International Conference on Computing, Communications and Control Technologies

August 14-17, 2004 - Austin, Texas, USA

PROCEEDINGS

Volume I

Edited by

Hsing-Wei Chu
Michael Savoie
Belkis Sanchez

THE UNIVERSITY OF TEXAS AT AUSTIN
Center for Lifelong Engineering Education

Sponsored by
The University of Texas at Austin
and organized by
International Institute of Informatics and Systemics
Member of the International Federation of Systems Research (IFSR)

IIIS

CCCT'04
International Conference on Computing, Communications and Control Technologies

VOLUME I

CONTENTS

A Wrapper-based Object Caching in CORBA

Uttam Kumar Roy
Information Technology, Jadavpur University Salt Lake Campus
Block-LB, Plot-8, Sector – III, Kolkata-700 098, India

and

Samiran Chattopadhyay
Information Technology, Jadavpur University Salt Lake Campus
Block-LB, Plot-8, Sector – III, Kolkata-700 098, India

ABSTRACT

It is necessary to improve quality of service (Qos), object-oriented middleware such as CORBA provides to support applications that require high degree of performance. However, traditional CORBA implementations incur high latency and low scalability. Caching is an important technique to enhance performance, letting clients to access distributed objects locally instead of accessing them remotely.

In this paper, we have provided a unified client/server object caching architecture and implementation. Our architecture uses object wrapper instead of interceptors to achieve a finer control on method invocations. A salient feature is that it can be used to cache different types of remote objects with a little modification of client and server code. This approach is based on cache-invalidate cache consistency policy, per process caching, object graph based data shipping and replication management. We have applied our architecture on applications with different types of system requirements (such as strong consistent, casual consistent and read only system) and made a comparison among them. We have demonstrated that the proposed approach of caching results in a significantly improved performance for applications that exhibit a reasonable amount of read operations.

Keywords: *CORBA, Distributed System, Caching, Consistency, Object Graph Serialization, Object Wrapper.*

1. INTRODUCTION

In the domain of distributed object computing, Common Object Request Broker Architecture (CORBA) provides several advantages such as programming language, operating system, network protocol, location transparencies, over existing protocols (e.g. RMI, socket). CORBA provides these transparencies by performing remote invocation for every request causing excessive marshalling-demarshalling, and overuse of network bandwidth. Examples of systems that need to deal with interoperability and heterogeneity issues, experience communication latency inherent in the CORBA architecture as well as software overheads associated with messages that are exchanged when a client node invokes methods of an object that is implemented by a remote server. Caching objects lets clients to invoke methods on distributed objects locally instead of invoking them from remote servers. However, to our knowledge, the CORBA [7] specification does not address object caching. By storing (i.e. caching) objects locally at the point of usage, communication bandwidth, excessive messaging and unnecessary marshalling-demarshalling can be reduced significantly and hence response time may be shortened since communications with remote servers are no longer necessary.

In this paper we have attempted to improve the performance of operations on objects implemented remotely as efficient as operations on locally implemented objects. This paper describes a general architecture for caching objects (or object graphs) with little modifications of the code using object wrapper that improves performance for applications. Depending upon the application requirements different cache consistency strategies have been adopted leading to more or less optimal cache exploitation ([21]).

In order to implement caching we have developed an architecture for caching that has several features. First, the architecture is designed to be used to implement caching with coherency for many types of objects. This is important because new types of objects can be introduced in the system at any time and we want to make the benefits of caching easily available to implementations of theses objects. This is different from systems in [9,17,11,18].

The second feature is that the caching and consistency modules have been isolated from the object, client and server implementations. Our caching architecture is completely transparent to client applications while consistency is transparently maintained without the knowledge of the server. If an application program happens to use an object that supports caching, then the object will be automatically cached, without the application needed to be aware of it. Applications are expected to work without the caching facility as before.

Third, in addition to the client side caching, server side caching has been proposed expecting further reduced response time whenever the server uses a remote database. It can be noted that server side caching facility is optional and for read operations it does not have any effect on access time if database server is located in the caching server.

Fourth, to accommodate large number of clients in a single node, cache replacement policy (LRU) has been used expecting high percentage of hit ratio for a moderate cache size.

Our architecture can cache directed object graph (a pictorial description a an object and its references to other objects) instead of caching standalone object expecting greater hit ratio. A hit ratio of 94.16% has been obtained for 6% cache size.

A Roadmap: This paper is organized as follows. The problems involved in building a caching system, possible solutions and the one we adopted are discussed in Section 2. We present an overview of our Architecture in Section 3. Section 4 contains the implementation details. We give experimental performance results in Section 5. The future research direction has been given in Section 6. We conclude the paper in Section 7.

2. ISSUES TO BUILD A CACHING SYSTEM

This section deals with the general issues that need to be addressed in building a distributed object caching system with various possible solutions.

Caching

To reduce the frequency of network interactions ([5, 1, 2, 17, 16]), caching lets clients to invoke methods on distributed objects locally instead of invoking them from remote servers.

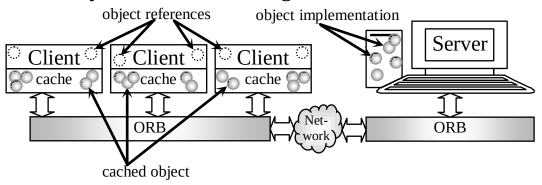


Figure 2.1: Object Caching Model

Figure 2.1 shows general object caching model in a distributed environment that uses CORBA as the middleware. A cache is essentially a container of the object, which resides with users, or physically closer to them than the original objects, e.g., within a separate server in the case of the Web [1,10]. Clients use the cache rather than the original object(s) as illustrated in Figure 2.1, to reduce the latency in accessing (possibly many) distributed objects. Obviously certain clients may continue to use the original object. The cache is typically populated on demand by obtaining a copy of the object either from the original location or another cache.

Cache Replacement

Whenever the cache is full and an object faulting occurs some object needs to be replaced. Several cache replacement algorithms (e.g. FIFO, LRU, LFU etc.) have been implemented in the systems from processors to Web. These policies exploit data localities (temporal or spatial). We can take into account many other parameters—such as frequency of use, download time and document size—for placement and replacement ([2]) policies. Thus, we can obtain more complex policies such as the LRU+ ([19]), LRU-MIN, LRU-HOLD.

Some distributed applications are different from the above systems in the fact that the locality of reference holds in a different manner. Examples include banking, railway reservation, airline reservation, hospital system, and University information system etc. where user requests come following a different locality of reference pattern for which more sophisticated cache replacement policies have to be devised. We have seen that LRU policy gives ~95% hit with 4% cache size.

Sharing Mode

Sharing mode can be classified according to the number of copies of the same data allowed in different nodes as follows.

- Single Reader/Single Writer (SRSW)
- Multiple Reader/Single Writer (MRSW)
- Multiple Reader/Multiple Writer (MRMW)

Proxy systems generally use MRSW approach, which is also used in the DSM systems; newer DSMs increasingly use the MRMW approach ([2]). Because the coherence maintenance solution strongly depends on the number of writers, coherence maintenance mechanisms for multiple writers are more complex and expensive.

Consistency

Systems such as banking need strong consistency where slightest inconsistency is not tolerable. There are some applications where the system is allowed to be inconsistent temporarily but the system should not be inconsistent for a long time. Railway reservation system is an example of such a type. Other systems such as University information systems where students read the information from the university database but they will not (rather are not allowed to) fire any write (update) operations, are read-only systems that do not require consistency-maintaining mechanism.

Many cache protocols employ a "*write-through"* policy, whereby updates, which are applied to the cache, are sent to the original object first. The server, in turn, *updates/invalidates* the other replicas that reside in different clients cache. The performance of updating or invalidating depends upon the object size being updated/invalidated, frequency of the cached object access, percentage of read operations etc. In the case of the Web, updates can only be made to the original object (document), and caches are not informed of updates; each document has a "*time-to-live*" associated with it and a cache will only fetch a new copy if it believes its current copy is out-of-date. Strict consistency models typically increase the number of coherence operations and consequently the network traffic. Different approaches are strongly linked with the system granularity.

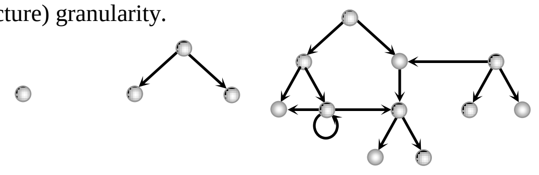
Consistency Initiation

Coherence policy determines how to keep all the shared data copies up to date. Responsibility can be imposed on the server (object implementer) or the client (object user) side. Both have their own set of advantages and disadvantages. If implementer takes the responsibility, more load on the server and hence longer the response time. On the other hand, designing a strict consistent system is more difficult if we impose consistency mechanism on the user.

We have implemented strict consistent system imposing consistency-maintaining mechanism on the object implementer while for weak consistent system.

Granularity

The size of basic data-item used to cache is generally referred to as "*granularity".* Granularity level can be one byte, a word, a page or a data structure. Hardware DSM solutions typically support smaller grain size (word or byte) and software DSM solutions support coarse grain size (page or complex data structure) granularity.



i) Stand alone *ii) Simple* *iii) Complex*

Figure-2.2: Different types of object graph

In our implementation, we use the coarse grain size for the following reason. Transporting only the state of the object is not enough. The server code that operates on the state may also be downloaded to the client machine. Again, it may so happen that an object to be serialized has references to other objects, which, in turn, have references to still more objects. This set of objects and the relationships among them form a directed graph (Figure 2.2).

The object "*serialization"* and "*deserialization"* facilities should be designed to work correctly in these scenarios. If we want to serialize an object at the top of an object graph, all the other referenced objects are recursively located and serialized. Similarly, during the process of "*deserialization"*, all of these objects and their references are correctly stored.

3. PROPOSED ARCHITECTURE

The following section describes various components of the proposed architecture and how they address all issues discussed in the previous section. Figure 3.1 shows the operation of an object wrapper, which is used for maintaining consistency.

Object Wrapper

Object wrapper feature allows us to define methods that are called when a client application invokes a method on a bound object or when a server application receives a request.

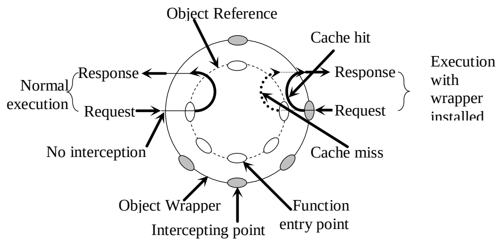


Figure3.1: Operation of an Object wrapper

Unlike the interceptor ([13]), which is invoked at the ORB level, object wrappers are invoked before an operation request has been marshaled. The difference is that here we can have finer control on the method invocations in the fact that only the necessary request can be intercepted.

Once an interceptor has been installed all the intercepting point (*preinvoke, posrinvoke* etc. in Visibroker CORBA) will be executed whether necessary or not. On the contrary a specific set of intercepting points can be installed (figure 3.1) using object wrapper as needed that eliminates unnecessary function call that can otherwise be unavoidable using interceptor. Any number of object wrappers can be installed in any side (client/server) depending upon the requirements.

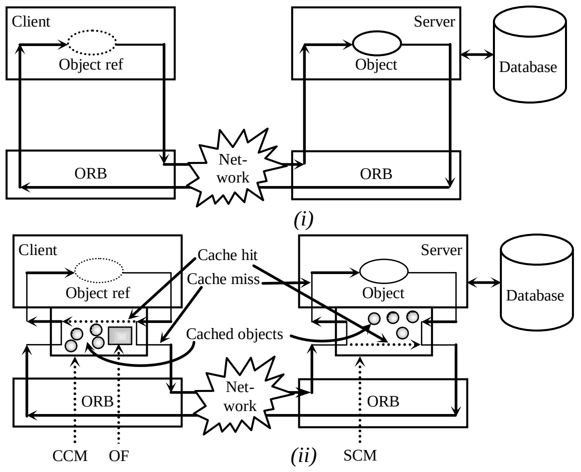
Two object wrappers--CCM and SCM, have been introduced in our architecture. CCM is responsible to detect object faulting, object shipping from the server side to the client side, object replacement (when the cache is full) etc. while the SCM takes care of the consistency mechanism. SCM can also be used to use caching facility in the server side.

Our architecture comprises of following four components:

- *Client Cache/Consistency Manager (CCM)*
- *Server Cache/Consistency Manager (SCM)*
- *Object Factory (OF)*
- *Implementer (Server)*
- *User (Client)*

Client Cache/ Consistency Manager

This module is responsible to managing the cache in the client side. For weaker consistent system CCM also takes the responsibility to maintain consistency. CCM, wrapping around the remote object reference in the user, intercepts the user request and searches the requested object in its local cache. If there is a hit, CCM serves the query locally without any network interactions. If the object is not in the local cache it consults SCM in the server side to get state of the requested object. CCM constructs the object locally from the serialized data sent by the SCM using Object Factory (OF), puts the object in the local cache and sends a "*register*" request to the SCM. If the request is a write/update operation, the request is sent transparently without any interception, to the sever side. CCM sets valid flag for the object to "*false*" upon receiving "*invalidate*" message from the SCM. While putting the object in the local cache CCM checks the cache's state. If it is full it removes some objects depending upon the application specific cache replacement policy and sends "*deregister*" message to the SCM telling that it cache no longer contains the object. If a user specifies weak consistency requirements, CCM creates a separate thread that updates all the object replicas in its cache after certain regular intervals (e.g. 10 sec.) or whenever necessary. Frequency of cache update specifies the degree/tightness of consistency. By decreasing interval time between two updates, less weak consistent system can be obtained. To obtain strong consistent system, responsibility can be imposed on the SCM rather than CCM.



CCM: Client Cache Manager
SCM: Server Cache/Consistency Manager
OF: Object Factory

Figure 3.2: Proposed architecture (i) without caching (ii) with caching

Server Cache/ Consistency Manager

SCM module is responsible to make the system strongly consistent. While CCM puts the object in the local cache and it sends a "*register*" request to the SCM, SCM in turn registers the CCM for the object. SCM in the server side intercepts the write operation request and sends the request (for strict consistency) to the object implementation and also sends the "*invalidate*" message to all the CCM registered for this object.

While CCM is used for client side caching, SCM can be used for server side caching. Often, server side object is populated by the data from some database. For heavily loaded server database, access time is not negligible. A similar caching concept can also be introduced in the SCM. SCM also maintains a local cache. While a request comes in the server side, SCM intercepts the request. It then checks whether the requested object in its local cache. If the object is in its local cache, it sends the response with the local copy without the database access. If it is not there, it fetches the data from the database, creates an empty object populating by the data it has just fetched and puts the object in its local cache. If the cache is full, SCM replaces some object depending upon the specified policy.

Note that server side caching policy is optional and is used for a specific set of applications mainly for applications where database is used.

Object Factory (OF)

OF (Object Factory) constructs an object from the serialized data. CCM/SCM uses this module whenever an object faulting occurs. When an object faulting occurs CCM sends a request to the SCM to return the serialized data containing the state of the object. After getting the raw data from the SCM, CCM gives the data to the OF. OF in turn constructs the appropriate object and returns it to the CCM. Java's serializations procedure has been used in our architecture.

The time required to serialize and reconstruct the object has a significant effect on response time for coarse-grained object. Similarly, for fine-grained object, more the cache miss, more the network access and hence the latency.

Implementer (Server)

This module is the implementer of the CORBA object. It creates a CORBA object, adds the consistency manager for this object and exports the object. A little modification of the server code is needed to add SCM. The SCM module can be added at runtime also so that the implemented methods can be intercepted.

User (Client)

User of an object first gets a remote reference of the CORBA object it want to use, installs the CCM module and invokes the methods transparently as if the object is locally available. CCM module intercepts the installed methods and then takes necessary actions as described earlier.

4. IMPLEMENTATION

This section describes an implementation of the architecture proposed in the previous section using Visibroker CORBA. We considered a bank application scenario with multiple read and update operations. A typical scenario involves an instance of a bank manager that creates account objects by opening accounts and exports that for the clients to use. An object wrapper(SCM) around the account manager is installed in the server side that notifies when the users update their accounts and the SCM sends requests to the registered CCMs to invalidate their cached copies. CCM is the object wrapper around the account manager object reference in the client side. It intercepts the read request and returns the result after invoking the read method on the objects in its local cache. If the object is not in the local cache it first registers with the SCM by sending a *"register"* request message. CCM, then gets the serialized data from the server object by calling *"serialize"* method, creates the objects locally using OF and puts the object in its local cache. It replaces an object using LRU policy if the cache is full.

To obtain object graph level granularity we have used Java's serialization procedure where the entire object graph can be shipped from the server side and can be reconstructed in the client side.

To provide caching service, a little modification in the object implementation is needed. The object that we want to cache must be created by extending the "*CachebleObject"* class that has a method "*serialize"*. The object must implement this method. CCM uses this method to get the serialized data and creates the objects locally with the help of the Object factory.

Caching

In our implementation, process level cache (requires less cache memory but more complexity in the Operating System call and greater overhead with respect to time) management has been used that uses Hash table (which will be main memory in Java during execution) as the cache memory. Note that since we did not have direct access to the cache memory, we have just used the main memory to get the simulation effect. CCM and SCM maintain their own Hash tables to cache the objects. It can be noted that for actual cache memory, access time would have been significantly smaller compared to main memory.

Cache Replacement

In our Implementation, LRU cache replacement policy has been used. Whenever an object is referred that resides in the cache, the current system time is attached with the page containing the object. The oldest page is replaced whenever the cache is full. We expect that the LRU cache replacement policy would give the best result. As mentioned earlier that a more sophisticated replacement policy (which we left for the future) has to be devised for practical applications that does not follow traditional locality of reference.

Sharing Mode

Number of readers/writers can also vary greatly with respect to an application type. As for example, for railway reservation system, an enquiry about a train may occur from a number places while enquiry about the balance of a bank account number will perhaps be from one place. In general, performance of an algorithm greatly depends upon the number of readers/writers. In this paper, we have implemented MRMW sharing mode. As the number of readers/writers increases, the performance decreases for a fixed percentage of read operation. We have shown that the rate of the increase of accessing an object decreases with the increase of percentage of read.

Consistency Maintenance

Again, we have implemented (i) a system with strong consistency, (ii) a system that is weakly consistent and (iii) a read only system. As mentioned earlier, better performance can be obtained with the cost of consistency while read only system gives the best performance and is invariant to the increase in number of reader/writer. CCM maintains the system weakly consistent while strong consistency is maintained by the SCM. In addition to the main thread, an extra thread is created to update its cache that updates its local cache after regular interval (e.g. 10 sec.). Interval time has a significant effect on the response time. If we increase the time interval, degree of inconsistency increases but network overhead decreases. Cache size also has an effect on response time. As the cache size increases, more the cache hit and less object serialization and reconstruction time but more the cache updating time. SCM maintains the system strict consistent. Depending upon the consistency requirements, CCM or SCM can be configured. We have shown that in case of weakly consistent system, performance increases with the increase in tightness in consistency.

Consistency Initiation

For weakly consistent system the consistency is initiated by the CCM. CCM updates its cache content in the regular interval. This interval time should be very long than the average interval between two read/write accesses. If the CCM updates its cache content frequently, more tight consistent system is obtained while the performance degrades, because the system remains busy to update its cache most of the time.

Granularity

We have implemented object graph level granularity for object shipping. Object graph level granularity provides robust solutions whereas data or single object level granularity provides better response time. Java's serialization procedure, which is responsible to sip the entire object graph recursively, has been used here taking the factors mentioned earlier into account. Not that the entire object graph may not always be needed and need not be cached. A possible solution may be partial object graph serialization that is yet to be implemented.

5. PERFORMANCE ANALYSIS

In this section we show some measurements of the performance with respect to different consistency policy and different cache capacity (cc). All measurements were carried out using average over large number (~4000) of iterations.

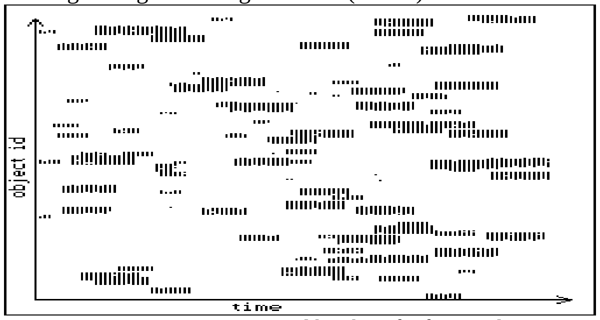


Figure 5.1: generated locality of reference data

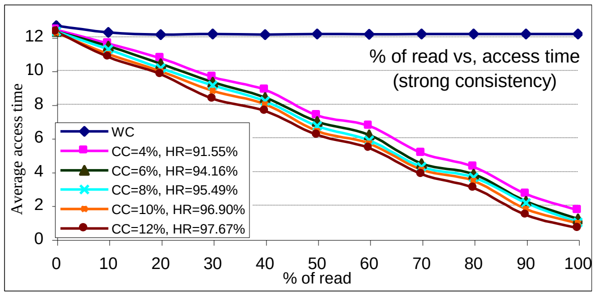


Figure 5.2: Performance of strict consistent system.

First, we describe the experimental setup. The experiments were done on HP & IBM running Windows2000 Server and IBM PC running Windows98. The stations are connected by 10Mbps Ethernet. The server is started on hp tc2100 server (1.13 GHz) that accesses the Oracle 9i database running on IBM server (xSeries 220, 1.2 GHz) and clients are started on IBM pcs. The clients then makes the CORBA calls. We have generated data set satisfying locality of reference. A typical reference pattern has been shown in Figure 5.1.

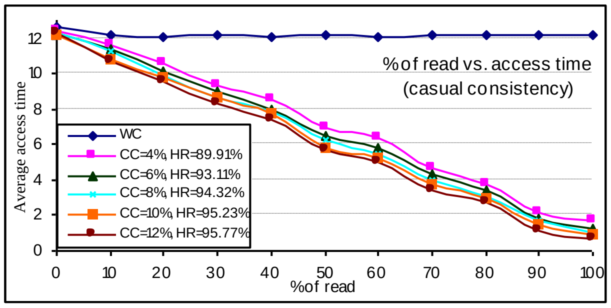
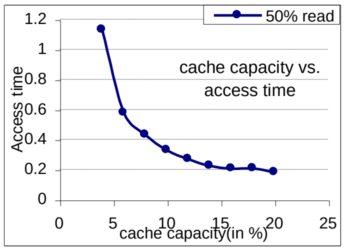


Figure 5.3 Performance of strict consistent system.



WC=Without Caching
CC=Cache capacity
HR=Hit Ratio
noc=number of clients
LRU=Least Recently Used

Abbreviations used

Figure 5.4 Effect of cache capacity on access time.

Figure 5.2 and Figure 5.3 show the access time vs. percentage of read operations using LRU replaced policy for strict and casual consistent system respectively.

In Figure 5.4, the effect of cache size on access time has been shown. The access time increases exponentially for smaller cache capacity and it is 30 times faster than the actual (without caching) time for 6% cache capacity.

We now present the timings related to casually consistent objects in Figure 5.3. The increased frequency of update operations fired by the clients can increase (even it may be quite large compared to original case) access time. A moderate interval time (~10sec) gives significantly less access time. Here also, access time is very less than the case for without caching for a significant amount of read operations (>10%). We have got a hit ratio 93.11 for 6% cache size using a locality of reference pattern as shown in Figure 5.1.

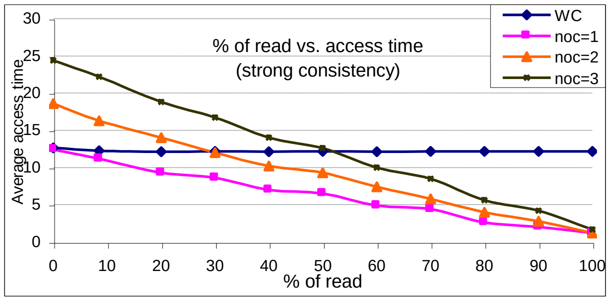


Figure 5.5: Effect of number of clients on access time.

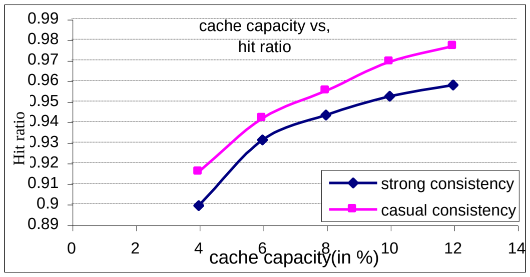


Figure 5.6: Effect of cache capacity on hit ratio.

Figure 5.5 shows that access time increases with number of replicated objects. On the other hand access tine for the clients without caching facility reduces (not shown). Figure 5.4 and 5.6 show hit ratio and access time obtained for different cache sizes respectively. It can be observed that the cache size (6%) gives higher hit ratio (~94%) for casual consistent system than strict consistent system (~93%).

6. FUTURE RESEARCH DIRECTION

As described earlier, when an object faulting occurs, entire object graph is shipped and installed in the client. As the size of the object graph increases, the overhead due to shipping procedure becomes significant. An empirical search results that the entire object graph rarely needed. So, partial object graph shipping scheme can carefully be implemented and that is expected to be better than the existing implementation. If a higher throughput or wider distribution is required then the platform is fully replicated and installed in the client site. In most of the cases, this approach is not feasible due to limited cache size. With the above scheme, a larger fraction of all objects can be cached resulting in improved performance.

A sophisticated cache replacement policy has to be devised for systems that do not follow traditional locality of reference pattern.

Our implementation cannot handle the situation whenever a fault (object or server) occurs. A fault tolerance mechanism has been left for future implementation.

Related Work:

In [19] Zahir Tari, Herry Hamidjaja and Qi Tang Li have outlined a cache replacement and consistency mechanism. They proposed a Least Recently Used algorithm extension, or Enhanced LRU (they call it LRU+) to minimize the overall overhead with a sorting process distribution and a variation of the Optimistic Two-Phase Locking (Optimistic 2PL) algorithm for consistency that shows the average access time is almost constant with the increase of number of clients but slightly less than the case with no caching. In our implementation access time, which is very less than the case with no caching, increases with number of clients but the rate of increase is very slow for moderate percentage of read operations.

In [22, 18], authors presented a caching approach (COCC) that is a transactional-based model where they have defined a transaction as a sequence of one or more read operations on one or more objects, followed by an update operation. There is an overhead to install the interface repository that can negate the performance over a critical processing. This model included a single update per transaction. Our architecture is applicable for all systems without any restriction on the number of read or write operations.

In [8], authors developed a caching system (ScaFDocs) considering both strict and casual consistency. Cache replacement policy, effect of cache size and hit ratio on access time has not been indicated there. We incorporated LRU policy that gives a hit ratio 95% for 6% of cache size.

In [17], authors mainly concentrated on effect of page size on access time. Though our system can cache object graphs with variable size, the effect of graph size on access time is left as a future exercise.

In [2], a qualitative discussion between distributed shared memory caching and proxy caching has been made without any implementation and hence quantitative discussion. We have designed and implemented the caching architecture that shows significant reduction in access time.

In [14], authors implemented a caching architecture that caches files. In this paper, cache size is assumed to be unbounded and no cache replacement policy has been implemented.

7. CONCLUSION

In this paper, we provided a unified object caching architecture and implementation based on object wrapper that can be used to cache different types of remote objects with little modifications of client code. This approach was based on cache-invalidate cache consistency policy, per process caching, object graph based data shipping and replication management. We applied our architecture on applications with different types of system requirements and made a comparison among them. We demonstrated that caching results in improved performance for applications that exhibit a reasonable amount of locality. Additionally, we document below what we considered as useful features.

- *Server side caching:* In addition to the client side caching we incorporated caching in the server side expecting better performance where remote database server is used.
- *Coarse-grained object shipping:* We ship the entire object graph form server side expecting more hit ratio.
- *LRU cache replacement policy:* High value of hit ratio obtained for moderate cache size.
- *Consistency level:* Systems with different consistency requirements have been developed showing various performance.

Acknowledgement

This work is partially funded by U.G.C sponsored M.R.P entitled “Performance Analysis of Distributed Component Platforms & Introduction of a Unified Distributed Architecture” (Ref. No. F.14-16 / 2000 (SR-I) dated Oct 12, 2000).

8. REFERENCES

[1] Charu Aggarwal, Joel L. Wolf and Philip S. Yu. Caching on the World Wide Web. *IEEE Transactions on knowledge and data processing, vol 11, no 1, January-February, 1999*

[2] Juan-Carlos Cano, Ana Pont and Julio Sahuquillo. The Difference between Distributed Shared Memory Caching and Proxy Caching. *IEEE Transaction on Computers, July-September 2000.*

[3] Gregory Chockler, Danny Dolev, Roy Friedman and Roman Vitenberg HUJI and Technion: Implementing a Caching Service for Distributed CORBA Objects (CASCADE).

[4] Pascal Felber, Benoit Garbinato and Rachid Guerraoui. The Design of a CORBA Group Communication Service. *Proc. Of the 15th Symposium on Reliable Distributed Systems (SRDS-15), Niagra-on-the-lake, Canada, 1996.*

[5] Amitranjan Gantait, Samiran Chattopadhyay and Uttam Roy. Event Service Based Architecture for Object Caching in CORBA.

[6] Aniruddha S. Gokhale and Douglas C. Schmodt. Measuring and Optimizing Latency and Scalability Over High-speed Networks. *IEEE Transaction on Computers, Volume 47, No. 4, April 1998.*

[7] Object Management Group (http://www.omg.org/). The Common Object Request Broker: Architecture and Specification.

[8] R. Kordale and M. Ahamad. Object Caching in a CORBA Compliant System. *Technical Report: GIT-CC-95-23.*

[9] M. C. Little and S. K. Shrivastava. Implementing High Availability CORBA Applications with Java. *IEEE Workshop on Internet Applications, WIAPP’99, San Jose, July 1999.*

[10] C Marchetti, L. Verde and R. Baldoni. CORBA Request Portable Interceptors, A Performance Analysis. *Proc. of the Third International Symposium on Distributed-Objects and Applications (DOA’01), 2001.*

[11] Veljko Milutinovic. Caching in Distributed Systems. *Guest Editor’s Introduction, IEEE Concurrency, July-September, 2000*

[12] T. Mowbray and R Malveau. *CORBA Design pattern.* John Wiley Computer Publishing, 1997

[13] Priya Narasimhan, Louise E. Moser and P. M. Melliar-Smith. Using Interceptors to Enhance CORBA. IEEE Transactions on Computers, 1999

[14] Michael N. Nelson, Graham Hamilton and Yousef A. Khalidi. A framework for Caching in an Object Oriented System. *The SMLI Technical Report series, 1993, Sun Microsystems Inc.*

[15] Robert Orfali and Don Harkey. Client Server Programming with Java and CORBA. *2nd Edition, John Wiley, 1998*

[16] Thomas Sandholm, Stefan Tai, Dirk Slama and Eamon Walshe. Design of Object Caching in a CORBA OTM System. IONA Technologies plc

[17] S. Selvakumar. Implementation and Comparison of Distributed Caching schemes. *The Proc. Of the IEE International Conference on Networks, 2000.*

[18] Zahir Tari, Slimane Hammoudi and Stephen Wagner. A CORBA Object-based Caching with Consistency. *Proc of the International Conference on Database and Expert Systems, Florence, September 1999.*

[19] Zahir Tari, Herry Hamidjaja and Qi Tang Lin. Cache Management in CORBA Distributed Object Systems. *IEEE Concurrency, July-September, 2000, p48-55.*

[20] Borland Technology. Visibroker for Java Programmers Guide, v5.2.1. www.borland.com.

[21] Oliver Theel and Markus Pizka. Distributed caching and Replication. *Proc. of the 32nd Hawaii International Conference on System Sciences-1999*

[22] Stephen Wagner and Zahir Tari. A Caching Protocol to Improver CORBA Performance. *Proc. Of the Australian Database conference, 31st January-3rd February, Canberra, Australia, 2000.*