

A CORBA Cooperative Cache Approach with Popularity Admission and Routing Mechanism

Zahir Tari

Herry Hamidjadja

School of Computer Science and Information Technology,
RMIT University,
City Campus, GPO Box 2476V,
Vic 3001, Melbourne, Australia,
Email: {zahirt,herryh}@cs.rmit.edu.au

Abstract

For many distributed data intensive applications, the default remote access to CORBA objects is often not acceptable because of performance degradation. Caching enables clients to invoke operations locally on distributed objects instead of through remote servers. Existing CORBA caching approaches are extension of well-known database and/or WWW techniques to deal with the specific architectural aspects of CORBA systems. As users (of distributed applications) always use the same distributed objects, sharing of these objects between caches become necessary to improve the overall performance of distributed applications.

This paper proposes a cooperative cache approach with solutions to two main problems: (i) eviction and (ii) routing. The proposed eviction technique takes into account several parameters, including the rate and cost of object invalidation. The routing protocol is based on the use of a skipping function (which efficiently determines the appropriate nodes to look up) and a hit factor function (which decides the appropriate routing path to follow based on aggregated probabilities provided at different caches). The proposed approach is implemented in JacORB system. The performance testings show how a saving of 45%-50% access time can be obtained.

Keywords: Distributed objects, performance, caching, and cooperative caching.

1 Motivation

The Common Object Request Broker Architecture (CORBA) provides several advantages over existing communication protocols (e.g. remote procedure call or sockets) such as location, operating system and programming language transparencies [Tari and Bukhres, 2001]. However the core of CORBA, the Object Request Broker (ORB), which is responsible for providing such transparency, has problems of messaging overhead and over-use of networking bandwidth. By default, a CORBA client application performs a remote invocation for every request. For many data intensive distributed applications, this default remote invocation of CORBA objects by clients is not acceptable as it causes performance degradation. Caching enables clients to invoke operations locally on distributed objects instead of fetching them from remote servers.

Though caching has been substantially addressed in several areas (e.g. database systems, network file servers, and WWW), very little work has been done in CORBA environments. Some of these are: Object Caching in Fresco system [Kordale et al., 1996], MinORB [Martin et al., 1999], CORBA Object Transaction Monitors (OTM) [Sandholm et al., 1999],

Copyright ©2001, Australian Computer Society, Inc. This paper appeared at the Thirteenth Australasian Database Conference (ADC2002), Melbourne, Australia. Conferences in Research and Practice in Information Technology, Vol. 5. Xiaofang Zhou, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

and our previous work [Tari et al., 2000, Wagner and Tari, 2000]. These approaches are useful, however the issue of sharing objects between caches has not been taken into account. This issue, as we have demonstrated in the performance testing, produces better performance because it increases the hit rate as objects can be found in neighbouring caches.

Two main issues are considered when dealing with cache management: (i) the design of a cache replacement technique (which is used to decide the objects to be removed whenever the cache is full), and (ii) the design of a consistency technique (which keeps objects in the cache consistent with the server objects). However, as we are dealing with the cooperation between caches, the issues of *routing* and *propagation* need to be addressed. Routing provides a way of finding objects in a network-based cooperative cache. If objects were not found in a given cache node, the request will be routed to neighbouring caches. The issue is therefore to find the best routing technique that will increase the hit rate. Propagation related to how the new information about objects is propagated from one cache node to another. When a new object is brought to the cache, this reference is propagated to the closest neighbours. This mechanism is used to speed up the routing process. This paper provides the following contributions:

- Proposes a Greedy Dual Size with Recent Popularity Admission (GDSRPA) cache replacement policy. GDSRPA incorporates a dynamic nature of object aging as part of utility value for sorting objects. It also takes into account the fact that objects can be updated and therefore performance degradation may occur. The utility function uses well-known parameters (such as frequency, size and retrieval time) as well as the rate and cost of invalidation (related to update operations).
- A routing technique that improves the hit factor by allowing caches to know about the objects of neighbouring caches. As they may exist several neighbours, a probability function is used to determine the appropriate routing path to find objects.

This paper is organised as follows. Section 2.1 provides details about the GDSRPA (Greedy-Dual Size with Recant Popularity Admission) model for object eviction. This section also describes the routing protocol as well as object consistency. The design of the cooperative cache system is described in Section 3. The performance analysis is proposed in Section 4. Finally we conclude our approach in Section 5.

2 The Proposed Cooperative Cache Model

The specific architectural aspects of CORBA systems drive the design of the caching technique towards a functional-based caching policy. Such a policy takes into account several parameters, such as size of objects, frequency of access, time since last access, latency, number of hop counts, cost of invocation execution on the server side, and average network bandwidth.

This section describes our solution for the issues of cache replacement, request routing and cache consistency. Algorithms are presented and examples are proposed.

2.1 GDSRPA – A Greedy Dual Size with Recent Popularity Admission

(A) Cache Replacement

This section proposes a formalisation of an extension of the Greedy-Dual Size with Popularity (GDSP) approach [Jin et al., 2000]. By having extra attributes, such as the invalidation rate and the cost of validation, taken into consideration in computing the utility function, the cache replacement problem can be partially solved.

The GDSP extension proposed in this paper is called Greedy-Dual Size with Recent Popularity and Admission (GDSRPA). The Greedy Dual (GD) algorithm and its variants [Young, 1991, Cao et al., 1997, Jin et al., 2000] do not take into account the update operations as they are designed to manage “read only” operations. This is not appropriate in the context of CORBA environments which exhibit high update transactions. By maintaining an extra data structure (e.g. lookup table) that keeps track of object references, update operations can be supported. The goal of the proposed cache replacement algorithm is to minimise response time: objects that account for a large fraction of the communication delays are retained in the cache. This can also be formalised by the following objective function:

$$\max \sum_{i \in I} (f_i \cdot c_i - r_i \cdot v_i) \quad (1)$$

where I is the set of objects selected for caching, f_i represents the computed frequency with decay rate, c_i is the cost of retrieving the object, r_i is the rate of invalidation, and v_i is the cost of invalidation. The function (1) attempts to provide a balance between advantages and disadvantages of keeping objects in the cache. The benefit of having a cache in the first place is subtracted by the complexity cost of maintaining consistency between cached objects. If the benefit of having caches can be maximised and the complexity involved in consistency can be reduced, the solution should provide good results.

The function (1) is subject to the following constraint:

$$\sum_{i \in I} s_i \leq S \quad (2)$$

where I describes the set of objects selected for caching, s is the size of the object, and S is the total cache size. Based on the above restriction we derive a new utility function to determine whether an object is worth caching. The newly derived utility function is defined as follows:

$$Utility(x) = \frac{f(x) \cdot c(x) - r(x) \cdot v(x)}{s(x)} \quad (3)$$

where f is the computed frequency of access of an object (i.e. with decay rate), c is the cost for retrieving such an object from the original server, s is the size of the object, t indicates the time since last access, r is the rate of invalidation, and v is the cost of performing invalidation. As defined in the utility function above, the benefit of caching comprises maximising the hit rate and minimising the update rate and cost. The maximal hit rate can be achieved by replicating popular objects locally, under assumption that popular objects exhibit high update rate and cost are not cached. Otherwise, it defeats the purpose of having a cache, since a high update rate and cost means higher network bandwidth usage.

Despite the utility value, any object in a priority queue must have a greater utility value than the one of the previously evicted one. Such a condition is considered to ensure that no obsolete popular object remains in the queue. This condition is described as follows:

$$Utility(x_{previous}) \leq Utility(x) \quad (4)$$

where $Utility(x_{previous})$ is the utility value of a previously evicted object, and $Utility(x)$ is the utility value of any object in the queue. We design our eviction algorithm to capture two main aspects: *popularity* and *rate of decay*.

- **Popularity information.** Our design is based on the popularity concept introduced in GDSP [Jin et al., 2000]. The main difference is that our model does not cache previously popular objects, as this is addressed by tracking the previously evicted utility values, which in turn helps to ensure that objects with the utility values below a certain threshold are not cached. Hence, our proposed caching mechanism allows some sort of “cache cleaning”.

It is true that traditional caches (e.g. LRU and LFU [O’Neil et al., 1993, Jin et al., 2000]) contain some kind of object pollution generated by previously popular objects. Our approach does not keep all statistical data about utility values, as a result, the purification process is not perfect in the sense that it still allows a certain degree of pollution. One advantage of keeping statistical data is to ensure that no objects in the cache has a utility value less than or similar to previously evicted objects. This allows a better cache management.

- **Frequency computation.** As mentioned earlier, there is a need to keep track of the access frequency for popular objects. Keeping a reference count on every object is simple, however may result in some inaccuracies. To produce a more accurate prediction, GDSP algorithm uses the decay rate as an additional attribute to compute the frequency access of an object instead of using a plain access frequency. In the model described in this paper, an adaptation of reference count incremental from the approach used in WWW systems [Jin et al., 2000]. The following is our formalisation of the frequency access function:

$$f(x+1) = f(x) \times 2^{(-t/T)} + 1 \quad (5)$$

where $f(x)$ is the current frequency of access, t is the time elapsed since last access, and T is a constant that controls the rate of decay. In the GDSP policy, T is a constant and is set manually at run time. This is not appropriate since the nature of the CORBA workload is not similar

to those of WWW systems. Therefore, a dynamically calculated rate of decay is proposed based on the value of T : T represents an average value of time elapsed since the last access. In this way we ensure that the rate of decay is not static, and this rate should be driven by the current flow of the object stream. We define T as follows:

$$T = \frac{t \sum_{i=0}^n}{n} \quad (6)$$

where t is the elapsed time since last access, and n is the total number of access of all objects in the cache. Having to use the average rate of decay, the number of previously popular objects polluting the cache will decrease against time.

GDSRPA algorithm sorts objects in ascending order. Objects with a low utility value are subjected to replacement first. In order to determine (I^*), that is the set of candidate objects to be in the cache, GDSRPA sorts all objects based on their profit and then, for caching, selects the profitable objects. This is performed under the assumption that cache fragmentation is minimal. Such a selection is made at eviction time when the cache buffer is full. As shown below, the set I^* returned by the algorithm satisfies the expression (1).

Remark. Let us assume the following parameters are known before hand: f_i (the frequency of access rate), r_i (the validation rate), c_i (the cost of fetching an object from a server), and v_i (the cost delay to validate an object). Then among all set of objects, the utility function (3) returns the one which satisfies expression (1).

Justification. We will show that I^* is the optimal solution that satisfies (1), subject to constraint (2) since we have:

$$\sum_{i \in I} f_i \cdot c_i - r_i \cdot v_i \leq \sum_{i \in I^*} f_i \cdot c_i - r_i \cdot v_i \quad (7)$$

GDSRPA algorithm selects a set with maximal profit (utility value), and it follows that

$$\sum_{i \in I} \frac{f_i \cdot c_i - r_i \cdot v_i}{s_i} \leq \sum_{i \in I^*} \frac{f_i \cdot c_i - r_i \cdot v_i}{s_i} \quad (8)$$

We assume that $I^* \cap I = \emptyset$. If however, such condition is not met, the intersection set can be eliminated from both sets while preserving (8). In order to do so, the following rules are defined:

$$\frac{f_{min} \cdot c_{min} - r_{min} \cdot v_{min}}{s_{min}} = \min_{i \in I^*} \frac{f_i \cdot c_i - r_i \cdot v_i}{s_i} \quad (9)$$

$$\frac{f_{max} \cdot c_{max} - r_{max} \cdot v_{max}}{s_{max}} = \max_{i \in I} \frac{f_i \cdot c_i - r_i \cdot v_i}{s_i} \quad (10)$$

These rules separate the potential of having a mixture between maximum solution (I^*) and the one not selected (I). By selecting the minimum utility value from the optimal set, and the maximum utility value from the opponent set, it can be seen that these extreme limits sit between both sets. Since I^* contains all the retrieved set with maximal utility value and $I^* \cap I = \emptyset$, it must be true that

$$\frac{f_{min} \cdot c_{min} - r_{min} \cdot v_{min}}{s_{min}} \geq \frac{f_{max} \cdot c_{max} - r_{max} \cdot v_{max}}{s_{max}}$$

Consequently, we have

$$\sum_{i \in I^*} f_i \cdot c_i - r_i \cdot v_i \geq \frac{f_{min} \cdot c_{min} - r_{min} \cdot v_{min}}{s_{min}} \cdot S \geq$$

$$\frac{f_{max} \cdot c_{max} - r_{max} \cdot v_{max}}{s_{max}} \cdot S \geq \sum_{i \in I} f_i \cdot c_i - r_i \cdot v_i \quad (11)$$

Therefore, we have shown that I^* is a set of objects with optimal utility values selected by GDSRPA algorithms and satisfies expression (1). The above proof holds with an assumption that $I^* \cap I = \emptyset$.

(B) The GDSRPA Algorithm

GDSRPA algorithm sorts objects based on the utility value in ascending order. The following steps describe the different stages of the GDSRPA algorithm with respect to cache hit and miss actions.

- (a) $L \leftarrow 0.0$
- (b) for each request for object p do
- (c) if p is in cache
 - then $H(p) \leftarrow L + (f(p) \times c(p) - r(p) \times v(p)) / s(p)$
- (d) else while there is not enough free cache for p
 - do $L \leftarrow \min\{H(q) | q \text{ is in cache}\}$
 - Evict q which satisfies $H(q) = L$
- (e) fetch p
- (f) $H(p) \leftarrow L + (f(p) \times c(p) - r(p) \times v(p)) / s(p)$

The variable L records a previously evicted utility value (a). In most of the situations, incoming requests are for read only transactions. If the object exists in the cache, the utility value of that object is updated by taking into consideration previously evicted utility values (c). Occasionally, eviction has to be performed due to insufficient space available for the cached objects. In this case, a search is performed to find an object with the lowest utility value to be evicted (d). This eviction process iterates several times till sufficient free space can be allocated for a new object. Finally, a new object is brought to the cache by fetching it from the server side (e).

2.2 The Cache Routing Protocol

Cache sharing techniques increase the hit rate globally by allowing clients to share the same cache buffer. However, this situation creates a bottleneck as each client shares the same cache buffer. In order to overcome this problem, multiple caches are employed. Another issue arises when searching for an object throughout multiple caches. This can be overcome through a balance of usage between the number of cache and the amount of time needed to search a particular object in the cache mesh to achieve an optimal solution.

A typical example of multiple caches is illustrated in Figure 1, where a node represents a cache node, a link between nodes models the search direction to retrieve an object. The basic idea behind the mesh construction is derived as an extension from cache sharing when objects are shared by cache nodes. In a complete cache mesh, where each node has links to other nodes, the search for an object can be forwarded to any node. This technique in theory can positively improve performance, however increasing complexity of query traversal reduces the gain of having the cache system in the first place. Therefore, such a complete cache mesh can not be modelled. A complete

mesh cache requires a higher degree of complexity in maintaining strong consistency between objects in the cache buffer and server.

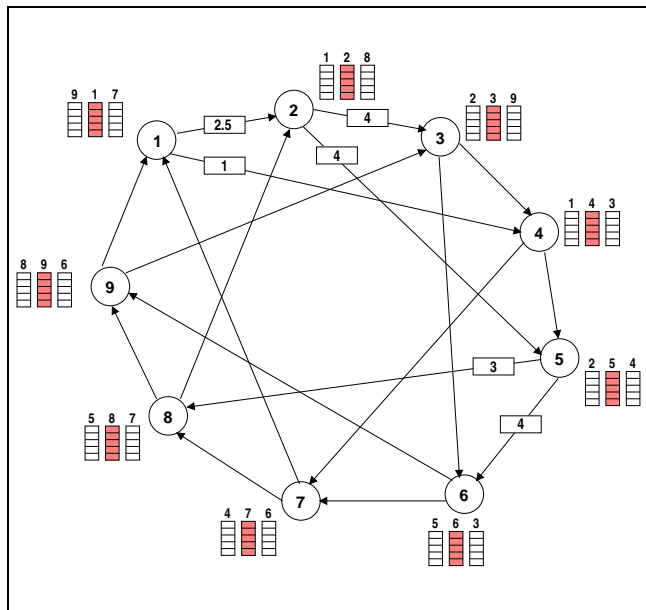


Figure 1: Cache Model with 9 Nodes

Besides, the more links a cache node has the more network bandwidth is consumed by messaging between nodes. Hence, more messages overhead means longer latency in retrieving objects. For the same reason, the cache is modelled to form a mesh from interconnected trees, where a search can be performed in a hierarchical manner which treats the start of the node as a root of the binary tree. Each node forwards the search to its children (i.e. neighbours), the trace of this process will form a logical binary tree. Virtually, the searching process can be triggered from any cache node. The point where the traversal search begins may be referred to as the “root” of the tree.

To speed up the search, each node is modelled in a way that it should seek one of its closest neighbour caches, and leap to a couple of caches ahead. By skipping the other close neighbourhood, the chance of finding an object in the mesh of caches is increased. In order to capture this behaviour of the routing mechanism, a **search skipping factor** is introduced. The search skipping factor is a number that indicates the number of nodes further a search task should start from.

The skipping factor is defined as follows:

$$SF = \left\lceil \frac{\log N}{\log 2} \right\rceil \quad (12)$$

where SF indicates the search skipping factor constantly used throughout the mesh network, and N is the number of nodes. Using the above formula, the skipping factor of a mesh is computed and the hierarchical form of the mesh can be constructed for searching and routing purposes. Referring to Figure 1, the computed skipping factor is 3, where node 1 searches node 2 and directs the other search process to node 4 by skipping 3 hop counts. Similarly, node 2 searches node 3 and skips seeking by 3 hop counts to node 5. This linking process iterates until all nodes are linked for traversal.

The proposed routing protocol uses directory service techniques, where each node knows the objects that are cached in its two neighbours (i.e. the state of two neighbours). This enables the amount of searching time traversing over the tree to be minimised since

it does not have to traverse all possible paths in order to find the objects. In addition to the skipping factor, each node computes its probability of finding a particular object locally. This probability provides useful information about objects in a cache node: if a node has most of the objects, which means that it has a higher probability, it is most likely that the object being requested will reside in that node. The computation of this probability is based on the following function:

$$P = \frac{N}{T} \quad (13)$$

where P is the probability of the node, N serves as the number of objects currently in the cache, and T as the maximum number of objects can be cached in the buffers. The definition of P is made under the assumption that the size of an object is not taken into consideration when defining the total number of objects. The size factor is not considered in counting objects since it may happen that at a particular instance, a cache node is only occupied with one or more large objects. This causes the probability computation less accurate and needs to be avoided by omitting the size of the objects.

Each link between two nodes is assigned a cost value, which exhibits a dynamic factor due to the nature of the network load. These two metrics (i.e. probability of the node and the cost of link) are combined to form a Hit Factor (HF) function serving as the routing decision maker. The newly derived function is defined as follows:

$$HF = e^{-c} \times P \quad (14)$$

where P is the probability of the node that has a link to current node, and c is the cost of the link. Every cache node stores a routing table which associates a certain hit factor (HF) used to improve the search for objects within the network cache. The routing procedure uses hit factor as a guide to choose one path over another: a path with a higher hit factor is chosen in preference to others. The node that has links to two other neighbours is responsible for computing the hit factor of each link.

There are several techniques which are used for traversing a mesh network: depth-first-search, breath-first-search, greedy search, A^* and its extensions are some examples. For the proposed routing protocol, depth-first-search has been selected because of its efficient search when network bandwidth and load are considered. Figure 2 illustrates the use of depth-first-search which traverses deep into a particular path to find objects.

Example Traversal from Node 1

The assumption made for searching is that the object will be found in Node 6, and the search is initiated at Node 1. As illustrated in Figure 2, the search will take a path towards Node 2. The reasoning behind this logic is:

- **Directory service** including the current node and the neighbour does not contain the object reference of the object being searched.
- **Cost of the link:** The link towards Node 2 has a lower cost of traversing compared to Node 4.
- **The probability of Node 2:** Node 2 has a higher probability value computed to represent the node than Node 4. The higher the probability of a node, the more likely the object will be found in it.

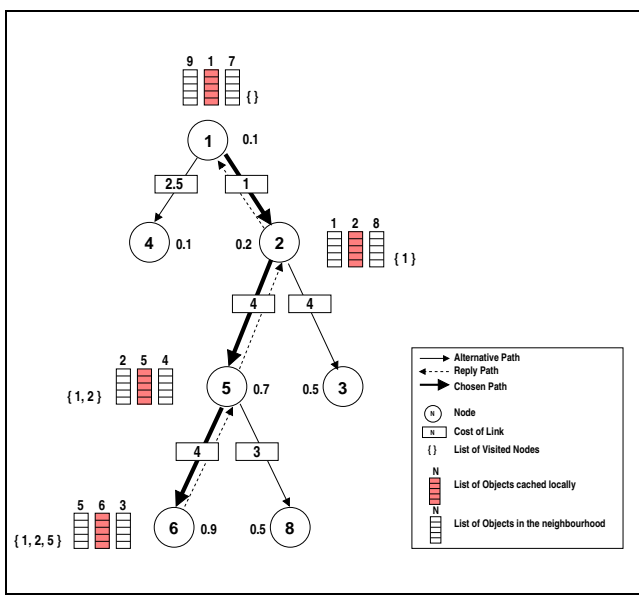


Figure 2: Cache Traversal Model

Let HF_i be the hit factor of Node 4 and HF_j be the hit factor Node 2. The different routing steps of a request from Node 1 to the target node are as follows:

$$\begin{aligned}
 HF_i &= e^{-c_i} \times P_i \\
 &= e^{-2.5} \times 0.1 \\
 &= 0.008205 \\
 HF_j &= e^{-1} \times 0.1 \\
 &= 0.073576
 \end{aligned}$$

where $P_i = 0.1$, $P_j = 0.2$, $c_i = 2.5$ and $c_j = 1$. Since, $HF_i < HF_j$, the path toward Node 2 is likely to be taken over a path that leads to Node 4. The search process is then resumed from Node 2 and the list of visited nodes is not empty: it contains Node 1. A similar process is repeated to route the searching towards a goal of finding the object. At Node 2, the search is forwarded toward Node 5 based on HF computation. The list of visited nodes contains Node 1 and Node 2 at this point. The same step is repeated to choose the right path, and finally the object matching the key is found at Node 6. Once the matching object is discovered, the object will return in a cascading manner to the origin root node (i.e. Node 1). Since, $HF_i < HF_j$, the path toward Node 2 is likely to be taken over a path that leads to Node 4. The search process is then resumed from Node 2 and the list of visited nodes is not empty, it contains Node 1. A similar process is repeated to route the searching towards a goal of finding the object. At Node 2, the search is forwarded toward Node 5 based on HF computation. The list of visited nodes contains Node 1 and 2 at this point. The same step is repeated to choose the right path, and finally the object matching the key is found at Node 6. Once the matching object is discovered, the object will return in a cascading manner to the origin root node (i.e. Node 1).

Since, $HF_i < HF_j$, the path toward Node 2 is likely to be taken over a path that leads to Node 4. The search process is then resumed from Node 2 and the list of visited nodes is not empty, it contains Node 1. A similar process is repeated to route the searching towards a goal of finding the object. At Node 2, the search is forwarded toward Node 5 based on HF computation. The list of visited nodes contains Node 1 and 2 at this point. The same step is repeated to

choose the right path, and finally the object matching the key is found at Node 6. Once the matching object is discovered, the object will return in a cascading manner to the origin root node (i.e. Node 1).

Example Traversal from Node 1 with One or More Node Failure

The proposed cache model supports a certain protection against the total black out where one or more nodes fail. Let us assume that Node 2 is not responding. Figure 3 depicts the failure of Node 2 during the search from Node 1. As Node 2 fails to respond back to Node 1, the searching process is routed through Node 4 instead. After a few adjustments, the search process resumes from Node 4 to Node 7, as it has a lower link cost and a higher probability. Finally, at Node 7, the object is found in the directory service of Node 6, therefore Node 6 is contacted to get the actual object. However, if both Node 4 and Node 2 fail at the same time, the routing process is forwarded to the original server to obtain a new copy of the object, and a new entry is made in the local cache of the current node (i.e. Node 1).

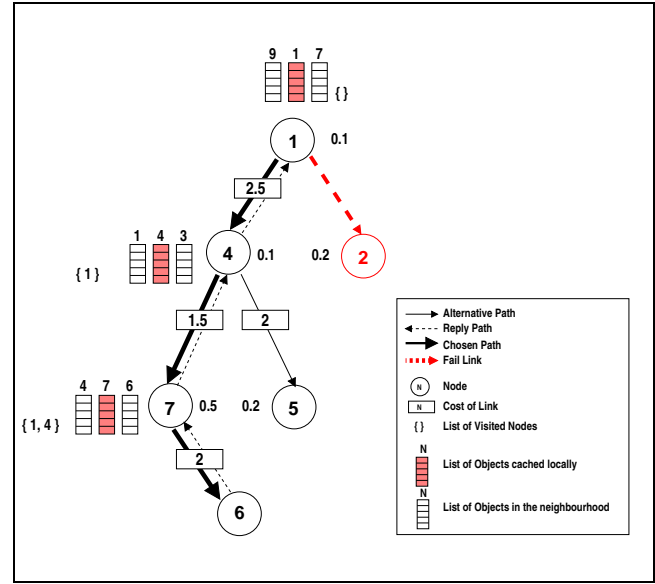


Figure 3: Cache Traversal Model with One or More Node Failure

Algorithm for Cache Traversal

The proposed routing approach traverses all possible paths based on $max(HF)$ which has maximal prediction of the probability of an object in the cache. The following pseudo code describes the behaviour of the routing procedure.

- (1) if p in Local Buffer
then return p
- (2) if p in Directory Service Current Node
then return p
- (3) else while p is not found and Node is not visited
do compute HF of each link
return forward request to link with highest HF
- (4) if p is not found in cache
then fetch p and return p

The first routing task is to find out whether or not the object p exists in the local cache buffer (1).

If it exists, the object is fetched and returned to the calling client. However, in the case of no matching object, the directory service that contains the object references from two closest neighbours is searched in the current node (2). If it does find the object, this object is fetched and returned to the calling client. However, if the object does not exist in the current node directory service and the next node is not yet visited, a decision needs to be made whether to traverse the tree by taking one path over another (3). To solve this conflict, the hit factor (*HF*) function is used. The current node is responsible for computing the Hit Factor of each link it has. Each link towards a different node has an associated value of HF. The higher HF for a node, the higher the change for an object is to be found. Otherwise, if all fail, the original server is contacted to perform the initial object cache.

The traversal algorithm model is complex as it has to perform a local search in either cache buffer and directory services before it can contact the other nodes or the original server to satisfy such a request. The same process is repeated in different nodes. This implies that once the search is forwarded to another node, that node is responsible to further route the request towards finding the goal.

2.3 Cache Consistency

We propose an adaptation of the version of Optimistic Two-Phase Locking Propagation (O2PL-P) that requires two steps of committing an update in the server side [Franklin et al., 1997]. Standard O2PL-P requires the server to know its client to enable the server notify all clients that are holding a local copy of the object. However, in this paper the model is updated such that only the nodes know the client, and from the server point of view, the server does not have to maintain a list of clients. Network of nodes even does not have to know their clients since objects are only cached at the nodes. The only trouble is maintaining consistency between the nodes and the servers.

This consistency model does not rely on the knowledge of being aware of the whole structure of interconnected cache nodes. Instead, it depends on the existing routing table to propagate update copies to other nodes and allows them to handle propagation to further the target.

Example Consistency Protocol and Propagation

Here we will refer to Figure 4 to illustrate the different steps of the consistency protocol. The client initiates update through Node 1, and propagates updated object X at the same time (1). Upon receiving the update transaction, Node 1 tries to put a write lock in object X and attempts to perform an update in the server side (2). Once Node 1 receives a reply from the server side (3), the update can be committed providing that no failure has occurred. Then, Node 1 replies to the client that fires the update transaction to notify that the update is completed (4). At the same time, Node 1 propagates an update to two other neighbours linked to it and allows them to continue further updates throughout the network (5, 6).

In the case of update failure in the server side, the transaction will be rolled back allowing the state of cache consistency. This temporal update as a result from this uncommitted transaction is cancelled and Node 1 returns to the pre-update transaction state. Consequently, the client has to restart the update transactions again, so that Node 1 can restart another write transaction.

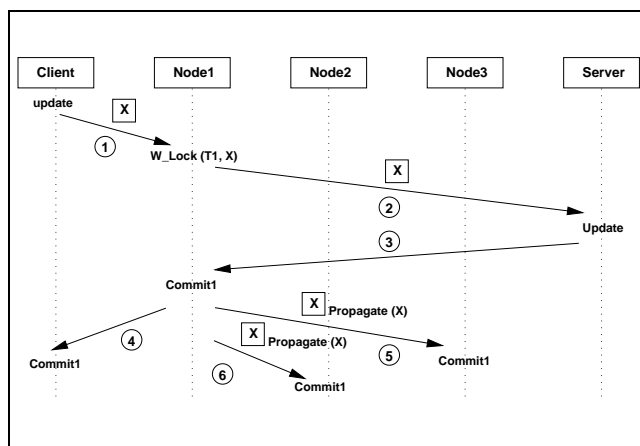


Figure 4: Cooperative Cache Consistency Protocol

3 Cache Design

3.1 Architecture

Figure 5 depicts the different components of the cooperative cache system. Every proxy cache node comprises of five major components: (1) a Client Manager, which executes the request invocation for the client and routes the request accordingly; (2) a Buffer Manager, which manages proxy buffer pool; (3) a Consistency Manager, which is in charge of consistency and concurrency control (i.e. locking) on the proxy; (4) a Directory Manager, which manages a list of object belongs to the neighbours; (5) a Resource Manager, which models the physical resources of the proxy cache. The server is modelled similarly to the proxy cache but with the following differences: the Consistency Manager maintains consistency and concurrent access of objects within the server; there is a Server Manager that coordinates the operation of the server; there is a Buffer Manager that performs objects pre-fetching from the disk.

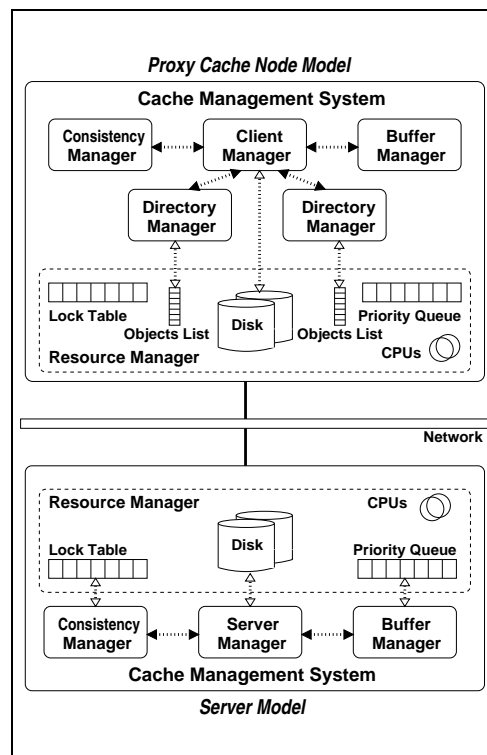


Figure 5: Cache Node Architecture

Client transactions are initiated on a separate en-

environment before being forwarded to the Proxy Cache Node for further processing. Upon receiving a request, the Client Manager queries the Buffer Manager for such an object reference. Request matching object reference will be satisfied from local buffer. Otherwise, the Client Manager will query the Directory Manager for a possible match. In the case of a matched situation, the request is answered by forwarding the retrieval of data from the Proxy Cache Node that has the object and is listed in the Directory Manager. When the object is found in the Directory list, it returns immediately, otherwise the request is forwarded to other Proxy Cache Node. Similar process is repeated until the object is found or no matching situation occur. As the client transaction of retrieving objects misses the cache, the object is retrieved from the original server, and a new entry is made accordingly in the cache buffer. The Buffer Manager ensures that the buffer is not full during the process of adding a new object. In a situation where the buffer is full and a new entry needs to be created, the Buffer Manager starts removing objects that are most likely not being used for long time compared to others. Once sufficient space is allocated, the Buffer Manager appends that new object to the cache buffer. Consequently, this proxy cache node notifies other two neighbour nodes by propagating object references to them. Upon receiving propagation, the proxy cache node appends object references into a list of object references maintained by the Directory Manager. However, it should be noted that the Directory Manager has the same capabilities with the Buffer Manager in terms of keeping object references (i.e. manages buffer including adding and removing).

When a client wishes to perform an update on the object, the transaction is forwarded to the proxy cache node. On the other end, the Client Manager receives the transaction and uses the Consistency Manager to put the object reference into the lock table and start propagating the updated value into the original server. Once the original server replies with a positive result, the Client Manager then propagates an update into two different neighbour nodes that it has links to. Upon receiving update propagation, the neighbour nodes put the object into their lock table, and try to update the object by querying the Buffer Manager. In contrast, any failures occurred during update, especially in the original server, will trigger a roll back action. Figure 6 depicts the workflow of a Cache Service node in more details.

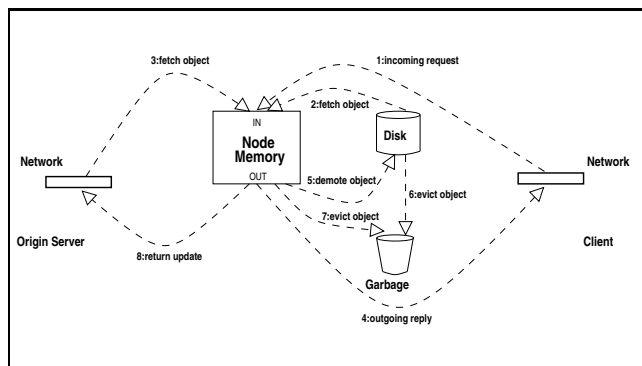


Figure 6: The Workflow of A Cache Node

3.2 Design

As depicted in Figure 7, the design of the caching model consists of the the following classes: `NodeImpl`, `BufferManagerFactory`, `GenericBufferManager`, `GenericCacheEntry`, and `ConsoleImpl`. The `NodeImpl` class

is the core functionality of the Cache Server, it provides both searching and routing functionalities. The `BufferManagerFactory`, `GenericBufferManager`, and `GenericCacheEntry` provides the queueing mechanism for storing objects temporarily, and they model the full behaviour of Buffer Manager including adding a new entry, getting the object, and the eviction functionalities. `GenericBufferManager` class is an abstract class that provides a clearly defined operation signature so that the implementation of a different kind of Buffer Management can be done easily. Similarly, `GenericCacheEntry` is also another abstract class for modelling different kinds of entries depending on the policy employed in the Buffer Management.

Directory class models the behaviour of Directory Manager that is to receive object propagation from the two closest neighbours. The propagation occurs only when a new cache entry is created in the current node. When the cache misses, the object is retrieved from the original server and this node which caches the object is responsible for informing the other nodes that it connects to. Assuming Cache Server `aNode1` is connected `aNode2` and `aNode3`, whenever a new object is replicated by `aNode1`, `aNode1` has to inform `aNode2` and `aNode3` that the object is available in `aNode1`.

The interface `ConsoleImpl` models the Console display behaviour, which determines where the output of debug messages is displayed. The `ConsoleImpl` class is a Java interface so that any realisation must implement all the method defined in the interface. In this implementation, the Cache Server has three different output console implementations including file, text console, and graphical window.

Figure 8 shows the interaction between Client and `BufferManager` in response to repeating attribute retrieval. In this scenario, invocation miss that implies the object is not in the cache buffer is expressed in sequence 1 to 6. When the object is not found in the local buffer, the search process is resumed toward seeking object in both Directory Managers (i.e. `aDirectory1` and `aDirectory2`). These directory contain objects that are cached in the current node's closest neighbours. As the request can not be satisfied using either `aDirectory1` or `aDirectory2`, the original server is contacted by forwarding the invocation accordingly. Hence, a new cache entry is created in the local cache buffer, so that the second invocation of the same object can be satisfied immediately from the local cache (refer to sequence 7 and 8).

4 Performance Digest

This section discusses the experiments performed on the proposed cooperative cache system. The implementation was tested on two different machines of the Computer Science Department of RMIT University: (a) a Sun Ultra Enterprise 3000 with 1GB of RAM running Solaris 2.6 (i.e. `numbat.cs.rmit.edu.au`). and (b) a Sun Ultra Enterprise 4000 with 2GB of RAM running Solaris 2.6 (i.e. `yallara.cs.rmit.edu.au`). To simulate a real environment, where the original server reside remotely, the application servers were executed on two different remote machines: (i) a Sun Ultra Enterprise 2 with 512MB of RAM running Solaris 2.6 (i.e. `bilby.cs.rmit.edu.au`). and (ii) a Sun Ultra 10 with 128MB of RAM running Solaris 2.8 (i.e. `nemesis.csse.monash.edu.au`).

Figure 9 illustrates the network setting of the test bed machines. The original servers are located at RMIT University (Bundoora campus) and Monash University (Caulfield campus). These servers are responsible for providing a set of functionalities that can be requested by a client. The Sun Ultra 10 machine is located at Monash, whereas the Sun Ultra

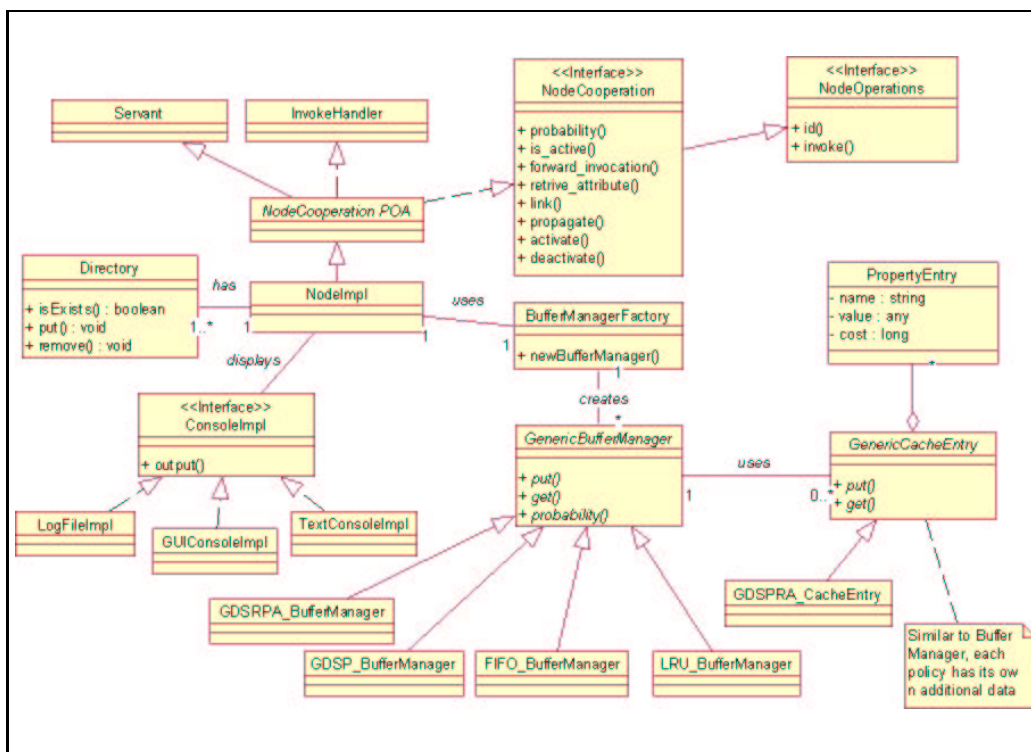


Figure 7: Class Diagram for a Cache Node

Enterprise 2 server is in RMIT University (Bundoora campus). There is no direct connection between these two campuses, and any request from these sites is forwarded via Victoria Regional Network (VRN) situated at Melbourne University [Wong, 2000]. Monash and RMIT Universities are linked via VRN with each end connected via 155 Mbps Synchronous Transfer Mode (STM) Optical Fiber. The connection between RMIT City campus and Bundoora campus is provided via STM-1 Microwave 155Mbps link. These infrastructures guarantee lowest response time provided that the bandwidth usage is low.

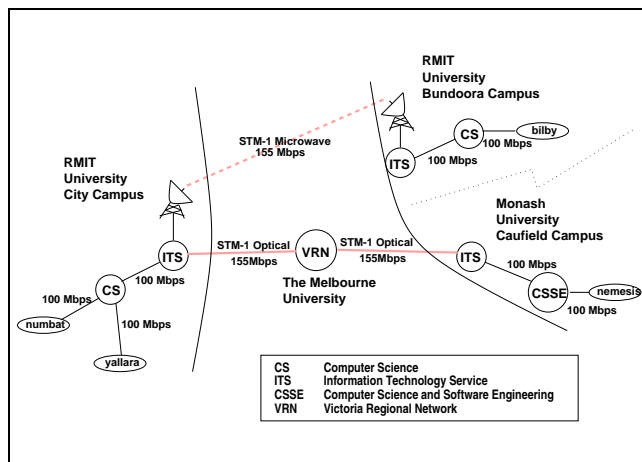


Figure 9: Base Network Setting

4.1 Workload Generation

Two types of workload are chosen: UNIFORM (which exhibits low or no data contention) and HICON (with high data contention). There are many more workload schemes that can be used in this experiment, however, they are mainly combination between HICON and UNIFORM workloads. UNIFORM workload has accessed pattern depicted in

Figure 10, which shows that every client views the working set (i.e. 5000 objects) in the same way. UNIFORM is used to examine how the cache service perform when there is little or no locality across transaction boundaries [Franklin et al., 1997]. In this workload, each client performs 200 object accesses, and all the objects in the working set are equally likely to be accessed. This workload is generated by having random distinct access patterns which ensure that less or no repeating request occurred. In this way, the effect of having a cooperative cache service in the case of a smaller number of cache hit occurrence can be analysed.

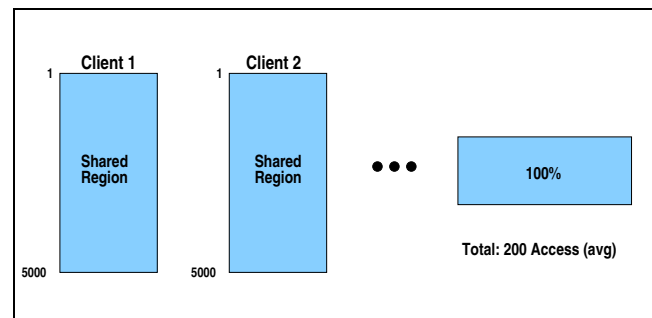


Figure 10: UNIFORM Workload

HICON Workload has access pattern depicted in Figure 11. It shows that the working set has been split into 20% "hot set" and 80% "cold set". The "hot set" will clearly have higher data contention compared to the later. HICON workload is used to examine the effect of cache service with high data contention. It is generated to ensure that 80% of object access patterns are repeating requests. From transaction average of 200 accesses, only 40 requests are repeating.

4.2 Experiment Results

Here we examine whether the average access time can be reduced by increasing the number of global hit

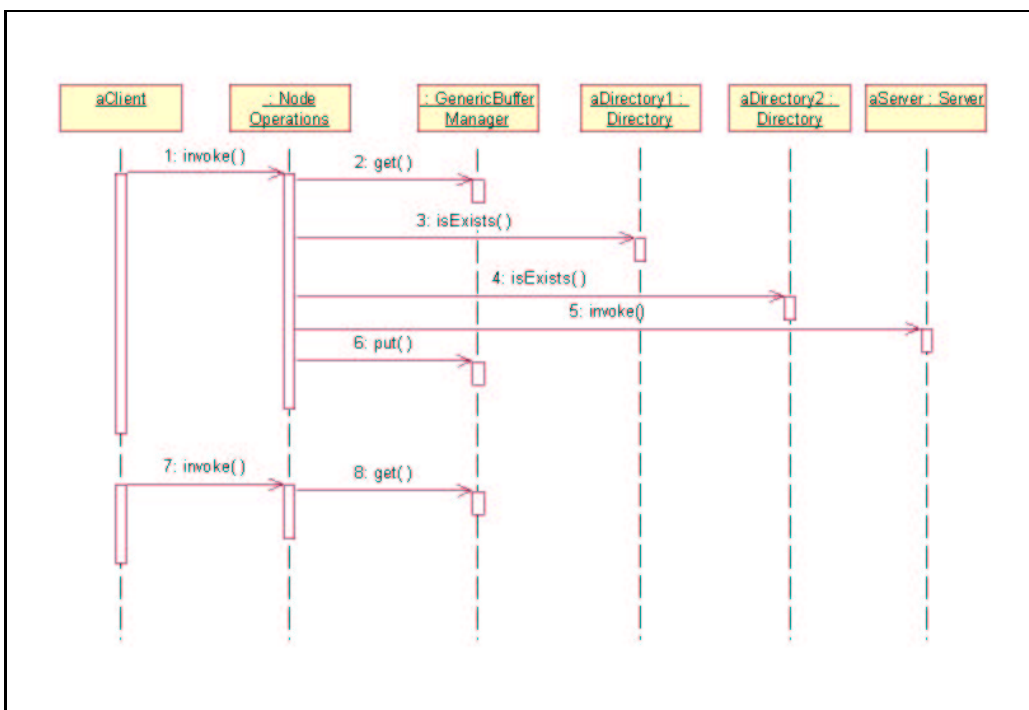


Figure 8: Trace of Internal Mechanism

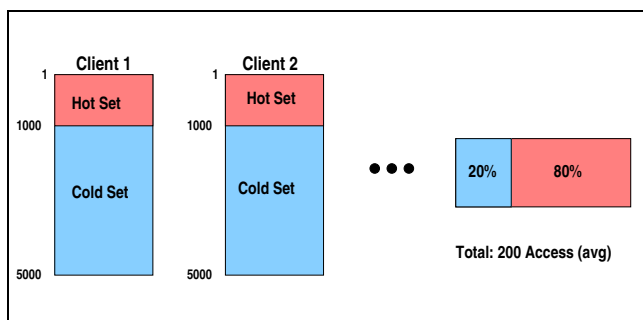


Figure 11: HICON Workload

rates, as client request can be satisfied locally. The main metric used in this experiments is access time or latency in time domain (i.e. milliseconds). Generally, a lower access time means the better the server throughput, since each transaction takes less time to be executed.

Figure 12 shows the initial testing which is performed to determine the remote server latency compared to the local one. The local server resides at RMIT University City campus, referred to as “yalara”. The remote server is located at Monash University (Caulfield campus), known as “nemesis”. Interestingly, both servers have a tiny latency gap which is not the case in most broadband networks. This phenomena is mainly due to the network infrastructure between both campus that deploys ATM (Asynchronous Transfer Mode) technologies which exhibits low latency.

HICON workload exercises the advantages of having a cache in the first place by having 80% of the accesses as repeating requests. Figure 13 outlines the experiment results based on HICON workload. The outcome is as expected, where caching approaches do reduce the communication latency between the original server and clients. The proposed GDSRPA cache replacement policy does perform better than conventional techniques, such as LRU and FIFO policies. GDSRPA policy is optimised for minimising the latency by considering object size, aging rate, fre-

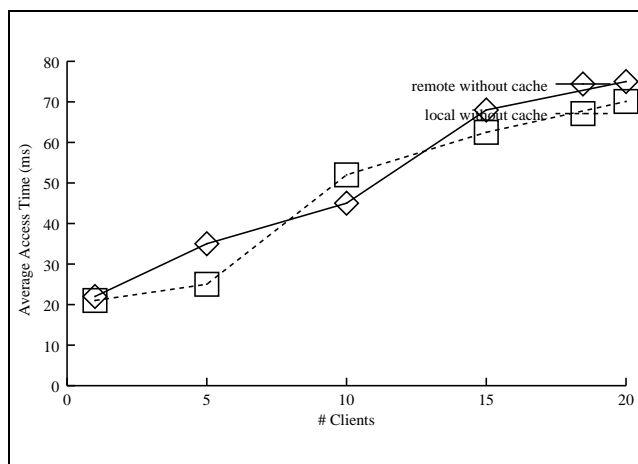


Figure 12: Local vs. Remote Server

quency, retrieval cost, invalidation rate, and invalidation cost. Thus, it enforces objects of the cache buffer to be the most eligible set of objects (i.e. objects that are big, frequently used, and expensive to retrieve). These experiments are conducted on a single node cache, where only one cache service serves the clients. In this scenario, a significant saving of access time can be achieved between 45% to 55%, which is better than the initial experiment using client cache system approach.

5 Conclusion

In this paper we have proposed a CORBA-based cooperative cache management system. This system introduces the notion of proxy, where every request for an object is forwarded to a proxy before aiming at the original server. This is aimed at ensuring any repeating request can be satisfied locally, hence it reduces the remote network access. Subsequently, this approach minimises the average access time and increases the overall transaction throughput. While traditional client cache systems focus on individual

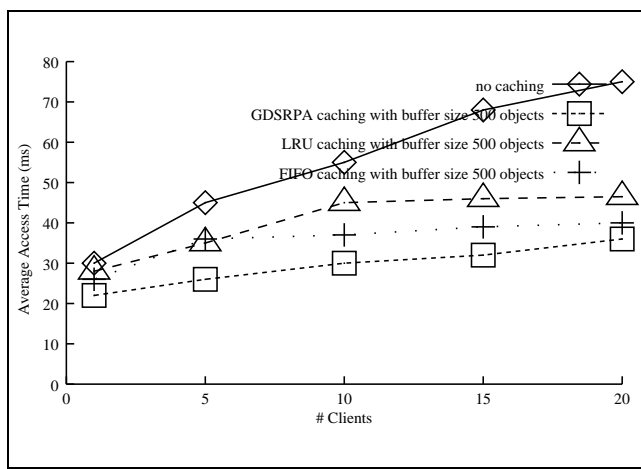


Figure 13: Client invocation of 200 transactions

access time, the proposed cooperative cache model does the opposite.

The proposed cooperative cache approach aims at reducing the average access time by increasing global hit rate. The argument is how this scheme affects the performance of each client compared to the traditional systems that focus on individual access. In a client cache system, each client has its own cache buffer where sharing is not allowed. On the contrary, the client cache system does not save remote network access since clients have different behaviours. The proposed model focuses on how to speed up the object retrieval on per client basis, where the global average access time is not the main focus. The cooperative cache permits the individual access time to be the secondary goal, as the main interest is to save network bandwidth and reduce the average access time globally.

The proposed caching approach is implemented using the standard CORBA protocol. Cache sharing protocol does improve the performance by saving the average access time up to 55% in comparison with standard client cache system. The maximum percentage of saving from standard no sharing approach is up to 45%. This means, the proposed sharing cache protocol works in CORBA and hence boosts the performance. Concepts related to the design and implementation of the cooperative cache system have been established and conceptualised. Issues related to the design of such approach such as cache replacement policies have been evaluated, along with their effects in consistency protocol.

Further work needs to be done on the proposed approach. First, propagation techniques as part of the cache sharing protocol need to be refined, and a new communication protocol between cache node needs to be redefined to avoid interference with the request processor mechanisms. The integration of a consistency protocol needs to be completed. Thus, a new way of reducing messaging overhead between cache node must be found since it contributes to the major performances degradation. Finally, the integration between the CORBA database adapter (CODAR [Surya, 2000]) and GDSRPA needs to be done to provide a larger scale cache management system which could handle virtually any size of cache buffer. This integration enhances the cache buffer size by using the notion of virtual memory where part of the cache buffer can be swapped to stable storage (e.g. databases, or disks).

References

- [Cao et al., 1998] Cao, P., Fan, L., Almeida, J., and Broder, A.Z. (1998). Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In *Proc. ACM Special Interest Group on Data Communication Annual Conference*. (SIGCOMM), pages 254–265, Vancouver, August 1998.
- [Franklin et al., 1997] Franklin, M.J., Carey, M.J., and Livny, M. (1997). Transactional Client-Server Cache Consistency: Alternatives and Performance. *ACM Trans. on Database Systems (TODS)*, 22(3):315–363, September 1997.
- [Jin et al., 2000] Jin, S., and Bestavros, A. (2000). Popularity Aware GreedyDual Size Web Proxy Caching Algorithm. In *20th IEEE Int'l. Conf. on Distributed Computing Systems (ICDCS)*, pages 254–261, Taipei, April 2000.
- [Kordale et al., 1996] Kordale, R., Ahamad, M., and Devarakonda, M. (1996). Object Caching in a CORBA Compliant System. In *USENIX Second Conference on Object-Oriented Technologies and Systems*, Toronto, June 1996.
- [Martin et al., 1999] Martin, P., Callaghan, V., Clark, A. (1999). High Performance Distributed Objects using Caching Proxies for Large Scale Applications. In *Int'l Symposium on Distributed Objects and Applications (DOA)*, pages 110–119, Edinburgh, September 1999.
- [Menaud et al., 2000] Menaud, J.M., Issarny, V., and Banatre, M. (2000). A Scalable and Efficient Cooperative System for Web Caches". *IEEE Concurrency*, 8(3):56–57, July 2000.
- [Menaud et al., 1998] Menaud, J.M., Issarny, V., and Banatre, M. (1998). A New Protocol for Efficient Transversal Web Caching". In *12th Int'l Symposium on Distributed Computing*, pages 288–302, Berlin, 1998.
- [O'Neil et al., 1993] O'Neil, E.J., O'Neil, P.E., and Weikum, G. (1993). The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *ACM Special Interest Group on Management of Data (SIGMOD)*, pages 297–306, Washington, May 1993.
- [Cao et al., 1997] Cao, P., and Irani, S. (1997). Cost-Aware WWW Proxy Caching Algorithms. In *USENIX Symposium on Internet Technologies and Systems*, pages 193–206, Monterey, May 1996.
- [Sandholm et al., 1999] Sandholm, T., Tai, S., Slama, D., and Walshe, E. (1999). Design of Object Caching in a CORBA OTM System. In *11th Conference on Advanced Information Systems Engineering*, Heidelberg, June 1999.
- [Surya, 2000] Setiawan, S. (2000). CODAR: A POA Based CORBA Database Adapter. Master's thesis, TR-00-3, RMIT University, Melbourne, 2000. <http://www.cs.rmit.edu.au/reports/2000/2000.html>.
- [Tari et al., 2000] Tari, Z., Hamidjaja, H., Lin, Q.T. (2000). Cache Management in CORBA Distributed Object Systems. *IEEE Concurrency*, 8(3):48–55, July 2000.
- [Tari and Bukhres, 2001] Tari, Z., and Bukhres, O. (2001). Fundamentals of Distributed Object Systems: The CORBA Perspective. John Wiley & Sons, 2001.
- [Young, 1991] Young, N. (1991). On-Line Caching as Cache Size Varies. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 241–250, San Francisco, CA, January 1991.
- [Wagner and Tari, 2000] Wagner, S. and Tari, Z. (2000). A Caching Protocol to Improve CORBA Performance. In *11th Australasian Database Conference (ADC)*, pages 140–148, Canberra, Australia, 2000.
- [Wong, 2000] Wong, E. (2000). Victoria Regional Network. <http://www.vrn.net.au>, August 2000.