GI 2006

The Implementation of a High-Performance User-Level Transport Protocol for omniORB2

Steve Pope, Sai-Lai Lo
Olivetti & Oracle Research Laboratory
24a Trumpington Street
Cambridge CB2 1QA
England

March 16, 1998

Abstract

The paper describes the design and implementation of omniTransport, a lightweight, user level transport protocol which has been tailored for the asymmetric communication requirements of a CORBA 2.0 compliant ORB, omniORB2. The protocol is shown to interwork with omniORB2 and has outperformed a leading in kernel TCP implementation.

1 Introduction

OmniORB2 is an implementation of the 2.0 specification of the Common Object Request Broker Architecture (CORBA). It forms the backbone of the distributed object infrastructure at ORL and is motivated by our need for a multi-threaded ORB to run over our in-house developed hardware and software platforms. OmniORB2 is the fastest CORBA 2.0 ORB currently available [5] and is publicly available under the GNU Public Licenses.

This paper will describe the implementation and performance of a lightweight, user level transport protocol for omniORB2 called **omniTransport**. The design of omniTransport is motivated by the need to support omniORB2 on platforms and networks which require a high performance, reliable transport protocol, but which do not support TCP or for which a TCP port would be inappropriate. For instance, the Medusa distributed multi-media system [8] was developed over what was a new ATM network infra-structure and was forced to use 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 omniTransport protocol has been designed for use over an ATM network, but would be suitable for any network which offers in-order delivery semantics. Unlike other user level transport protocols, omniTransport is not a general purpose design, but is instead targeted to support asymmetric interactions where one end always plays the role of the client and the other the server. This asymmetry means that it is possible for omniTransport to be *single* *threaded* at both the client and the server side. This should be contrasted with other general purpose user level transport protocols, such as [9, 3, 7] where it is necessary for a thread to always be available for the processing of incoming data and to generate acknowledgements for received data. Inter-thread communication is a major performance bottleneck in a user-level transport protocol. By its elimination, omniTransport is able to outperform a leading TCP implementation within the bounds of our interaction style.

The performance of omniTransport does not require a complicated implementation and it should be noted that this paper will describe in full its algorithms and state transitions. The protocol, while simple is able to perform well, fulfills the architectural considerations embodied in [2] and support the requirements of omniORB2 over the whole range of call types, sizes and supports exactly-once and at-most-once delivery on a per call basis.

2 OmniORB2 Overview

The primary function of an ORB is to provide the means for object invocation between two address spaces. Where the two address spaces are separated by a network, the ORB must manage the transfer of data over the network. OmniORB2 uses the thread-per-connection model in preference to other threading models in order to maximise the possible degree of concurrency while keeping thread overheads to a minimum and to avoid the interference by the activities of other threads on the progress of a remote invocation. Each connection communicates with the remote address space through the **strand** abstraction of a bidirectional data channel. The strand allows a client to specify whether a call should be invoked with exactly-once or at-most-once semantics and is able to handle bulk data, such as a sequence of octets, with very little overhead (zero copy). We believe that exactly-once semantics are desirable for request-reply interactions and at-most-once semantics are more suitable for one-way calls. This is particularly true when one-way calls are used to transmit isochronous data. With this data type, retransmission should be avoided because undesirable jitters would otherwise be introduced.

The scope of this paper is restricted to describing the omniTransport protocol used by omniORB2 in the reliable transfer of data over an unreliable network. A more complete description of omniORB2 is presented in [4, 6].

2.1 The OmniTransport Interface

OmniTransport has been designed to work in a general request-response style environment and so can be thought of as logically separate from the strand abstraction as shown in Figure 1. The basic interface to omniTransport is through three functions, **send, receive** and **reclaim**. The omniORB2 strand passes the **send** function a number of buffers which are to be reliably delivered to the other side; calls the **receive** function to accept buffers from the other side and calls the **reclaim** function to reclaim any acknowledged buffers which are held by the transport. Additionally, the **send** function may be passed a parameter

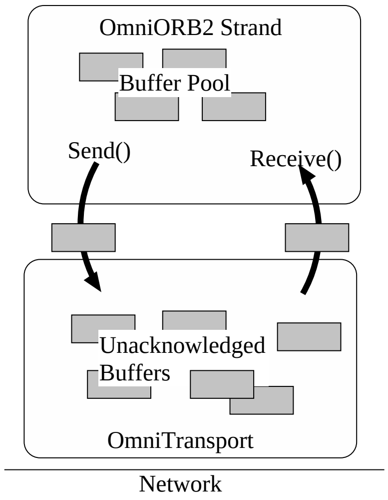

Figure 1: omniTransport and omniORB2

which indicates that a request is either **complete** or **partial**. The **receive** function may be passed a parameter which indicates that the omniORB2 strand is expecting a **new** response.

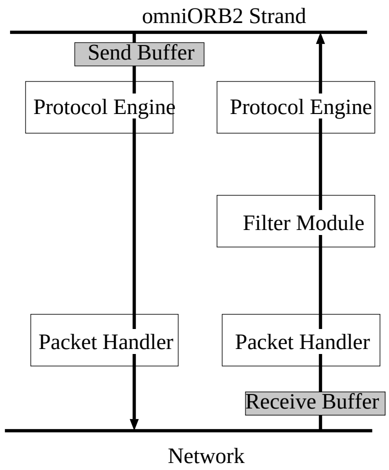

Figure 2: Structure of the omniTransport Implementation

This interface is sufficient to implement a reliable request-response protocol for omniORB2 and is used in this paper for clarity. However, it should be noted that for performance reasons, the interface between omniTransport and the strand is collapsed in the implementation.

3 Structure of omniTransport

The structure of the omniTransport implementation is as shown in Figure 2 and comprises of:

1. A **protocol engine**, which manages the transport's state machines and timers and is responsible for high level protocol decisions, such as acknowledgement (**ack**) or negative-acknowledgement (**nack**) generation.

2. A **filter module**, which is used during packet reception to examine incoming packet headers, dealing with situations such as repeated or dropped packets. The filter module adds contextual information to a received packet before input to the protocol engine. For example, a *received packet* might be passed to the protocol engine as an *out of sequence received packet*.

3. A **packet handler**, which is responsible for the fragmentation, transmission and reception (with timeout) of packets and the insertion of omniTransport headers.

Packet Type	Scope	Description
EOF	External	*End of file*
BAD	External	*Hard communication problem*
ABORT	External	*Abort stub processing of call*
PARTIAL	External on-wire	*Partial request made*
COMPLETE	External on-wire	*Complete request made*
CALL ACK	Internal on-wire	*Acknowledge buffers*
NACK	Internal on-wire	*Request retransmission of buffers*
RECLAIM	Internal on-wire	*Request acknowledgement for received buffers*
RECLAIM ACK	Internal on-wire	*Acknowledge received buffers on a reclaim*
TMO	Internal	*Timeout has occurred*
SEQ	Internal	*Call or fragment was out of sequence*

Table 1: omniTransport Message Types

The type of messages sent between each level of the implementation is restricted as indicated by the **scope** field in Table 1. Between the omniORB2 strand and the protocol module, messages have **external** scope, between the protocol engine and the filter module, messages have **internal** scope and to or from the packet handler with **on-wire** scope.

All packets generated by omniTransport contain a header as shown in the Table 2. The **message type** may be one of **on wire** scope. If the **exactly-once semantics** field is set, then the transport should attempt to ensure reliable delivery for the call. The *message size* indicates the size of the packet in bytes, including the header. The **call sequence** and **packet sequence** numbers are used to maintain the correct sequence of buffers for and within each call using the standard *sequence space* operations. Finally, a header can be marked as having been **fragmented** by omniTransport.

Number of Bytes	Type	Name
1	**CARD8**	*message type*
1	**BOOL**	*exactly-once semantics*
2	**CARD16**	*message size*
1	**CARD8**	*call sequence number*
1	**CARD8**	*packet sequence number*
1	**BOOL**	*fragmented*
1	**CARD8**	*padding*

Table 2: omniTransport Header Format

OmniTransport does not support the re-ordering of out of order packets. This assumption is in line with our target ATM networks. However, out of order packets could be re-ordered within omniTransport without changes to the protocol on the wire. OmniTransport also assumes the presentation of checksummed packets and that the underlying network discards packets with the wrong checksum. An extension to provide checksumming by omniTransport would require extra space to be reserved in the header for a checksum field.

The next section will describe how the protocol supports the reliable transfer of data using the above network primitives.

4 OmniTransport Protocol and Algorithms

4.1 Normal Protocol Operation

The state machine of the **protocol engine** starts in the **idle** state, with the transport awaiting either an exactly-once or at-most-once call to be initiated by a client omniORB2 strand. This section will describe the state transitions for the client and server during both an exactly-once (or request-reply) and an at-most-once (one-way) call. It should be noted that a client may interleave at-most-once with exactly-once calls.

The exactly-once client side state transitions are reflected in the state machine shown in Figure 3. The following describes the actions and state transitions used in making an exactly-once call.

- If the client has a request which fits into a single buffer, it will call **send** (i) passing in the buffer with the **complete** flag set. OmniTransport will ensure that the buffer is sent on the network and will retain the buffer in case its retransmission is required. The state machine will move into the **waiting for reply** state, with the transport awaiting a reply from the server for the request.

- If a number of buffers are required to satisfy the client's request, then the client must call **send** (ii) with the **partial** flag set. This will move the state machine into the **request in progress** state, from which any number of

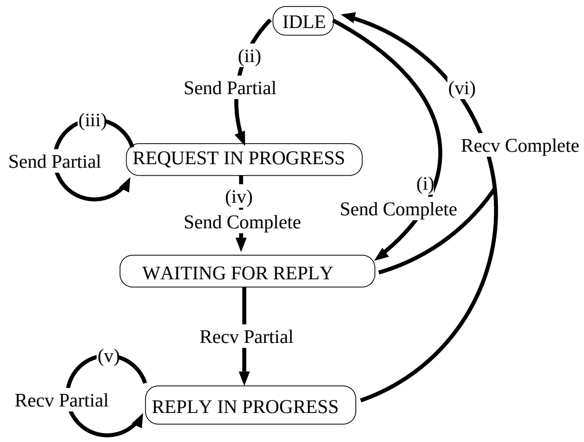

Figure 3: Client State Machine (Exactly-Once Semantics)

calls (iii) to **send** with the **partial** flag set can be made by the client to transmit further buffers corresponding to the request. On the final call to **send** for the request, the client must set the **complete** flag (iv), which will move the transport into the **waiting for reply state**.

- The client will then call **receive** to await the response from the server, passing in a buffer which will be filled with data from the server. Again, a number of calls to **receive** (v) might be required to consume all the data transmitted by the server. Finally, a packet will be received which was sent with the **complete** flag set. This information is used by the transport to make the state transition to the **idle** state (vi) and await a new request.

The protocol makes use of the property that the $N+1$ call is an implicit acknowledgement for the N call and so explicit acknowledgements are not generated on the wire for an on-going interaction between a client and a server. However, if the client goes idle for a significant amount of time, then the protocol requires that an **ack** be generated by the client for the last response from the server. Hence, on each call, the client side is required to set and clear an acknowledgement timer, which when fired will generate the required ack. It is only in this situation that omniTransport is not single threaded, since the client thread is processing elsewhere. Care has been taken in the implementation to ensure that in most cases no synchronisation between threads is required to either set or clear a timer.

Further optimisations to the acknowledgement timer have also been made in order to eliminate time from the bottleneck path of a call. For example, the acknowledgement timer is set by the client before the results which it is to acknowledge have been received even though this causes some awkwardness in the case when results are late. These optimisations mean that there is little overhead in an omniTransport call over and above transmitting the raw data.

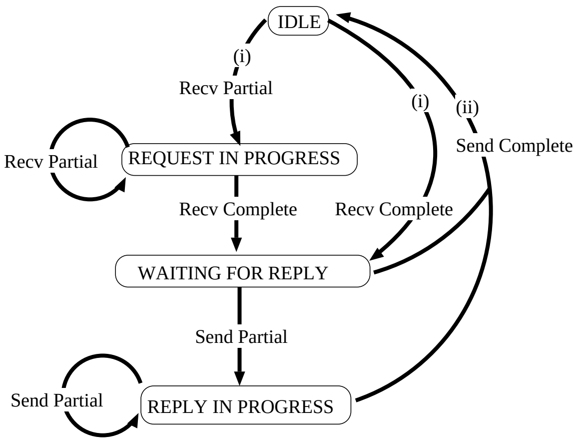

Figure 4: Server State Machine (Exactly-Once Semantics)

The exactly-once server side state transitions are shown in Figure 4. The server omniORB2 strand first calls **receive** (i) to await a request from the client. Buffers corresponding to a request are returned and the strand will up-call, making the request to an application level object. A response is then made and the omniORB2 strand will call **send**, indicating to omniTransport when the response is **complete** (ii).

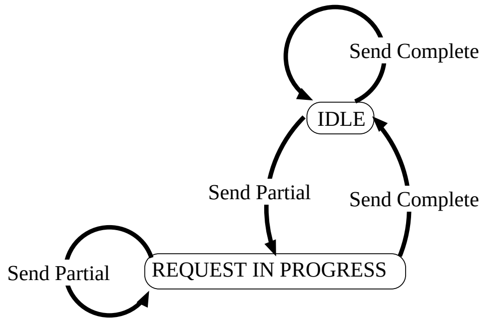

Figure 5: Client State Machine (One-Way Semantics)

The one-way client side state transitions as shown in Figure 5 are a subset of the transitions for an exactly-once client side call. After any **complete send**, the protocol moves back to the **idle** state and awaits a new call.

The one-way server side state transitions as shown in Figure 6 are the same as the client side except that the inputs to the machine are driven by calls to **receive**. Once a one-way request has been completely received, the transport moves back into the **idle** state to await another request.

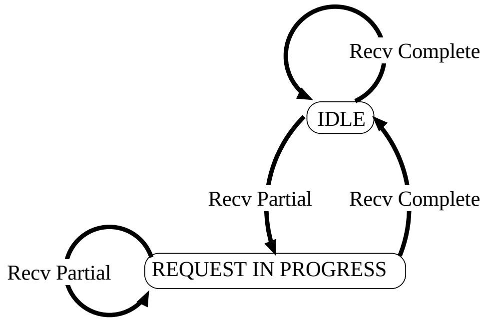

Figure 6: Server State Machine (One-Way Semantics)

4.2 Error Conditions

It is in the cases that errors have occurred that the protocol engine is required to perform non-trivial actions. The algorithms and state transitions used in an error recovery situation are shown for the client side in Figure 7 and for the server side in Figure 8.

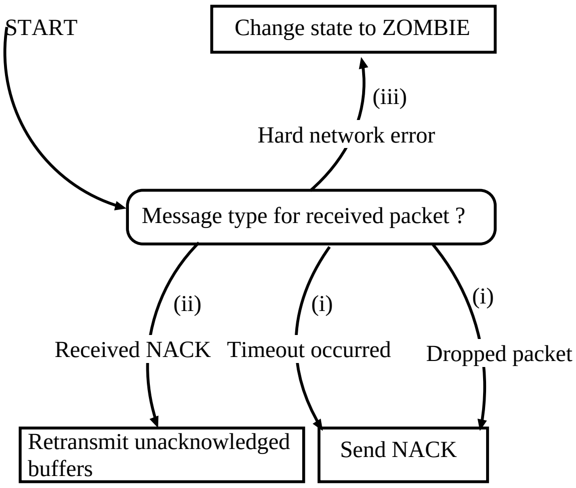

Figure 7: Client Error Handling Protocol Engine

4.2.1 Client Side Errors

- If the client is expecting a response, but instead receives a timeout or an out of sequence packet, then the client sends a **nack** to the server (i). The sequence numbers of all packets indicates the call to which the packet refers, thus the call sequence number of a nack indicates the call which is being nacked. The packet sequence number of a nack is used to calculate the buffers which require retransmission, namely all buffers for the call with a packet sequence number $\geq$ the packet sequence number of the received nack.

- Similarly, on receipt of a nack from the server (ii), the client will retransmit appropriate buffers which have been held for retransmission.
- If a hard network error is received (iii), omniTransport informs the stubs and changes state to **zombie**. Once in the zombie state, no further operations are allowed for the connection.

This simple retransmission scheme is based on the low error rates on our target networks and simplifies our implementation over other schemes such as requesting the retransmission of lists of sequence pairs.

4.2.2 Server Side Errors

Handling errors on the server side is similar to the client side, with some complications. Firstly, the server must handle the differences between exactly-once and one-way calls. This is not an issue for the client since it never will receive a response from a one-way call. Secondly, because of the asymmetry in the protocol, whereby all requests are initiated by the client, only the client can *advance* the call sequence number. This means that the client should never receive packets corresponding to the next call while processing the current call, while the server must handle this condition.

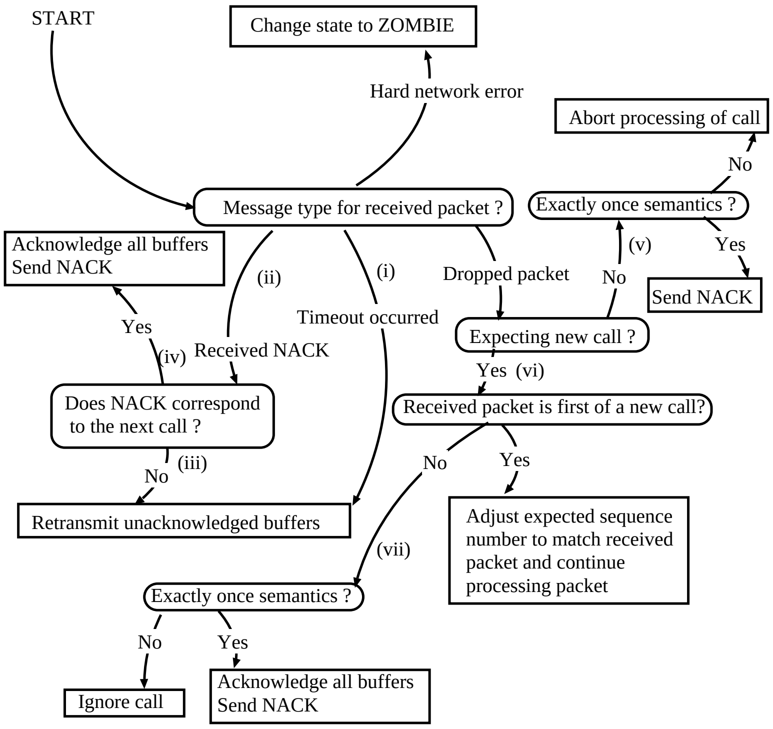

Figure 8: Server Error Handling Protocol Engine

- In the case that a **timeout** has occurred (i), the server should simply retransmit any buffers which are unacknowledged. When the timeout corresponds to an idle client, all buffers will already have been acknowledged, so there is no action.

- If the server has received a **nack** (ii), then the client must have experienced a dropped packet or timeout while awaiting a response from the server. If the call sequence number corresponds to a call which the server has just handled, then the client has not correctly received the response, hence all buffers should be retransmitted (iii). Conversely, if the call sequence number of the nack corresponds to a call which the server has not yet started to handle, then the server must not have received the new call and the client is awaiting a response. In this case (iv), the received nack implies that the response previously sent by the server has been correctly received by the client and the client has sent a new call which has not been received by the server. The server should acknowledge and free all the buffers held from the previous call and nack the client to request a retransmission of the lost request.

- If the server has noticed a dropped packet, then the correct response depends on whether processing has been already started by the server on the call. If so (v), then for an exactly-once call, the server should nack the client for a retransmission. For a one-way call, there is no possibility of a retransmission from the client and some buffers will have already been passed up to the omniORB2 strand for unmarshalling. Hence, the server must inform the strand to abort processing of the call. If processing had not started on a call (vi), then if the received packet is the first packet of a new call, the client and server are simply out of step with their call sequence numbers. Since the out of step sequence number could have been caused by the loss of a complete one-way call from the client, the server should proceed with the call. Finally, in case (vii), the server has noticed a dropped packet from the start of a call. If the call is one-way, then the server must ignore any further packets corresponding to the call, since it cannot request a retransmission from the client. There is no requirement to inform the omniORB2 strand since it has not started to process the call. However, if the request has specified exactly-once semantics, then for the same reasons as in case (iv), the server should acknowledge its previous response and send a nack to the client to request a retransmission.

4.3 Buffer Reclamation

During operation of the protocol, buffers circulate between a buffer pool, held by the omniORB2 strand and the transport; with the transport holding unacknowledged buffers for retransmission. In the case that a sender is required to send more data than there are buffers available in the buffer pool, the **reclaim** function of the transport must be used. A call to the reclaim function sends the **RECLAIM** protocol message which requests the other side immediately acknowledge all unacknowledged buffers with a **RECLAIM ACK** message[1]. When the acknowledgement has been received, these buffers can be passed back to the buffer pool for re-use by the sender. The reclaim function can also be thought of as a very crude, high level flow control mechanism as no more buffers can be in flight than have been made available to a connection.

4.4 Buffer Fragmentation

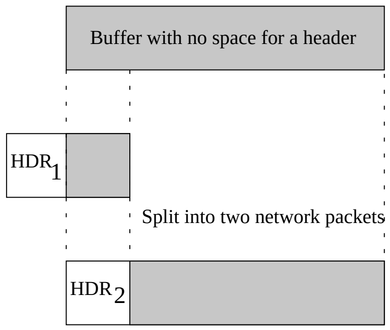

Figure 9: Header Insertion into a Zero Copied Buffer

The fragmentation feature of the packet handler is used by omniORB2 in cases where the ORB is able to perform a zero copy operation from the application to omniTransport. In this case, omniTransport will not be passed one of the pre-defined buffers which contains space for the insertion of a header. To avoid introducing a copy of the complete buffer, omniTransport copies out a small amount from the buffer into a second buffer as shown in Figure 9. The second buffer is sent as one network packet, leaving room at the head of the original buffer for the inclusion of an omniTransport header. The original buffer is then sent as second network packet.

5 Performance

The omniTransport protocol has been implemented over the Linux ATM API [1]. Our experimental configuration consists of two Pentium Pro 200Mhz processors interconnected using the Efficient Networks 155Mbps adaptor and an ATML Virata VM1000 switch. Both processors were running the Linux kernel version 2.0.25 and Linux ATM version 0.23. The system was also configured with the single copy TCP and large IP window options set.

5.1 Null Echo

The time to echo a zero length (null) string is measured. The following IDL interface is used:

[1] A separate message was used rather than overloading the ACK message to simplify normal ACK processing.

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

The average round trip time for 10000 invocations of a Null Echo server was measured. Table 3 shows the results for omniORB2 over raw ATM (which indicates the maximum performance of a transport on our platform), TCP over ATM and omniTransport over ATM. This result shows a significant improvement in latency over the Linux kernel TCP.

Transport	Time per call (us)
omniOrb2/TCP/ATM	440
omniOrb2/omniTransport/ATM	380
omniOrb2/ATM	360

Table 3: omniORB2 Null Echo

5.2 Bulk Data Transfer

The performance in bulk data transfer is measured by sending multiple sequence of octets using either one-way or exactly-once operations. The following IDL interface is used:

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

Throughput against sequence size was measured for a byte stream with one-way semantics for a total transfer of 10 Mbytes is shown in Figure 10. A byte sequence uses the zero copy capabilities of omniORB2 and hence does not require any marshalling or copying within the ORB. This test is able to saturate the link at 135 Mbps for a 4K sequence size[2]. An artifact of the Nagle algorithm used in TCP[3] accounts for the TCP throughput which is greater than that of raw ATM for small sequence sizes. This is of course at the expense of latency.

Throughput against sequence size was measured for a byte stream with exactly-once semantics for a total transfer of 10 Mbytes is shown in Figure 11. Again, the byte sequence uses the zero copy capabilities of omniORB2 and hence does not require any marshalling or copying within the ORB. This test differs from the previous in that a reply from each request must have been

[2]Direct socket programming will saturate at 3K.
[3]Delay the sending of small packets while there are outstanding acknowledgements.

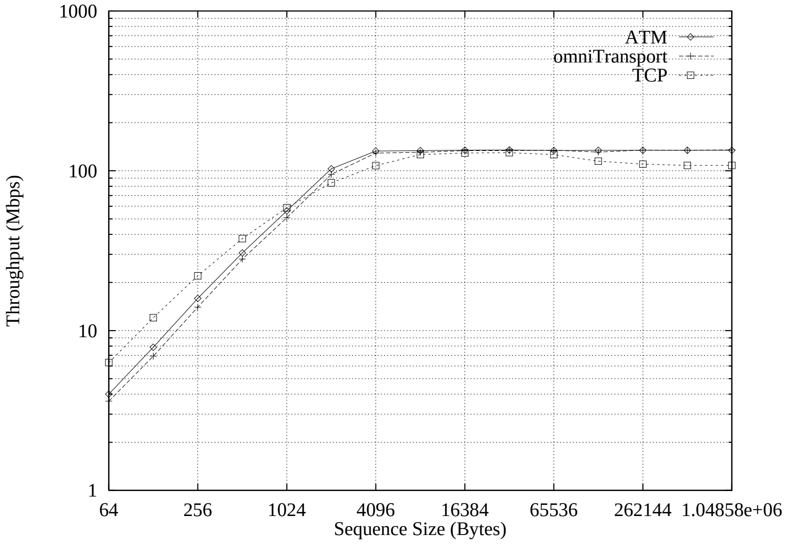

Figure 10: Inter-machine one-way throughput

processed by the client before a subsequent request is made. Nevertheless, the test saturates the link at a sequence size of 16k.

There is no effect from the TCP Nagle algorithm in a request-response interaction. However, it should be noted some tuning was required to obtain the traces. For the TCP over ATM trace, a collapse occurred at 4k unless the **nodelay** option was set[4]. Also for TCP over ATM, a marked throughput drop occurred around 32k unless the socket send buffer size was adjusted from 64k to 32k. A similar drop in throughput at 32k occurred for the raw ATM and omniTransport traces and was fixed with an adjustment of the maximum AAL5 packet size to 8k. This problem appears to be a function of the buddy scheme used for memory allocation in the Linux Efficient Networks ATM driver, since this test required both very large and very small blocks to be allocated for each request-response.

As a further comparison, the same test was carried out for TCP over ATM between a pair of 166 Mhz Sun Ultra workstations on the 155 Mbps ATM network. The plot reached an asymptote of 61 Mbps at a sequence size of around 128K.

Throughput against sequence size was measured for a marshalled sequence from a sustained burst of 10 Mbytes is shown in Figure 12. Each sequence element consists of two 32 bit integers, hence omniORB2 stubs will copy and marshall the sequence before transmission. As for the previous test, a reply is expected from the server. This test is CPU bound for all sequence sizes. The performance gain by using omniTransport over TCP is a result of the reduction in protocol processing overhead.

[4]This was considered strange since nodelay simply disables the Nagle algorithm.

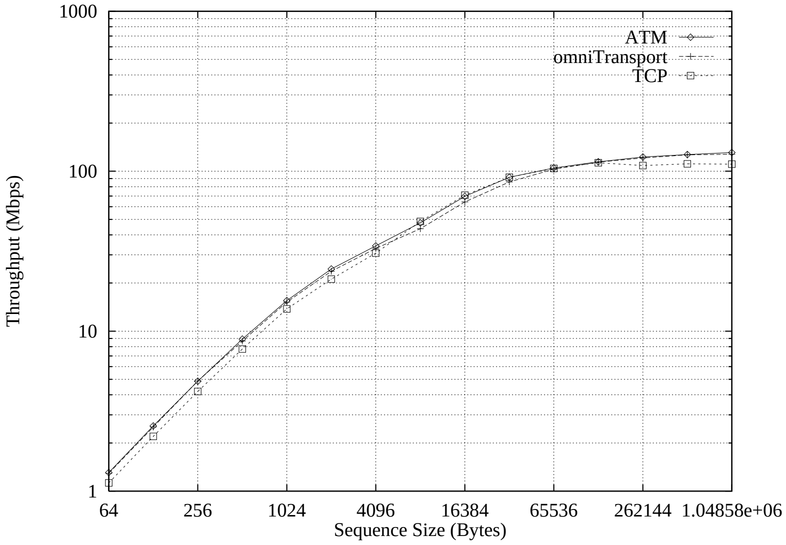

Figure 11: Inter-machine request-response throughput

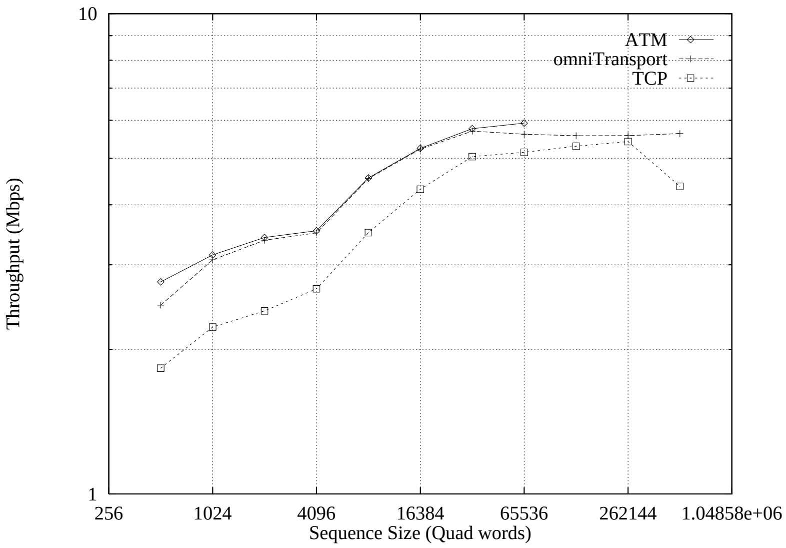

Figure 12: Inter-machine marshalled sequence request-response throughput

6 Conclusions

Our results indicate that the omniTransport protocol outperforms the Linux in-kernel TCP implementation over a wide range of throughputs and argument types. The poor performance of the Solaris TCP implementation provides a motivation for porting omniTransport onto native ATM programming interfaces available for Solaris, such as X-Open XTI. We are continuing with work which will provide dynamic transport protocol selection for omniORB2 whilst retaining its CORBA 2 compatibility.

More generally, this paper has shown that in the domain of request-response applications, it is possible to use a lightweight user level transport protocol and outperform a highly optimised general purpose in-kernel transport protocol.

References

[1] W Almesberger. ATM on Linux, 1998. http://lrcwww.epfl.ch/linux-atm/.

[2] D Clark and D Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM90 Conference*, pages 200–208, 1990.

[3] A Edwards and S Muir. Experiences implementing a high-performance TCP in user-space. Technical Report 95-110, HP Laboratories, 1995.

[4] S. Lo. The omniORB2 version 2.5 User's Guide, 1998. http://www.orl.co.uk/omniORB.

[5] S. L. Lo. The Implementation of a Low Call Overhead IIOP-based object request broker. In *ECOOP '97 Workshop CORBA: Implementation, Use and Evaluation*, 1997.

[6] S. L. Lo. The Implementation of a High Performance ORBover Multiple Network Transports. In *Submitted to Middleware 98*, 1998.

[7] G Mapp and S Pope. The design and implementation of a high-speed user-space transport protocol. In *Proceedings of the IEEE GLOBECOMM97 Conference*, 1997.

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

[9] XTP Forum. *Xpress Transport Protocol Specification*. XTP Forum, March 1995. Rev 4.0.