Networking Abstractions and Protocols Under Variable Length Messages *

Stephen Milliner
School of Information Systems
Queensland Univ. of Technology
Brisbane, QLD 4001
Australia

Alex Delis
Department of Computer and
Information Science
Polytechnic University
Brooklyn, NY 11201

Abstract

The size of data shipped over networks of distributed databases has increased substantially with the introduction of multimedia and high performance systems. Therefore, there is a need to understand the behavior and the trade-offs of available data communication abstractions in such settings. In this paper, we consider parameters associated with the abstractions, the impact of their implementation on throughput, and we compare the performance of BSD sockets (TCP, UDP), System V TLI (TCP), and SunOS RPCs (TCP) in the presence of large messages. In such environments, segmentation overhead of UDP messages at the user level outweighs other considerations such as optimal `mbuf` *allocation and full ethernet bandwidth usage. In addition, it is found that the size of buffer areas allocated to TCP sockets gains further importance when sending large messages. We also find that communication induced paging significantly deteriorates the data transmission throughput and suggest a policy to avoid this problem.*

1 Introduction

As there is a strong trend for distributed database applications to move towards the service of multimedia [20], real–time delivery [12], interactive [16], active [25], high–availability [1], and replicated information systems [11, 8], the importance of shipping large massages across the network becomes paramount. New paradigms for distributed database organizations such as client–server [15, 9, 5, 10] may also impose heavy requirements at the sending and receiving end points of a networked system.

In this paper the effects of limited main memory on communication performance are investigated. This is done by measuring data transmission times and throughput rates as the size of messages increases. Experiments are performed on an ethernet local area network using communication abstractions including : BSD Unix sockets based on TCP (Transmission Control Protocol), BSD Unix sockets with UDP (User Datagram Protocol), Unix system V TLI (Transport Layer Interface) with TCP, and SUN RPC (Remote Procedure Calls) with TCP. A comparison is carried out and the effects of paging on the sending of large messages is reported. Since we are interested in communication performance between applications, all measurements are taken at the user application level. Although the implementation of communication abstractions, the ethernet policy, and the way in which the operating system handles communications cannot be directly modified, a number of parameters at user level can be adjusted to tune performance. These parameters include the size of the segments that UDP transfers, the number of synchronization messages between senders and receivers, and the size of the socket TCP and UDP buffers.

The paper is organized as follows: section 2 discusses the communications abstractions and the underlying communication protocols. In section 3, we briefly describe previous research performed in the area. Section 4 discusses experimental goals and design. Section 5 discusses our experimental results. Conclusions and future work can be found in the last section.

2 Fundamentals of Data Transmission

The communication process can be envisaged in terms of several layers. The communication abstractions, TLI, RPCs and Sockets provide an interface to the lower communication levels. The communication protocols, UDP and TCP can then be selected by the user applications to provide unreliable connectionless datagram, or reliable connection–oriented byte–stream services, respectively. The IP (Internet Protocol) provides the elementary communication services upon which TCP and UDP build their functionalities. The network interface then provides access to the underlying physical communications medium.

2.1 The Communication Abstractions

Berkeley sockets and the System V designated TLI provide elementary interfaces for accessing the communication protocols. TLI is *stream* based (see [23, 22]), but in SunOS 4.1.3 the TLI library actually serves as an interface into the sockets abstraction and transport provider (TCP/IP) [18]. Hence, TLI is

*This research was partially supported by grant NRGS-1816-50008

0-8186-7216-1/95 $04.00 © 1995 IEEE

not expected to out–perform socket based communications. The UDP flavor of TLI has not been implemented in SunOS 4.1.3.

RPCs run on top of the socket interface and inherits its services. Because of the overhead associated with *marshaling* and *unmarshaling*, and its placement on top of sockets, it is expected that RPC transmission performance will be inferior to *pure* sockets based transmission times. TLI based RPCs are not available in SunOS 4.1.3 [18].

2.2 The Transport Layer Protocols

TCP acknowledges received packets and performs re–transmissions when necessary, it checks for duplicate message fragments and ensures fragments arrive in the correct sequence. TCP also implements a *sliding window* that allows only a maximum number of message segments to be un-acknowledged at any one time. The size of the socket's receive buffer dictates how big the window can become. A buffer size equal to the size of data within a single ethernet packet will result in a sequential send packet/receive acknowledgment mode of operation. The receiving end of TCP processes data faster than it can be sent [21, 7]. This is because the sending end controls the transmission and must deal with windowing, acknowledgments, re–transmissions and so on. TCP overheads can be reduced by increasing the allowable number of un-acknowledged message fragments and by acknowledging several fragments at a time. No such synchronization process occurs in UDP transmissions unless explicitly coded into the user application. Therefore, UDP transmissions are inherently unreliable and, (partially) because there is no synchronization overhead, faster.

In order to provide a fair comparison of TCP and UDP, the reliability of the transfers must be considered in addition to the transfer rates achieved. In our experiments UDP transmissions took place between two machines, and proved reliable. In such an environment, the throughput of the *optimistic* UDP is expected to surpass that of TCP. However, there are two implementation aspects of UDP which must also be considered with respect to its performance.

The size of UDP messages used was held constant at the system default of 9000 bytes. This introduces additional overhead as large messages must be fragmented in the user application. In addition it was found that, in the SunOS 4.1.3 UDP implementation used, the un–interrupted sending of a large message resulted in packets being dropped at the receiving end and the freezing of the receiving process. To prevent this, synchronizing messages were sent regularly, thereby introducing a second UDP overhead. This need for synchronization is discussed further in a later section. The following subsection briefly discusses the overall communication stages between two user processes.

2.3 Application Level Data Flow

Figure 1 summarizes the various stages in socket based message transmission from one machine to another (only data queues are shown [21]).

There may be at most four context switches re-

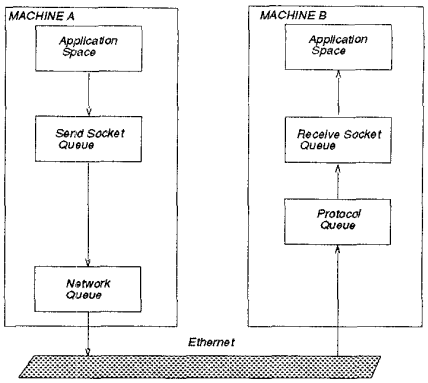

Figure 1: Communication of User Processes

quired to send a single message assuming all processes sleeping and reliable connection. Transmission performance is thus, in part, determined by the OS scheduling policy/algorithm and the nature of processes which are competing for CPU time. Large reliable messages will require more CPU time than short reliable ones. As the Unix system tends to favor jobs which are lightly CPU intensive, short messages will be scheduled faster than large messages. In addition, large messages may induce paging overheads. Paging is a non-interruptible process which results in memory resident pages being transferred to and from disk swap space. This additional I/O means that large message processing will result in performance degradation of competing processes (even if their pages are pinned into memory). The effect of background CPU noise on large message transmission is discussed in [19].

Passing of data between the buffers depicted in Figure 1 proceeds using a fixed amount of kernel memory, which corresponds to the maximum setable receive/send socket buffer size of approximately 50 kBytes (default is around 4 kBytes). The data structure used to manipulate this buffer space is the `mbuf`. A single `mbuf` contains 112 bytes of data, or it may reference a 1024 byte page within the kernel memory (its original 112 bytes are not used). The former is termed a plain `mbuf`, the latter a cluster `mbuf`. SunOS provides an optimization of `mbufs` whereby the pages referenced by a cluster `mbuf` may be external to the kernel memory. Hence, data is sent/copied directly to main memory where it is referenced by kernel `mbufs` and the applications also. This cuts the number of memory-to-memory copies from 2 to 1 (or 4 down to 2 for data passing from sender to receiver). Memory-memory and user-memory coping represents a significant amount of the communications overhead (approx. 48 percent) [7]. Thus, optimal manipulation of `mbufs` is an important consideration. Because TCP is based on a byte sequence, it is impossible to allocate `mbufs` optimally (for messages lager than 1024 bytes). UDP, on the other hand, is datagram based and a priori knowledge of message boundaries does allow optimal `mbuf` allocation.

Transmission throughput degrades significantly when ethernet traffic causes packet collisions and re–transmissions. In these experiments the effects of the ethernet carrier sense multiple access with collision detection (CSMA/CD) and binary exponential back off policy are assumed equal across all experiments (given the constant small amount of background traffic).

3 Previous Work

A large amount of research work has been performed in the area of communication interface performance over the last few years. However, none of this works has examined the behavior of these communications protocols under large data transfers - an overhead question pertinent to database systems. In addition there appears to have been no comprehensive comparison of the communications abstractions studied here.

Areas covered in recent works include performance measurements of OSI based and TCP/IP protocols and application services over LANs [24], application protocol performance [13] such as FTP, FTAM etc., understanding the overheads of the TCP protocol [7, 21, 4], the effects of transport system architectures on communications processing [2, 17, 22, 14], the effects of background CPU and/or ethernet loads [4, 17, 3], the effects of socket buffer size [21], and finally issues regarding scalability [17, 6].

In [22, 14], it is noted that the `mbuf` structure is biased towards small messages. Memory-to-memory copying is a significant overhead and several papers have noted its potential to create a bottleneck in the communications process [7, 2, 17]. However, the role of the `mbuf` structure in large message transmission was not considered. In [4] both background ethernet and CPU loads were varied. Both checksum and memory-to-memory copying were found to be the prime causes of performance degradation. It was also found that TCP messages of size 1024 bytes resulted in much lower overheads than messages of 1023 bytes. This is due to the fact that 1024 bytes fit in a single cluster `mbuf`. The 1023 bytes would require multiple 112 byte plain `mbufs`. The effects of the `mbuf` structure are reported in [24] as well. But, once again, no mention is made of the impact of `mbuf` allocation on large message throughput.

In [21] the effects on performance of changing the socket buffer size are studied. Throughput increases when the buffer is increased from the default 4 kBytes to 16 kBytes, and then only marginally when the buffer size is changed to around 50 thousand bytes. It is also found that the TCP protocol can process data more quickly than a 10 mbps ethernet can deliver it. When a background CPU load is introduced at the sending node, the socket queue bottlenecks (Figure 1). A CPU load at the receiving node also causes data to build up in the socket queue. Finally, for the default buffer size (4 kBytes) 12 connections can be supported (in SunOS 4.0.3) before buffers begin to overflow. If the send socket buffer is increased to 16 kBytes this drops to 3 [21].

4 Experimental Objectives

The main aims of this work were to:

1. Examine communication performance as message size increases.
2. Determine optimal parameters of existing communications abstractions for sending large messages.
3. Compare the performance of a comprehensive set of abstractions and protocol flavors.
4. Determine the best method for sending large messages based upon the results obtained.

In order to resolve the above questions we run three major experiments.

- The first experiment addresses the trade offs between various segment sizes for UDP transfers, the allocation of `mbufs`, and ethernet bandwidth use.
- The purpose of the second experiment is to investigate the role of the TCP socket buffer sizes as message size increases.
- Finally, the third experiment compares the performance of RPC, TLI, and sockets based on various protocols.

The first experiment was run to determine the best message segmentation size for UDP based socket transfers. The default UDP send buffer size of 9000 bytes was used. Thus larger messages must be segmented within the user application. The following factors were considered with regard to determining the optimal segment size :

1. `mbuf` allocation.
2. IP fragmentation (minimal traffic).
3. Segmentation processing overhead.

Optimal `mbuf` allocation minimizes memory-to-memory copying. To achieve this the segment size should be a multiple of 1024—the size of a cluster `mbuf`. Optimal IP fragmentation results in full utilization of the 1460 bytes in each ethernet packet that maximizes bandwidth. To achieve this the segment size should be a multiple of 1460 bytes. The fewer segments sent the fewer socket write/read calls and iterations of the loop which is used to segment the message. Segments should be the default maximum of 9000 bytes to minimize segmentation overhead. In this experiment runs are made using segment sizes based upon the numbers quoted above. Because an acknowledgment mechanism must be incorporated into UDP based socket transfers, runs are also made to determine the effect of sender/receiver synchronizations on performance. In order to ensure that data had been transmitted reliably, the data was sent back to the sending node for comparison with the original message. UDP timing was performed at the sending node and messages between 1 and 5 million bytes were used. Timing began with the sending of the first segment and concluded upon receiving of the final acknowledgment.

The second–TCP socket–experiment was performed to determine the best send and receive socket buffer size for large messages. Runs were made using buffers which were equally varied at the sender and receiver from 4 kBytes up to a maximum size of 50

thousand bytes. The message size ranged from 1 to 24 million bytes. Transmission timing began with invocation of the user application socket writing function and finished upon its return.

In the final experiment a comprehensive comparison is made among TCP sockets, UDP sockets, TCP based RPCs, and TCP based TLI communications. The optimal TCP and UDP settings, determined in the previous two experiments, were used along with default settings for TLI and RPC transmissions. TLI was timed in the same way as the TCP socket case. Timing of RPC runs began with invocation of the remote procedure call and terminated upon its return (no parameter was returned by the RPC).

The experimental testbed consisted of two networked SUN workstations running the SunOS 4.1.3 operating system. The two machines had 20 and 16 mega bytes of RAM and local disk swap spaces of 68 and 57 mega bytes respectively. A 10 megaBits/sec ethernet was employed along with AMD Am7990 LANCE ethernet controllers.

5 Experimental Results

The following three subsections summarize the results of our experiments.

5.1 Varying UDP Segment Sizes

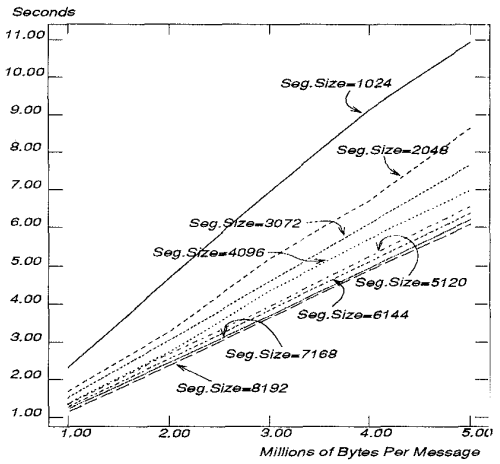

Figure 2: Socket UDP Transfers with Variable Segment Sizes (multiple of 1024)

Figure 2 shows the transmission time (in terms of secs) as the size of messages increases from 1 million to 5 million bytes, and as the UDP message segment size varies between 1024 and 8192 bytes. The segment sizes used are multiples of 1024 bytes resulting in optimal `mbuf` allocation. As the segment size increases, the overheads due to message segmentation and shipment decrease, as does the the number of synchronization messages. However, this decrease is relative to the fixed time it would take to transmit data optimally (ie. in a single UDP message). Thus, the performance increases as the segment size grows. But the relative gains in performance decrease in a non-linear way as the fixed transmission overhead value is approached.

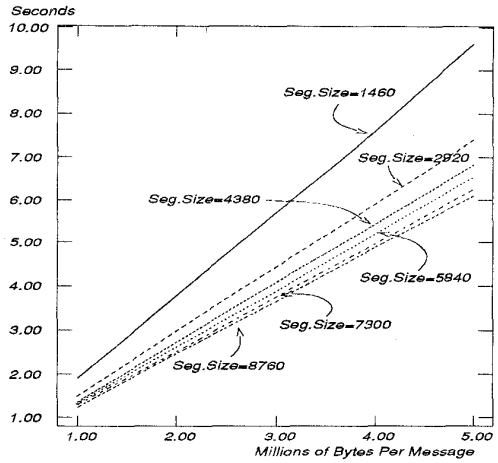

Figure 3: Socket UDP Transfers with Variable Segment Sizes (multiple of 1460)

Similarly, in Figure 3 the best performance is achieved with the largest segment. Segments in Figure 3 are multiples of 1460 bytes to optimize ethernet packet usage and IP fragmentation. In our experiments, 1460 bytes is the maximum amount of data carried by each ethernet packet.

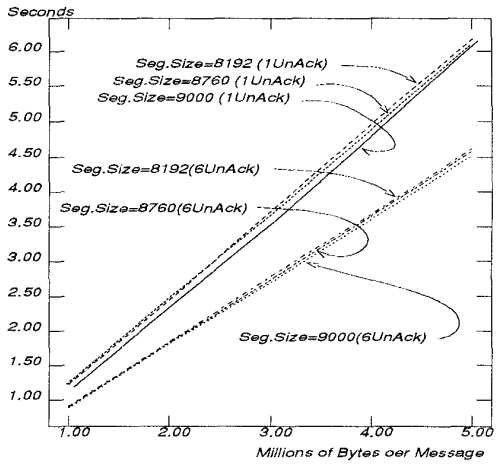

Figure 4: Comparison of Segment Sizes and Synchronization Overheads

In Figure 4 the best runs from Figures 2 and 3 are compared with a run using a segment size of 9000 bytes. The cost of synchronization is also considered. The set of curves marked "1UnAck" corresponds to runs in which only one segment may be unacknowledged at any point in time. Runs marked as "6UnAck" allow 6 segments to be sent before a synchronizing ac-

knowledgment is sent back. Changing the number of acknowledgments did not alter the order to the response curves. A segment size of 9000 bytes results in the smallest transmission time, closely followed by the transmission times based on 8760 byte and 8192 byte segments.

From Figure 4 we see that less than optimal `mbuf` allocation (98 versus 100 percent) and a lower degree of ethernet packet usage (88 versus 94 percent), does not counteract the overhead incurred by the segmentation of the message. For this reason a segment size of 9000 bytes was chosen as the default in all other experiments. Obviously this may not be prudent if network traffic is high as 9000 byte segmentation produces significantly more ethernet packets.

One other obvious and rather expected influence on UDP based socket performance is the affect of synchronization. Unlike TCP, UDP has no built in flow control mechanism. Thus when the network or socket buffers are exceeded packets will be dropped. To prevent this a acknowledgment/synchronization step was incorporated at the user level. This consisted of a 2 byte message encapsulated in a single ethernet packet which was sent after a certain number of segments had been received. It was found that the largest number of 9000 byte segments that could be sent "reliably", before packets were dropped, was 6. Thus, in all other experiments a default of 6 unacknowledged 9000 byte segments was used.

Although the network interface can store up to 73000 bytes, the receive socket buffer can only store 52000 bytes and there is no guarantee that the receiver's read will be scheduled before data in the receive socket buffer is overwritten. There is no flow control between these layers and data is simply discarded when buffers are full. In TCP a maximum window size of 64 kBytes is set to prevent/restrict such overflow. If large messages are to be sent using UDP an equivalent flow control mechanism is required. Thus, there is a need for synchronization messages and the limit on the number of unacknowledged segments. Why 6 (54000 bytes) rather than 7 (63000 bytes) unacknowledged segments results in reliable UDP transfers is uncertain. Presumably, because the UDP protocol at the receiving end is more simplistic than TCP, its ability to flood the receiving socket buffer is increased - thus the need for a smaller "window". An alternative explanation is that this form of overflow also occurs in TCP, but is hidden by the re-transmission mechanism.

5.2 Impact of the TCP Socket Buffer Size

The results of the second experiment are shown in Figure 5 and 6. Send and receive TCP socket buffer sizes in 5 are varied from 4096 bytes up to 50000 bytes. It has been found previously [21] that increasing the buffer sizes to 16 kBytes has a significant effect on performance for messages less than 1 million bytes. The gain in throughput for buffers of greater sizes is minimal (see Figure 5). A larger buffer size means that there will be fewer acknowledgment messages and better use of the ethernet bandwidth. The larger the receiving buffer, the wider the flow control's sliding window can open. A larger buffer thus enables a larger number of packets to be unacknowledged and fewer sender/receiver synchronizations via acknowledges.

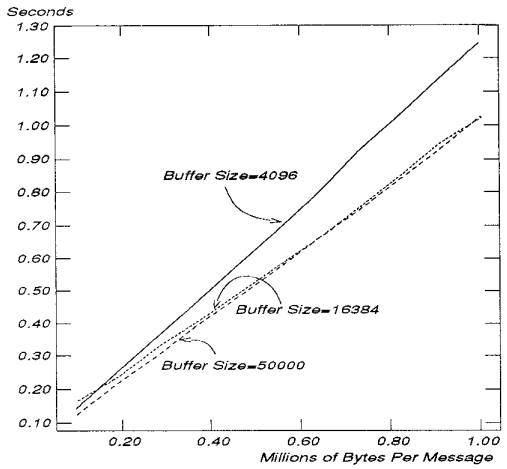

Figure 5: Low Space for TCP Transfers with Variable Buffer Sizes

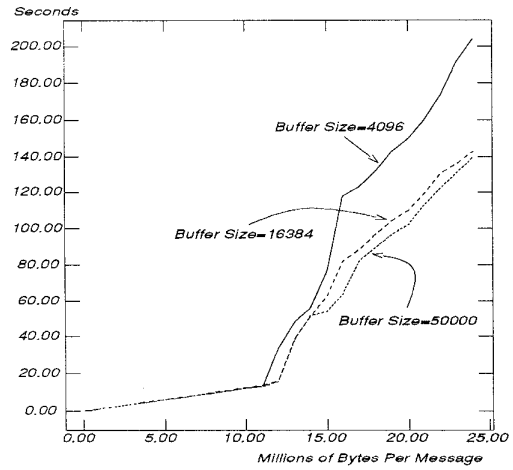

Figure 6: TCP Transfers with Variable Buffer Sizes

Fewer synchronizations results in better use of the network bandwidth and less acknowledgment traffic. Papadopoulos et. al. [21] have found that when a 50 thousand byte buffer is used instead of a 4 kByte buffer, approximately half the number of acknowledgments are produced. The inability of TCP to make use of the larger buffers suggests that optimal pipeline parallelism is achieved when 11 ethernet packets (16060 bytes) have been transmitted. That is, the time taken for the sender to process an acknowledgment and transmit 16 kBytes, corresponds roughly to the time taken for the receiver to do protocol processing, pass the information on to the user application and send an acknowledgment. This leads to a steady

state situation in which queues do not back up and there is no need to reject and re-transmit data. This is not so in the previous case of UDP where synchronizations are generated at the user application level. This simplistic stop-and-wait flow control means that transmissions are sequential - thus in the UDP case the larger the buffer/window the better.

In Figure 6 the effect of buffer size is not significant, for messages less than approximately 12 million bytes. But for message sizes greater than 12 million bytes, increasing the buffer size results in a distinct performance advantage. But once again there is relatively little advantage gained in increasing the buffer size above 16 kBytes. Paging effects of transmission are clearly evident after 12 million bytes. A buffer size of 4096 bytes limits the the window size to one unacknowledged packet. This results in sequential stop-and-wait transmissions similar to UDP base on one synchronization per segment. Poor performance is exacerbated when paging begins because the "wait" now also includes the paging overhead. Thus, the effects of buffer size gain even more importance in the case of large message transmission. In all other experiments a 50 thousand byte buffer was chosen as the default.

5.3 Abstractions Comparison

The results of this experiment are summarized in Figures 7 and 8. Figure 7 depicts a performance comparison of the abstraction/protocol combinations for message sizes up to 10 million bytes. As was expected, socket based UDP (6 un-acknowledgments) has the fastest transmission time. This is followed by the UDP socket communications using one acknowledgment per segment and, the reliable TCP based socket transmissions (using 50 thousand byte buffers), the RPC based on TCP sockets, and lastly, the connection oriented TLI. By increasing the buffer size TCP can almost match unreliable UDP performance based on one synchronization per segment but TCP is surpassed, if UDP takes advantage of the reliable environment and reduces the number of synchronizations. RPC performance, being based on TCP, is expected to be worse than TCP. In addition, RPCs' incur argument marshaling costs, and use the default TCP buffer size of 4096 bytes. TLIs poor performance, however, is surprising. Either some artifact has been introduced by the test code, or there is an implementation problem associated with SunOS 4.1.3 TCP based TLI. This anomalous behavior was not observed when a test was run between two SunOS 5.3 machines indicating the latter of these two explanations is probably the more likely.

In Figure 8 a performance comparison of the abstraction/protocol combinations is presented for messages up to 24 million bytes in size. Transmission time increases almost linearly until a message size between 12 and 13 million bytes is reached. At this point, in all cases (bar that of TLI), paging overhead becomes a major consideration. Once again UDP based socket transmissions (6 "UnAck") are fastest. However, TCP transmissions now outstrip UDP sockets based on 1 synchronization per segment. One possible explanation for this is that the segment size used

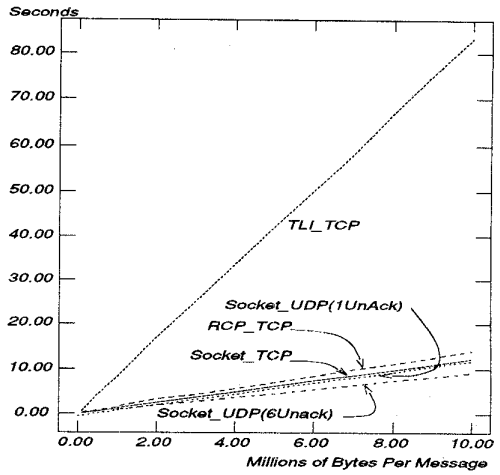

Figure 7: Response Time for the Various Protocols up to 10 Million Bytes

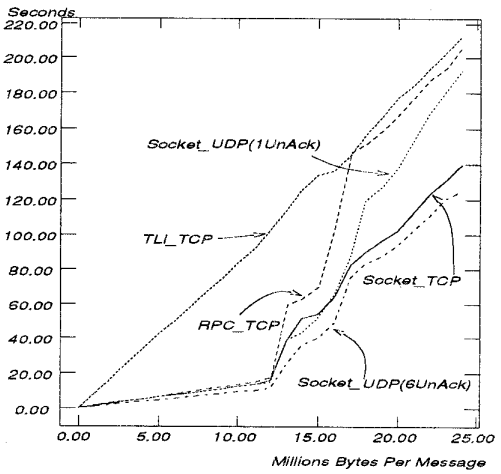

Figure 8: Response Time for the Various Protocols up to 24 Million Bytes

(9000 bytes) conflicts with page size (4096 bytes). If we assume a message is stored in logically contiguous pages in virtual memory, then 3 page faults will occur each time a 9000 byte segment is referenced. This results in a relatively high proportion of irrelevant information retrieval (approximately 25 percent). TCP socket transmissions are based on a bit stream. So the amount of data sent and received at any one time is uncertain. However, it was found that the vast majority of socket reads return the same number of bytes as the maximum socket receive buffer (50000). This corresponds to a relatively low proportion of potentially unused data being retrieved (about 6 percent). The TLI curve tends towards the RPC curve as the message size is increased. This kind of behavior is expected as TLI is also based on TCP sockets.

Initially throughput close to the ethernet maximum of 10 mega bits per second (due to the absence of significant background traffic) is observed in Figure 8. It then plunges at around 12 million bytes, slows slightly for messages of about 13 to 14 million bytes and then plunges again before stabilizing for message sizes greater then 17 to 18 million bytes. Throughput decreases by a factor of approximately 5 for almost all of the curves as the message size goes from 10 to 20 million bytes.

Paging overhead increases rapidly as more and more of the message is stored in virtual memory. When the entire message has been paged out page faulting will be at its maximum - after this point there will be only a linear increase due to paging overhead, paging has reached a "saturation" level. One could speculate that the initial plunge in throughput, at around 12 million bytes (see graph 8), is due to page faulting in both the sending and receiving machines. The slowing in throughput degradation, at approximately 14 million bytes, could possibly be due to page faulting saturation on the machine with only 16 mega bytes of RAM. Similarly, the final *leveling off* of throughput, at around 17 million bytes, is possibly due to maximum page faulting at the 20 mega byte RAM machine.

In conclusion, the degradation of throughput due to paging overheads has been clearly demonstrated (reduces throughput by a factor of about 5). As it was expected, the socket abstraction outperformed all others, and the more "optimistic" UDP protocol achieved a higher throughput than the reliable/pessimistic TCP protocol. In a paging environment, the relative differences between TCP and UDP were maintained. Before paging commences, it is recommended that the UDP based socket abstraction be used (in reliable environments). However in the region where swapping happens the choice of the abstraction is less significant.

One way to prevent paging overhead is to restrict the amount of memory taken up by the transmitting and receiving processes. If, for example, a 10 million byte message can be sent without inducing paging of the transmitting process or any other database processes, then a 20 million byte message could be sent in two halves. To achieve this half of the message could be written to, and later read, from disk. In doing this the paging algorithm of the system is overridden. There are several advantages in doing this :

- all database processes remain memory resident and the risk of thrashing is circumvented
- while transmission time may still be increased (due to disk I/O) there will be no paging overhead for any of the other database processes
- a specific message paging algorithm can be developed.

The last of these advantages is most likely to have a significant influence. A "message paging" algorithm can take advantage of the strictly sequential nature of message transmissions. Thus, instead of bring in a page at a time as is done a general paging algorithm, a large segment of the message can be written to memory at the one time. This reduces the amount of transmission induced I/O compared with OS paging which cannot, and therefor dose not, assume highly sequential page accessing. However, the realization of this potential performance improvement will be dependent upon a number of factors. OS scheduling of paging processes may, for instance, differ from the scheduling of processes which include I/O (although disk I/O has a similar kernel priority to paging).

5.4 Experiment Error Rates

During the experiments collisions were found to be very low (less than 4%). Thus we assume that the effect of ethernet transmission is a constant, and is not considered when interpreting the experimental results.

It has been noted [4] that repetitions of several thousand are required when performance measurements are made using messages of several thousand bytes. However, a UDP experiment using 1 to 25 million byte messages, with a 1 million increment and 20 repetitions, took approximately 48 hours. Hence, the number of repetitions was limited. The standard deviation of our runs varied from 5 to less than 1 percent, an acceptable error rate.

Factors contributing to this variation may have included sporadic background traffic which, although minimal at the time of night the experiments were run, could not be totally eliminated due the shared network environment. Performance was measured using the gettimeofday system call which has a millisecond resolution.

6 Conclusions and Future Work

Network technology has increased transmission rates greatly and has resulted in a shift of performance bottlenecks away from the network to the sending and receiving end points. This has resulted in additional performance pressure on the higher levels of communication models. In this paper, we have examined the trade–offs and the behavior of the available data communication abstractions using large messages. More specifically, we have compared the performance of of BSD sockets (TCP,UDP), System V TLI (TCP), and SunOS RPCs (TCP). Our main conclusions are:

- Segmentation overhead of UDP messages at the user level outweighs other considerations such as optimal `mbuf` allocation and full ethernet bandwidth usage.
- The size of buffer areas allocated to TCP sockets gains further importance when sending large messages.
- Communication induced paging significantly deteriorates the data transmission throughput.

The detrimental effects of paging on message transmission have been demonstrated, indicating a need to more closely integrate networking and operating system components. We proposed a message paging policy that circumvents the operating system paging policy and eliminates paging creating by message shipments. In future work we will be running experiments to determine to what extent these optimizations are feasible. In addition, we will be considering the effect

on large message transmission of UDP buffer size, and the introduction of background CPU noise and ethernet traffic. Finally, experimentation will be performed using different operating systems. In particular, we plan to reproduce our experiments using the thread based Solaris 2 environment.

References

[1] R. Alonso, D. Barbara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM–Transactions on Database Systems*, 15(3):359–384, September 1990.

[2] S. Banerjee, V.O.K. Li, and C. Wang. Distributed database systems in high-speed wide-area networks. *IEEE Journal on Selected Areas in Communications*, 11(4):617–630, May 1993.

[3] A. Braccini, A. Del Bimbo, and E. Vicario. Interprocess communication dependency on network load. *IEEE Transactions on Software Engineering*, 17(4):357–369, April 1991.

[4] L. Cabrera, E. Hunter, M.J. Karels, and D.A. Mosher. User-process communication performance in networks of computers. *IEEE Transactions on Software Engineering*, 14(1):38–53, January 1988.

[5] M. Carey, M. Franklin, M. Livny, and E. Shekita. Data Caching Tradeoffs in Client–Server DBMS Architecture. In *ACM–SIGMOD–Conference on the Management of Data*, May 1991.

[6] I. Chlamtac and A. Ganz. A study of communication resource allocation in a distributed system. In *IEEE 10th International Conference on Distributed Computing Systems*, pages 530–536, May 1990.

[7] D.D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, pages 23–29, June 1989.

[8] A. Delis and N. Roussopoulos. Server Based Information Retrieval Systems Under Light Update Loads. In *Proccedings of the 1991 IEEE International Conference on Systems, Man, and Cybernetics*, Charlottsville, VA, October 1991.

[9] A. Delis and N. Roussopoulos. Performance and Scalability of Client–Server Database Architectures. In *Proccedings of the 19th International Conference on Very Large Databases*, Vancouver, BC, Canada, August 1992.

[10] D. DeWitt, D. Maier, P. Futtersack, and F. Velez. A Study of Three Alternative Workstation–Server Architectures for Object–Oriented Database Systems. In *Proceedings of the 16th Very Large Data Bases Conference*, pages 107–121, 1990.

[11] H.M. Gladney. Data Replicas in Distributed Information Services. *Transactions on Database Systems*, 14(1):75–97, March 1989.

[12] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan–Kaufman, San Mateo, CA, 1992.

[13] P. Gunnungberg, S. Pink, M. Bjorkman, P. Sjodin, E. Nordman, and J. Stromquist. Application protocols and performance benchmarks. *IEEE Communications Magazine*, pages 30–36, June 1989.

[14] N.C. Hutchinson and L.L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[15] W. Kim, J. Garza, N. Ballou, and D. Woelk. Architecture of the Orion Next–Generation Database System. *IEEE–Transactions on Knowledge and Data Engineering*, 2(1):109–124, March 1990.

[16] J. Leggett and J. Schnase. Viewing DEXTER with open eyes. *ACM–Communications*, 37(2), February 1994.

[17] E. Mafla and B. Bhargava. Communication facilities for distributed transaction-processing systems. *IEEE Computer*, pages 61–66, August 1991.

[18] Sun Microsystems. *Sun OS 4.1.3 Answer Book*. SunSoft, 1992.

[19] S. Milliner and A. Delis. A LAN based comparison of IPC abstractions. In *18th Australian Computer Science Conference*, pages 399–406, February 1995.

[20] J. Murray. K12 network: Global education through telecommunications. *ACM–Communications*, 36(8), August 1993.

[21] C. Papadopoulos and G.M. Parulkar. Experimentation of sunos ipc and tcp/ip protocol implementation. *IEEE Journal on Selected Areas in Communications*, 11(4):489–505, May 1993.

[22] D.C. Schmidt and T. Suda. Transport system architecture services for high-performance communications systems. *IEEE Journal on Selected Areas in Communications*, 11(4):489–505, May 1993.

[23] W.R. Stevens. *UNIX Network Programming*. Prentice-Hall, 1990.

[24] L. Svobodova. Measured performance of transport service in LANs. *Computer Networks ISDN Systems*, pages 31–45, 1989/1990.

[25] K. Wilkinson and M.A. Neimat. Maintaining Consistency of Client–Cached Data. In *Proceedings of International Conference on Very large Data Bases*, August 1990.