TODS: Cluster Object Storage Platform Designed for Scalable Services

Feng Zhou [a], Chao Jin [b], Yinghui Wu [b], Weimin Zheng [b]

[a] *Computer Science Division, University of California, Berkeley, USA*

[b] *Dept. of Computer Science and Technology, Tsinghua University, Beijing, China*

zf@cs.berkeley.edu, {jinchao99, wuyinghui01}@mails.tsinghua.edu.cn, zwm-dcs@tsinghua.edu.cn

---

Abstract

In this paper we present the design and implementation of Tsinghua Object Data Store, a cluster object storage system to support the building of scalable Internet services. TODS provides a unified, transparent and object-oriented view of the storage devices of the whole cluster, which greatly simplifies cluster service development. In the meantime, it is designed to be scalable and efficient. Services built on it can simply inherit these properties in a lot of cases. TODS supports ACID transactions, which facilitates the building of complex transactional services. TODS abstracts away from service logic most complexities of data management, which have often become major obstacles in developing high quality Internet services. The design principles, architecture and implementation of TODS are 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. We also demonstrate that the programming interface is significantly easier to use than that of RDBMS with a head-to-head comparative experiment.

Keywords: Object store, cluster storage, transparent persistence

---

1 Introduction

Server cluster is a natural platform for building large-scale Internet services [3]. Clusters are incrementally scalable, inherently parallel and easily made fault-tolerant. Recent years have already seen lots of successful scalable Internet services like Google [11] and Yahoo that are built with clusters. However, building cluster services has never been an easy task. One of the most challenging problems is data management. The data store must be scalable, efficient, easy to use and tolerant to node failures. Unfortunately, most current cluster services use home developed service-specific solutions to achieve these properties. For example, the Porcupine cluster email system [9] managed to achieve the above properties by using its own distributed storage manager and doing replication of critical data all by its own. Although these techniques do solve the problem, this approach tends to mix application logic with low-level data management details, which in turn undermines portability and maintainability of the code as the services evolve. Therefore it is desirable that a separate, reusable data management layer exists to abstract away all the complexities and let developers focus on service logic.

The separation of data management from application logic is by no means a new idea. RDBMS and distributed file systems are standard examples of it. However, the *status quo* of Internet service construction described above reflects the fact that these standard data management means do not easily fit into the cluster service scenario.

In this paper we present Tsinghua Object Data Store, our attempt to build an object-oriented data management layer as a cluster infrastructure software, specifically for the construction of Internet services. It behaves as a scalable and fault-tolerant object store with transaction support. Moreover, it supports *transparent persistence* in the Java programming language. 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 data objects persistent. Objects are automatically fetched from store when they are accessed, and modified objects are automatically written back. The client interface is compatible with Java Data Objects API [8], which is the emerging standard for transparent data access in Java.

Behind the scene of the easy-to-use interface, TODS maintains the consistent shared state of the cluster store in a scalable and fault-tolerant way through a decentralized software architecture. It also appeals to properties of clusters such as high speed, low latency interconnection and incremental scalability. Well-known characteristics of Internet services are kept in mind when we designed TODS, including huge data volume, relatively small data unit sizes, high concurrency, good access locality and diversified consistency requirements.

2 TODS Overview

A TODS system is defined as a self-contained data management layer running on a server cluster to handle storage requests of Internet services running on the same cluster. Figure 1 shows how services interact with TODS.

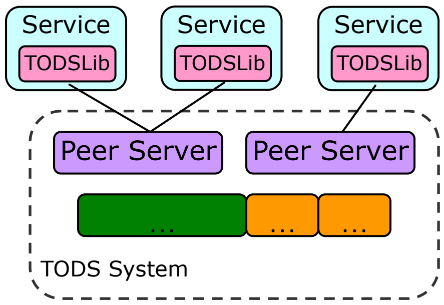

Fig. 1. TODS Overview

Services connect to TODS components known as Peer Servers to access data. They are called Peer Servers because they are inherently identical to each other from users' view, each presenting a single image of the whole TODS system and they communicate with each other in a peer-to-peer style. There are also other components (the lower three blocks), which the service instances do not connect to directly.

Within the service process, a library named TODSLib maps user API calls to messages sent to Peer Servers and parses results from them. Currently a Java version of TODSLib is implemented. As mentioned before, it implements automatic persistence control named *transparent persistence*.

From a functional point of view, TODS provides similar capabilities found in RDBMS and distributed file systems. RDBMS provides completely self-contained data management capacities and strict consistency guarantees to users. Distributed file systems, on the other hand, provide distribution abstraction and sometimes fault-tolerance support, which are very important properties to Internet services.

However, for Internet services, it is argued that **RDBMS** (and also parallel RDBMS) are often too complicated, provides overly powerful semantic guarantees and employs generality such as SQL at the cost of performance [4]. Another concern raised by researchers is a phenomenon called *impedance mismatch*, between relational databases and object-oriented programming languages [2]. Application developers are forced to write repetitive and error-prone code to convert data from database records to in-memory objects and vice versa. On the other hand, **distributed file systems** are often considered too general in the sense that the system knows nothing about the structure of user data, thus leaving all the work of organizing, accessing and querying data to application developers. Another important drawback of distributed file systems is the lack of transaction support, forcing developers to use ad-hoc ways such as custom locking servers to maintain data consistency under concurrent access.

TODS strives to solve some of these problem through an object-oriented data model and a decentralized architecture. We believe that an object-oriented data model have at least two advantages over a relational model for Internet services. First, it is in most cases easier to use for service programmers. While theoretically whether object-oriented model or relational model better matches application data is always a problem of active debate, the better matching between object-oriented model and widely-used object-oriented programming languages is clear. Second, the more "explicit" and fine-grained nature of object-oriented data access eases more aggressive data access optimization, such as caching and prefetching. Most operations are done at object (tuple) level in an OO store while at relation level in a relational systems. For example, in relational systems, the basic operation of accessing one single data entry, which makes up a significant portion if not most of data accesses in a lot of Internet services, has to be done using a query to select the corresponding tuple from the whole relation. Therefore every data access including this simple one-tuple access incurs the overhead of query parsing and processing, which is often a headache for Internet services using RDBMS. Trying to cache the tuple in application level in order to reduce this overhead is not easy, because as data access is done in the granularity of relations, timely notification of tuple updates to the application level is often not available. In TODS, as data accesses are done in object(tuple) granularity, objects can be cached on different nodes and events will be delivered to the nodes when objects are updated.

The rest of the paper is organized as follows; Section 3 of this paper describes related work. Section 4 presents design principles, architecture and implementation of TODS. Section 5 presents performance results. Section 6 discusses application examples. Section 7 talks about future work and concludes the paper.

3 Related Work

TODS incorporates ideas from previous research work that are not directly aimed at Internet services. Shore [1] is a well-known object storage repository built to support large applications such as Geographic Information Systems and satellite data repositories. Also using decentralized cluster architecture, Shore provides good scalability and availability. Shore provides OODB-like features such as a type system and transaction support. However, access to persistent objects in Shore is not transparent, as in TODS. 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 (Java byte-code) processing to extract meta-data from user classes, so users never have to maintain two copies of class definition, one in

SDL files and the other in C++ source files.

Distributed Data Structure (DDS), described in [4], stores distributes data structures within a cluster. DDS shields service authors from the complexities of cluster storage management. 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. All the inner structures of data items are left to users to organize. Update of one data item is atomic, but transaction is not supported.

Active Disks [7] is a research effort to exploit computing power of embedded processors on disks. The most obvious use is to do database queries. This greatly improves scalability compared to traditional server-based systems. Actually the local storage manager in TODS (called a Brick) is much like an Active Disk. It does primitive data management tasks such as get, put and filtering. Although currently Brick is implemented as a software daemon, it is an interesting topic to embed it into a device or even a powerful disk drive.

4 TODS Design and Implementation

4.1 Design Principles

Several principles are followed when we design TODS.

Separation of concerns. Apart from abstraction of data management tasks from service logic, TODS itself is designed to be layered and modular, with different level of abstraction for each layer and clear interface between modules.

Abstraction without losing key performance hints. “Performance hints” such as which objects are physically ”near” are propagated to the more ”abstract” upper layer, which uses these hints to more effectively arrange its work.

Exploiting properties of clusters, such as that network partition never happens, network RAM is faster than local disk, all nodes are under central control, etc..

4.2 TODS Architecture

The structure of an Internet service using TODS as storage layer is shown in figure 2. Figure 3 shows the distribution of components on a cluster running TODS and an Internet service.

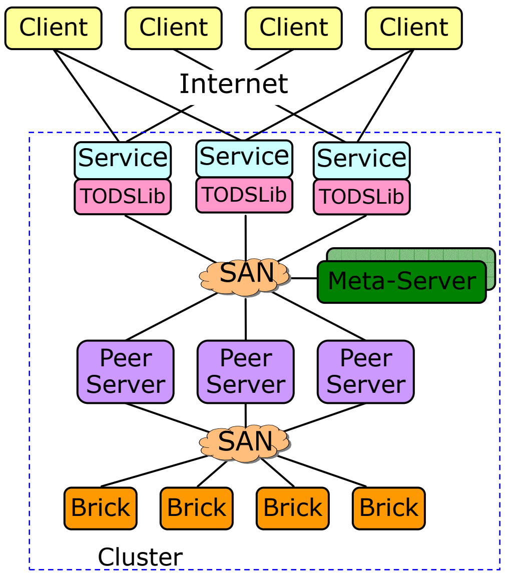

Fig. 2. TODS Architecture

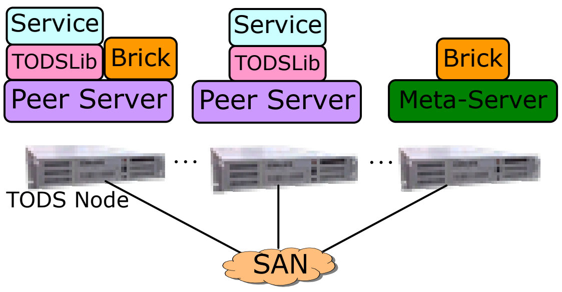

Fig. 3. Typical TODS Configuration

The **clients** in figure 2 can be Web browsers, email clients or other applications. They connect to service instances through the Internet by Internet application protocols such as HTTP, IMAP, etc.

Services are multiple instances of the same service. Reverse proxy or DNS rotating can be used to make the cluster nature of the service transparent to clients. Techniques such as IP take-over by ARP spoofing [12] can be exploited to hide transient service failures, as long as service instances maintain no state or only *soft state* that can be recovered easily by another service instance from persistent data.

Peer Servers are connection points into the TODS system. Each Peer Server presents the global image of the whole TODS system, abstracting the internal distribution of data. So clients connecting to any one of the Peer Servers see all data objects in the system. Peer Servers also act as transaction managers in distributed transactions. The first reason for using more than one peer server is that it prevents Peer Server from becoming performance bottlenecks, which is often the case in classic client-server systems. The second reason is for increased availability. The client library can transparently switch to another Peer Server in case one fails. Another reason is for efficiency. 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.

Bricks are single-node transactional data stores. They manage local data in a key-value fashion, and provide local transaction and primitive query functionalities. Brick is currently implemented upon the Berkeley DB [5] 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 Servers currently cache which objects, and invalidates data on corresponding Peer Servers when certain objects are updated.

The **Meta-Server** maintains system configuration and meta-data. It is replicated and thus assumed fault-tolerant, providing a safe place for critical global information. System configuration includes location (IP, port) and parameters of all components such as Peer Servers and Bricks. This ensures centralized management of the whole system. Meta-data includes information about name space structure, users and groups information, persistent class list and meta-data (structure) for each user class. Current implementation is based on Java Remote Method Invocation (RMI), which is similar to RPC. Data replication is implemented using the built-in replication feature of Berkeley DB (version 4).

TODSLib is a Java class library that service developers use to access persistent objects in TODS. It implements the transparent persistence logic and communicates with Peer Servers. To make access to external objects stored in TODS identical to accessing in-memory objects, synthetic code is inserted into user classes to track status change and intercept certain operations such as getting a referenced object that is not currently in memory. The above process of automatically modifying existing code is often mentioned as *code enhancing*. Thanks to the capabilities of modern Java Virtual Machines and the simple format of Java byte-code, this process is done on-the-fly by TODSLib at run-time, without any modification to the Virtual Machine or user code. Using a custom class loader, TODSLib enhances every persistent class before actually loading it, thus making the process completely transparent to users.

TODSLib communicates with Peer Servers using a message-oriented protocol. Consecutive requests and responses are grouped to form messages to be sent over the wire. The underlying network protocol is pluggable. Currently we

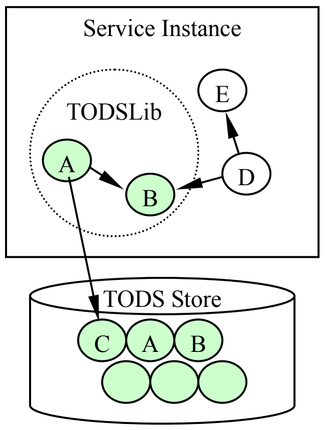

Fig. 4. Object Reference

have implemented the transport layer on both standard TCP and the GM messaging protocol of Myricom, with the latter delivering much shorter latency. The encoding/decoding logic and request handling framework of this process are encapsulated in a module which is reused in both "TODSLib-Peer Server" and "Peer Server-Brick" communication.

4.3 Data Model and Application Programming Interface

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.

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 4 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.

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.

A simple code example of using TODS is shown below. It uses the Java Data Objects API implemented by TODSLib. After initializing the TODSLib, the example first inserts a *Point* object, using a transaction. Then it lists all *Point* objects in the store.

```
import javax.jdo.*;
import edu.tsinghua.tods.client.TODSPMF;
import java.util.Iterator;

public class Simple {
  public static void main(String[] args) {
    // -1- Initialize TODSLib
    // Persistence Manager Factory, analagous to a connection
    // factory of RDBMS
    TODSPMF pmf = new TODSPMF("tods://127.0.0.1", "zf",
                              "_password_");
    // Persistence Manager, analagous to a connection of RDBMS
    PersistenceManager pm = pmf.getPersistenceManager();

    // -2- Insert a Point into the store
    Transaction tx = pm.currentTransaction();
    tx.begin();
    Point p = new Point();
    p.setX(5);
    p.setY(10);
    pm.makePersistent(p);
    tx.commit();

    // -3- List all Points in the store
    Extent e = pm.getExtent(Point.class, false);
    for (Iterator it = e.iterator(); it.hasNext(); )
    {
      Point p2 = (Point)it.next();
```

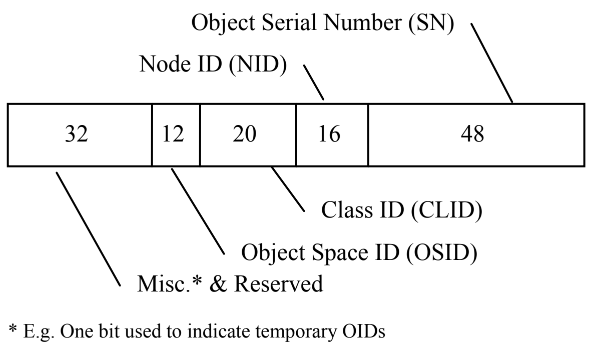

Fig. 5. 128-bit OID Format

```
      System.out.println("X="+p2.getX()+", Y="+p2.getY());
    }

    // -4- Close the Persistence Manager
    pm.close();
  }
}

// User data class
class Point {
  private int x;
  Private int y;

  public int getX() { return x; }
  public int getY() { return y; }
  public void setX(int v) { x = v; }
  public void setY(int v) { y = v; }
}
```

4.4 Object Identification and Location

An Object ID is a 128bit integer, whose structure is shown in figure 5. 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.

4.5 Data Consistency, Transactions and Failure Recovery

TODS supports non-transactional and transactional modes of access. It is determined by whether data accesses are enclosed in a transaction. For non-transactional accesses, data caching on Peer Servers are enabled, which results in much better performance. Cache invalidation between Bricks and Peer Servers is used to properly update the caches. However there isonly limitedguarantee about data consistency under concurrent access. Different Peer Servers may report different value for the same object at some moment due to asynchronous cache invalidation. Consistency level of non-transactional access is PRAM Consistency [6], using parallel computing terminology, i.e., writes made by each specific client are seen by others in the original order, but the global order is not guaranteed.

In contrast, transactions in TODS are fully ACID but incur additional overhead. As previously mentioned, Bricks are designed to be transactional. So transactions involving one brick (local transaction) are easily done. Distributed transactions on multiple bricks are managed using the two-phase commit protocol. This approach is chosen in favor of global concurrency control because the 2-phase commit protocol is not prohibitively expensive given the low latency property of modern LAN [4], and it is a relative simple protocol. TODS prefers to do local transaction whenever possible. E.g., multiple objects inserted in a transaction always go to the same brick. When distributed transaction must be done, the corresponding Peer Server acts as the transaction manager, and participating Bricks act as resource managers. All status information of ongoing distributed transactions is stored persistently in a simple database managed by the Peer Server, in order to make both Peer Server and Brick failures recoverable.

Graceful recovery from all types of node failures is both very important and difficult to achieve. General solution to the problem is way beyond the scope of the project. So a few of assumption are made, which we think are reasonable in the cluster environment. First, it is assumed that node storage corruption never happens. Redundant storage like RAID can be used to approximate this.

Second, network partition never occurs. LAN equipments are very reliable today and often redundant in clusters. However, the obviation of this problem is mainly due to its complexity. With these two assumptions, the possible failures are fortunately only those that do not cause data corruptions, such as power losses, software crashes and hardware faults. Below we discuss recovery processes of each component of a system.

As previously mentioned, Meta-Servers are replicated. Actually they are organized as a synchronized 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.

4.6 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 is 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 [4]. For other operations like queries, Peer Servers and Bricks access the internals of the data block. Without knowing about class meta-data, Bricks are able 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. Shown below is part of the de-compiled code of the *enhanced* Point class as shown in section 4.3.

```
class Point implements javax.jdo.PersistenceCapable
{
  ...

  public int getX() {
    return jdoGetPoint_x(this);
  }

  public void setX(int i) {
    jdoSetPoint_x(this, i);
  }

  private static int jdoGetPoint_x(Point point) {
    if(point.jdoFlags <= 0)
      return point.x;
    int i = jdoInheritedFieldCount + 0;
    if(point.jdoStateManager.isLoaded(point, i))
      return point.x;
    else
      return point.jdoStateManager.getIntField(point, i, point.x);
  }

  private static void jdoSetPoint_x(Point point, int i){
    if(point.jdoFlags == 0){
      point.x = i;
      return;
    } else {
      point.x = point.jdoStateManager.setIntField(point,
      jdoInheritedFieldCount + 0, point.x, i);
      return;
    }
  }

  ...
}
```

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.

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

4.7 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 non-trivial 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

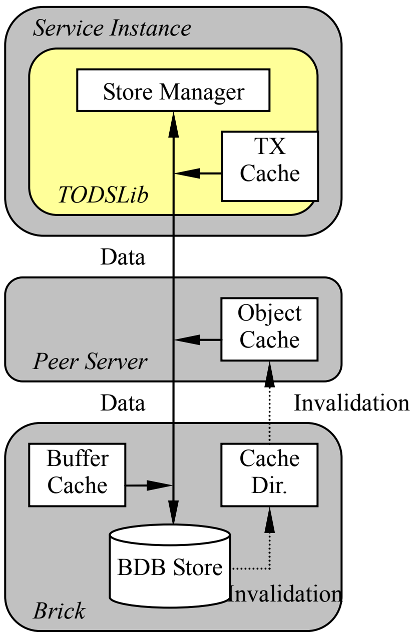

Fig. 6. Cache Interations

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 are 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.

4.8 Cache

Caches are maintained by components of TODS to improve performance. Figure 6 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* operation that hits the Object Cache can be as low as a local IPC call such as a shared memory access.

A cache directory is maintained at each Brick to ensure that copies of objects in corresponding Peer Servers are up to date. It tracks which Peer Servers cache which objects and invalidates the copies if they are updated. On the other hand, the directory should also be updated when certain items are replaced in the caches. However, how to efficiently maintain the validity of this dir is a subtle problem. If a lot of traffic is used to maintain it, the benefits of the Object Cache will be compromised. However the directory is allowed to contain some redundant items, i.e., a cache item that does not exist on a Peer Server. This property is exploited to reduce the maintaining cost. Rather than sending notification as separate messages immediately when cache items are replaced, Peer Servers lazily piggyback all notifications about cache replacement with normal Brick request messages. Although this results in some harmless inaccuracy in the directory, total message counts decreases substantially, thus network traffic is reduced.

Bricks also maintain Buffer Caches which are page-based caches for table files. This is effective for both transactional and non-transactional operations. Although the underlying OS often has its own buffer cache, it still makes sense to use our own because without it, every read operation will result in a system call, which incurs the overhead of CPU entering and quitting kernel mode. So we prefer to use our own application level buffer cache and bypass the OS buffer cache if possible, using the *Direct I/O* or *Raw I/O* features which are available on many systems.

4.9 To Fail-Stop or Not?

Sometimes a trade-off between availability and consistency must be made about a clustered Internet service. Letting the service run when some part of the storage fails will certainly improve availability, but will also potentially confuse the users or corrupt existing data if the service logic is not designed very carefully. Consider the example of an online directory. If it continues running after part of the storage becomes unavailable, some items will mysteriously disappear, which is confusing. Moreover, data corruption may occur if, for example, a sub-category with the same name as an existing but currently unavailable one is created.

However, the counterpart – fail-stop operation also does not always work. Consider a clustered Web-mail system with a lot of nodes. It is totally intolerable that the system should become completely unavailable if one storage node fails. Thus TODS supports both types of operation. If configured to run after partial failure, the service will be notified when failure happens and recovers. If fail-stop is enabled, TODS simply fails if any part becomes unavailable.

Replication may be the answer to the above dilemma. But it may not be practical for a large part of Internet services because of the multiplied cost. Replication of user data is not yet implemented in TODS but should be relatively easy if replication of entire Bricks were to be done, which could be done transparently to above layers.

5 Performance

Performance experiment results of the TODS prototype 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.

5.1 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 7. 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 8 and figure 9. 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

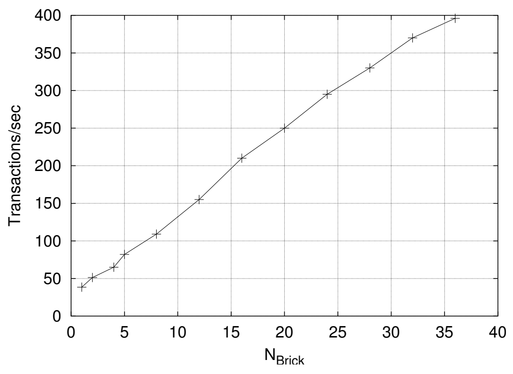

Fig. 7. In-memory Read Scalability

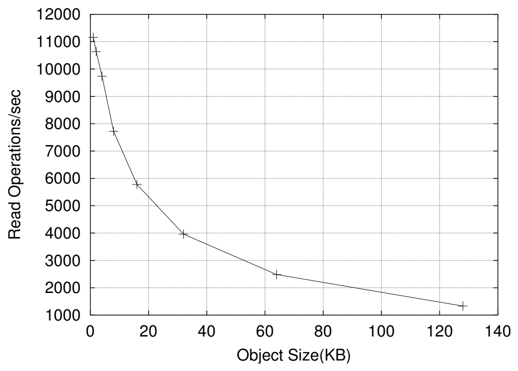

Fig. 8. In-memory Read Throughput

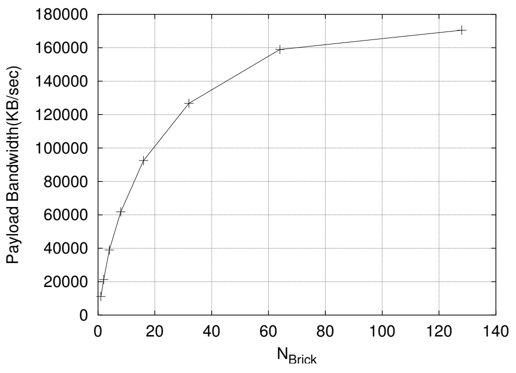

Fig. 9. In-memory Read Payload

adequate for any conceivable Internet services apart from media and file services.

5.2 *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 10 and figure 11, 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.

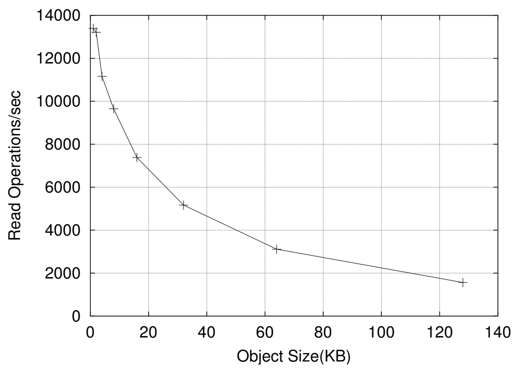

Fig. 10. On-disk Read Throughput

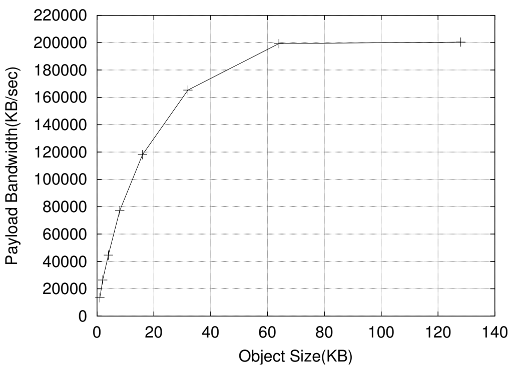

Fig. 11. On-disk Read Payload

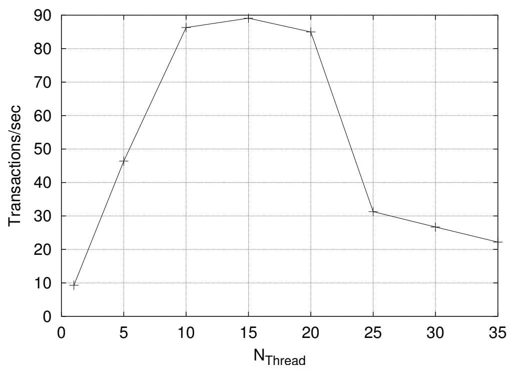

Fig. 12. Cache-hit Read Throughput

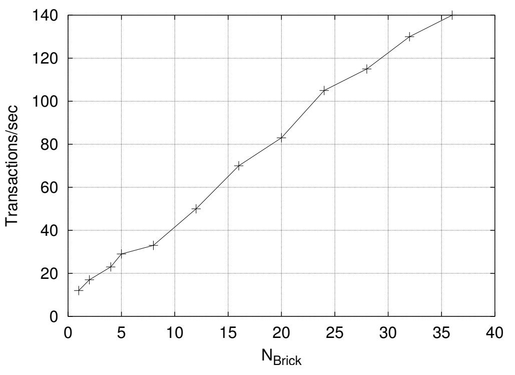

Fig. 13. Cache-hit Read Payload

5.3 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 12 and figure 13. 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.

5.4 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

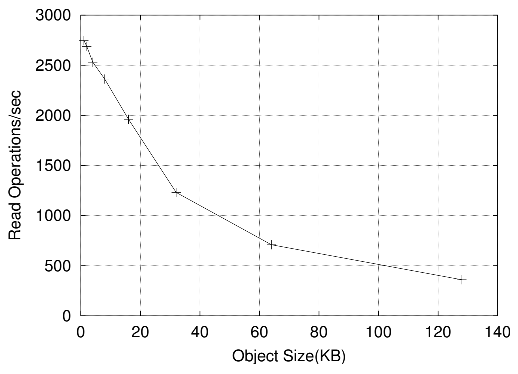

Fig. 14. Transactional Write

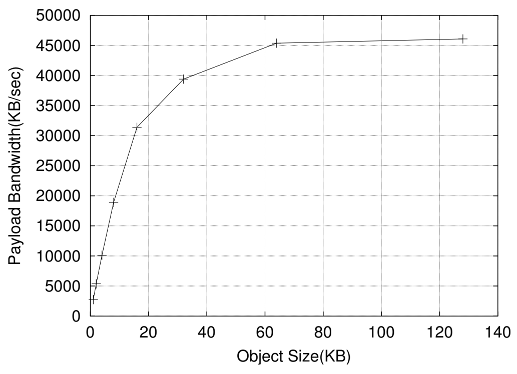

Fig. 15. Distribute Xact. Scalability

possible. From the results shown in figure 14, 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.

5.5 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 15. The performance is roughly 1/3 of that of local transactions.

6 Experience with TODS

One Internet service we have built with TODS is a simple Web-based music jukebox service. Registered users can upload their music, browse the category with artist/album/genre, build play-lists and listen to them. As an experiment to compare the programming interface of TODS and RDBMS, two undergraduates with intermediate Java skill and no experience of JDBC (Java API to RDBMS) or JDO were invited to write it, using JDBC and JDO, respectively. The framework code, E-R graph of the data model and HTML template pages were already done. It turns out that the JDO version took half the time of the JDBC version to write (about 20 hours v. 40 hours) and the total lines of code is about 15% less. This demonstrates that the TODS programming interface is both easy to learn and productive, compared to RDBMS.

7 Future Work and Conclusion

There are many issues not addresses in current version of TODS. Large data files (like a PDF document, a TAR archive) are not handled efficiently. They are loaded totally into memory if managed by TODS. The addition of a stream oriented access method (like what we have in file systems) is 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 implementing more *transports* such as one based on VIA [10] 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.

In conclusion, we have presented the design and implementation of TODS, a cluster storage platform specifically designed for Internet services. It is designed with the requirements of scalable services in mind and appeals to many advantageous properties of modern server clusterss. A decentralized architecture is used which proves to be scalable and efficient. The object-oriented programming interface of TODS implements transparent persistence which completely relieves developers from writing I/O code. Different levels of consistency are supported that makes TODS suit the requirements of different services.

References

[1] Carey, M. J., DeWitt, D. J., Franklin, M. J., Hall, N. E., McAuliffe, M. L., Naughton, J. F., Schuh, D. T., Solomon, M. H., Tan, C. K., Tsatalos, O. G., White, S. J., and Zwilling, M. J. Shoring up persistent applications. In *ACM-SIGMOD 1994 International Conference on Management of Data* (1994), pp. 383–394.

[2] Chen, J., Huang, Q., and Sajeev, A. Interoperability between object-oriented programming languages and relational systems. In *Proceedings of TOOLS Conference* (December 1995).

[3] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. Cluster-based scalable network services. In *Symposium on Operating Systems Principles* (1997), pp. 78–91.

[4] Gribble, S., Brewer, E., Hellerstein, J., and Culler, D. Scalable, distributed data structures for internet service construction. In *Symposium on Operating Systems Design and Implementation* (2000).

[5] M. Olson, K. B., and Seltzer, M. Berkeley db. In *Proc. of USENIX Techinal Conference, FREENIX Track* (Monterey, CA, June 1999).

[6] Raynal, M., and Schiper, A. A suite of formal definitions for consistency criteria in distributed shared memories. In *Proceedings Int Conf on Parallel and Distributed Computing (PDCS'96)* (Dijon, France, 1996), pp. 125–130.

[7] Riedel, E., and Gibson, G. Active disks - remote execution for network-attached storage. Tech. Rep. CMU-CS-97-198, 1997.

[8] Russell, C. Java data objects specification 1.0. `http://access1.sun.com/jdo`, 2001.

[9] Saito, Y., Bershad, B. N., and Levy, H. M. Manageability, availability and performance in porcupine: A highly scalable, cluster-based mail service. In *Symposium on Operating Systems Principles* (1999), pp. 1–15.

[10] VIA. *Virtual Interface Architecture Specification, Version 1.0.* available at: www.viarch.org, 1997.

[11] Wagner, M. *Google Bets The Farm On Linux.* `http://www.internetweek.com/lead/lead060100.htm`.

[12] Zhang, W. Linux virtual server for scalable network services. In *Linux Symposium, Ottawa* (2000).

Feng Zhou is a PhD student at CS division, University of California, Berkeley, CA. He received his Master's degree in 2002 and Bachelor's degree in 2000 from Tsinghua University, Beijing, China. His research interests are in runtime systems for scalable services, storage systems and wide-area distributed systems.

Chao Jin is a 4th year Ph.D Candidate at Department of Computer Science and Technology, Tsinghua University, Beijing, China. His major research interests include parallel and distributed systems, cluster computing and storage systems. Now he is working on a project named Storage and Organization of Massive Data in Network Environment, supported by National Grand Fundamental Research (973) Program of China.

Yinghui Wu is a Master student at Department of Computer Science and Technology, Tsinghua University, Beijing, China. His research interests include networking and operating system. Now he is working on a project named Storage and Organization of Massive Data in Network Environment, supported by National Grand Fundamental Research (973) Program of China.

Weimin Zheng is a professor at Department of Computer Science and Technology, Tsinghua University, Beijing, China. His major research interests include parallel / distributed and cluster computing, compiler techniques and run-time system design for parallel processing systems. He and his group, with sixteen faculty members and research staff members and 90 graduate students, are currently doing a number of R&D projects supported by the Natural Science Foundation of China, and the National High-Tech Program in these areas.