# An Interceptor-based CORBA Object Proxy Caching System

Uttam Kumar Roy and Samiran Chattopadhyay
Information Technology
Jadavpur University
Kolkata, India
Email: {u_roy, samiran}@it.jusl.ac.in

*Abstract*—**Next-generation distributed CORBA applications and services must be flexible, reusable, and capable of providing scalable, low-latency quality of service (QoS) to delay-sensitive applications. Caching is an important technique to improve performance for critical and real-time distributed applications. In this paper, we provide a unified object caching architecture (based on caching proxies that use interceptors), easily extendable to mobile system. This approach is based on write-through-invalidate cache consistency policy, per-process caching, object-graph based data shipping and replication management. We apply our architecture on applications with various system requirements (e.g. strong/casual consistent and read-only system) and make a comparison among them. We demonstrate that the proposed approach of caching results in a significantly improved performance for applications that exhibit a reasonable amount of locality and read operations.**

***Keywords-CORBA; distributed system; caching; consistency; object graph serialization; interceptor***

## I. INTRODUCTION

**C**ommon **O**bject **R**equest **B**roker **A**rchitecture (CORBA) provides several advantages over other distributed technologies (e.g. RMI, Socket etc.) by providing location transparent method invocation on remote objects, and by taking care of all issues such as programming language, operating system, network protocol those arise from the heterogeneity of distributed systems. CORBA undertakes these burdens by performing remote invocation for every request causing excessive data conversion, marshalling-demarshalling, and overuse of network bandwidth. As a result, systems, which hope to address interoperability and heterogeneity issues, inherent in mobile environment, experience communication latency. This default latency is not acceptable for critical and real-time systems and applications that require high degree of performance.

Caching [Fig. 1.1] objects lets clients to invoke methods on distributed objects *locally* instead of invoking them from remote servers. In [9, 10, 11], we provided various caching architectures. This paper describes a proxy-based general architecture for caching objects or object graphs [Fig. 3.2] with little modifications of existing code using *interceptors*.

The proposed architecture has several salient features. First, the architecture can be used to cache many types of objects with consistency. 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 [7,13].

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. Applications are expected to work without the caching facility as before.

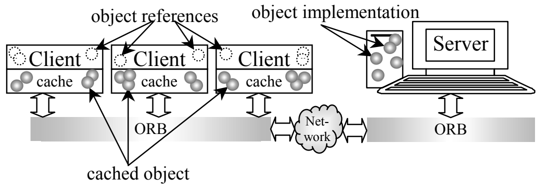


Figure 1.1: Object Caching Model

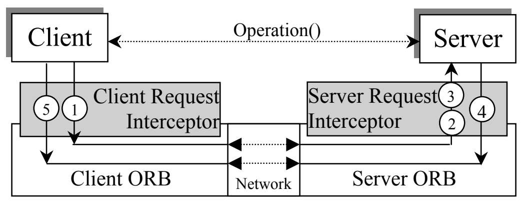
Third, in addition to the client side caching, server side caching has been proposed expecting further reduced response time whenever the server uses a remote database.

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.

## II. OVERVIEW OF PORTABLE REQUEST INTERCEPTORS

Portable Interceptors (PIs)[Fig 2.1] are a set of interfaces, which provide a framework for plugging in additional ORB behavior such as caching, security, transactions, or logging. PIs are classified in *client request interceptors* and *server request interceptors*. Formers are installed in client-side ORBs and can intercept outgoing requests and contexts as well as incoming replies and exceptions while the latter are installed in server-side ORBs and can intercept incoming requests and contexts as well as outgoing replies and exceptions.

*This research work is partially funded by Center for Mobile Computing and Communications, Jadavpur University, India under the UGC Scheme "Universities with Potential for Excellence".*



1 send_request/send_poll()
2 receive_request_service_context()
3 receive_request()/receive_poll()
4 send_reply()/send_exception()/send_other()
5 receive_reply()/ceceive_exception()/receive_other()

Figure 2.1: Operation of an Interceptor

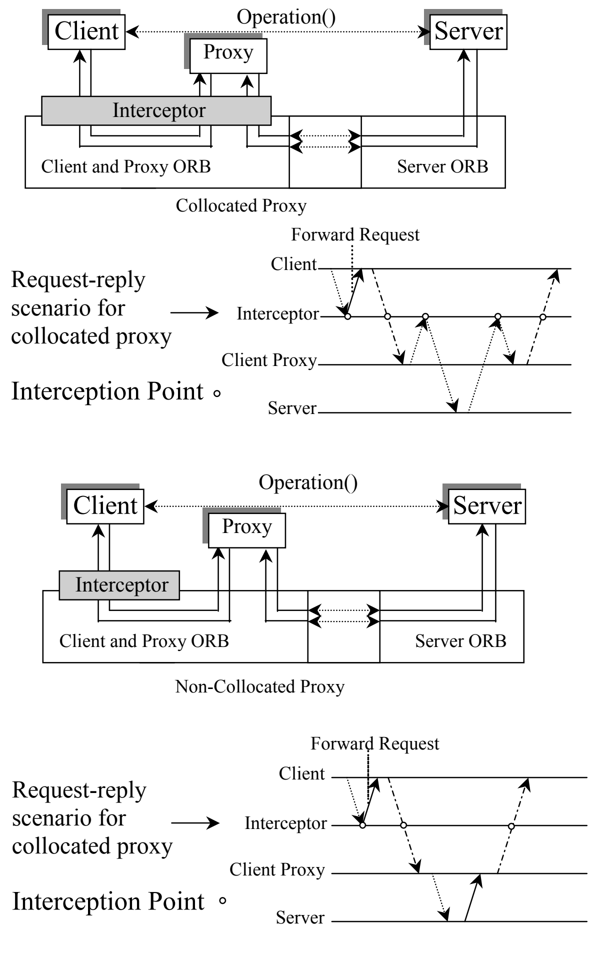


Figure 2.2. Collocated and Noncollocated proxies

## A. Portable Interceptors Limitations

The main limitation of Java implementations of the PIs specification consists of the impossibility of accessing some important fields of a request or of a reply from a request interceptor. To overcome these limitations, we have used proxy pattern that increases the flexibility of the implemented solution, by decoupling the request redirection aspects from the application dependent ones.

## B. PI-based Proxy Techniques

In such a scenario a client request PI redirects each intercepted client request to a local proxy, which is able to read the request content and perform operations to meet the application requirements (caching in our case). However, depending on such requirements, different design choices [Fig. 2.2] concerning both the interceptor behavior and the proxy deployment can be made.

## C. Collocated and Noncollocated proxies

As shown in Fig. 2.2, a proxy is *collocated* if compiled in the same client addressable space. Otherwise it can be *non-collocated* (which is considered in our architecture), i.e. deployed in a distinct process running on the client host (thus allowing many clients to use the same proxy). Requests outgoing from collocated proxies are intercepted by the underlying client request PI. In this case a mechanism to avoid recursive scenarios has to be implemented. Collocated proxies are compiled with the client, reducing modification flexibility but improving performance. On the other hand, non-collocated proxies can handle requests coming from more than one client and are modifiable without recompiling the clients (which was one of our design criteria).

## III. PROPOSED ARCHITECTURE

It comprises of following components [Fig. 3.1].

*Proxy Client Object (PCO), Proxy Server Object (PSO), Client Request Forwarder (CRF), Server Request Forwarder (SRF), Object Factory (OF), Object Implementer (OI), Object User (OU).*

## A. Proxy Client Object (PCO)

This module is client side proxy, which is responsible to managing cache in the client side. For weaker consistent system, PCO also takes part to maintain consistency. The entire procedure is divided as: *Initialization* and *method invocation*. In the initialization phase (it is executed once for the first time when client tries to invoke a method using remote object reference) Client Request forwarder intercepts the request and forwards it to PCO by throwing a LOCATION_FORWARD exception, a standard CORBA mechanism. Successive requests from client will then redirected to PCO, which in turn, checks whether the requested object is its local cache. If it is there *(hit)*, PCO serves the query locally without any network interaction that eliminates the remote invocation latency. If the object is not in the local cache *(miss)* it consults *Server Proxy Object (PSO)* in server side to get current state of the requested object. PSO serializes the object graph[1] (Fig. 3.2) with the requested object at the root and sends it back to the PCO. PCO in turn constructs the object locally using *Object Factory (OF)* and then puts the object in the local cache and sends a *register* request to the PSO. PSO in turn registers the PCO for the object. If the request is a write/update operation, the request is sent transparently to the sever side without any interception. PSO intercepts the *write* operation request, sends the request to the object implementation and also sends the *write* operation to

[1] Java's *serialization* procedure has been used in our architecture to bring the entire object graph from server side to client side.

all the PCOs registered for this object except the one from which the request has just arrived. PCOs update their local objects. While putting the object in the local cache PCO checks the cache's state. If it is *full* it removes some object(s) depending upon the application specific cache replacement policy (e.g. LRU, FIFO etc.) and sends *deregister* message to the PSO telling that its cache no longer contains the object. If a user specifies weak consistency requirements, PCO creates a separate thread that updates all the object replicas in its cache after certain regular intervals (e.g. 10 msec.) 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. But this scheme increases network traffic. To obtain strong consistent system, responsibility is imposed on the PSO rather PCO.

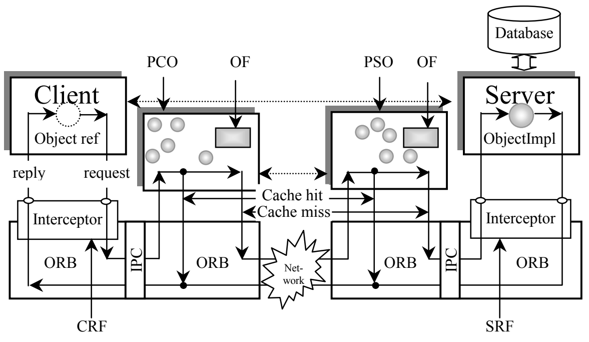


Figure 3.1: Proposed architecture

## B. *Proxy Server Object (PSO)*

PSO module is responsible to make the system strongly consistent. PSO keeps track of the objects that are stored in client's local cache and eventually updates them. For write/update operations, those are sent transparently to the server side without any interception in the client side, PSO intercepts the *write* operation request and sends the *write* message to all PCOs registered earlier for this object.

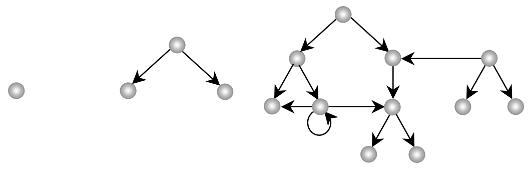


Figure 3.2: Different types of object graph

While PCO is used for client side caching, PSO is used for server side caching to reduce access time further. Often, server side object is populated by the data from some database. For heavily loaded database, access time is not negligible. A similar caching concept is introduced in the PSO. PSO also maintains a local cache. While a request comes in the server side, PSO intercepts the request. It then checks whether the requested object in its local cache. If it is there, PSO serves the request with the local copy without any database access. If it is not there, it fetches the data from the database, creates an object populating by the data it has just fetched and puts the object in its local cache. If the cache is full, PSO replaces some object(s) depending on the specified policy. Server side caching policy is optional and is used for a specific set of applications mainly for applications where database is used.

## C. *Client Request Forwarder (CRF)*

In the initialization phase, *Client Request Forwarder* (CRF) and *Server Request Forwarder* are loaded in the client and server side respectively. CRF and SRF are *Client Request Interceptor* and *Server Request Interceptor* respectively. These interceptors are loaded using *Client Request Forwarder Loader* and *Server Request Forwarder Loader* that sets ORB properties so that the specific functions (e.g. receive_request_service_contexts(), send_request(), etc. as described in section I) are called whenever a method invocation request is made. For the first time CRF intercepts the request, initializes Proxy Client Object (PCO) and forwards the request to PCO. Any further request will then transparently be redirected to PCO.

## D. *Server Request Forwarder (SRF)*

Server Request Forwarder (SRF) is loaded by Server Request Forwarder Loader during the ORB initialization. SRF initializes the Proxy Server Object. Once PSO is initialized accepts the request sent by the PCO and forwards the request to the server implementation. If the request is a write operation, it updates all other replicas that reside in the other PCOs. SRF can be made optional if one initializes PSO during the object initialization. We have used SRF so that initialization of PSO can be done transparently without any knowledge of server.

## E. *Object Factory (OF)*

*OF* constructs object with the serialized data. PCO uses this module whenever an object faulting occurs. When an object faulting occurs PCO sends a request to the PSO to return the serialized data containing the state of the object graph. After getting the data from the PSO, it constructs the appropriate object with the help of OF. If server side caching is implemented then PSO uses the OF same way the PCO does.

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.

## F. *Object Implementer (OI)*

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. PSO can be added at runtime also so that the methods implemented subsequently can be intercepted.

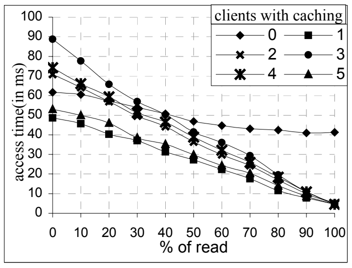
## G. *Object User (OU)*

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

## IV. PERFORMANCE ANALYSIS

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 using Nortel switch. 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). For the experimental purpose we have considered 5 clients started on IBM pcs. The clients then makes the CORBA calls. We generated data set satisfying locality (temporal and spatial) of reference. A typical reference pattern is shown in Fig. 4.2(ii).

Abbreviations used: *CC: Cache capacity*
*HR: Hit Ratio*



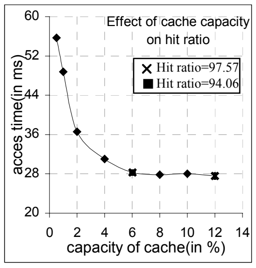
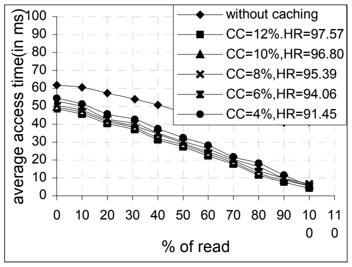


Figure 4.1: (i) access time vs. % of read (ii) *Access time vs cache size*



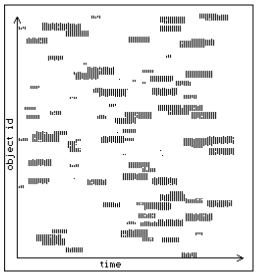
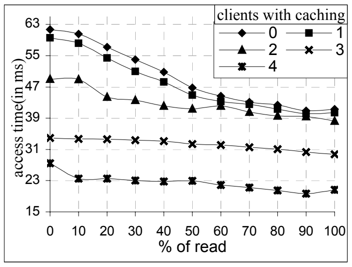


Figure 4.2: (i) *Effect of cache size on access time (ii)Data set used*



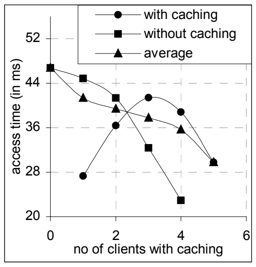


Figure 4.3: (i) *Performance of strict consistent system.(ii)Effect of number of clients on access time*

In Figure 4.1(i), effect of % of read on access time for different client with caching facility is shown. In Fig 4.2(i) the same is shown for different cache capacity with hit ratio obtained. Access time decreases exponentially with cache capacity as shown in Fig 4.1(ii). Fig. 4.3(i) shows that the access time for the clients without using caching facility also decreases as the number of clients with caching facility increases. Fig 4.3(ii) is the average of Fig. 4.1(i) and Fig. 4.3(i).

## V. CONCLUSION

In this paper, we provide a unified object caching architecture easily extendable to mobile environment based on interceptor using caching proxies. This approach was based on *write- through-invalidate* cache consistency policy, *per process* caching, *object graph* based data shipping and replication management. We demonstrated that caching results in improved performance for applications that exhibit a reasonable amount of locality and read operations. The proposed architecture can easily be extended for mobile environment. For successful use of the architectures in mobile systems, assumption about the existence of a mobile-middleware must be made. This assumption is not critical assumption as mobile middlewares are now available in the market.

## ACKNOWLEDGMENT

This research work is carried out in the project "Capability Enhancement & Interoperability of Mobile Middleware Systems" funded by Centre for Mobile Computing and Communication, Jadavpur University, India under the UGC Scheme "Universities with Potential for Excellence".

## REFERENCES

[1] A. S. Gokhale and D. C. Schmodt. *"Measuring and Optimizing Latency and Scalability Over High-speed Networks"* IEEE Transaction on Computers, vol 47, No. 4, April 1998.

[2] C. Aggarwal, J. L. Wolf and P. S. Yu. *"Caching on the World Wide Web",* IEEE Transactions on knowledge and data processing, vol 11, no 1, Jan-Feb, 1999.

[3] C. Marchetti, L. Verde and R. Baldoni. *"CORBA Request Portable Interceptors, A Performance Analysis",* Proc. of the Third International Symposium on DOA, 2001.

[4] J. C. Cano, A. Pont and J. Sahuquillo. *"The Difference between Distributed Shared Memory Caching and Proxy Caching",* IEEE Transaction on Computers, July-Sept 2000.

[5] O. Theel and M. Pizka. *"Distributed caching and Replication",* Proc. of 32nd Hawaii International Conf on System Sciences-'99.

[6] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith. *"Using Interceptors to Enhance CORBA",* IEEE Trans on Computers, '99.

[7] S. Selvakumar. *"Implementation and Comparison of Distributed Caching schemes",* Proc. of IEEE Conference on Networks, '00.

[8] S. Wagner and Z. Tari. *"A Caching Protocol to Improve CORBA Performance"*, Proc. of Australian Database conf, Jan-Feb, 00.

[9] U. Roy, S. Chattopadhyay and A. R. Gantait, *" Event Service Based Architecture for Object Caching in CORBA"*, HiPC, 02 *p 583-584.*

[10] U. K. Roy and S. Chattopadhyay, *"Wraper based Object Caching in CORBA",* CCCT '04,

[11] U. K. Roy and S. Chattopadhyay, *"Some studies on Object Caching in CORBA",* DPN '04, IIT kgp, India page no 18-22.

[12] Z. Tari, H. Hamidjaja and Q. T. Lin. *"Cache Management in CORBA Distributed Object Systems",* IEEE Concurrency, July-Sept. '00, vol 8, no 3, p48-55.

[13] Z. Tari, S. Hammoudi and S. Wagner. *"A CORBA Object-based Caching with Consistency",* Proc of the International Conference on Database and Expert Systems, Florence, Sept. '99.