A High Performance CORBA ORB with Support For Multiple Network Transports

Sai-Lai Lo, Steve Pope
Olivetti & Oracle Research Laboratory
24a Trumpington Street
Cambridge CB2 1QA
England
(email: s.lo@orl.co.uk)

16 March, 1998

Abstract

This paper describes the implementation of a high performance Object Request Broker (ORB)– omniORB2. The discussion focuses on the experience in achieving high performance by exploiting the protocol and other characteristics of the CORBA 2.0 specification. The design is also highly adaptable to a variety of network transports. The results of running the ORB over TCP/IP, shared memory, Scalable Coherent Interface (SCI) and ATM Adaptation Layer 5 (AAL5) are presented. In both null calls and bulk data transfers, the performance of omniORB2 is significantly better than other commercial ORBs.

1 Introduction

In this paper, we describe the implementation of a high performance Object Request Broker (ORB)-omniORB2. OmniORB2 is the second generation ORB developed at ORL (hence the name "omniORB two"). The initial goal was to produce a standard conforming ORB that can deliver the performance required by the applications developed in-house.

In the past, we have used different RPC-style communication systems in various projects. For instance, the Active Badge system [6] was built on top of the Ansaware. Our experience have shown that while a good programming paradigm for building distributed system is important, it is of little practical use if the implementation cannot deliver adequate performance. For instance, the Medusa system [7] was developed with its own message-passing system partly because the RPC systems at the time were found not to be able to deliver the performance required.

The ability to reuse a distributed system infrastructure over a new network transport is also an important consideration. Medusa was designed to run directly on top of ATM transports. Existing RPC systems at the time were usually designed to run on top of UDP or TCP. The effort required to add the ATM support is almost the same as writing a message-passing system from scratch. The need to support a new transport is not an one-off requirement. The laboratory has always been interested in new networking technologies. ATM is extended into the wireless domain by Radio ATM [3]; a low-power and short range radio networking system (Piconet) [2] is being developed; and the potential of high performance interconnects is being explored. Clearly, it is important to design the ORB so that a new network transport can be added without changing the majority of the code base.

It is becoming technically and economically feasible to connect a cluster of (inexpensive) computers by very fast interconnects to create a high performance distributed computing platform. These interconnect technologies, such as SCI and fiber channel, offer very high bandwidth and, more importantly, very low latency user-space to user-space data transfer. This has the effect of removing the dominant part of a RPC round-trip time, i.e. the delay introduced by the kernel and the network protocol stack. In this environment, the overhead incurred by the so called "middleware" will become the dominant factor in deciding the performance of remote invocations. Although omniORB2 is primarily used with conventional networking, we try to minimise the call overhead as much as possible and at the same time stay compliant with the CORBA [4] specification. We hope that in future the design will prove to be useful in low-latency computer clusters. The initial results of running omniORB2 over SCI are encouraging.

OmniORB2 has been deployed for lab-wide use since Mar 1997. It implements the specification 2.0 of CORBA. The IDL to C++ language mapping is provided. It is fully multi-threaded. In May 1997, the ORB was released externally as free software under the GNU public licences[1]. Since then, two more public releases have been made. The development of the ORB is continuing. Its user base is growing steadily.

In this paper, we present the considerations and choices made in the design of omniORB2. The discussion will focus on the experience in achieving high performance by exploiting the protocol and other characteristics of the CORBA specification. The results of several performance tests are included. The data show that in both null calls and bulk data transfers, the performance of omniORB2 is significantly better than other commercial ORBs. The results of running omniORB2 over several transports: ATM/AAL5, shared memory and SCI, are presented. The data show that omniORB2 is highly adaptable to the characteristics of these transports and can achieve a significant improvement in performance over the TCP/IP.

2 Internal Architecture

CORBA provides a standard framework for building distributed applications. Application components interact via user-defined interfaces specified in an Interface Definition Language (IDL). The ORB is the "middleware" that allows a client to invoke an operation on an object without regard to its implementation or location.

In order to invoke an operation on an object, a client must first acquire a reference to the object. Such a reference may be obtained as the result of an operation on another object (such as a naming service or a factory) or by conversion from a "stringified" representation previously generated by the same ORB. If the object is remote (i.e. in a different address space), the ORB has to create a local representation of the object- a "proxy" in the client's address space. From the client's view point, the proxy object is indistinguishable from the object itself. When the client invokes an operation on the proxy object, the ORB delivers the invocation to the remote object. Exactly how the ORB arranges to deliver the invocation is not specified by CORBA. Nevertheless, in order to permit interoperability, CORBA mandates that IIOP (Internet Inter-ORB Protocol), a RPC protocol layered over TCP/IP, be supported by all ORBs. However, the standard does not preclude the use of other transport mechanisms when available.

Figure 1 outlines the main components that are involved in the delivery of remote invocations. The functions of the components are as follows:

2.1 Proxy and Implementation Skeleton

For each IDL interface, the IDL compiler generates a proxy object class and an implementation skeleton class. The proxy object provides a local represent of the remote object. The application

[1] More information is available at *http://www.orl.co.uk/omniORB2/omniORB2.html*

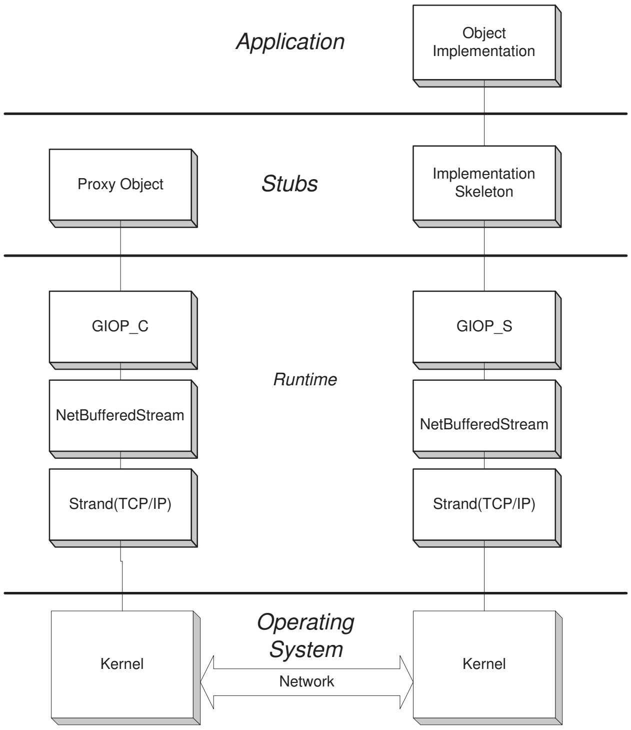

Figure 1: Internal Architecture

provides the object implementation and connects to the ORB upcall structure via the implementation skeleton class.

2.2 GIOP_C and GIOP_S

GIOP_C and GIOP_S together drive the General Inter-ORB Protocol (GIOP). IIOP is a mapping of GIOP over TCP/IP. We choose to layer GIOP, not just on top of TCP/IP, but on all the network transports available to the ORB. Of course it is possible to use an entirely different protocol for a non-TCP/IP transport. While we think GIOP is hardly the most optimised protocol and a more efficient Environment Specific Inter-ORB Protocol (ESIOP) (a term used in CORBA to refer to other non-GIOP protocol) could be designed, the overhead incurred by the increased complexity to accommodate two protocols may outweigh any performance benefit. For instance, the overhead in multiplexing incoming requests via different protocol paths to the same upcall structure on the server side could be substantial.

Using GIOP-only allows us to tightly integrate the run time and the IDL interface specific stubs. One benefit of the tight integration is that some of the more complex and less performance critical functions in driving the GIOP protocol can be delegated to the stubs. One example is in handling location forwarding. In response to an invocation, the server could response with a location-forward message to point the client to the new location of the object. The client is expected to retry the invocation, without any application intervention, at the new location. If the runtime is to retry the invocation, it must keep the complete request in its own buffer until a reply comes back from the server. For calls with very large arguments, this will result in excessive buffer allocation. In omniORB2, GIOP_C does not perform the retry, control is passed back to the stub code and the call arguments are re-marshalled. There is no need for the runtime to buffer the whole request message. As a matter of fact, the runtime handles large call arguments by sending it off to the network while the rest of the request is being marshalled by the stubs; only a small amount of buffer space is needed.

2.3 NetBufferedStream

NetBufferedStream performs data marshalling from call arguments to a bidirection data connection. Marshalling functions for all IDL primitive types and for a vector of bytes are provided. Ideally, we would like to generate optimal marshalling code from an IDL interface. In practice, we found that the scope to speed up data marshalling is somewhat limited by the coding rules of GIOP.

GIOP specifies that all primitive data types must be aligned on its native boundaries. Alignment is defined as being relative to the beginning of a GIOP message. For instance, a *double* which is 8 bytes in size must start at an index that is a multiple of 8 from the beginning of the GIOP message. Where necessary, padding bytes must be inserted to maintain the correct alignment. Unfortunately, a GIOP request or reply message starts with a variable size header. The size of the header is only known during the invocation because it contains fields that are variable length and context dependent. It is therefore not possible to precompute at compile time whether padding bytes have to be inserted to make a call argument correctly aligned. Also, fixed size *struct* and *union* constructor types cannot be marshalled in bulk and has to be done for each field member individually[2].

Despite of these limitations imposed by the GIOP specification, there is still scope for further optimisation in the marshalling code. Future omniORB2 releases will further improve the stub code to speed up data marshalling. Techniques used in previous work such as Flick [1] may be useful.

3 Threading Model

OmniORB2 uses the thread-per-connection model. The model is selected to achieve two objectives. Firstly, the degree of concurrency is to be maximised while any thread overhead is kept to a minimum. Secondly, the interference by the activities of other threads on the progress of a remote invocation is to be minimised. In other words, thread "cross-talk" should be minimised within the ORB. The thread-per-connection model helps to achieve these objectives because the degree of multiplexing at every level is kept to a minimum.

Figure 2 shows the call-chain of a remote invocation. On the client side of a connection, the thread that invokes on a proxy object drives the GIOP protocol directly and blocks on the connection to receive the reply. On the server side, a dedicated thread is spawned per connection (this is why the threading model is called thread-per-connection). The server thread blocks on the connection until a request arrives. On receiving a re-

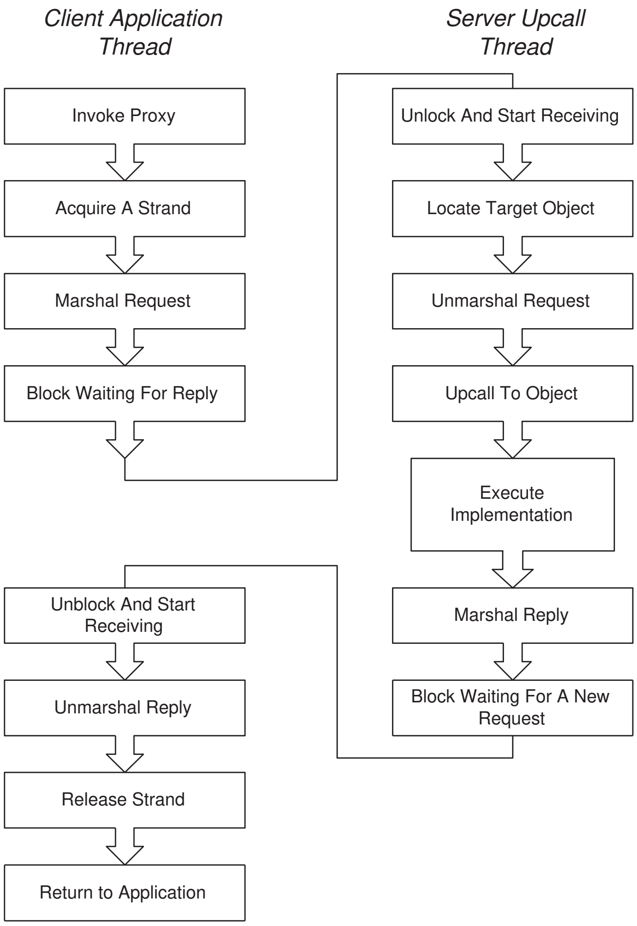

Figure 2: Call-chain of a Remote Invocation

[2]This is because the memory layout of constructor types is both compiler and processor dependent. Even if the memory layout agrees with the GIOP alignment rules, the internal alignment between field members will be affected by what precedes the structure in the GIOP message.

quest, it performs the upcall to the object implementation and sends the reply when the upcall returns. There is no thread switching along the call chain.

Note that there is at most one call "in-flight" at any time in a connection. The GIOP specification allows the client to have multiple outstanding requests down the same connection at the same time; the replies can be returned in any order. To use this call multiplexing scheme, both ends must implement two levels of thread dispatch. In the lower level, each end must have a dedicated thread to (de)multiplex the call. Once demultiplexed, the call has to be dispatched to the upper level: the application threads on the client side and the upcall threads on the server side. To process a call, this involves four thread switchings (two on each side). The overhead is simply too high and there is little/no advantage over the "one-call-per-connection" scheme used in omniORB2. Worst, the two-level thread dispatch structure introduces undesirable "cross-talk" among different threads because all the remote invocations are multiplexed down to a single thread at both ends.

If there is only one connection, concurrent invocations to the same remote address space would have to be serialized (because at most one call is "in-flight" per connection). This severely limits the degree of concurrency. Also, a deadlock can occur[3]. To eliminate this limitation, omniORB2 creates multiple connections to the same remote address space on-demand. In other words, if two concurrent calls are made to the same remote address space, the ORB would establish two connections to handle the call.

Instead of one thread per connection, some ORBs use a fixed pool of "worker" threads on the server side to dispatch upcalls. The idea is to limit the degree of concurrency or to put an upper bound on the resources (in this case the thread objects) used by the applications. Like the multiplexing of multiple calls onto the same connection, this model incurs a significant thread switching overhead. Furthermore, it is unclear whether the number of threads used by an application is more likely to be a limiting factor than other resources, such as the number of network connections it is allowed to open. For these reasons, we think thread-per-connection is more suitable as the default threading model.

4 Dynamic Connection Management

In order to invoke an operation on an object, the ORB arranges for invocations on the proxy object to be transparently mapped to equivalent invocations on the implementation object. In CORBA, as in most distributed systems, remote bindings are established implicitly without application intervention. CORBA does not specify when such bindings should result in a connection being established between the two address spaces. Instead, the ORBs are free to implement implicit binding in a variety of ways.

In omniORB2, connections are established only when invocations have to be passed on to other address spaces. It is obviously a better approach than, for instance, to always open a connection to the remote object whenever the ORB is building a proxy object in the client's address space[4].

As discussed in the previous section, multiple connections to the same remote address space may be established when there are multiple calls in progress concurrently. These connections are managed collectively by a "Rope". The Rope abstraction provides an association between two address spaces. The connection point of each address space is identified by an Endpoint. On de-

[3]Consider the case where object A is in address space 1 and object B and C are in address space 2. In address space 1, a thread invokes on object B; B then performs a nested invocation on A; A in turn performs a nested invocation on C. At this point, there are two calls originating from address space 1. The first call can only be completed after the second call returns. If the second call is serialised, a deadlock situation occurs.

[4]Consider a server, such as a naming service, which acts only as a repository of object references for other clients; the server would not invoke on these objects. The naive approach of connecting to all the remote objects would quickly overwhelm the server with unnecessary connections.

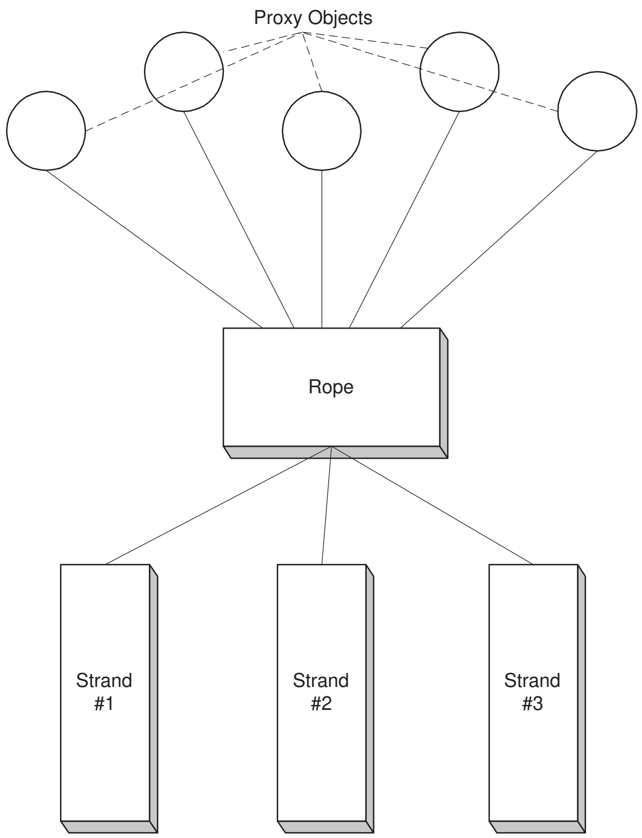

Figure 3: Dynamic Connection Management

mand, a Rope creates a "Strand" to connect the two Endpoints. The Strand abstraction is a bidirectional data connection. Each Strand is mapped directly onto a network connection. In the later part of this paper, we'll discuss the implementation of the Strand abstraction over multiple network transports.

Just as a real rope often consists of multiple strands, a Rope may be composed of one or more Strands. The relationship between the entities are further illustrated in figure 3. As indicated in the figure, every pair of address space is always associated by one Rope. This association is shared by multiple proxy objects that bind to the implementation objects in the other address space. When the client invokes an operation via a proxy object, exclusive access to a Strand is acquired from the Rope. At this point, the Rope may have to create a new Strand in order to satisfy the demand. When the invocation is completed, the Strand is returned to the Rope.

Once created, a Strand is left opened in case it might be reused again by other invocations shortly afterwards. However, it is wasteful to leave a Strand opened when it has been left unused for a considerable time. It is particularly bad on the server side because too many idle network connections could block out new connections when the server process runs out of spare channels. For example, most unix platforms has a limit on the number of file handles a process can open; 64 is the usual default limit although this can be increased to about 1000. For this reason, the Strands are scanned periodically. A Strand found to be unused for a period of time would be closed. To decouple this housekeeping action from the critical path of executing an operation on a remote object, the task is dedicated to two separate threads running in the background. One thread is responsible for outgoing connections and the other looks after incoming ones. All opened connections are scanned every "scan period". If a connection is found to be idle for two consecutive periods, it will be closed. The threads use mark-and-swipe to detect if a connection is idle. The details are as follows. When a connection is checked, a status flag attached to the connection is set. Every remote invocation using that connection would clear the flag. So if a connection's status flag is found to be set in two consecutive scans, the connection has been idled during the scan period.

To support a large number of clients, it is important for the server to be able to shutdown idle connections unilaterally. However, this function is only useful when the client is able to distinguish such an orderly shutdown from those caused by abnormal conditions such as server crashes. When omniORB2 is about to close a connection on the server side, it uses the *GIOP closeConnection* message to inform the client this is an orderly shutdown. The client should recognise this condition and retry an invocation transparently if necessary. This is in fact the compliant behaviour. What it should not do is to treat this as a hard failure and raise a *COMM_FAILURE* system exception. Unfortunately, very few of the ORBs we have tested for interoperability are able to response correctly.

As illustrated in figure 3, the default behaviour of omniORB2 is to multiplex proxy objects to the same remote address space onto a common pool of connections. Some applications may prefer a dedicated connection per proxy object. For instance, a proxy object may be used to push live video to the other address space. The problem with implicit binding, i.e. the default behaviour of omniORB2, is that it provides no opportunity for the application to influence the ORB's binding policy on a per-binding basis. This makes it impossible for applications with specific performance requirements to control, for instance, the timing of binding establishment; the multiplexing of bindings over network connections; and the selection of the type of network connection itself. This is an issue that has not been addressed by existing CORBA standards. We believe one solution is to augment the binding model with explicit binding. In other words, the application is given direct control on setting up bindings via proxy objects. This is work in progress and the details are beyond the scope of this paper.

5 The Strand

The Strand abstraction provides a bidirectional data connection. We will discuss the implementation of this abstraction on top of several network transports. But first, we describe the distinct features of its interface.

In this section, the term "client" refers to the entity that invokes on the interface.

5.1 Receive

To receive data, the client invokes the `receive` method:

```
struct sbuf {
  void*  buffer;
  size_t size;
};
```

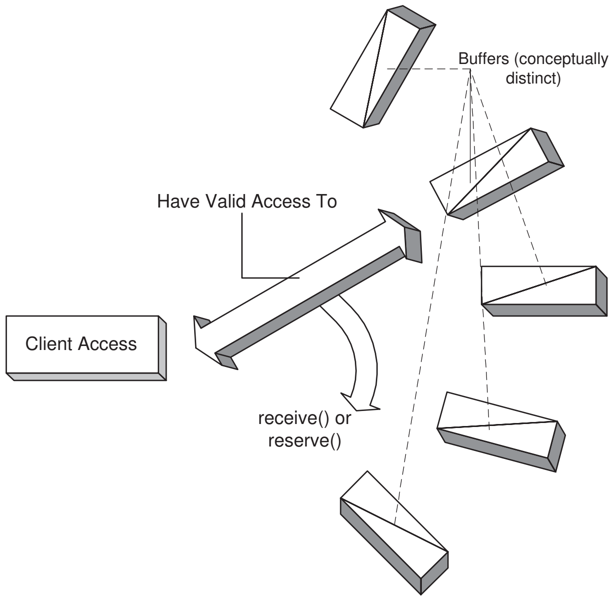

Figure 4: Buffer Management Scheme

```
sbuf
receive(size_t size,
      CORBA::Boolean exactly,
      int alignment,
      CORBA::Boolean startMTU);
```

Apart from the buffer management scheme, the semantics of the method is quite straightforward. The client specifies how many bytes it expects to receive, whether the strand should return when it has less in its buffer and whether the returned buffer should be 8, 16, 32 or 64 bits aligned. The method returns a structure which contains a pointer to the data area and the number of bytes in that area.

The buffer management scheme is special. The buffers are allocated and deallocated inside the Strand. Furthermore, the buffer returned by `receive` is only guaranteed to contain valid data until the next `receive` call. In other words, the next `receive` call automatically invalidates the data pointer returned by this call. Figure 4 provides an illustration of how buffers are managed. There are two reasons for adopting this scheme.

Firstly, because buffer allocation is hidden behind the interface, a Strand implementation could be optimised for a network interface that provides memory-mapped network buffers and zero-copy receive semantics. The clients could unmarshal data directly from the network buffers returned by the Strand.

Secondly, the interface inherently puts a limit on the buffer space that the Strand has to commit to the clients. At any time, there is only one buffer the client can access. Any buffers that have previously been given out to the client are released implicitly and can be recycled immediately. Furthermore, the size of each buffer is limited to a maximum fixed by the Strand. To unmarshal large call arguments, the client will have to call `receive` repeatedly to fetch all the data. There is no need for the Strand to allocate extra buffers to handle the call. Instead the unmarshalling of data by the client can be pipelined with the fetching of more data from the network by the strand.

We believe the buffer management scheme has a direct impact on performance. It is possible to take advantage of zero-copy network buffers to improve data marshalling efficiency. Because there is no need to cater for special cases, such as calls with very large data arguments, the program logic is simple and can be implemented efficiently. It also helps to avoid the contention for buffer resources among different strands. For instance, in all the strand implementations we have done so far, we are able to avoid dynamic buffer allocation completely. Each connection is allocated one or two fixed size buffer statically[5]. From the point of view of minimising the interference among different thread of executions, the absence of dynamic buffer allocation helps by eliminating an obvious point of resource contention.

5.2 Send

To send data, the client first requests for a buffer with the `reserve` method:

```
sbuf
reserve(size_t size,
      CORBA::Boolean exactly,
      int alignment,
      CORBA::Boolean transmit,
      CORBA::Boolean endMTU);
```

[5] To avoid unnecessary fragmentation in the operating system, the size of each buffer is set to accommodate the maximum PDU size of the network interface.

Roughly speaking, `reserve` is the converse of `receive`. The client specifies how much space is needed, whether the strand should return when it has less than required and whether the returned buffer should be 8, 16, 32, or 64 bits aligned. The method returns a buffer for the client to write data into.

The buffers are managed in the same way as `receive`. The client can write to the buffer returned by `reserve` until the next `reserve` call. In other words, the next `reserve` call automatically invalidates the data pointer returned by this call (figure 4). Moreover, the strand is free to transmit the implicitly relinquished buffers any time hereafter. The client can also cause the transmission to be done synchronously by setting the `<transmit>` flag to true.

The discussion on buffer management in the previous section is also valid in this case. However, the performance gain by avoiding excessive buffer allocation is partly offseted by the need to walk through the marshalling arguments twice! This is necessary because GIOP version 1.0 does not permit data fragmentation. In other words, a request or a reply must be sent as a single GIOP message. To transmit a call with large arguments, it is possible that the first part is transmitted by the strand while the client is still marshalling the rest of the arguments. Unfortunately, the first part of a GIOP message– the header – contains the size of the message body. It is therefore necessary to walk through all the arguments to calculate the total size and put the value into the message header. After this pass, the arguments have to be accessed again to perform the actual data marshalling. The performance penalty due to the size calculation varies with the type of arguments. It can be substantial in cases such as a long sequence of strings. Having said that, this performance penalty will soon be history with the newly revised GIOP specification. It now allows a request or a reply to be sent in multiple and consecutive GIOP messages. With the new specification, there is no need to process the arguments in two passes as the total argument size does not have to be pre-calculated.

5.3 Bulk Data Send/Receive

Two methods are provided for bulk data transfer.

```
void
receive_and_copy(sbuf b,
      CORBA::Boolean startMTU);

void
reserve_and_copy(sbuf b,
       CORBA::Boolean transmit,
       CORBA::Boolean endMTU);
```

`receive_and_copy` is equivalent to a `receive` and a memory copy to the buffer provided by the client. Similarly, `reserve_and_copy` is equivalent to a `reserve` and a memory copy from the buffer provided by the client. A naive implementation may have to copy the client data to/from its internal buffer. Depending on the network interface, this copy action may be optimised away. For instance, with a BSD-style TCP socket interface, the `reserve_and_copy` can be collapsed into a single `send` system call directly from the client's buffer; the `receive_and_copy` can be collapsed into a single `recv` system call. Using this optimisation, the ORB is able to handle bulk data, such as sequence of octets, with very little overhead (zero copy within the ORB).

5.4 Exactly-Once and At-Most-Once Semantics

The strand abstraction is not intended to be a general purpose bidirectional data connection. Instead, the goal is to specialise the interface and encapsulate sufficient functionalities to support request-reply and oneway interactions efficiently.

When implemented on top of a reliable stream transport (such as TCP/IP), both request-reply and oneway calls are exactly-once by default. On the otherhand, invocations implemented on top of an unreliable transport (such as ATM/AAL 5) are by nature at-most-once because data packets could be dropped in transit. Exactly-once semantics can be obtained if the sender is able to retransmit lost-packets and the receiver is able to remove duplicates. We believe that while exactly-once semantics is desirable for request-reply interactions, at-most-once semantics is more suitable for oneway calls. This is particularly true when oneway calls are used to transmit isochronous data. With this data type, retransmission should be avoided because undesirable jitters will be introduced otherwise.

For this reason, the strand allows the client to specify whether a call should be invoked as exactly-once or at-most-once. Using this information[6], the strand decides whether lost-packets are retransmitted internally.

6 TCP transport

The implementation of the tcp transport is quite straightforward. The noteworthy features are:

- Each connection is allocated a fixed size buffer (8Kbytes). Once the buffer is filled up, the data are transmitted.
- The transfer of sequence and array of primitive types are optimised. The stub code marshals these data types with `receive_and_copy` and `reserve_and_copy`. If the data size is bigger that the buffer space available, the data in the buffer are flushed to the network and the sequence/array is then sent directly from the application buffer.

[6] The method: `reserve_and_startMTU` is provided. This method is the same as `reserve` except that the former takes an extra boolean argument to indicate if the call semantics should be exactly-once. Also, the `startMTU` and `endMTU` boolean arguments in the `receive` and `reserve` methods are used to indicate the beginning and the end of an invocation. This is necessary because exactly-once calls may be interleaved with at-most-once calls, the strand must be able to distinguish the packets that can be dropped, i.e. those from at-most-once calls, from those that have to be retransmitted. Of course, none of these arguments have any effect when the strand is implemented on top of a reliable stream transport.

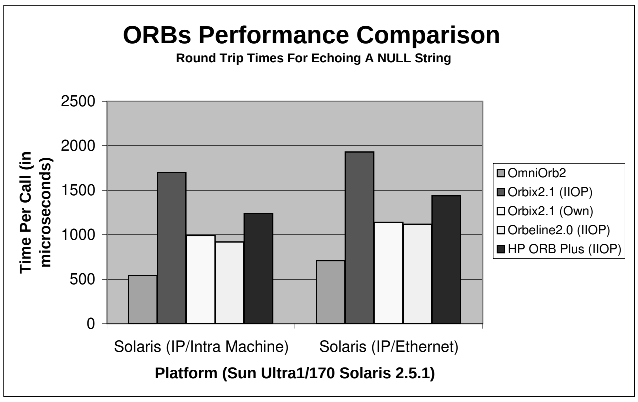

Figure 5:

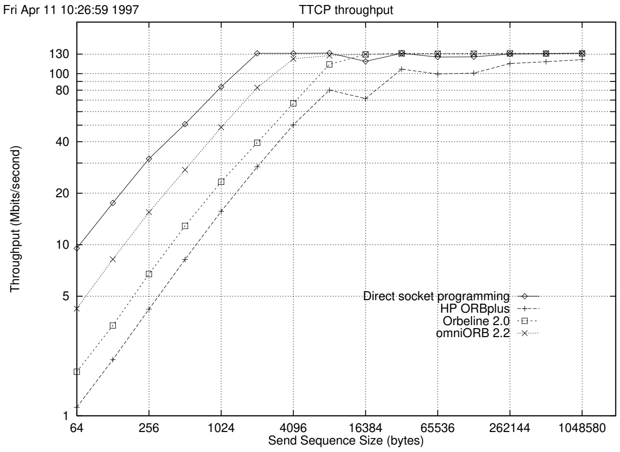

Figure 6: ORBs Bulk Data Throughput Comparison

6.1 Performance

6.1.1 Null Echo

The time to echo a zero length (null) string is measured . This is a measure of the overhead incurred by the ORB and the operating system. The following IDL interface is used:

```
interface echo {
   string echoString(string mesg);
};
```

The measurements taken from a number of platforms are shown in table 1. The data are the average of 5000 invocations.

To put the measurements into context, the same test is used to measure the performance of several commercially available ORBs. All the tests were conducted on a pair of Sun Ultra 1 running Solaris 2.5.1. The results are shown in figure 5. OmniORB2 takes 540μsec intra-machine and 710μsec inter-machine. Compare to omniORB2, other ORBs take 70% to 214% longer for intra-machine and 58% to 172% longer for inter-machine calls.

6.2 Bulk Data Transfer

The performance in bulk data transfer is measured by sending multiple sequence of octets using oneway operations. The following IDL interface is used:

```
interface ttcp {
   typedef sequence<char> Buffer;
   oneway void receive(in Buffer data);
};
```

The test is conducted using two Ultra 1 that are connected via 155Mbits ATM. The machines are connected via three ATM switches in this order: ATML Virata VM1000$\longleftrightarrow$Fore ASX-200WG$\longleftrightarrow$ATML Virata VM1000. Classical IP over ATM is used to provide TCP/IP connectivity. The MTU size of the ATM interfaces is 9180 bytes. The default socket send and receive buffer size is used. The socket option `TCP_NODELAY` is not set. The time to send 100Mbytes of data in multiple sequences of octets was measured. Different sequence sizes were used. Figure 6 plots the throughput against the send data size in each test run. The throughput from using direct socket programming is also shown. This curve represents the performance envelope obtainable from the test platform.

At small send data size, the throughput is CPU-bound. As the send data size increases, the throughput changes from CPU-bound to link-bandwidth-bound. The yield point where the throughput changes from CPU-bound to link-bandwidth-bound is directly proportional to the ORB's overhead, i.e. the higher the yield point, the higher the ORB's overhead.

Platform	Transport	Time per call(μsec)
Linux Pentium Pro 200MHz (gcc-2.7.2 no compiler optimisation)	TCP/intra-machine	340
	TCP/ethernet(ISA card)	1000
	TCP/ATM	440
Windows NT 4.0 Pentinum Pro (MS Visual C++ -O2)	TCP/intra-machine	360
	TCP/ethernet(ISA card)	1000
Digital Unix 3.2 DEC 3000/600 (DEC C++ -O2)	TCP/intra-machine	750
	TCP/ethernet	1050
Windows 96 Pentium 166MHz (MS Visual C++ -O2)	TCP/intra-machine	1000
	TCP/ethernet(PCI card)	1250
Solaris 2.5.1 Ultra 1 167MHz (Sunpro C++ -fast)	TCP/intra-machine	540
	TCP/ethernet	710

Table 1: The Null Echo Round Trip Time of omniORB2 on Various Platforms

With 64 bytes data size, the throughput of omniORB2 is 39% of the direct socket throughput. The figure rises to 63% at 2048 bytes where the throughput is still CPU-bound. This is an indication that the ORB overhead does not increase linearly with the size of the call arguments. In other words, the ORB's overhead has quickly become insignificant at moderate data size, the operating system overhead is the dominant factor.

The throughput of other ORBs are also shown in figure 6[7]. All the ORBs remain CPU-bound well beyond the point when omniORB2 is already able to saturate the link. Before the yield point, the CPU-bound throughput of other ORBs are 52% to 67% less than that of omniORB2.

7 Shared Memory Transport

A shared memory transport was designed as a complementary transport, useful where two address spaces may communicate through a segment of shared memory. While this usually corresponds to the client and server being located on the same machine, we have also used the transport over a high performance interconnect which presents an abstraction of shared memory for inter-machine communication.

[7]The test cannot be performed using Orbix 2.1MT because of a memory leakage problem in its runtime.

A server instance of the transport creates a large shared memory segment which is carved up into a number of communication channels, each channel with a semaphore set to allow for synchronisation. One channel is reserved for new clients to rendezvous with the server. During the rendezvous, the server allocates a channel for the client and passes a key to the allocated channel back to the client through the rendezvous channel. All subsequent communication between the client and server takes place through the allocated channel.

7.1 System V shared memory

The shared memory transport is implemented using the standard System V primitives for shared memory and semaphores. These are now widely accepted and are supported on all our Unix platforms. The round trip times of *null echo* using the shared memory transport are shown in table 2.

The results of the bulk data transfer test using the shared memory transport is shown in figure 7. The results are obtained on the Linux Pentium Pro platform.

These results show that the shared memory transport yields a performance improvement in the region of 20% over all platforms for intra-machine communication.

Platform	Time per call (μsec)	
	Shared Memory	Local TCP/IP loopback
175 Mhz Digital Unix Alpha Uniprocessor	510	750
167 Mhz Solaris UltraSparc Uniprocessor	510	540
168 Mhz Solaris UltraSparc Multiprocessor	370	480
200 Mhz Linux Pentium Pro Uniprocessor	270	340

Table 2: Shared Memory Null Echo Round Trip Time

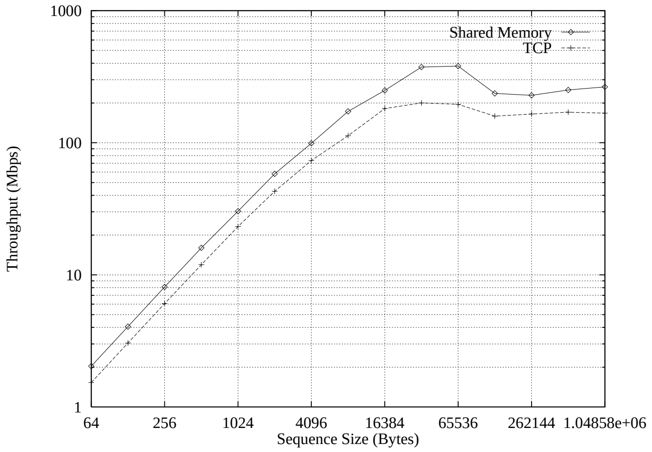

Figure 7: Intra-machine oneway throughput comparison

7.2 SCI

The shared memory transport was ported over the SCI interconnect from Dolphin Interconnect Solutions. SCI presents an abstraction of shared memory for inter-machine communication. The interconnect will write a single **dword** to a remote memory location in 2.5 μsec and has a raw-write performance on our Pentium-Pro 200Mhz platforms of 220 Mbps.

The *null echo* test was performed on two Linux Pentium Pro machines connected via SCI. The round trip times are shown in table 3. Also included is the inter-machine time for Linux TCP over a 155 Mbps ATM network.

These results illustrate a significant improvement in latency when using the SCI transport. It has been estimated that for the Linux platform, 110us is spent in the ORB with protocol and marshalling overhead, the remainder being spent in performing transmission and synchronisation through the shared buffer. It is interesting to note that the inter-machine time using SCI is significantly faster than intra-machine using standard shared memory primitives. This is due to the parallelisation of the GIOP and the avoidance of a context switch between the client and server.

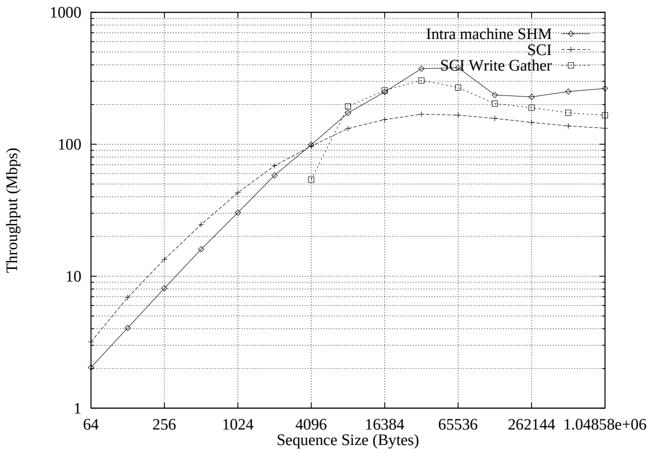

Figure 8: SCI v Shared Memory Bulk Data Throughput Comparison

The results of the bulk data transfer test using the System V shared memory and the SCI transport are shown in figure 8. The System V shared memory trace *SHM* shows the throughput dropping as the transfer size approaches 256 Kbytes. This is probably due to the effect of cache trashing as the Pentium Pro secondary cache is only 256 Kbytes in size. The two SCI traces– *SCI* and *SCI write gather*– show that the throughput peaks at around 32 Kbytes, which is the size of the shared memory buffer. This would have been im-

Platform	Transport	Latency(us)
Intra 200 Mhz Linux Pentium Pro	SYS V Shared Memory	270
Inter 200 Mhz Linux Pentium Pro	SCI Shared Memory	156
Inter 200 Mhz Linux Pentium Pro	TCP/ATM	440

Table 3: SCI Null Echo Round Trip Time Comparison

proved had circular buffering been used. Using SCI with write-gathering (*SCI write gather*)[8], the throughput is higher that *SHM* probably because of the avoidance of context switching. However, we were unable to use the write-gathering option for block sizes of less than a page without throughput being significantly reduced. Without using write-gathering, the throughput is significantly less. We believe the throughput is currently limited by the PCI implementation of our test machine.

8 ATM AAL5 Transport

The ATM AAL5 transport is intended to run directly on top of the native ATM API of the host platform. For the moment, the transport has been implemented over the Linux ATM API. It should be straight forward to port the implementation to another platform API. Unlike the transports described so far, the underlying transport– AAL5– is unreliable although the packets are guaranteed to arrive in-order.

As discussed in section 5.4, we would like to provide exactly-once semantics for request-reply and at-most-once semantics for oneway calls. As the size of each packet is limited to 9180 bytes[9], a GIOP request or reply message that exceeds this limit would have to be transmitted as multiple packets. To support exactly-once semantics, the transport must be able to retransmit any packets that are lost in transit. The retransmission is only done for the request-reply calls and not for the oneway calls. In general, request-reply calls may be interleaved with oneway calls, the transport must be able to adapt and alter the retransmission policy on the fly.

To satisfy these requirements, a very lightweight protocol– omniTransport– is designed to layer on top of the ATM API. A detail discussion of omniTransport and its implementation can be found in [5]. Unlike other transport protocols designed to run in the user address space, omniTransport is not a general purpose design but is targeted to support asymmetric interactions where one end always plays the role of the client and the other the server. This greatly simplify the design because the number of possible protocol states is much smaller than a more general purpose protocol.

Although omniTransport is first implemented on top of an ATM API, the design is well suited for use over any network which offers in-order packet delivery semantics. By inference, one should be able to port omniORB2 to run on top of these networks.

8.1 Performance

The performance of omniTransport is discussed in detail in [5]. For completeness, the round trip times of null Echo are shown in table 4.

The measurements were taken from two Pentium Pro 200Mhz Linux machines interconnected using the Efficient Networks 155Mbps adaptor and an ATML Virata VM1000 switch.

The timing of *raw/ATM* is the round trip time for running the test without the exactly-once guarantee. This represents the performance upper bound achievable on this platform. The timing of *TCP/ATM* is the round trip time using Classic IP (CIP) over ATM. The timing of *omniTrans-*

[8] Where successive discontiguous writes are amalgamated by the host PCI bridge into a contiguous block of given size before bursting the writes to the SCI card.

[9] This is the MTU size of the Linux ATM network interface.

Transport	Time per call (μsec)
TCP/ATM	440
omniTransport/ATM	380
raw ATM	360

Table 4: omniTransport/ATM Null Echo Round Trip Time Comparison

port/ATM is the round trip time with exactly-once semantics. The protocol adds an extra 20 μsec overhead which is well below the overhead of the kernel-based CIP.

9 Conclusion

Our experience with omniORB2 has confirmed that it is possible to design a high performance and compliant Object Request Broker. Good performance is achievable by tailoring the design to the protocol and other characteristics of CORBA. At ORL, omniORB2 has been in use for over a year. It is the software backbone for controlling the Virtual Network Computer (VNC) via the Active Badge and for building the distributed infrastructure of a fine-grain 3D location system– Active Bat. It is also an integral component of Ouija– a CORBA-database integration toolkit that provides seamless integration of an object-oriented frontend, developed in CORBA, with the Oracle 7 relational database server. Given the experience we have with its deployment, we are confident in the robustness and scalability of omniORB2.

We are interested in enhancing omniORB2 to include some notion of "quality of service". This would allow applications with specific performance requirements to influence the way the ORB performs some actions that are necessarily transparent to the applications. For instance, the applications may specify the performance requirements for streaming real-time audio and the ORB should select the suitable network transport and binding model with the appropriate performance guarantee.

We have demonstrated that omniORB2 is adaptable to a variety of transport mechanisms. There is currently an interest at ORL in high bandwidth and low latency interconnect technologies. OmniORB2 has been ported to run on top of one of these technologies– SCI. We believe a software backbone based on CORBA has a lot of potential in passing on the performance improvement provided by the underlying interconnect technology to many distributed applications. We'll continue to explore new improvements to omniORB2 for running on top of fast interconnects.

References

[1] Flick idl compiler, 1998. http://www.cs.utah.edu/projects/flux/.

[2] The piconet project, 1998. http://www.orl.co.uk/piconet/.

[3] The radio atm project, 1998. http://www.orl.co.uk/radio/.

[4] OMG. Common Object Request Broker Architecture and Specification, July 1996. Revision 2.0, Available electronically via `http://www.omg.org`.

[5] S Pope and S. L. Lo. The Implementation of a Native ATM Transport for a High Performance ORB. In *Submitted to Middleware 98*, 1998.

[6] R. Want and A. Hopper. Active badges and personal interactive computing objects. *IEEE Transactions on Consumer Electronics*, February 1992.

[7] S. Wray, T. Glauert, and A. Hopper. The medusa applications environment. In *International Conference on Multimedia Computing and Systems*, May 1994.