and scalability issues and is however rather simplified on issues about data model and consistency. The most useful data structure it supports is a distributed hash table. DDS provides atomic operation, but without transaction.

SHORE (Scalable Heterogeneous Object REpository) [7] is also a persistent object store system built to support large applications such as Geographic Information Systems and satellite data repositories. SHORE represents a merger of object data base and file system technologies. By a symmetric peer-to-peer architecture, SHORE provides good scalability and availability. Access to persistent objects in Shore is not transparent. Users have to explicitly retrieve every object they use and have to call a method to notify the update of an object. Second, Shore has its own data definition language (SDL) and statically compiles SDL files to import user classes, while TODS uses binary code processing to extract meta-data from user classes, so users never have to maintain two copies of class definition.

THOR [13] is also a persistent object store which supports atomic transaction. But the aim of THOR is achieve good performance in a wide-area distributed environment. Unlike clusters, wide-area systems must deal with heterogeneous, network partitions, untrusted peers, high latency and low throughput network. Because of these differences, THOR has relaxed consistency semantics and low update rates.

Active Disk [8] is also a method to take advantage of processing power on individual disk drives and storage parallelism to reduce network traffic. Request and corresponding application-level code are sent to Active Disk, so the Disk can execute the code on the requested data. Active Disk is more similar to a low-end computer than a disk device. Compared with Active Disk, the object interface of intelligent disk device in TODS makes it more scalable and across platform. Weaker ability of intelligent disk makes it implements more realism.

6. Conclusion

In this paper, we present the design and implementation of TODS, a distributed persistent object storage platform specifically designed for scalable services. It is designed with the requirements of scalable services in mind and appeals to many advantageous properties of modern server clusters. A decentralized architecture is used which proves to be very scalable and efficient. The user interface of TODS implements transparent persistence which relieves developers completely from writing I/O code. Different levels of consistency are supported that makes TODS suits the requirements of different services.

There are many issues not addresses in current version of TODS. Large block of data (like a PDF document, a TAR archive) are not handled efficiently. A stream oriented access method (like that of files) is more desirable. Garbage collection of persistent objects is not implemented. Thus users have to explicitly delete useless objects. Schema evolution is also not supported now, which is very important for long-run data management systems. Other interesting topics include implement more *transports* such as one based on VIA [12] and one based on local shared memory messaging, and making the Object Caches inside difference Peer Servers cooperate to further improve hit rate, i.e. one Peer Server accessing another one's cache.

References

[1] Armando Fox and Steven D. Gribble and Yatin Chawathe and Eric A. Brewer and Paul Gauthier. Cluster-Based Scalable Network Services, In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, 1997

[2] Mitch Wagner, Google Bets The Farm On Linux, available at *http://www.internetweek.com/lead/lead060100.htm*

[3] Yasushi Saito and Brian N. Bershad and Henry M. Levy. Manageability, Availability and Performance in Porcupine: A Highly Scalable, Cluster-based Mail Service, *Symposium on Operating Systems Principles*, 1999

[4] C. Russell, Java Data Objects 1.0 Proposed Final Draft, JSR12, Sun Microsystems Inc., available from http://access1.sun.com/jdo, 2001

[5] Steven D. Gribble and Eric A. Brewer and Joseph M. Hellerstein and David Culler. Scalable, Distributed Data Structures for Internet Service Construction, *4th Symposium on Operating System Design & Implementation*, San Diego, 2000

[6] J. Chen and Q. Huang and A. Sajeev. Interoperability between object-oriented programming languages and relational systems, *Proceedings of TOOLS Conference*, Melbourne, 1995

[7] Michael Carey, David DeWitt, Michael Franklin, Nancy Hall, Mark McAuliffe, Jeffrey Naughton, Daniel Schuh, Marvin Solomon, C.K. Tan, Odysseas Tsatalos, Seth White, Michael Zwilling, Shoring Up Persistent Applications, In *Proceedings of ACM-SIGMOD 1994 International Conference on Management of Data*, Minneapolis, Minnesota, 1994.

[8] Riedel, E. and Gibson, G. Active Disks - Remote Execution for Network-Attached Storage, *Technical Report CMU-CS-97-198*, December 1997

[9] Wensong Zhang. Linux Virtual Server for Scalable Network Services, *Linux Symposium*, Ottawa, 2000

[10] Sleepycat Software Inc., Berkeley DB Programmer's Tutorial and Reference Guide, available at www.sleepycat.com, 2001

[11] M. Raynal and A. Shiper. A Suite of Formal Definitions for Consistency Criteria in Distributed Shared Memories. *ISCA Proceedings of the International Conference PDCS*, Dijon France, pages 125--130, September 1996.

[12] Compaq Computer Corp., Intel Corporation, Microsoft Corporation. Virtual Interface Architecture Specification, Version 1.0. available from www.viarch.org, 1997

[13] Barbara Liskov, Miguel Castro, Liuba Shirra, Atul Adya. Providing Persistent Objects in Distributed Systems. In proceeding of ECOOP'99.