Cache Invalidation and Propagation in Distributed caching

Pooja Kohli Rada Y. Chirkova
pkohli,chirkova@csc.ncsu.edu

Abstract

Replication and caching strategies are increasingly being used to improve performance and reduce user perceived delays in distributed environments. A query can be answered much faster by accessing a cached copy than by making a database roundtrip. This setting creates a number of important issues such as maintaining consistency among copies of the same data item. Numerous techniques have been proposed to achieve caching and replication while maintaining consistency among the replicas. A closer investigation of these schemes reveals that no one scheme can be optimal for all environments. In this study we look at invalidation protocols for achieving consistency in systems that use distributed caching. We propose heuristics for dynamic adaptation of these protocols for cache consistency. These heuristics aim at propagating invalidations while reducing the cost of data transfer.

1 Introduction

1.1 Overview

In many client-server scenarios, data is often stored at a central database server. This central server supports various client servers which interface with the end user. In case the client servers are remote, data may be retrieved from long distances. This often results in problems such as shortage of communication bandwidth, expensive trips to the database and unacceptable response times. A key idea for solving this problem is to store copies of the data locally on the remote servers in order to reduce the number of data retrieval operations over long distances. Locally storing data copies is called *caching*. Caching avoids the overhead of maintaining the transactional consistency involved in maintaining a database on each site. By locally storing data at the point of usage, communication bandwidth is saved and response time is shortened. Data is often distributed and replicated over different caches to improve performance [14]. With distributed caching, problems

arise over time if copies of data become inconsistent, causing the local copy of the data to be out of sync with the contents of the central database, or with the other cached copies. These inconsistencies may result from local updates on the cached data or due to an outdated version of the cached data compared to what is stored on other caches, see [15]. To overcome this problem, some caches employ an invalidation policy [19] for informing all the other caches that the data has been updated and causing them to update the invalidated data. The goal of this thesis is to propose heuristics, aiming at propagating invalidations while reducing the costs of data transfers for several scenarios with varying parameters, such as network topology or number of transactions.

1.2 Motivation

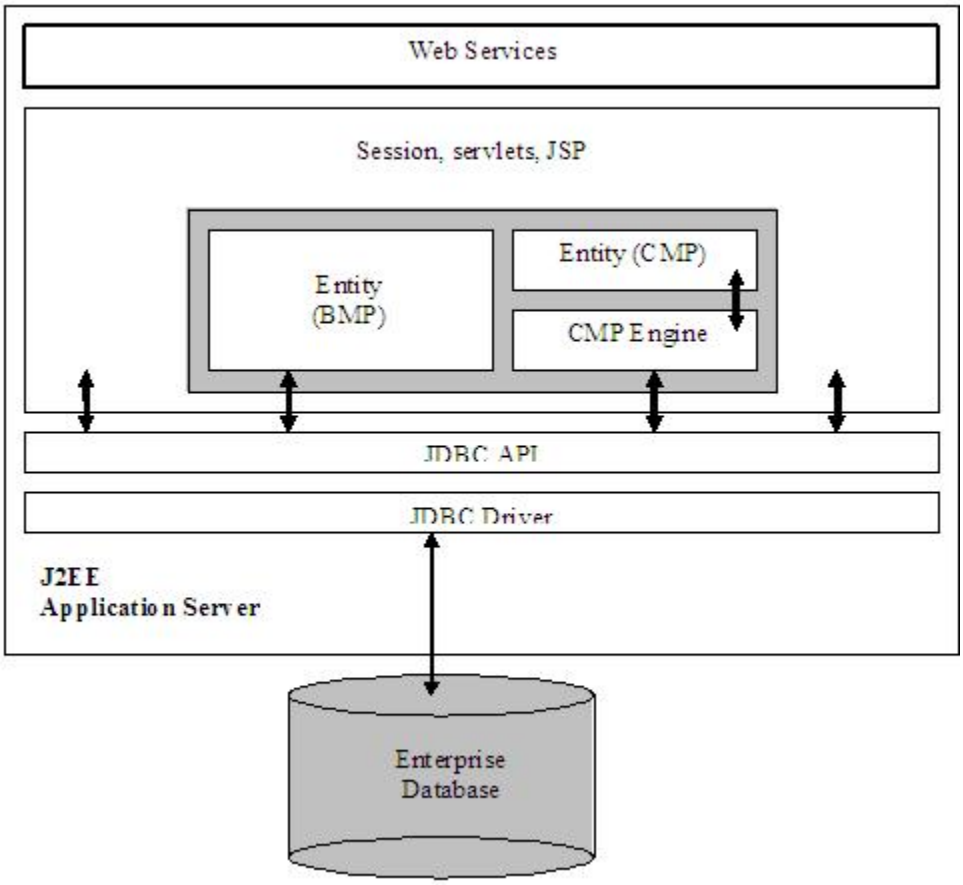

Figure 1: J2EE configuration without caching

Distributed caching has important implications in many real life applications, including J2EE applications [5] for clustered environments. These are increasingly used to enable scalability and

high availability of data. J2EE has significant provisions for caching to improve performance, and all existing enterprise-class application servers provide multiple levels of caching to enable faster applications. Websphere [1], Bluestone Total-E-Server [8], Sybase Enterprise Application Server [21], SilverStream Application Server [20] are some of the middleware applications that implement this technology. Beyond what the J2EE specification and application servers provide, there are circumstances where there might be a need to implement further application-level caching to gain still faster response times. In a cluster, each node could end up maintaining its own copy of the cache, which could eventually run out of sync with the others. Hence, there is a need to provide effective schemes for cache invalidations in these scenarios. Caching has been implemented to improve performance of J2EE applications in products such as Tangosol [24] and the Livestore Data Cache [7]. Sekkapu pattern [**?**] provides an illustration of an EJB entity bean distributed cache. It provides a JMS or broadcast method to handle the problem of invalidation in its transactional caches.

Fig 1 shows a typical example of J2EE architecture without caching. Enterprise Java Beans (EJB), which are special Java objects, encapsulate the enterprise data. This data is stored on the enterprise database which may be residing on a remote server. As a result, for every data request in a J2EE session, a JDBC call or a network connection needs to be established with the database. This can prove costly and induce delay while answering the request. A solution to this problem is depicted in Fig 2, which shows the J2EE architecture with an application cache, such as an EJB cache, which stores a copy of the information residing on the database. Now whenever a user queries the enterprise data, it can be answered locally without making a JDBC call to the database. As a result, the query is likely to be answered much faster in most of the cases.

1.3 E-Bidding: A motivating example

We use E-bidding and online auctions as a motivating example to illustrate the concept of distributed caching and the need for having effective invalidation schemes in place. On an online bidding site, often for load balancing puporses the informational content is distributed over multiple servers. Since this is a logical distribution, we can have multiple bidders bidding for the same object over different servers without knowing the difference. An update of the current bid by one bidder may not be reflected in time to the other bidder. As a result he may be getting stale values and may base his decision on the stale value returned. Hence we see that there is a need to maintain consistency among all the replicas. E-bay [**?**], Yahoo Auctions [**?**], Sothebys [**?**] are some of the popular online

bidding sites maintanined on the internet.

1.4 Contributions

The main contribution of this thesis work is to help decide effectively which algorithm would perform better for propagating updates and invalidations for several scenarios in distributed caching. A lot of work has been done in terms of cache invalidation for file systems, hierarchical caches and web caching. Given the potential and advantages for distributed caching, this thesis provides a reference for creating distributed cache invalidation design. The heuristics comparing the performance of contemporary schemes shall help designers choose the right kind of scheme for their setting. The results of this study should be useful to real world applications such as enterprise servers based on clustered J2EE technology, which need to provide highly scalable and available software. In general, applications that make expensive trips to the database can use these heuristics to optimize performance and resources. Other scenarios that can directly benefit from the results include distributed database systems, client-server systems and application servers.

2 Related Work

Caching has been seen as a source for improving performance for a long time [15]. [16] presents an extensive bibliography of some of the work done in the field of web caching. Distributed caching is a relatively new architecture. Some motivation for adopting this architecture can be derived from [22] where two out of four design recommendations for improving cache performance are replication and distribution of data. [2], [14], [24] are some of the references which describe the general architecture of distributed caching along with its advantages and disadvantages. [14] explains some of the common terms associated with distributed caching. Some related problems such as accessing replicated objects can be found at [17] and [12]. Once the need for caching has been established, and distributed caching seen as an important component in architectures for providing caching in cluster based application severs, the next logical question is how to maintain consistency among caches. In distributed caching, this is more crucial than traditional caching as objects are replicated at many servers. A lot of work has been done in the field of cache consistency for web caching and file based networks [10], [6], [13], [27], [25], [26]. At the same time, very little work has been done in the field of distributed caching.

[9] and [3] are two studies very similar to ours but in the setting of web caching. While the

authors of [9] only compare the TTL based schemes with the Alex protocol [4], the authors of [3] compare three classes of schemes – client polling, invalidation and TTL based. Another work which compares TTL and invalidation schemes is that of Worrell [23]. All these studies are concentrated on web caching architectures. They assume large network sizes and a hierarchical cache architecture. Our work differs from the above studies as we focus on the distributed caching architecture. In our settings we assume flat cluster-based networks. As a result, we assume the network size is small (not greater than twenty hosts). What further distinguishes our work is that we specifically look at comparing different types of invalidation schemes – client-based invalidation and server-based invalidation. Client-based invalidation is a pull-based scheme, in which invalidations are initiated or pulled by clients at different times, whereas server-based invalidation is a push-based scheme, in which invalidations are initiated or pushed by the central server. Also, our results are somewhat different from [3] and [9]. While in [3] the authors conclude that polling each time is the least competitive scheme, they argue that invalidations is the best approach. In [9] the authors propose the use of Adaptive TTL over all else. We see that neither one of the schemes outperforms all of the other schemes for each and every setting during our experiments.

3 Preliminaries

3.1 Caching

In a typical client-server environment data returned by clients to end users is typically retrieved from remote servers. This could often result in shortage of communication bandwidth as well as unacceptable response times. To reduce the number of remote data retrieval operations over long distances, copies of data objects are stored locally. This concept of locally storing copies of remote data objects for faster retrieval is called *caching*.

3.2 Pure distributed caching

In *pure distributed caching* only one copy of each data object is maintained across all the cache servers. The objects are distributed among the existing cache servers. Each cache server is responsible for maintaining the objects it caches. When an object needs to be accessed, the cache server that has the object is contacted and the object is retrieved. While this scheme makes the task of cache consistency easy, it makes the lookups relatively more expensive. This scheme could also lead to uneven distribution of workload when one object becomes much more popular than the others,

resulting in an increase in the number of queries targeted towards that particular cache server.

3.3 Replication

In *replication* objects are replicated over all cache servers. In this scheme each cache server maintains a copy of the object. Replication makes the system redundant and available. It also eliminates the single point of failure which occurs when the 'one copy per object' scenario is implemented. Replication also improves response time as the workload can be distributed among the different servers caching the objects. This however introduces complexity in maintaining the cache consistency as more than one server is now accountable for the same object.

3.4 Replicated distributed caching

We use the term *(replicated) distributed caching* to refer to schemes that contain some elements of both pure distributed caching and replication. Both pure distributed caching and replication offer two extremes. While pure distributed caching offers a scenario that is easy to maintain, it also has drawbacks. First, it hinders performance as the server with the copy of the object needs to be located and contacted. This makes the lookups relatively more expensive. Second, it may lead to uneven workload distribution and finally, presents a single point of failure. With replication, the system has increased redundancy and availability, but has higher complexity in maintaining consistency among the various replicas. By combining the two we avoid the drawbacks offered when any one of these schemes is applied just by itself. To implement distributed caching, we replicate the objects at more than one cache server but not all. We refer to the number of servers an object is replicated at as *degree of replication.*

3.5 Invalidations

With multiple serves caching the same object, there is a need for maintaining cache consistency. Changes to a cached object occur when it is updated as a result of a transaction. These changes cause the object to be out of sync with the other cached copies. To rectify this problem there is a need to propagate the change to all other caches. The policy employed to notify all caches about an update is called *invalidation policy.*

4 Cache Invalidation Scenarios

In this chapter we describe situations that illustrate the problem of maintaining cache consistency. Distributed caching has replication as one of the core ideas. Suppose a single object X with an initial value 50 is cached at two sites.

Scenario I Read - Read Situation
Site A:

```
SELECT X FROM table_employee;
```

Site B:

```
SELECT X FROM table_employee;
```

The 'SELECT' statement above, is a Structured Query Language (SQL) [18] representation of a read transaction. "SELECT X FROM table_employee" is a statement that is requesting to read all values of column 'X' from a table called as 'table_employee'. This situation means that on both cached objects the 'read' transaction is performed. In this case, the same value of X = 50 is returned to both users and the data is consistent. Thus, there is no need to invalidate the objects.

Scenario II Write - read situation
Site A:

```
UPDATE table_employee SET X = 100;
```

Site B:

```
SELECT X FROM table_employee;
```

The 'UPDATE' statement above, is a SQL representation of a 'write' transaction. "UPDATE table_employee SET X = 100" is a statement that is requesting to write or update all values of column 'X' from a table called as 'table_employee' to 100.

This is a situation of interest to us, as inconsistent values are returned to the two users. From Fig 3 we see that both sites have cached the initial value of 50. Now User 1 updates the object value to 100. Since this is a local copy, the value is changed at Site A but is not reflected at Site B. This means, because of the update of object X at Site A, all the others sites caching object X now need to be invalidated. There is a inconsistency in the value of the object stored at other sites, and the actual value of the object. Now when User 2 queries Site B for reading the value of object X it shall be returned as 50, not reflecting the updated value at Site A.

Scenario III Write - write situation
Site A:

```
UPDATE table_employee SET X = 2*X;
```

Site B:

```
UPDATE table_employee SET X = X/2;
```

In this scenario there is an update on the local copy of object X at both sites. In the absence of a cache invalidation policy, Site A will update X to 100 and Site B will update X to 25. This is not correct assuming that the transaction at Site B occurred after the transaction at Site A. The correct values should be X = 100 at Site A, and X = 50 at Site B.

Scenarios II and III illustrate the problem of inconsistency. To resolve this problem, when as an object is updated, all other copies of the object need to be invalidated. This problem is more pronounced in distributed caching than traditional caching schemes as there are multiple copies of the same object existing at different cache servers. This example illustrates that cache invalidation messages are an important part for distributed caching to be fully effective.

These scenarios also pose the independent problem of cache concurrency, such that no two read-write and write-write transactions occur concurrently.

5 Comparison of Schemes

In this chapter use the term *origin server* to refer to the server on which the original content resides. We refer to the term *cache servers* as the servers that maintain cached copies of the objects originally residing on the origin server.

5.1 Approaches based on the weak consistency model

In the *weak consistency model*, a stale object may be returned to the user. We borrow the definition of weak consistency from [3].

5.1.1 TTL based schemes

In this scheme whenever a cached object is retrieved from the database, it gets an associated *time to live* (TTL) with it. This is a number indicating the time interval during which the object is considered to be fresh. When an object is requested for the first time the server also sends with it its associated TTL. When the object is to be referenced its TTL is checked and if the TTL is still valid then the object is retrieved otherwise a new value is requested from the server. The client caches both the object and its TTL. Whenever the object is requested, the client checks the TTL value for that object. If the TTL has expired, a fresh copy of the object is brought from the server. This is an instance of the weak consistency model. When the data is distributed and replicated, it may have been updated on another client machine. The other object caches shall be informed of the update only when the TTL for the object expires. We compare and contrast two kinds of TTL-based schemes.

Fixed TTL scheme In the Fixed TTL scheme, when an object is retrieved from the server, its TTL is set to this fixed value. After this TTL has expired on a client, the object is treated as stale and a fresh value is retrieved from the server.

Pseudocode for origin server

```
1. While (no message );do nothing

2. If message==Request object {
```

```
send object to the client; TTL = max
}

3. if message==TTL Expired{
resend object to the client; TTL = max
}

4. if message==Object updated {
Update database;
}

5. Goto Step 1
```

Pseudocode for cache servers

```
1. /* Request object(s) from the origin server*/

      Send "Object request" to the origin server

2.  While (no transaction && no message); do nothing

3. if transaction == READ {
If (TTL == Valid); return object
If (TTL == Expired){
 Send "TTL Expired" to origin server;
 Goto Step 3;
}
}

4. If transaction == WRITE{
update object in the local cache;
```

```
Send "Object updated" to all other caches;
Send "Object updated" to the origin server;
}

5. If message == Object updated{
Update object in local cache; TTL = Max;
}

6. Goto step 2
```

5.1.2 Adaptive TTL

This scheme was originally proposed in [4]. Another variation of the scheme is described in [3]. In our variation of this scheme, the cached object when retrieved has an associated TTL with it. When the object is to be referenced its TTL is checked and if the TTL is still valid then the object is retrieved, else a new value is requested from the server. This scheme is different from fixed TTL in that when the object is referenced from the cache its TTL is reset. In other words, the TTL is always a function of the frequency of access of the object.

Pseudocode for origin server

```
1. While (no message );do nothing

2. If message==Request object{
send object to the client; TTL = max
}

3. if message==TTL Expired{
resend object to the client; TTL = max
}
4. if message==Object updated {
Update database;
```

```
}

5. Goto Step 1
```

Pseudocode for cache servers

```
1. /* Request object(s) from the origin server*/

      Send "Object request" to the origin server

2.  While (no transaction && no message); do nothing

3. If transaction == READ{
If (TTL == Valid){
return object;
reset TTL;
}

If (TTL == Expired){
 Send "TTL Expired" to origin server;
 Goto Step 3;
}
}

4. If transaction == WRITE{
update object in the local cache;
reset TTL;
Send "Object updated" to all other caches;
Send "Object updated" to the origin server;
```

```
}

5. If message == Object updated{
Update object in local cache; TTL = Max;
}

6. Goto step 2
```

5.1.3 Client polling

In the client polling scheme clients or the application servers poll the origin server for the validity of the object at regular fixed time intervals. In this, each client sends a 'poll-object' request to the origin server at fixed predetermined time intervals.

Pseudocode for origin server

```
1. While (no message );do nothing

2. If message==request object{
send object
}

2. If message == poll object; return object;

3. if message==Object updated {
Update database;
}

4. Goto Step 1
```

Pseudocode for cache server

```
1. /* Request object(s) from the origin server*/

      Send "request object" to the origin server

2. Set Timer = Polling interval

3.  While (no transaction && no message && Timer=Not Expired); do nothing

4. if transaction == READ {
return object;
}
5. if transaction == WRITE{
Update local cache;
send "object updated" to origin server;
}

6. If transaction == Timer Expired{
send "poll object" to origin server ;
update local cache ;
Reset Timer;
}

7.Goto Step 3
```

5.2 Approaches based on the strong consistency model

In the *strong consistency model*, a stale object is never returned to the user. We borrow the definition from [3].

5.2.1 Polling each time

In the polling every time approach the client servers send a 'poll object' request to the origin server, every time a request for an object hits the cache. The poll is to ask the database server if the cached copy of the object is still valid. Since every transaction invokes a poll, a stale object is never returned.

Pseudocode for origin server

```
1. While (no message );do nothing

2. If message==request object {
send object + serverTimestamp to the client;
}

2. If message == poll object{
if (serverTimestamp > ClientTimestamp); return "Invalid" and object
else; return Valid
}

3. if message==Object updated {
Update database;
update serverTimestamp;
}

4. Goto Step 1
```

Pseudocode for cache servers

```
1. /* Request object(s) from the origin server*/

      Send "request object" to the origin server
```

```
2.  While (no transaction && no message); do nothing

3. if transaction == READ {
send "poll object" + ObjectTimestamp to the origin server;
Wait for response;
if response == Valid; return object
if response == invalid{
update local cache;
update ObjectTimestamp = ServerTimestamp;
return object;
}
}
4. if transaction == WRITE{
Update local cache;
send "object updated" to origin server;
update objectTimestamp = CurrentTimestamp;
}
```

5.2.2 Server-side invalidation

In this approach, the origin server keeps track of all the clients that have cached the object. When the object is modified, the cache server informs the origin server, which in turn sends invalidation messages to all other clients. A write is considered complete when the invalidation messages reaches all the relevant clients. This ensures strong consistency.

Pseudocode for origin server

```
1. While (no message );do nothing

2. If message==request object {
send object
```

```
}

3. if message==Object updated {
Update database;
notify all caches by sending "object updated"
}

4. Goto Step 1
```

Pseudocode for cache server

```
1. /* Request object(s) from the origin server*/

      Send "request object" to the origin server

2.  While (no transaction && no message && Timer=Not Expired); do nothing

3. if transaction == READ {
return object;
}
4. if transaction == WRITE{
Update local cache;
send "object updated" to origin server;
}
5. If message == Object updated{
update local cache ;
}

6.Goto Step 2
```

5.2.3 Client-side invalidation

A stateless variation of the server-side invalidation is the client invalidation scheme in which the client on which the object is modified is responsible for informing all the other clients. As this scheme does not assume a centralized server exists, it is well suited for peer to peer networks.

Pseudocode for origin server

```
1. While (no message );do nothing

2. If message==request object {
send object
}

3. if message==Object updated {
Update database;
}

4. Goto Step 1
```

Pseudocode for cache server

```
1. /* Request object(s) from the origin server*/

      Send "request object" to the origin server


2.  While (no transaction && no message && Timer=Not Expired); do nothing

3. if transaction == READ {
return object;
}
```

```
4. if transaction == WRITE{
Update local cache;
send "object updated" to origin server;
send "object updated to all other caches;
}
5. If message == Object updated{
update local cache ;
}

6.Goto Step 2
```

6 Implementation

6.1 Testbed description

The basic testbed for simulating distributing caching was implemented on three workstations (PIII, 863Mhz) and an IBM xSeries 335 server(Intel Pentium 2Ghz) hosting a database management system (DB2 Version 7.0) which acts as the origin server. We simulate a distributed caching environment with a central server and a cluster of workstation serving as client servers.

6.1.1 The basic building blocks of the testbed

Origin Server The server on which the original content resides or is to be created is called the origin server. Whenever a cache server needs to cache an object, it contacts the origin server and retrieves the object to be cached. Also, whenever an object gets invalidated, the update needs to be propagated to the origin server. There is only one origin server in the environment and it hosts the central database for all the cache servers.

Cache Server These are the remote servers that interface with the end users. Each cache server implements an object cache which stores one or more objects. The cache server is responsible for maintaining the state of the object it caches. Whenever a transaction needs to access an object, the query is addressed by the nearest cache server that is maintaining the object. There are multiple cache servers attached to the origin server to simulate a distributed network. For simplicity, we do

not consider the problems of choosing the caching policy, i.e deciding which object to cache, or of cache replacement. We assume the cache size and caching policies to be independent problems on their own, we do not address them here.

Objects In our scenario, objects are Java objects encapsulating tuples of a database residing on the origin server database.

Cache Each cache server implements an object cache. The object cache is implemented as a Java Hash Table data structure [11]. Each object stored has a key and value pair. The key is generated on the primary key defined for the object in the database. Using this key, the lookup of the value of the object can be done in constant time.

Transaction A transaction is defined as a read or write/update request for an object. Transactions are simulated as SQL queries which either read or update an object.
An example of a read transaction is:

```
SELECT age FROM employee_table WHERE salary > 1000000;
```

An example of a write transaction is:

```
UPDATE employee_table SET dept='Accounting' WHERE dept='Auditing'
```

Fig 4 shows the layout of the testbed. The origin server is hosting the database. The rows of the table in the database are encapsulated as objects, they are referred to as A, B, C, D and E in the figure. The workstations serve as cache servers, each server has multiple instances of the cache running to simulate multiple cache servers. The objects are replicated over these instances. All the components are connected via a LAN to form a cluster.

6.1.2 Design assumptions and simplifications

- Cache size is infinite: For simplicity, we do not consider the problems of choosing the caching policy, i.e deciding which object to cache, or cache replacement. These problems are considered as complex independent problems and studied extensively in the literature. For simplicity we assume 2PL to be implemented to resolve concurrent transactions.
- We assume that there are lots of shared data items and lots of frequent access clashes on these data items.

- To minimize the number of workstations needed for simulating cache servers, in our network a single workstation is capable of mapping to many caches. This is required to simulate concurrent operations on multiple caches.
- We assume flat cluster based networks, as a result we expect the network size to be small no more than 20 cache servers.
- The network traffic is not considered as a criterion for evaluating performance, since the cluster size is never assumed to be very large.

6.1.3 Testbed design details

Cache Manipulator This is a Java class written to do operations on the local cache. Specifically it handles the following:

- **Cache initialization:** Creation of cache and allocation of the data structures. This is done at the time the cache is first instantiated on a workstation.
- **Filling the cache:** Once the cache has been simulated, the objects that need to be cached are retrieved from the origin server.
- **Object retrieval and updates:** When a transaction requests to read or update an object, the cache manipulator accesses the local cache and addresses these requests.

Cache Manager The cache manager is a Java class responsible for maintaining the cache consistency for the local cache and for sending invalidation and update messages to the rest of the caches and the origin server. It is designed to handle the following:

- **Write Invalidation Message:** This is sent to all cache managers for informing them of object invalidations as well as propagating the updated value. This is triggered when an update takes place.
- **Read Invalidation Message:** This is received by the cache manger informing it that the object in its cache is invalidated and that it needs to be updated.
- **Update Local Cache:** When the cache manager receives a *write invalidation message* from another cache manager informing it that an object in its local cache has been invalidated,

then the cache manager issues an *update local cache* message to the cache manipulator to update its current contents.

- **Update Database:** This message updates the database to the current updated value of the object. This message is sent only to the application server hosting the database. An update database message is triggered when an invalidation message is received.

Fig 5 describes how the cache manipulator and the cache manager modules interact with each other and with the database. The database resides on the origin server. The cache manager interacts directly with the cache manipulator and with the cache managers residing on other cache servers. It also sends and retrieves objects from the origin server. The cache manipulator is mainly responsible for maintaining the local cache and performs initialization, retrieval and updating of the cache.

6.2 Prototype design and implementation

Our prototype for the invalidation protocol is built on top of the testbed described above. For each scheme we studied, new methods/messages specific to that scheme are integrated into the testbed.

6.3 Evaluation Methodology

6.4 Terminology

In this and future chapters we shall use the following terminology:

- **U** denotes the number of updates.
- **R** denotes the number of reads.
- **T1** denotes the probability that an object expired in the TTL window.
  T1 = value of [U/(U+R)] in one timer window. This is derived from the fact that the expiry of an object is directly related to the number of updates in the transaction. Also, the probability that a stale object is returned to the user is proportional to the number of read transactions following an update in a timer window.
- **T2** is the number of "timer expired" messages received.

6.5 Parameters

6.5.1 Number of transactions

A transaction is defined as a read or update of an object requested by a query. All transactions are performed in batches and run together. This is indicative of the load run at the servers. The server load is classified as following

Low The number of transactions is between 1 to 10

Medium The number of transactions is between 10 to 100

High The number of transactions is greater than 100

6.5.2 Latency

This is defined as the end-to-end delay in milliseconds required to complete a bulk transaction by a cache server.

6.5.3 Server CPU load

This is the CPU utilization of the servers to complete a bulk transaction.

6.6 Evaluation

The schemes are evaluated mainly by measuring the performance in terms of average latency experienced while completing a bulk transaction. The transactions are in the form of SQL queries which either read or update an object. The SQL queries are batched together and run on a client server machine. The number of transactions in each batch is varied to measure the performance of the scheme under different load settings. Specifically, for each scheme the average latency values are recorded for low, medium and high number of transactions. The CPU utilization of the servers is also recorded during the processing of these transactions. The data points for the experiments can be found in Appendix A.

Scheme	Stateless	Stale object returned	No. of messages [1]
Fixed TTL	YES	MAYBE	U+T1*R
Adaptive TTL	YES	MAYBE	U+T1*R
Client polling	YES	MAYBE	U+T2
Polling each time	YES	NO	R+U
Server side invalidation	NO	NO	U
Client side invalidation	YES	NO	U

Table 1: Qualitative analysis of schemes

7 Experimental Results

Table 1 shows a qualitative comparison of the schemes we have studied. The TTL based schemes, i.e, fixed TTL and adaptive TTL, fall under the weak consistency model. If an object is requested before the TTL window expires, there is a chance that a stale object might be returned. The probability that this might happen can be calculated as U/(U+R) where U and R are the number of updates and reads received in one timer window. Client polling too can return a stale object if the value of the timer window after which it polls is greater than the frequency of the updates. The invalidation schemes – server-side invalidation and client-side invalidation fall under the strong consistency model as they push the invalidations as soon as an update occurs on the local copy. Of all the schemes only server-side invalidation is a stateful scheme, where the central server needs to remember which client cached which object. This may be appropriate for a centralized approach, where a master server has to keep a copy of the state of all the clients attached to it. In terms of the number of messages passed, we see that the invalidation schemes have the lowest number of traffic generated, as they send a message only when an update occurs. Whereas, other schemes like the TTL based schemes may also generate messages when the TTL expires.

The CPU measurements recorded in Table 2 were taken by averaging out samples of CPU bursts over the period of time corresponding with the completion of bulk transactions on the client machines. The CPU measurements are quite similar for the schemes, but the schemes where the updates are propagated to other clients via the central server have slightly higher CPU utilization. An example of this is server-side invalidation where the central server has the responsibility of pushing the invalidations to the other cache servers. In this scenario, if a bulk transaction occurs, the central server has to process each batched transaction and to take appropriate action.

Scheme	CPU Performance in %
Fixed TTL (100 ms)	75
Fixed TTL (1000 ms)	70
Adaptive TTL	65
Polling each time	75
Polling at fixed intervals (100 ms)	60
Polling at fixed intervals (1000 ms)	55
Server side invalidations	85
Client side invalidations	55

Table 2: CPU performance

Fig 6 compares the average latency time experienced while the client server completes a bulk transaction of 10 reads and updates. This is indicative of low load running on the server. Fig 7 compares latency time when a medium load is presented to the client machines. Fig 8 compares latency times for a high load on the client machines. We have compared three classes of schemes – TTL based schemes, push based schemes and pull based schemes. An investigation of the results indicates that the TTL based schemes and the push based schemes are very close in terms of performance, while the pull based schemes are much more expensive. This is especially true when the transactions occur in large bursts. In all scenarios, the client polling schemes are the most expensive. For the medium and high loads, the polling each time scheme has the highest latency. The rest of the client polling schemes are also more expensive than the others. The invalidation schemes in all low, medium and high load settings give similar performance. The client-side invalidation is slightly better than server-side invalidation. For low load, fixed TTL with higher values for TTL performs better than schemes with lower values for TTL and adaptive TTL. As the load increases, adaptive TTL performs better than the fixed TTL schemes.

Fig 10 combines results of Figures 6, 7 and 8. We see that polling each time performs the worst in terms of elapsed time. Intuitively polling each time does not perform as well as the other schemes is that it checks with the origin databases before answering any query. As a result, the total time to complete a transaction increases. This time delay is more noticeable in larger burst values as each transaction that it receives invokes a poll increasing the response time. This increase is the cause for the bad performance of the scheme. The rest of the client polling schemes also prove more costly than the other two classes of schemes. The TTL based schemes have their

performance tied with the length of the TTL field. A very large value may cause lower elapsed times but the probability that a stale object is returned is increased. Adaptive TTL shows steady performance throughout and maintains the number of trips to the database directly proportional to the frequency of updates. Invalidation schemes perform well, but continue to have the disadvantage that they need to maintain the state of other clients.

8 Case study

A online bidding site similar to E-bay.

This example depicts the set of basic operations designed to exercise transactional web system functionality in a manner representative of internet commerce environments. These basic operations have been given a real-life context, portraying the activity of a website that supports user browsing, searching and online bidding activity. The application portrayed is an online bidding store on the internet. Customers visit the website to look at the products, see the bids placed, make a bid, see the status of the bid and put items for bidding.

We specifically focus on the bidding of a single object. The item in question is available to all users for bidding till a certain period of time. For load balancing, the object and its properties, which include the highest bid value as well as the time left for the bid are replicated over several web servers maintained by the company. There is need to make sure that the copies are in sync with each, as different customers shall place their bid according to the present value of the highest bid attribute of the object. Hence there is need to update this value as soon as it changes to all the replicas of the object.

We design a series of transactions that update and read the bid value for the object. The efficiency of the schemes are measured by how fast the update propagates to all the other copies as well as whether any end user is being returned a stale value.

Transaction update to place a bid

```
update User 1.current_bid = current_bid
If current_bid > obj A.highest_bid then:
Update obj A.highest_bid=current_bid where highest_bid>current_bid
```

Transaction read to retrieve the current state

Scheme	Response time in ms
Fixed TTL = 100ms	49
Fixed TTL = 1000ms	47
Adaptive TTL	47
Polling each time	80
Client polling = 100ms	64
Client polling = 1000ms	63
Server side invalidation	47
Client side invlaidation	49

Table 3: Comparision of schemes

```
Read obj A.highest_bid


If obj A.highest_bid >= User 1.current_bid then return : you are highest bidder
Else return: you need to bid higher to win
```

8.1 Comparision of schemes

We implemented read and update transaction for each of the scheme evaluated here. The reads and writes were implemented on a single object. The reads implemented the scenario where users wanted to see the current value of the bid. Often, the users would want to place a new bid or update the value of their current bid. These transactions translated to an update transaction as described above.

The measurements were taken when multiple sites were trying to access a common object. The object was cached at the client servers against which the transactions were issued. The transaction were a combination of reads and writes directed towards the object. The order of the transaction was dependent on the order the transactions were recieved by the origin server. Table 3 describes the behaviour of the schemes in the context of a e-commerce application that make frequent reads and writes on an object. The trend we see is that the client polling schemes prove to be the most costly. The TTL and invalidation schemes are similar in terms of performance.

No. of Updates	No. of reads	Latency (ms)
5	5	47
10	10	78
100	100	781
500	500	3178

Table 4: Latency: fixed TTL (1000 ms)

9 Conclusions and Future Work

In our experiments we examined three basic classes of schemes: schemes based on timestamps, push-based schemes where the central server pushes invalidations, and pull-based schemes where the clients pull invalidations from the central server.

Our experiments show that in the three classes the scheme which always proves to be most costly is polling each time, which is a pull-based scheme. It performs 150% times worse than the other schemes in terms of average times. Also the values suggest that all push based schemes perform better than pull based invalidation schemes. The best performance is offered by timestamp based schemes. In timestamp based schemes we see as the number of transactions increases, adaptive TTL offers the best response time. We conclude that server side invalidation is highly recommended for small sized networks as it is guaranteed to never return a stale object. But if maintaining the state of clients in the server is a problem then adaptive TTL is the best alternative and is recommended over fixed TTL approaches. As per [9], adaptive TTL returns a stale object five out of hundred times.

The next step could be to design a scheme that combines the benefits of adaptive TTL and server-side. As this work focuses on small sized cluster networks, another direction of interest is to study the behavior for larger networks.

A Data Points for Latency Measurements

Here we discuss raw data points we obtained for latency measurements in our experiments. The elapsed time is reported in milliseconds. For all the schemes, latency values are measured for low (5 reads and 5 updates), medium (20 reads and 20 updates) and high (100 reads and 100 updates) loads. The data is presented in tabular form for all the schemes and is self explanatory.

No. of Updates	No. of reads	Latency (ms)
5	5	63
10	10	93
100	100	875
500	500	3153

Table 5: Latency: fixed TTL (100 ms)

No. of Updates	No. of reads	Latency (ms)
5	5	63
10	10	79
100	100	844
500	500	3040

Table 6: Latency: adaptive TTL

No. of updates	No. of reads	Latency (ms)
5	5	110
20	20	515
100	100	4672
500	500	11547

Table 7: Latency: polling each time

No. of updates	No. of reads	Latency (ms)
5	5	78
20	20	140
100	100	2453
500	500	7016

Table 8: Latency: client polling with time interval = 10000ms

No. of updates	No. of reads	Latency (ms)
5	5	125
20	20	165
100	100	2985
500	500	5125

Table 9: Latency: client polling with time intervals = 100ms

No. of updates	No. of reads	Latency (ms)
5	5	63
20	20	141
100	100	813
500	500	2797

Table 10: Latency: server side invalidations

No. of updates	No. of reads	Latency (ms)
5	5	62
20	20	140
100	100	782
500	500	3069

Table 11: Latency: client side invalidations

References

[1] Websphere application server. http://www-306.ibm.com/software/websphere/.

[2] Kyle Brown. http://members.aol.com/messagingpattern/distributedcacheupdate.pdf.

[3] Pei Cao and Chengjie Liu. Maintaining strong cache consistency in the world wide web. *IEEE Transactions on Computers*, 47(4):445–457, 1998.

[4] Vincent Cate. Alex – A Global File System. In *Proceedings of the USENIX File System Workshop*, pages 1–11, Ann Arbor, Michigan, 1992.

[5] J2EE concepts. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/.

[6] D. Dai and D. K. Panda. Reducing cache invalidation overheads in wormhole routed DSMs using multidestination message passing. In *Proc. of the 1996 Int'l Conf. on Parallel Processing (ICPP'96)*, volume 1, pages 138–145, 1996.

[7] Livestore data cache. http://www.isocra.com/livestore/index.php.

[8] Bluestone Total e server. http://h21022.www2.hp.com/HPISAPI.dll/hpmiddleware/products/Total-e-Server/default.jsp.

[9] James Gwertzman and Margo I. Seltzer. World wide web cache consistency. In *USENIX Annual Technical Conference*, pages 141–152, 1996.

[10] Hideki Hayashi Takahiro Hara and Shojiro NISHIO. Cache invalidation for updated data in ad hoc networks. *CoopIS/DOA/ODBASE*, pages 516–535, 2003.

[11] Java hash map implementation. http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashMap.html.

[12] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[13] P. K. McKinley, H. Xu, A.-H. Esfahanian, and L. M. Ni. Unicast-based multicast communication in wormhole-routed direct networks. *IEEE Transactions on Parallel and Distributed Systems*, 5(12):1254–1265, December 1994.

[14] Markus Pizka Oliver Theel. Distributed caching and replication. *32nd Annual Hawaii International Conference on System Sciences*, 8, 1999.

[15] R. S. Hall P. B. Danzig and M. F. Schwartz. A case for caching file objects inside internetworks. *In Proceedings of SIGCOMM '93*, pages 239–248, 1993.

[16] Guillaume Pierre. A web caching bibliography. citeseer.ist.psu.edu/pierre00web.html.

[17] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[18] Structured query language. http://www.sql.org/.

[19] Pablo Rodriguez and Sandeep Sibal. Spread: Scalable platform for reliable and efficient automated distribution. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(6):33–49, 2000.

[20] SilverStream Application Server. http://www.novell.com/products/extend/.

[21] Sybase Enterprise Application Server. http://www.sybase.com/products/ developmentintegration/easerver.

[22] Renu Tewari, Michael Dahlin, Harrick Vin, and John Kay. Beyond hierarchies: Design considerations for distributed caching on the Internet. In *IEEE ICDCS'99*, 1999.

[23] Kurt Jeffery Worrell. Invalidation in large scale network objects caches. Master's thesis, 1994. citeseer.ist.psu.edu/worrell94invalidation.html.

[24] www.tangosol.com. http://www.tangosol.com/coherence-featureguide.pdf.

[25] Jian Yin, Lorenzo Alvisi, Michael Dahlin, and Calvin Lin. Volume leases for consistency in large-scale systems. *Knowledge and Data Engineering*, 11(4):563–576, 1999.

[26] Jian Yin, Lorenzo Alvisi, Mike Dahlin, and Calvin Lin. Hierarchical cache consistency in a WAN. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[27] Haobo Yu, Lee Breslau, and Scott Shenker. A scalable web cache consistency architecture. In *SIGCOMM*, pages 163–174, 1999.

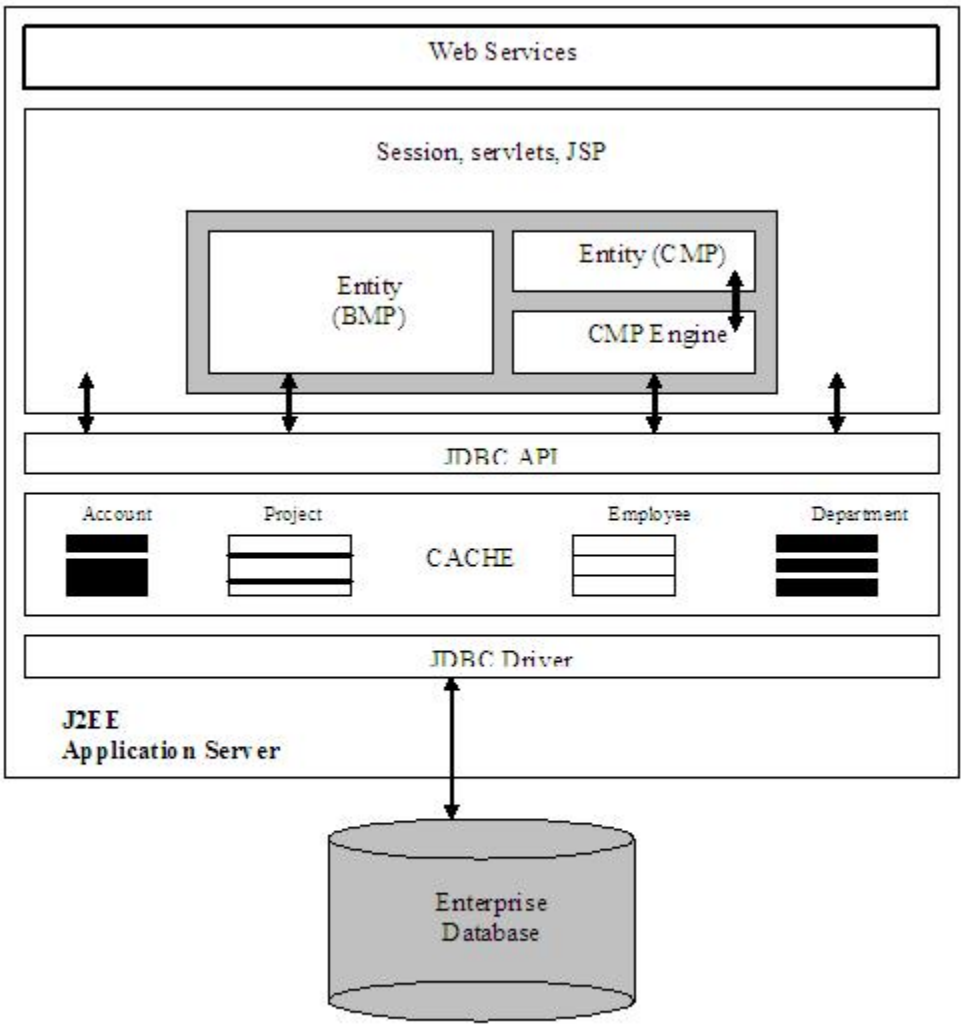

Figure 2: J2EE configuration with caching

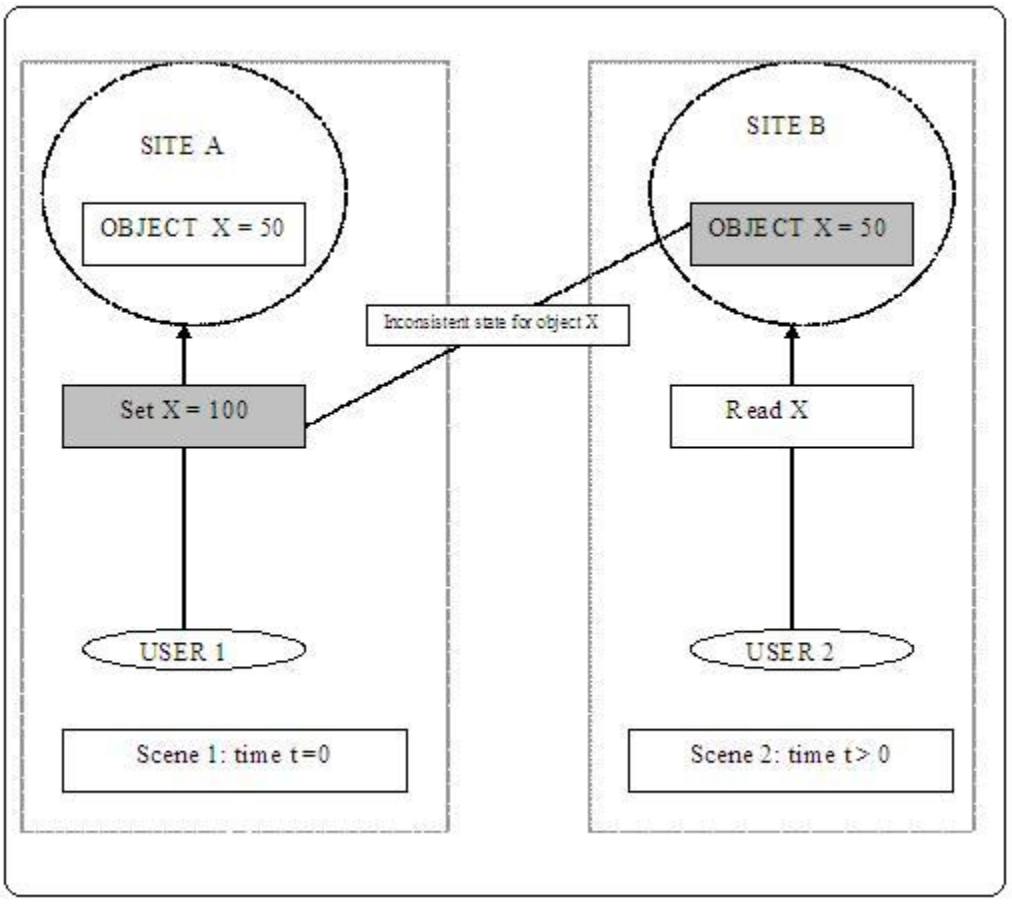

Figure 3: Write - read scenario

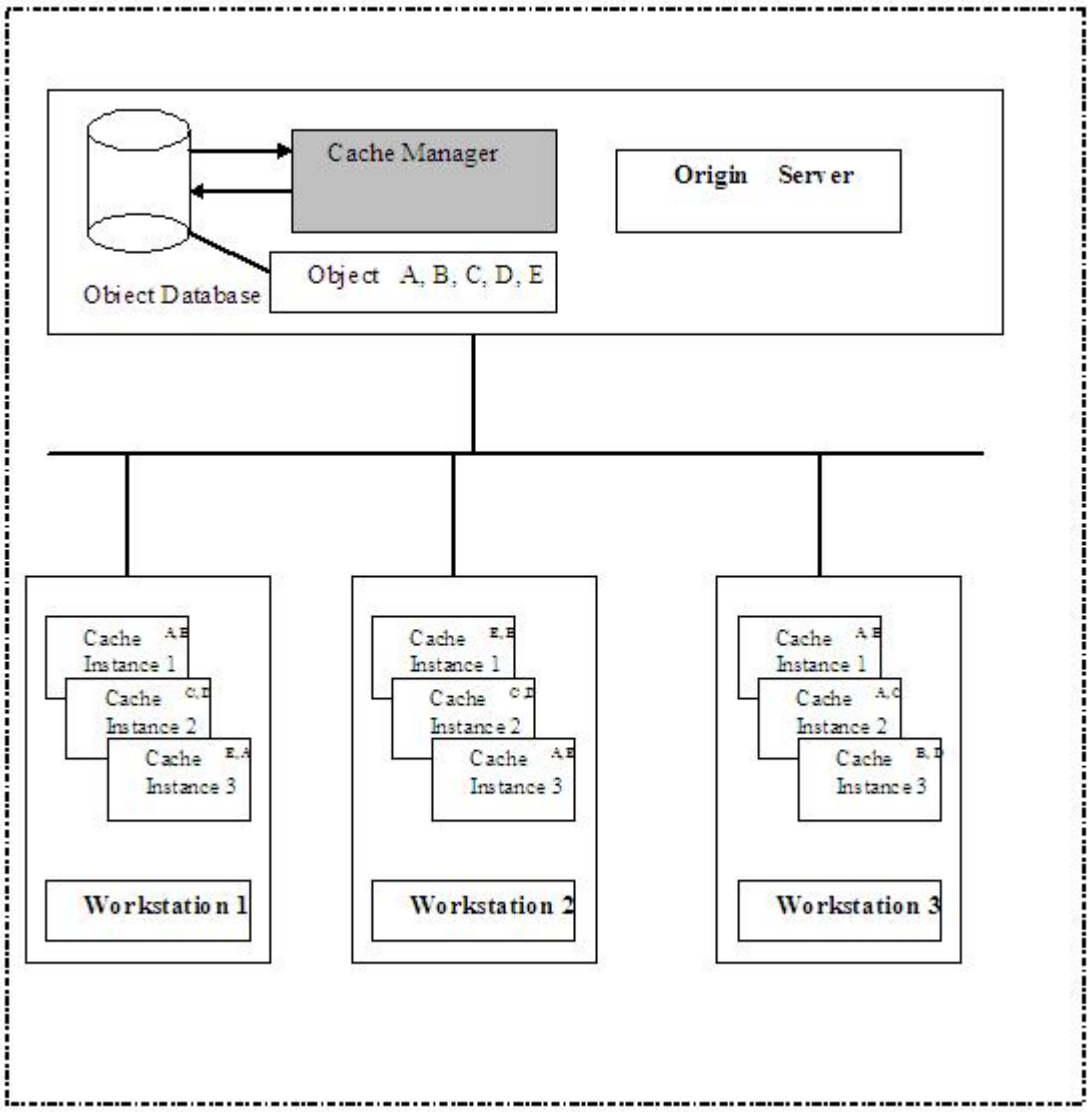

Figure 4: Layout of the testbed

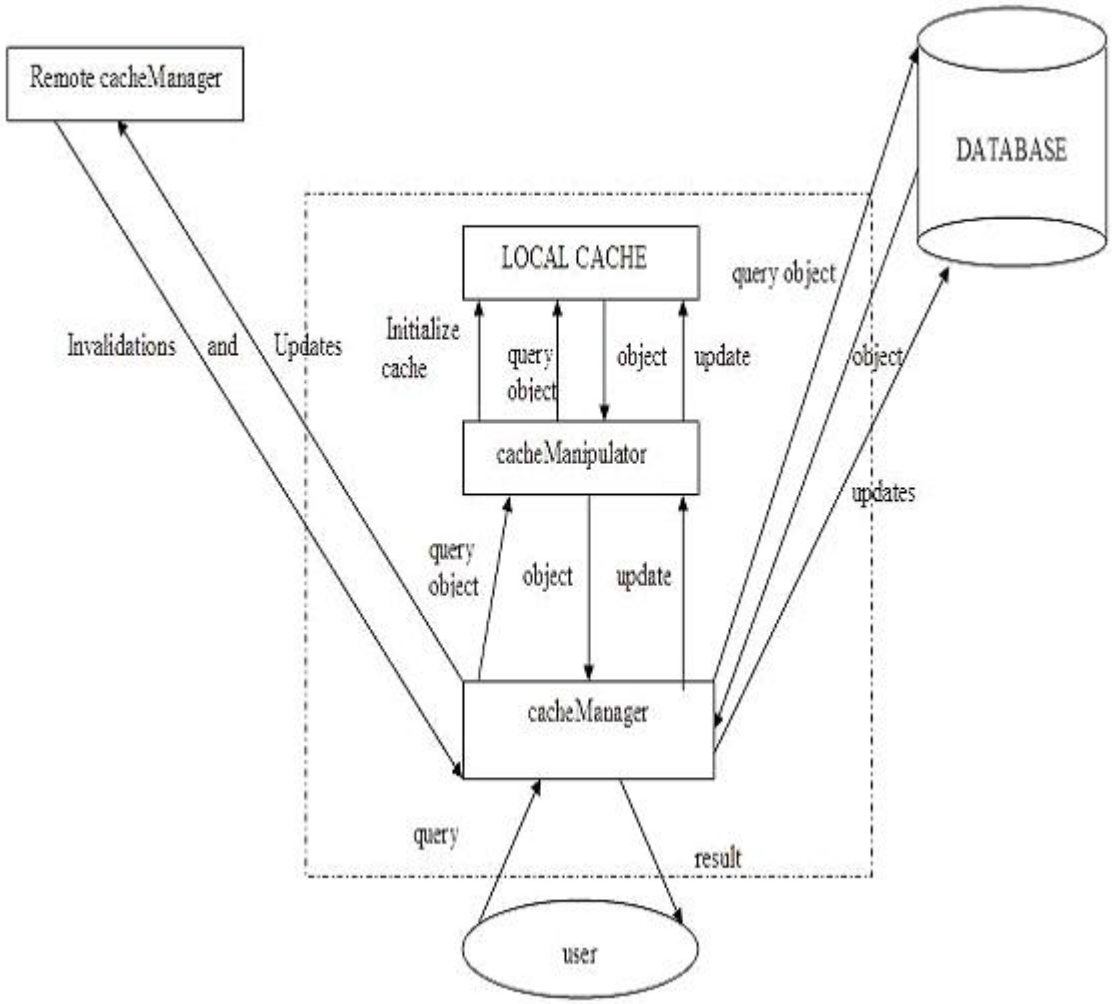

Figure 5: Interprocess communication

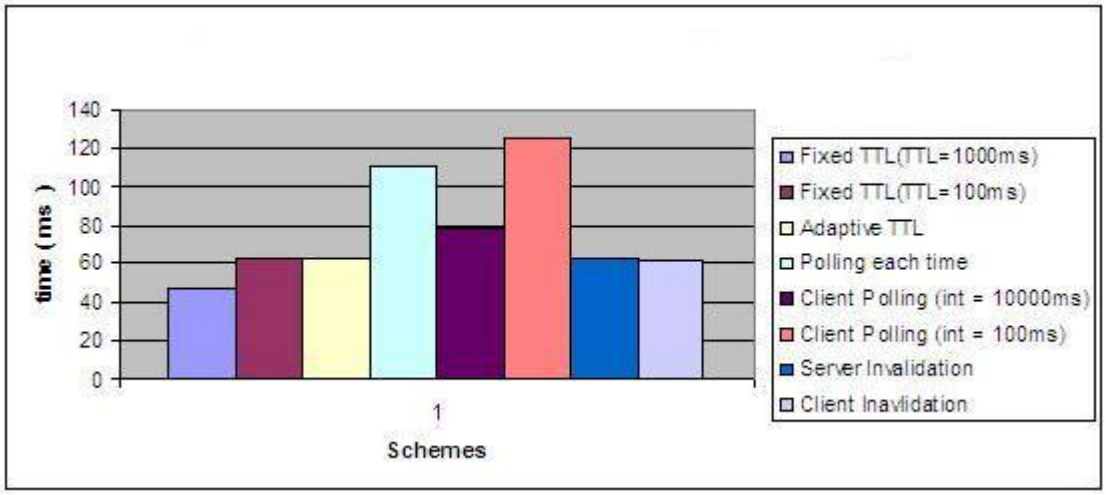

Figure 6: Comparison with low bulk load: 5 reads and 5 updates

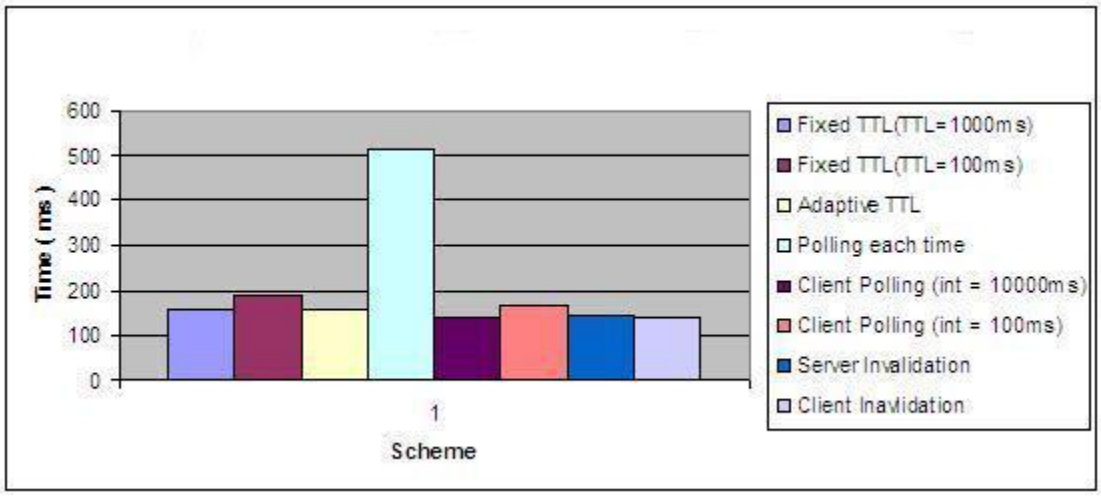

Figure 7: Comparison with medium bulk load: 20 reads and 20 updates

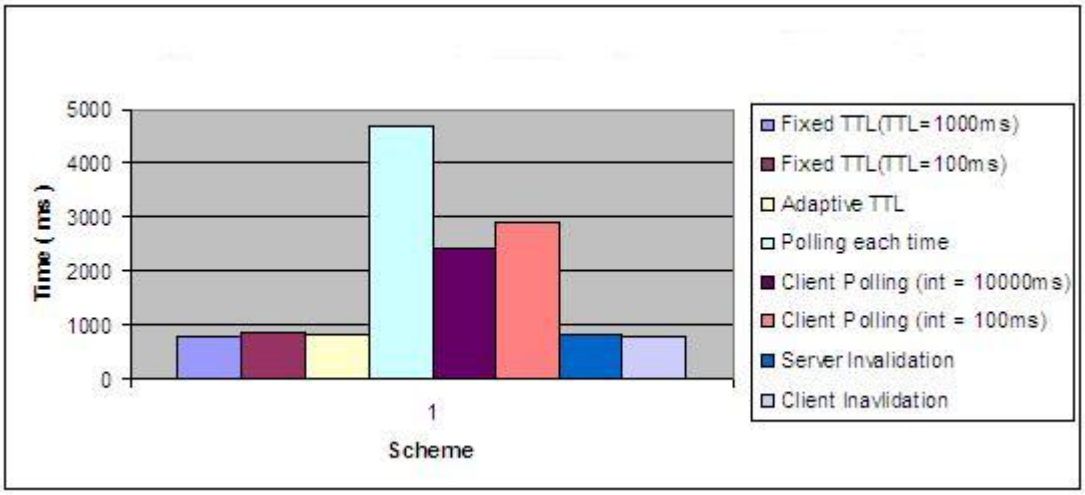

Figure 8: Comparison with high bulk load: 100 reads and 100 updates

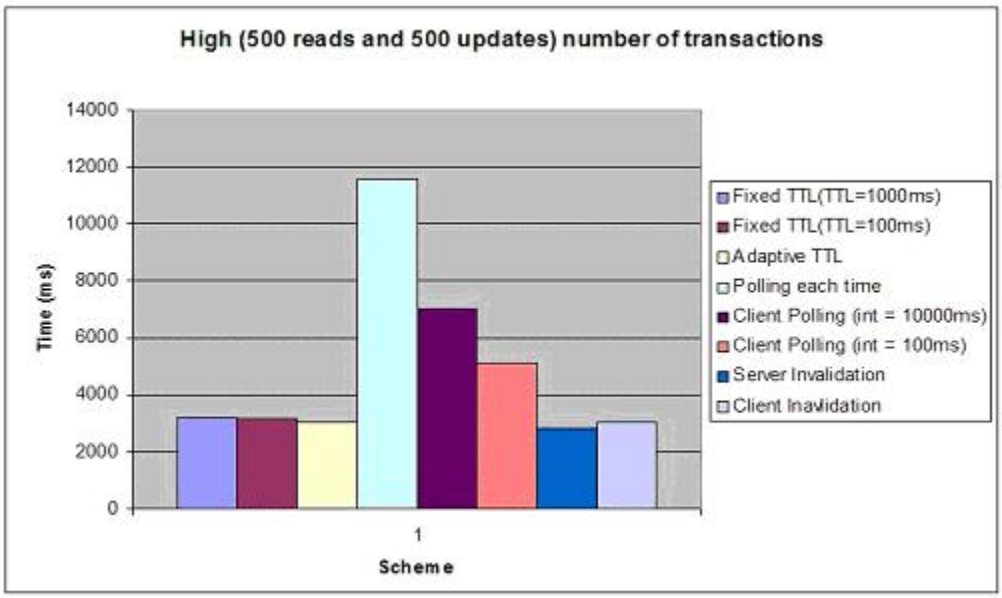

Figure 9: Comparison with high bulk load: 500 reads and 500 updates

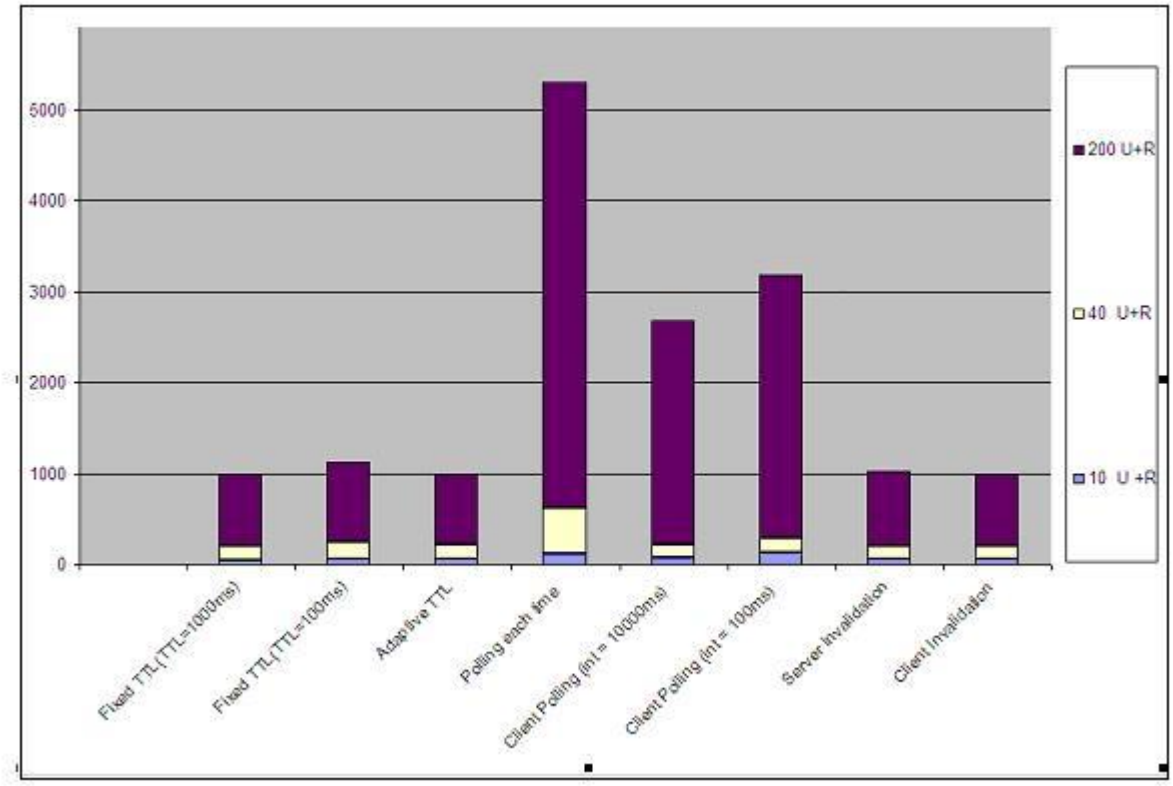

Figure 10: Cumulative results