"High Performance Distributed Objects using Caching Proxies for Large-Scale Applications", International Symposium on Distributed Objects and Applications (DOA'99), Edinburgh, Scotland. 5-6th September 1999

High Performance Distributed Objects using Caching Proxies For large scale Applications

Paul Martin
Network Intelligence Engineering Centre
BT Labs
Martlesham Heath
Suffolk, IP5 7RE
United Kingdom

Victor Callaghan, Adrian Clark (ESE)
Dept. of Computer Science and Electronic Systems Engineering
University of Essex, Wivenhoe Park
Colchester CO4 3SQ, United Kingdom

Abstract

Initial implementations of Middleware based on standards such as CORBA have concentrated on host and language transparency issues in order to demonstrate interoperability. They have largely adopted a No-Replication approach and have frequently neglected performance-at-scale issues. This has lead to a continuing deployment of either non-scalable Full-Replication approaches or ad-hoc messaging-based Middleware for applications such as Intelligent Networks, WWW applications and Collaborative Virtual Reality. These applications require millions of objects globally distributed across hundreds of hosts and demand a very high throughput of low-latency method invocations. Our main research aim is to be able to reason about the performance of such applications when using scalable Partial-Replication and Object-Oriented approaches to Middleware.

Our approach is to capture in a simulator, the behaviour of distributed systems hardware and software components such as networks, hosts, multithreading, caching and RMI mechanisms. We then use the simulator to explore potential design and implemention choices. Our current simulator-driven design, called "MinORB", has been fully implemented and tested. MinORB supports scaleable high-performance by a combination of techniques including weak and application-specified consistency and partial replication using fine-grained proxy caching. Experimental results show that our work compares very favourably with other leading implementations such as OmniORB. Scalability is unparalleled with up to 1,000,000,000 objects per address space, a maximum throughput of 42,000 invocations per second and service times as low as 4 microseconds.

Keywords*: Distributed Objects, Weak Consistency, Partial Replication, Proxy Caching, Performance, Scaleability*

1. Introduction

The increasing availability of low-latency networks connecting high-speed hosts, allied to the demand for next-generation applications such as Intelligent Networks (IN) Call Control, World-Wide Web (WWW) applications and Collaborative Virtual Reality (CVR) has lead to a need for scaleable high-performance, constrained-latency application support that currently doesn't exist. The numbers of objects and wide-area access requirements of these applications often necessitate a distributed approach. These requirements have motivated the evolution of Distributed Object Programming Systems [1] and the resultant need for standardising architectures, such as CORBA [2]. CORBA provides a framework for automating repeated network programming tasks such as object location, binding and invocation. Unfortunately, conventional No-Replication implementations of CORBA incur a restrictive network latency speed penalty for applications that are performance sensitive and do not scale to large numbers of objects [3]. In order to achieve the necessary throughput and latency, programmers are forced to use either resource hungry Full-Replication approaches or resort to low-level transport-specific mechanisms [4].

An illustrative example would be IN services, such as 0800 Services or Personal Numbering, which are widely implemented as closed single-vendor systems with a single service running on a customised computing platform [5]. If a higher throughput or wider distribution is required then the platform (and the data on it) is often fully replicated and installed at a new site. This Full-Replication approach has so far been the most appropriate way to satisfy applications throughput and latency requirements as any host can satisfy any query. However this approach has also lead to scalability limitations in terms of the number of objects supported per site and also a high data-management overhead of keeping the data between sites consistent. Therefore seamless IN applications, that require global access to millions of objects, have not been possible using either No-Replication or Full-Replication approaches and accordingly our work explores Partial-Replication approaches.

The recent explosion in the accessibility of the Internet, and the WWW that runs over it, has led to some interesting approaches to satisfying the throughput demands of such applications. A well-known feature of popular sites is that request latency can be excessively high due to network and computational overload. Currently some of this overload is relieved of by use of caches which can be in the browser, server, or between the two, in proxy-servers. Caching proxy-servers provide a read-only cache of recently accessed Web pages and provide improved overall performance but, in common with any replication approach, introduce the possibility of inconsistency. Current access patterns to Web resources are strongly read-dominated, allowing servers to be stateless, but the advent of new applications such as Web-based electronic commerce and dynamic documents may change this simple model which impacts the approaches to handling consistency in the caches. Leading approaches under research such as Globe [6] seek to provide frameworks for supporting WWW applications via distributed shared O-O approaches and propose enabling consistency policies to be applied on a per-object basis.

CVR is a third application family that often relies on Full-Replication of in-memory databases on each collaborating host in order to satisfy throughput and latency goals. Recent years have show a growing research interest in scalability issues associated with supporting VR worlds. Leading projects include DIVE [7] and MASSIVE [8] who seek to support thousands of users with hundreds of thousands of simulation entities. They have exploited a number of methods for controlling the computational and network load of adding additional users and entities. Firstly, they have exploited applications semantics by restricting entity state updates to only be delivered to those parties that have expressed and interest rather than to all parties. Secondly, they have

determined that strict consistency between replicas is difficult to combine with real time interaction and have accepted some inconsistency tolerance in order to maintain rapid response times.

Therefore in order for any Middleware to be adopted for these target applications it must provide:

- **High Invocation Throughput**. Middleware implementations must be able to support sufficiently high invocation throughput to match the applications requirements. An example global IN application would potentially have billions of objects spread across 1000 hosts with 1000 call-attempts per second arriving at each host. Each call-attempt potentially results in 10 nested method invocations so the total throughput would be 10,000,000 invocations per sec [9].
- **Low Invocation Latency**. Middleware implementations must be able to provide low response times for method invocations. Typical OSF DCE RPC latency across a LAN is of the order of 5 milliseconds but this is strongly dependent upon hardware configuration. Our work assumes global distribution and overall latency increases with wider area distribution due to higher network latency as WANs become necessary.
- **Support for Large Numbers of Objects.** Middleware must scale efficiently to large numbers of objects in total. This infers both large numbers of hosts and large numbers of objects per host. In the context of the IN, WWW and CVR applications addressed by this paper, up to 1000 hosts and up to 1,000,000,000 objects per host are considered appropriate.
- **Reliability, Interoperability etc.** These additional topics are beyond the scope of this paper.

It has been widely recognised that use of DOPS has the potential to overcome the scalability limitations for our target applications. For example the Telecommunications Information Networks Architecture Consortium (TINAC) encourages interoperability and the advancement of the technology in this arena [10]. TINA has adopted CORBA as part of its architecture but as yet very little in-depth work has been published on performance-at-scale issues.

Real-time (RT) CORBA has recently been receiving a lot of research attention [11][3] but this early work is focused on applications such as avionics and process-control that require Hard Real-time (HRT) constraints for relatively small numbers of objects. Also some high-performance implementations of CORBA are under continuing development but these are not aimed at supporting very large numbers of objects and often have no consideration of latency constraints. Another approach that has been taken by vendors of existing CORBA implementations is to port their systems to Real-time Operating Systems (RTOS). However this does not address the sources of latency and scalability and throughput limitations in their Middleware implementations.

In this paper we describe our own CORBA-like Middleware called MinORB, which employs a number of techniques to produce a low-latency, high-throughput implementation that provably supports 1 billion objects per host at run-time and is expected to scale beyond this. Also described in this paper is our simulator, which is used to predict the run-time performance of an application with a given Middleware implementation on a known infrastructure and the appropriateness of using either RMI or Smart Proxy-Execution approaches.

The remainder of this paper is organised as follows: Section 2 describes our simulation of execution policies. Section 3 describes the run time MinORB optimisations. Section 4 describes our simulation and experimental results using OmniORB and MinORB. Section 5 briefly describes related work and Section 6 presents concluding remarks.

Section 2 - Simulation of RMI and "Smart Proxy" Execution Policies

This section describes our simulator which is used predict optimal run-time configurations for distributed applications on available hardware. The experimental methods that are used to validate our simulator and to test our implementation are also outlined.

Simulator Overview

One of the aims of our research is to build a distributed applications simulator that can be used for both pre-implementation design and pre-deployment capacity planning. Current approaches rely on rules-of-thumb and post-installation measurements, which can easily result in expensive under or over-capacity [12].

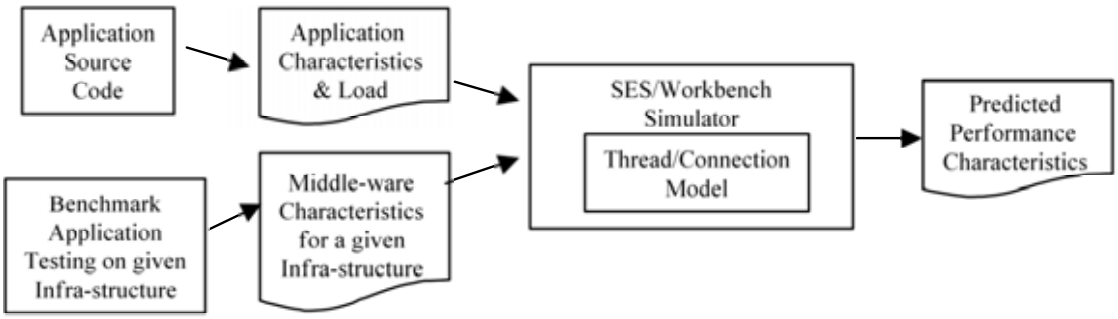

Figure 1: Overview of Middleware Simulator Process

The simulator requires knowledge of the application characteristics/load and the Middleware characteristics for a given infrastructure. Currently we can simulate a single application class with singly nested invocations using the both the RMI and caching or "Smart" Proxy execution policies. We model system behaviour as an event driven Finite State Machine (FSM) and example states would be RMI-Request-Writing and Object-Locating. The simulator is based on SES/Workbench [www.ses.com]

and we have built an abstraction layer to emulate the state machine of Solaris multithreaded executables [13] and the common ORB implementation "thread per connection" approach [14]. The simulator will return the average invocation latency in microseconds, invocation throughput in invocations/second and the server-to-server bandwidth demand.

Application Characteristics

The simulator must understand some application characteristics including object sizes, object counts, caching proxy counts and the run-time ratio of read operations to write operations (rw_ratio). Also required is the "lag", in microseconds, which is the time-based inconsistency that an application will tolerate between a write operation editing an object's data-members and the update being enforced at the caching proxies. Operations are described to the simulator in terms of either the number of data-member bytes edited for each write operation or required for each read operation.

A simple "Echo" application, as employed in our benchmarking tests, is used to illustrate these characteristics. An Echo object holds a single data-member, which clients can either query using the read operation `getValue()` or set using the write operation `setValue()`. The Echo class is defined below.

```
const int dm_size=4;

class Echo{
        char value[dm_size]; // lag <= 1000 usecs;
public:
        Echo();
        char* getValue() const;
        void setValue(char* v);
};
```

In this case the size of the Echo object is dependent on the single data-member array size and the tolerated divergence is limited to a maximum value of 1000 microseconds. Our focus of interest is on invocation throughput rather than the effects of operation parameter count, which has already been researched [15].

Infrastructure and Middleware Characteristics

The simulator also needs to have an understanding of the characteristics of the target Middleware implementation when used on the target infrastructure. Our approach requires access to a minimal subset of the infrastructure and the Middleware source code, which we can then instrument to capture timings. The infrastructure subset is a pair of networked hosts on which a series of benchmark "Echo" applications are run to capture the Middleware/Infrastructure characteristics.

The benchmark applications exercise various aspects of the Middleware implementation and the results are captured in terms of state dwell times such as RMI-Request-Write time or Object-Location time. To our great advantage Solaris 2.x is equipped with a high-resolution timer, `gethrtime`(), which returns results in nanosecond granularity. This function call incurs an overhead, which must be subtracted from the overall invocation latency.

RMI and Proxy Execution Polices

Our simulator allows us to specify a choice of two execution policies, which are illustrated in Figure 2.

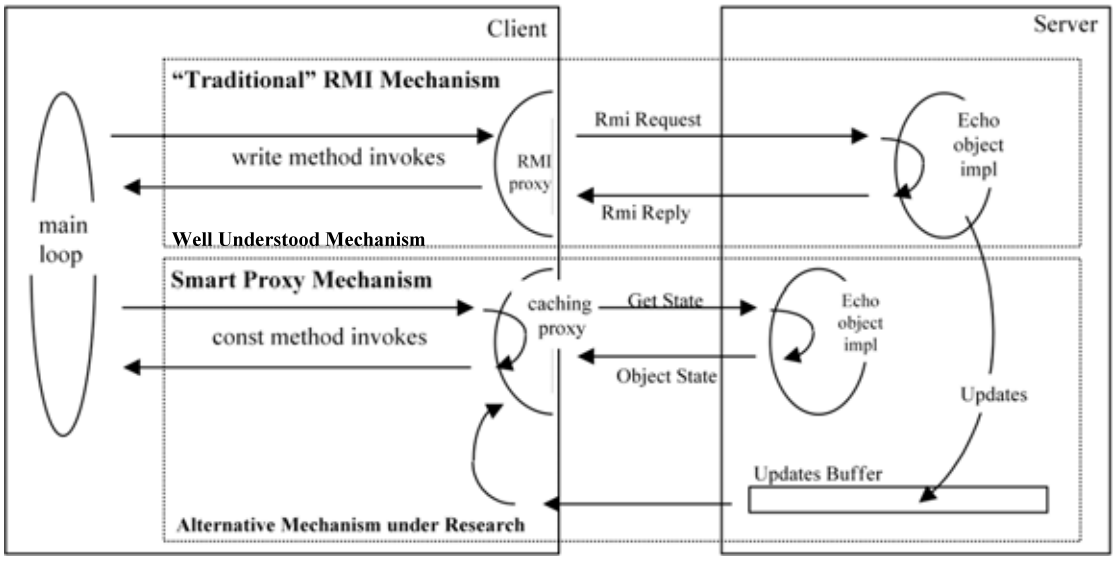

Figure 2. Schematic of RMI and Proxy-Execution

RMI (No-Replication). In this case each object exists on only one host and all invocations are directed to that host, which ensures single-copy serialisation. This approach has the advantages of having the minimum memory requirement and is a mature

and understood mechanism. However the invocation to remote objects will always involve O/S calls, a processing overhead and network latency [15]. The example benchmark dwell times using MinORB on our Dell test bed for RMI are given below.

```
Find RMI Proxy =                0 us
Write RMI Req. =               75 us.
Read RMI Req. =                61 us.
Channel Time =                115 us.
Find Object =                   8 us.
Execute Application Logic =    18 us.
Write RMI Reply =              74 us.
Read RMI Reply =               61 us.
Thread Switch =                65 us. (not used in OmniORB)
```

Proxy Execution (Partial-Replication). Partial-Replication means maintaining a portion of the whole remote object population on the local host. Our chosen implementation route involves taking advantage of the caching possibilities of object proxies. A number of researchers have proposed the use of object proxies [16][17][18] as an extension to RMI where object's state is transmitted across the network to the caching-proxies so read-only methods can be executed locally. Our research explores the numerous trade-offs necessary for this approach to provide a performance benefit. Amongst the important factors are: the ratio of read operations to update operations, the amount of data changed for each update operation, network latency, availability of memory for caching and application tolerance of cache inconsistency. In particular we have adopted existing update-based inconsistency techniques [19] and modified them to express and enforce time-based tolerances, which we feel is a necessary addition.

Section 3 - MinORB Implementation Optimisations

Given the experiences with Full and No-Replication approaches described in the foregoing sections, we could then employ our simulator to design a throughput optimised Partial-Replication ORB core. The MinORB executables are deployed on the target hosts on a one-process-per-host basis and are heavily multithreaded to take advantage of Symmetric Multiprocessors (SMP) performance. MinORB embodies a number of throughput optimisations that are not widely found in other implementations.

- **"Smart Proxy" Execution.** The most significant optimisation is to cache sufficient data members in smart proxies in order to satisfy client-side read operation execution. This requires MinORB to transfer state across the network, from objects to proxies, at proxy initialisation and when object's state is updated. Recognition of which object data members have been edited and the derivation of the minimum state to transfer is performed by the pre-compiler [20] and RTE in a manner that is transparent to the programmer. Our implementation of Proxy-Execution requires two new mechanisms in addition to the existing RMI mechanism. The Proxy-Initialisation mechanism is invoked in the event of a read operation on a remote object that is not in the local proxy-cache. The Proxy-Update mechanism is transparently invoked in the event of a write operation and ensures that updates are eventually sent to the caching proxies.
- **Application Specified Consistency.** MinORB allows the application programmer to relax the consistency constraints between the object's state and the cached proxy's state. This is manifest by the programmer supplied `"lag"` comment against data members in the C++ class definition. The tolerance of cache inconsistency allows the run-time system to use Application Layer Framing for updates and hence minimise network traffic and processor load. Our current implementation violates strict serialisation of concurrent invocations and future work will establish the serialisation/throughput trade-offs.
- **Zero-message copy.** Following the experience with OmniORB [21] and TAO [3], care has been taken to avoid multiple message copying. Messages are left in their thread-specific incoming buffer and only copied if there is a need to overwrite the buffer.
- **Near C++ speed invocations for co-located objects.** In common with a number of other systems, MinORB uses early recognition of objects being in the same address space and avoids unnecessary message passing. This optimisation is completely transparent to the application programmer.
- **Rapid Object and Operation Demultiplexing.** Existing systems use searching or hashing algorithms to locate objects and operations whereas MinORB uses knowledge of object naming to enable direct memory lookups and C++ `switch` statements.
- **Infrastructure Optimisations.** The MinORB RTE makes a minimum number of calls to system routines in order to minimise the execution mode switch overhead. Also the code of MinORB's RTE has been carefully designed to occupy the minimum space (~ 65Kbytes) and still provide sufficient functionality. Accordingly the RTE can fit in most modern processors L2 CPU text cache avoiding the need for main memory text fetches at run time. Lastly, MinORB executables can have a tailored data size in order to fit into the main memory of the target host. Once initialised, MinORB executables have a fixed footprint of a size that is intended to avoid the need for the O/S to page.

A number of other small or obvious optimisations have not been described for reasons of brevity.

Section 4 - Simulation and Test Results

This section presents our experimental and simulation results using both MinORB and OmniORB, mainly on a Dell test bed and some focussed experiments on a highly configured Sun E6000 test bed.

Dell Test Bed. Consists of two Dell 6100s configured with 4 Pentium Pro CPUs. Each CPU operates at 200MHz has 1Mbyte of L2 cache. Additionally each computer is equipped with 2 Gbytes of RAM and no VRAM, meaning no swap disks were configured. The O/S was 32 bit Solaris 2.6 with the Solaris threads library. The compiler was Sun Pro and was used with the "–fast" flag set. The computers were connected by a dedicated 100 Mbps NetGear switched Ethernet network.

E6000 Test Bed - This test bed was kindly loaned by Sun Microsystems [www.sun.com] and consisted of a highly configured Sparc E6000 that acted as the server and a client E4000. The E6000 had 16 250MHz Sparc CPUs each with 1Mbytes of L2 cache. Additionally, the E6000 had 18 Gbytes of high-speed RAM with 100 Gbytes of VRAM configured to swap across 30

disks. The E4000 contained 8 333Mhz CPUs with 512Kbytes of L2 cache and 1Gbyte of RAM. Importantly, the E6000 was selected by Sun as it is a server optimised for I/O dominated applications and incorporates the "Gigaplane" system interconnect, which can operate at 2.6 GB/sec. The network was a dedicated 100 Mbps switched Ethernet. The O/S was a beta release of 64-bit Solaris 2.7 with a beta release of the 64-bit "v9" Sun-Pro compiler. At the time of testing, the compiler was not fully optimised and the "–fast" flag was not operational.

Middleware - Firstly, and in most detail, an instrumented version of MinORB has been tested using both its RMI mode and then with Proxy Execution enabled. Secondly, and for comparison, OmniORB 2.6.1 has also been tested. OmniORB was selected as it met our criteria of being a high-performance ORB [15] and its source was available so it could be instrumented to establish state dwell times. Source inspection led to the identification of appropriate points in the code for instrumentation.

Test Case 1: RMI Latency vs. Number of Application Objects per Server

As a base line for our performance comparison, blackbox testing of the "traditional" RMI approach was undertaken on our Dell and E6000 test beds. The server object count was varied from 100 to 1,000,000,000. A serialised load was applied from the single client host using the random target object distribution and performed `getValue()` read operations or `setValue()` write operations. In both cases the computational load is equal and low. The results given below represent the average latency taken from the second 100,000 invocations (that is invocations numbered in the range 100,000 to 199,999).

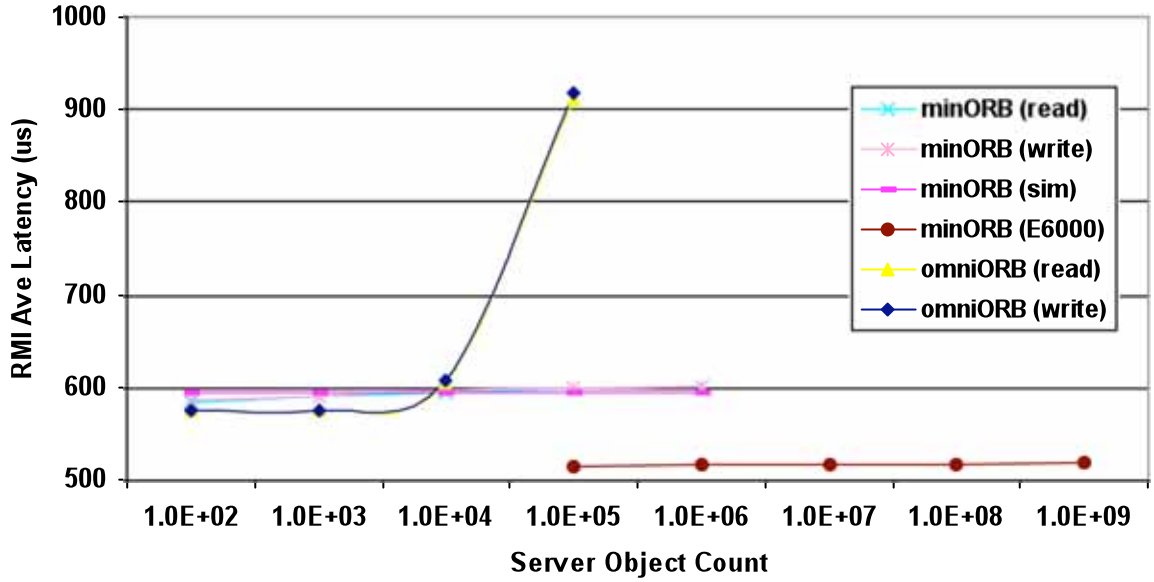

Figure 3: Variation of RMI latency with Object Count per Server, Dell Test Bed

The results show, as expected, that the rw_ratio has no significant effect on the RMI latency for any server object count with either implementation. This is because the read and write operation application code results in a similar computational load.

The OmniORB latency with 100 objects is very low at 575us which we feel confirms the claim that OmniORB is "probably the fastest CORBA2 compliant ORB available" [15][21]. From 100 to 500,000 objects, OmniORB latency increases with the object count up to a large latency of 2,731 us (nearly 3 milliseconds). Our whitebox testing reveals the reason for this latency increase is hash-table overflow during object de-multiplexing and could probably be reduced with tuning. The MinORB latency with 100 objects was 585us, which is slower than OmniORB. However as the number of objects increases the MinORB latency increases only slightly to 600us at 1,000,000 objects, giving better performance at scale which is the research goal of our work.

OmniORB supports up to 953 thousand objects on our Dells before memory exceptions prevented further testing. An OmniORB overhead of approximately 256 bytes per object, and a similar amount per proxy, leads to this memory exhaustion and, on systems configured with swap space, would lead to O/S paging. It should be noted that the OmniORB demand is less than other tested ORBs [15]. MinORB has a 24-byte overhead per object (or 88 byte on 64 bit Solaris 2.7) and accordingly can support at least 1 million objects on the same infrastructure. Significant testing beyond this point on the Dell test bed would exhaust the 32-bit address-space of Solaris 2.6.

OmniORB uses a simple single send-and-receive "thread-per-connection" approach, which allows only one invocation to be "in-flight" at any one time per connection. This contrasts with the less direct MinORB approach of one thread that sends a RMI request and blocks on a mutex, and another dedicated thread that reads the RMI replies and wakes the original blocked thread, thus incurring a thread switch overhead. We believe the MinORB approach makes more efficient use of the connection by allowing many invocations to be in-flight concurrently and thus provides throughput-at-scale benefits. Accordingly, we are prepared to accept the initial latency penalty. Our simulator results show good co-relation with experimental results.

E6000 Testing: In order to determine the efficiency and scalability limits of our object de-multiplexing scheme, we repeated this case on the E6000 test bed. The E6000 limitation was not the 64-bit address-space, but that 18 Gigabytes of physical RAM became exhausted at 558 million application-level objects and hence O/S paging started. The results confirm the trend observed

on the Dell test bed with the latency of MinORB RMI increasing only slightly with object count. In absolute terms, the average latency of E6000 MinORB RMI being around 520 us, which is only 120 microseconds over the time for a simple RTT (401 us). Also the latency variation with object count is encouraging, showing latency increasing from 516 us at 100,000 objects to 520 us at 1,000,000,000 (1 billion) objects. When O/S paging takes place the latency increases sharply, from approximately 520 us for a page-hit to 12,096 us for a page-miss, which represents a 24-fold degradation.

Test Case 2: RMI Throughput vs. Number of Concurrent RMI Clients

Concurrent execution is important to our target applications as it strongly affects invocation throughput and the RMI test case was repeated with MinORB on the Dell and E6000 test beds with 1 to 9 random invocation pattern clients. For the purposes of our investigation a mutual-exclusion lock was obtained on the whole object for the duration of the `setValue()` operation. Server object count was set to 100,000, clients operated in an asynchronous mode with separate send and receive threads and the invocation pattern was fully write-dominated.

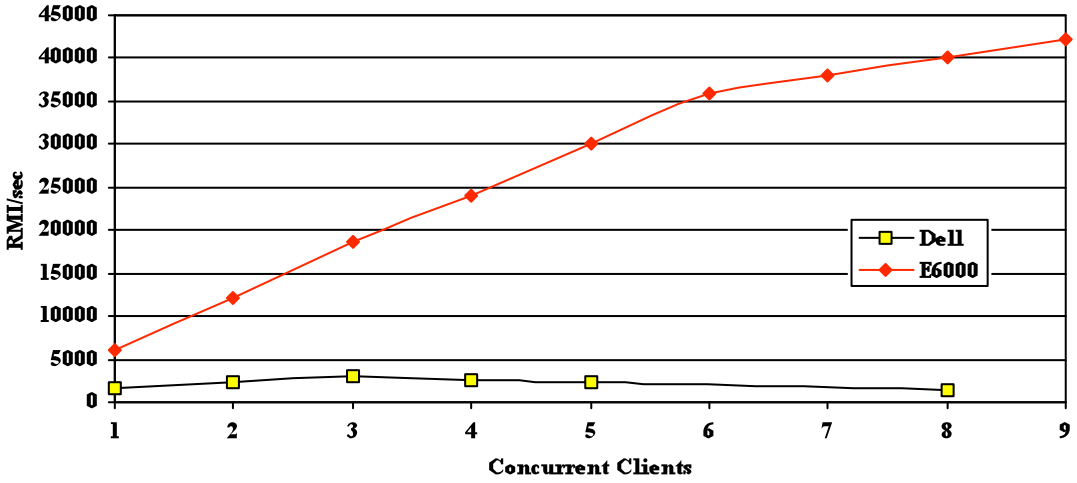

Figure 4: Variation of MinORB RMI Throughput with Multiple Clients.

Using MinORB on the Dell test bed produced linear scalability up to 3 clients, which demands 3 threads on the server and 6 threads on the client host are active. Accordingly beyond the 3-client point, thread contention for CPUs is evident and has the effect of reducing overall throughput.

On the E6000 test-bed almost linear scalability is demonstrated until the network became saturated at 6 clients and clients 6 to 9 ran on the server host. A single point result using 9 clients recorded a maximum throughput of approximately 42,000 invocations per second. With 16 CPUs there was no contention for thread to CPU mapping, which goes some way to explaining the linear scalability demonstrated. It is expected that with a greater throughput LAN and more highly configured client hosts then the linear scalability of throughput would have continued beyond 5 clients. In absolute terms the throughput of approximately 6,000 invocations/sec per client and would be satisfactory for the most demanding of IN applications. However it should be remembered that these invocations are not nested and are only invoked across a LAN rather than an end-to-end IN transaction.

It can be seen that the multithreaded design of MinORB complements the design of Solaris and the hoped for scalability of throughput has been realised. This remains the case until the number of highly active threads exceeds the number of CPUs leading to contention.

Test Case 3: rw_ratio and Caching

Strong factors in deciding the appropriateness of Proxy-Execution are the rw_ratio and the availability of caching storage. For this case the server object count is fixed at 100,000 and the rw_ratio and caching are varied. The update tolerance was set high meaning that all updates are buffered, and the amount of changed state per write operation was 4 bytes.

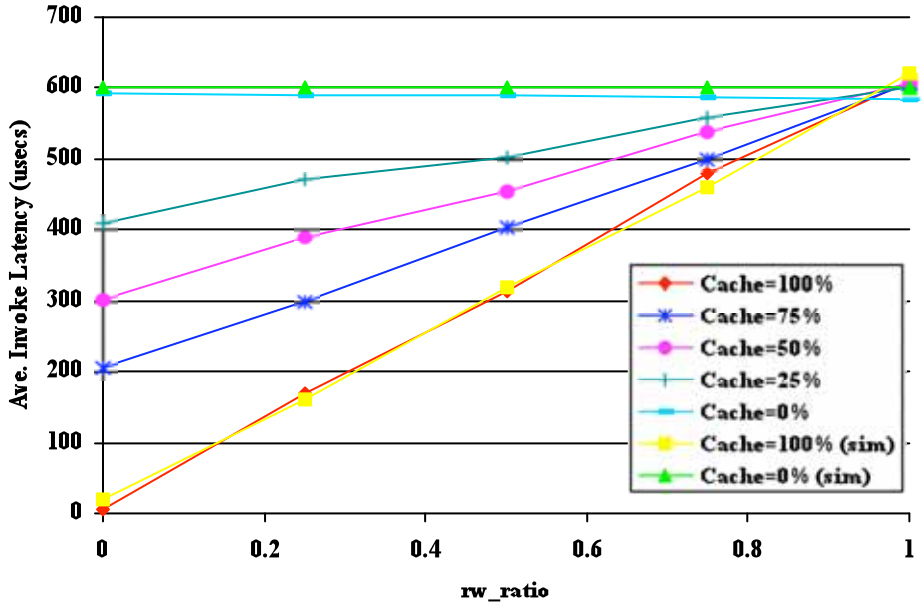

Figure 5: Invocation Latency Variation with rw_ratio and Caching on the Dell test bed

For the zero caching case, which forces the RTE to use only the RMI mechanism, the latency doesn't vary with the rw_ratio which confirms the results from Case 1. Where 100% caching is used, the latency varies greatly with the rw_ratio. When read-dominated, the invocations can all be satisfied by cached proxies on the client-side and the latency drops to 6us for the Dell test bed or 4us for the E6000 test bed. This is a 100-fold speed-up and as no messaging takes place this represents a very encouraging reduction in both latency and bandwidth demand. The E6000 results (not shown) confirm the trends seen on the Dell test bed.

With 100% caching and fully write-dominated, the latency increases beyond the RMI case by a small amount. The increase is due to the overhead of update messages that are required for each caching proxy after a write operation. It is notable that the update mechanism overhead is not prohibitive even for this worst case. Coulouris [22] anticipates that the update mechanism has "some" effect, but we have demonstrated this effect is minimal on suitably configured SMPs.

Where partial caching/replication is used the latency varies between the two above cases roughly in proportion to the amount of caching employed. Satisfactory agreement is achieved between experimentation on the Dell test bed and our simulator [13]. Also, with serialised load, throughput is the direct inverse of latency, so the 100-fold reduction in latency will relate to a 100-fold increase in throughput.

Test Case 4: Latency Overhead of Updating Proxies vs. Time Based Inconsistency Tolerance

This test was performed on 100,000 server objects, each with a single 4-byte data member using the random target object distribution. To expose the worst case we have selected a fully write dominated invocation pattern that will force continual updates.

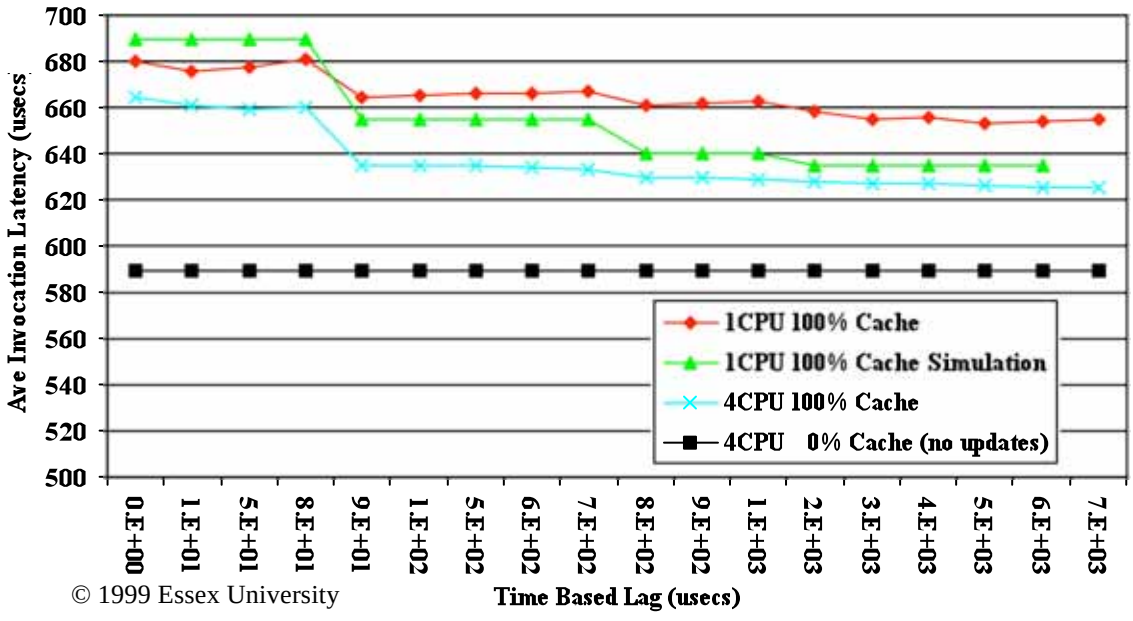

Figure 6: Write Operation Latency vs. Time Based Lag

A clear stepped behaviour can be observed as the tolerated lag is increased. For instance the highest latency of about 680 us is at a lag of 0 to 80 us, followed by a step down to 663 us where the lag is 90 to 700 us. After this latency stabilises at about 652 us for increasing values of tolerated lag. Observation of the update message count led us to the clear conclusion that the steps are directly related to the number of updates that are sent in each message. For example, up to a lag of 80 us, the buffer was being sent for every update. The second step represented a send for every two updates and produced a reduced latency. After this step, three-update sends and greater are more difficult to separately identify but the general trend is a reduced latency and a reduced message count. As the inconsistency tolerance increases to 6000 us, the update buffer, which is sized to the Maximum Transfer Unit (MTU) size, is dispatched as it becomes full rather than as the time lag expires so beyond this point no further reduction latency is expected or observed. We repeated the test using all 4 Dell CPUs and the average latency was reduced by 25-30 us. This was due to the drop in thread context switching and lead us to the conclusion that optimum performance is more likely to be achieved if the number of CPUs equals or exceeds the number of highly active threads.

It can be seen that the reduction in latency of increased lag yields some limited latency benefits, but another metric is the reduction in message count and hence bandwidth demand. Increasing lag allows the RTE to buffer updates, which results in a maximum 20-fold reduction in server-to-server update messages when only 4 bytes are edited per write operation.

Section 5: Related Work

The ACE Orb (TAO): The work at Washington University has been the focus of real-time research on CORBA implementations is very well documented [3] and much of their experience with threading and buffering approaches has influenced our work. Additionally, they have performed a number of insightful benchmark performance tests of existing ORBs, which has lead to a wider appreciation of the true bottlenecks (which have been masked on low speed networks) and sources of non-determinism, such as de-multiplexing and marshalling, in ORB invocation implementations. Finally, experience with TAO has guided the development of methods to express and enforce real-time Quality of Service (QoS) requirements, which are now being down-streamed into CORBA specifications.

Whilst their work is highly relevant, the initial focus is on Hard Real Time applications, such as Avionics, that require only thousands of objects. Accordingly their approach is unlikely to satisfy the scalability and throughput requirements of our target applications.

OmniORB: OmniORB2 is a CORBA2.0 compliant ORB implemented and freely distributed, including source code, by AT&T UK Research Labs. OmniORB has a number of advantages, which include relative bug-freeness, scalability and high performance. The OmniORB performance is well-documented [15] and their threading and buffer management strategies have been strongly influential on our work. A notable achievement of their work is that it has demonstrated that the use of IIOP, as the native protocol, is not prohibitively expensive for the cases demonstrated in their benchmark applications. Also they have demonstrated OmniORB working across a number of transports including TCP/IP, Shared Memory and ATM AAL5.

Unlike OmniORB, we wish to take advantage of the GIOP specification that allows the client to have multiple outstanding requests and replies can be returned in any order. This would compliment our high-throughput multithreaded clients. Accordingly, we believe that higher throughput can be yielded from a hybrid Thread(s)-per-Connection/Shared Pool architecture.

Globe: Globe is a current project running at Virje University that has identified that developing large-scale wide-area applications requires an infrastructure that is presently entirely lacking [6]. In common with our work, their most important goal is to enabling distributed, shared objects with global scalability. Importantly, they too propose a number of approaches to cached objects, including weak-consistency and server initiated update mechanisms.

This work is, in principle, closely related to the work of this paper. However, it is possible that the Globe approach of providing per-object policies will entail a large per-object data overhead, leading to an objects-per-host limitation and a larger than necessary per-invocation computational overhead. Implementation work on Globe is incomplete and therefore no experimental results can be expected.

Section 6 - Summary

Traditional Full-Replication approaches lack scalability and No-Replication RMI approaches are inherently slow. MinORB RMI implementation demonstrates low latency (approximately 100 us over RTT on either test bed), high-throughput and good scalability, which has been tested up to 1,000,000,000 objects per server, and compares well with other leading RMI based implementations such as OmniORB. Good agreement is demonstrated between our simulation and experimentation results, which goes some way to validating our simulator. However RMI alone can never overcome the problems of network and end-system latency.

For our test bed, Proxy-Execution approaches can reduce the latency and increase the throughput 100-fold for read-dominated applications with small amounts of update information, as long as sufficient RAM and CPU resource is available. As the write domination or the size of edited data increases the latency and throughput benefits reduce, but generally outperform the RMI case. The overhead of updating caching proxies has been shown to be tolerable in a uni-processor environment and is reduced further in a sufficiently configured multiprocessor and multithreaded environment.

The envelope of operations under which Proxy-Execution is beneficial can be extended by the use of application-defined time or update-based inconsistency tolerance. Latency and throughput is improved as the tolerance increases, up to the size of an update buffer reaching the size of an MTU. More significantly, the number of server-to-server messages is greatly reduced, reflecting a reduced demand for WAN bandwidth. As the network latency increases, so do the benefits of Proxy-Execution and Inconsistency Tolerance.

In conclusion, it is clear then that Proxy-Execution and Inconsistency-Tolerance are lucrative techniques applications that are read dominated, globally distributed, have limited edits and tolerate "stale" data. Therefore applications such as IN, CVR and WWW, that share these characteristics are likely to benefit.

As part of future work we intend to validate our simulator when modelling larger scale applications and extend it to cover a larger sample of infrastructure and Middleware. We also intend to explore, through simulation and experimentation, the benefits of unifying the currently separate RMI, proxy-initialisation and update mechanisms. In the longer term, we aim to introduce fault tolerance and transactional mechanisms to MinORB.

Acknowledgements

We would like to thank many colleges at BT Labs and, in particular, David Eastaugh for his work on the simulator. Also we would like to thank Sun Microsystems for their kind loan of equipment and especially Phil Harman for his help with Solaris multithreading.

References

[1] Roger S. Chin and Samuel T. Chanson. "Distributed Object-based Programming Systems". ACM Computing Surveys, Vol23, No 1, March 1991.

[2] Steve Vinoski. "New Features for CORBA 3.0". Communications of the ACM. October 1998/Vol41 No 10.

[3] Douglas C. Schmidt et al. "A High-Performance End System Architecture for real-time CORBA". IEEE Communications Magazine Feb 1997.

[4] N. Bogunovic. "Visual Resource management in Heterogeneous Distributed Computing Environments". Proceedings of the 10th IASTED PDCS Conference. ISBN 0-88986-231-1

[5] T W Abernathy and A C Munday. "Intelligent Networks, Standards and Services" Chapter 1 of Network Intelligence, Chapman and Hall, ISBN 0-412-78920-5

[6] Steen, Homberg and Tanenbaum. "The Architecture of Globe: A Wide-Area Distributed System" Vrije Universitieit, Internal Report IR-0422 March 1997

[7] E. Frecon and Marten Stenius. "DIVE: scalable network architecture for distributed virtual environments". Distributed Systems Engineering Vol5 No 3 September 1998.

[8] Chris Greenhalgh. "Awareness based communication management in the MASSIVE systems", Distributed Systems Engineering Vol5 No 3 September 1998.

[9] Hack. H. Kim. "Intelligent Network Services in the Next Decade". IN Symposium Estoril '98.

[10] Fabrice Dupuy, Gunner Nilsson, and Yuji Inoue. "The TINA Consortium: Toward Networking Telecommunications Information Services" IEEE Communications Magazine, November 1995.

[11] Victor Fay Wolfe et al. "Expressing and Enforcing Timing Constraints in a Dynamic Real Time CORBA System". Dept of Computer Science. University of Rhode Island. Kingston RI 02881. 1997.

[12] Brian L. Wong. "Configuration and Capacity Planning for Solaris Servers". Prentice Hall, Upper Saddle River, NJ 07458. ISBN 0-13-349952-9

[13] David Eastaugh. "An Investigation into the Performance and Scalability of a Distributed Computing Environment tailored for use in the Telecommunications Industry". Loughborough University Dept. of Mathematics. MSc dissertation, September 1998.

[14] Douglas C. Schmidt. "Evaluating Architectures for Multithreaded Object Request Brokers". Communications of the ACM October 1998/Vol41 No 10.

[15] MLC Systeme GmbH & Distributed Systems Research Group. "CORBA Comparison Project - Final Project Report". Dept. of SE, Faculty of Mathematics & Physics, Charles University, Malostranske namesti 25, Prague, Czech Republic. WWW http://nenya.ms.mff.cuni.cz.

[16] S.J. Caughey, G.D.Parrington and S.K. Shrivastava. "SHADOWS - A Flexible Support System for Objects in Distributed Systems". 1993 IEEE 0-8186-5270-5/93.

[17] Henri E. BAL, M. Frans Kaashoek and Andrew S. Tanenbaum. "Replication Techniques for Speeding up Parallel Applications on Distributed Systems". Concurrency: Practice and Experience, Vol. 4(5), 337-355 (August 1992). John Wiley 1040-3108/92/050337.

[18] Matthew J. Zelesko and David R. Cheriton. "Specialising Object-Oriented RPC for Functionality and Performance". Proceeding of the 16th ICDCS. 1063-6927/96.

[19] Y Huang et al. "Divergence Caching in Client-Server Architectures", IEEE 0-8186-6400-2/94.

[20] P Martin, V Callaghan and A Clark. "DC+1: An approach to automated application partitioning and distribution transparency". Proceedings of the 10th IASTED PDCS Conference. ISBN 0-88986-231-1.

[21] Sai-Lai Lo and Steve Pope. "Implementation of a high-performance ORB over Multiple Transport Networks". Proceedings of Middleware'98, Springer ISBN 1-85233-088-0.

[22] Coulouris, Dollimore and Kindberg. "Distributed Systems, Concepts and Design 2nd Edition". Addison-Wesley. 9780201624335.

