A HIGH-PERFORMANCE TRANSPORT PROTOCOL FOR LOCAL AREA NETWORKS

SADHNA AHUJA
GTE Laboratories Inc.
40 Sylvan road
Waltham, Massachusetts 02451-1128, USA
Email: sadhna.ahuja@gte.com

ROBERT D. RUSSELL
Department of Computer Science
University of New Hampshire
Durham, New Hampshire 03824-3591, USA
Email: rdr@unh.edu

ABSTRACT

This paper describes LAMP, a high-performance transport protocol designed to provide a reliable, in-order, message-oriented service on local area networks. It incorporates many of the features described in the literature as desirable for high-speed networks. So that LAMP can be easily integrated into existing application programs, it presents the standard BSD socket interface to users. In order to be able to utilize any existing networking technology, it utilizes the standard interface to network device drivers offered by the system. It collapses the transport and network layers, and bypasses but coexists with IP. The first implementation of LAMP is for Linux, and shows improved performance over TCP in terms of both bandwidth and latency (response time).

Keywords: transport protocol, local area network, high speed networks.

1 INTRODUCTION

A number of studies have been done to determine desirable properties of transport protocols for high-speed networks [1] [2]. According to the list given in Doeringer *et al.* [2], TCP, which is by far the most widely utilized transport protocol, is deficient in a number of areas. One of the fundamental sources of these deficiencies is that TCP was originally designed at a time when bandwidth was an expensive resource relative to computation speed, and transmission errors occurred frequently. These deficiencies are particularly relevant to LANs (Local Area Networks), which today have very high bandwidth, often measured in gigabits per second, and extremely low error rates.

Another issue with existing protocols is that they are often designed with WANs (Wide Area Networks) and internets in mind, which may result in poor performance on LANs. In particular, many of the algorithms for flow control and congestion control in TCP, such as the Nagle algorithm, delayed ACKs, *etc.*, perform best on a WAN, and can be detrimental to LAN operation in some cases.

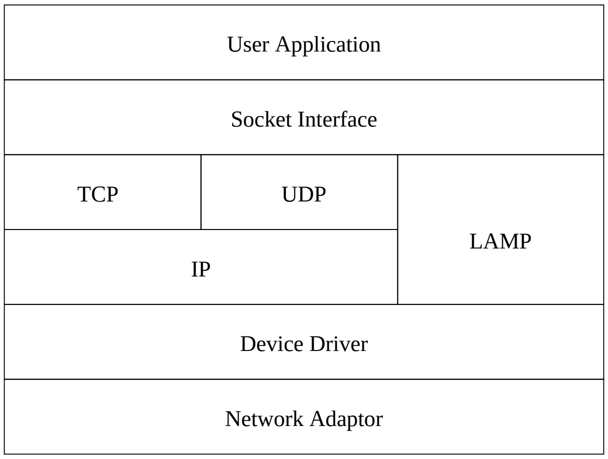

FIGURE 1: LAMP and the Linux network layers.

This paper reports on the development of LAMP, (Local Area Message-oriented Protocol), an alternative to TCP and UDP for local area networks. LAMP provides users with a reliable, in-order stream of messages of arbitrary size. It was designed to incorporate most of the features found in the literature to be desirable for high-speed networks. These include use of out-of-band signaling for acknowledgements and error reporting, labels for fast packet dispatching, minimum header processing, streamlined flow control and others. This protocol was designed to present the standard BSD socket interface to users and, in turn, to utilize the standard interface to network device drivers offered by the system. Figure 1 illustrates how LAMP fits into the Linux network layering model. By limiting its scope to local area networks, routing was not a concern, so that the implementation bypasses but coexists with TCP, UDP and IP. This further improves performance by collapsing the transport and network layers into one.

2 USER-LEVEL API

A user-level application program utilizes LAMP via calls to the BSD socket interface in a manner similar to the way it would utilize TCP or UDP. One major difference is in the specification of the parameters to the `socket()` function: the first (domain) parameter has the value `AF_LAPF` (`PF_LAPF`), which indicates the "Local Area" address (protocol) family; the second (type) parameter has the value `SOCK_SEQPACKET`, which indicates a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; the third (protocol) parameter has the value `0`. The complete call is:

```
fd = socket(AF_LAPF, SOCK_SEQPACKET, 0);
```

Once a socket has been created with the `socket()` function, a client process will use the `connect()` function to establish a connection to a remote server, and the server process will use the `bind()`, `listen()`, `accept()` sequence of functions to accept connections from clients. For purposes of the client-server rendezvous, standard IP addresses are used, and LAMP maintains a port number assignment parallel to those of TCP and UDP. Once connected, clients and servers use the standard `read()` and `write()` (or `recv()` and `send()`) functions to transmit data, and `close()` or `shutdown()` to terminate connections.

The other major difference between LAMP and TCP is that LAMP transmits messages rather than a stream of bytes. This means that on a `write()` operation, all the bytes specified by the user are considered to be one message that is delivered intact to a single `read()` operation on the other end of the connection. If the message written is longer than the buffer provided in the `read()`, the extra bytes are truncated without error. In this respect, data transmission is analogous to UDP (although LAMP provides for reliable, in-order delivery, which UDP does not). The user can set the size of the maximum possible message that can be sent (received) by using the `SO_SNDBUF` (`SO_RCVBUF`) option to the `setsockopt()` function.

3 PROTOCOL DESIGN

The design of LAMP was guided by a number of the principles delineated by Doeringer *et al.* [2] The first example of this is the fact that LAMP is implemented as two separate protocols at the data link (MAC) level: one to carry data, the other to carry control information (i.e., a signaling protocol). This design simplifies the packet format and processing, making both more efficient. It also enables SMP platforms to process data in parallel with control information, and allows these protocols to travel in parallel over different links, although this has not yet been tried.

0	16	31
Source Label	Destination Label	
More to Follow	Payload Length	
Sequence Number		
Unused		

FIGURE 2: Fixed-size LAMP data packet header.

3.1 Data Protocol

A packet in the data protocol consists of the fixed-size 16-byte header shown in Figure 2, followed by a variable length payload whose maximum size is determined by the MTU of the underlying data link layer. For Ethernet, which has an MTU of 1500 bytes, the maximum LAMP payload is 1484 bytes. The header layout conforms to another of Doeringer's principles: align all fields on easily accessed boundaries. As shown in Figure 2, fields in the LAMP data header are either 2-byte or 4-byte integers, and a field of size "x" is aligned on an address boundary that is a multiple of "x". Furthermore, since the header itself is 16 bytes, a 16-byte alignment of the data link payload is preserved for the LAMP payload, a factor that makes it faster for some hardware to copy information.

Another feature of LAMP is that when a connection is first established, each end of the connection dynamically assigns a unique internal addressing label to that connection, and it is these labels that are used in the data packet header for addressing purposes. In particular, the destination label is used by the receiving side as a direct index into an internal table to obtain the pointer to the control block for this connection. It is impossible to demultiplex incoming packets any faster.

User-level messages are fragmented by LAMP on the sending side into the appropriate number of data packets as required by the MTU of the underlying data link. These fragments are reassembled by LAMP on the receiving side before being delivered to the user as a single message. Each data packet is assigned a sequence number that is used to ensure reliability and in-order delivery. This number is 4 bytes long, permitting 4 gigapackets to be transmitted before wrap-around occurs, which means that on Ethernet, where the LAMP payload is 1484 bytes, more than 4 terabytes can be transmitted in one direction before the sequence number wraps. This is significant, because out-of-order packets are differentiated from duplicate packets by whether their sequence number is more or less than

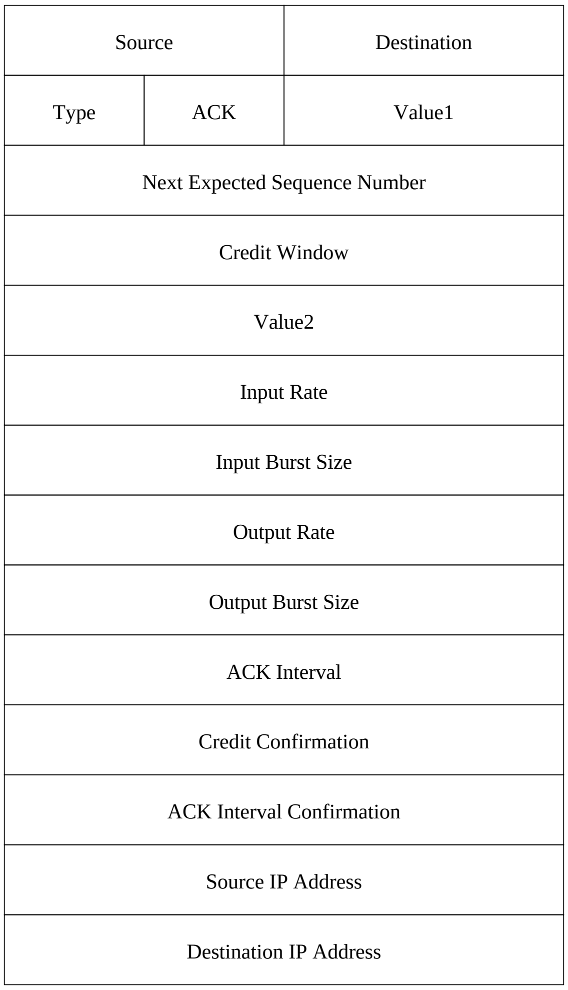

FIGURE 3: Fixed-size LAMP control packet.

one-half the maximum sequence number ahead of the expected sequence number. Therefore, the maximum sequence number times the payload size cannot exceed twice the bandwidth-delay product of the link. Four terabytes should be sufficiently large for the foreseeable future of LAN technology.

Note that the data packet header contains only information needed to multiplex and process the payload data in that packet. There are no piggy-back acknowledgements, no window advertisements, no connection control information. All that information is carried in separate control protocol packets.

3.2 Control Protocol

Packets used by the control (signaling) protocol are shown in Figure 3. Each packet contains the same fixed number of fields, although not all fields are used in all packets. There are 6 control packet types, as indicated by the type field in the header. All fields are either 1-, 2- or 4-byte integers that are aligned on appropriate byte boundaries for rapid access.

To establish a connection between client and server, LAMP uses a 3-way handshake similar to that used by TCP. As part of this handshake, each side establishes a local label that defines the connection on that side, and exchanges this label with the other side for use in demultiplexing all subsequent data and control packets. Other information exchanged and confirmed is the initial sequence number, credit window size, and the currently unused traffic-shaping parameters (rate and burst size for input and output flows). A connection is closed by using a 2-way handshake.

Once data packets are flowing from one side to the other, the receiver simply counts the number of packets it has received and sends an acknowledgement control packet after receiving `ack_interval` packets correctly. A negative acknowledgement control packet is sent back immediately whenever a dropped packet is detected. Error recovery is "go-back-n", which was found by Doshi and Johri [1] to be the best error recovery scheme when error rates were very low, as is the case in modern LANs. This scheme permits the receiving side to simplify its processing and buffering by not retaining out-of-order packets.

4 IMPLEMENTATION

There were two basic approaches that could have been taken in implementing LAMP: implement it as a user-level transport protocol on top of UDP, or implement it as a kernel-level protocol. XTP [3] is an example of the former approach. Although a user-level protocol is clearly more portable, we felt that that approach would sacrifice efficiency. UDP uses port multiplexing, but we felt that the use of address labels would be more efficient. (Although it can be disabled, UDP also normally performs checksumming, which is also unnecessary and therefore inefficient, given the highly reliable CRC mechanism provided by modern LAN technologies.) UDP in turn utilizes IP, which provides two main functions, routing and fragmentation, neither of which we needed. Routing was not an issue, since we were specifically targeting LAMP for LANs. Because of its expense, IP fragmentation is intentionally avoided by TCP and has been eliminated entirely from IPv6. Finally, the ability to collapse the two-layer UDP/IP structure into one further increases efficiency. Therefore, LAMP was implemented as a set of C routines for the Linux 2.2.1 kernel that coexist with, but bypass, both UDP and IP.

The coding task was simplified as much as possible, with the intention of making the resulting code more efficient. Following the recommendations in [4], use of timers and other facilities provided by the operating system was minimized, although these could

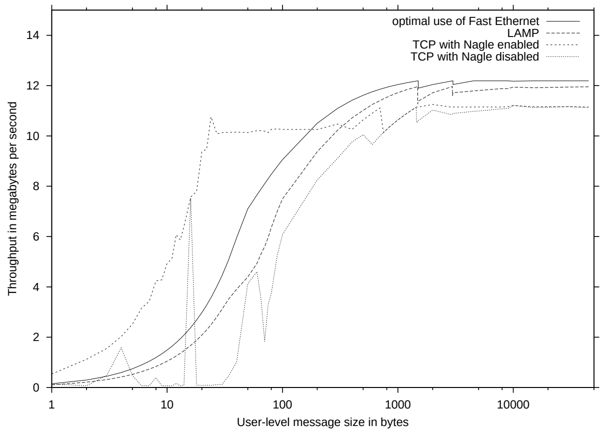

FIGURE 4: Throughput in megabytes per second.

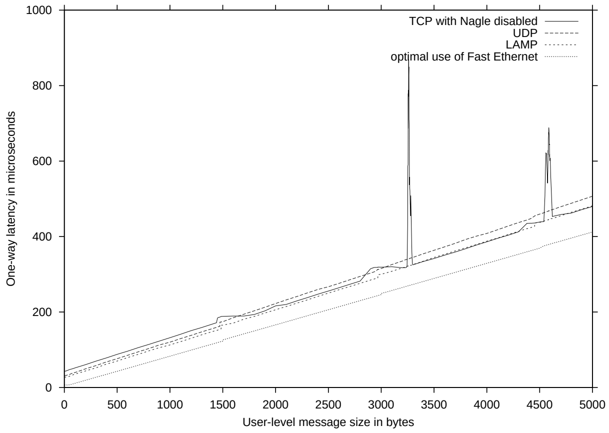

FIGURE 5: Latency in microseconds per message.

not be entirely avoided because of the need for LAMP to conform to the requirements of both the socket interface and standard network device drivers. Bandwidth was intentionally sacrificed for processing efficiency: headers are perhaps longer than necessary in order to align information on conveniently addressable boundaries and to keep it always in the same place in a packet; control information is sent separately from data via a separate signaling protocol; out-of-order packets are retransmitted rather than being retained by the receiving side, *etc.*

5 PERFORMANCE

Much has been written about various ways to evaluate networking protocols in general and TCP in particular [5] [4]. To date we have performed very limited testing of LAMP *vs.* TCP by using two benchmarks, one to measure throughput, the other to measure latency or response time. These tests were all run on 600 MHz Pentium III processors having 256 Mbytes of memory that are interfaced via DEC DS21140 “tulip” adaptor cards to 100 Mbps fast Ethernet.

5.1 Throughput

Throughput was measured by using a “blast” program in which the client sends, as fast as it can, a sequence of fixed size data messages to the server, which just throws them away. After the final message has been received, the server sends back a short “done” message to the client. The total time is measured by the client from just before it sends the first data message to just after it receives the “done” message. Throughput in bytes per second is computed as the number of data messages sent by the client times the fixed message size divided by the total time. Figure 4 shows a plot of the throughput for various user-level message sizes. There are four curves in that figure: one for LAMP, one for TCP with its Nagle algorithm enabled, one for TCP with its Nagle algorithm disabled, and one for the optimal throughput (*i.e.*, the entire Ethernet payload of 1500 bytes is available for user data and there is no processing delay at either end of the connection). The maximum optimal throughput is 12.19 Megabytes per second and occurs for user-level messages that are multiples of 1500 bytes.

For messages smaller than about 200 bytes, Figure 4 shows that TCP with the Nagle algorithm enabled gives the best performance. This is to be expected, because the Nagle algorithm causes the sending side to buffer consecutive user messages, sending a data frame only when it is full. Therefore, the actual TCP frames are much bigger than the individual messages they contain. Neither LAMP nor the optimal calculation allows for extra buffering, so their performance is worse than that of TCP for small messages. In practice, however, applications do not “blast” data using small messages, and when messages larger than about 300 bytes are used, LAMP performs better than TCP with Nagle enabled. TCP with Nagle disabled gives very erratic throughput — near zero for most message sizes less than 40 bytes, but with extreme differences in performance for messages that differ only slightly in size. LAMP gives consistently better throughput than TCP with Nagle disabled (except for two of the TCP peaks). Although all the curves show abrupt drops at frame boundaries, the LAMP curve is much smoother than either TCP curve, indicating that with LAMP slight differences in message size do not translate into big differences in throughput as they do with TCP. Asymptotically, LAMP achieves 11.965 MBps, more than 98% of the optimal ethernet throughput, compared to 11.250 or 92% for TCP.

5.2 Latency

Latency (response time) was measured by using a "ping" program in which the client sends one fixed size data message to the server, which immediately sends it back to the client, where it is immediately sent to the server again, *etc.* The total time is measured by the client from just before it sends the first data message to just after it receives the last echo. The round-trip latency in seconds per message is the total time divided by the number of data messages sent by the client. One-way latency is half the round-trip latency. Figure 5 shows a plot of the one-way latency for various message sizes. There are four curves in that figure: one for LAMP, one for UDP (which is reliable in a "ping" scenario on a LAN), one for TCP with its Nagle algorithm disabled, and one for the optimal use of Ethernet. We also ran TCP with its Nagle algorithm enabled, which is the normal (default) mode of running TCP. However, for message sizes just larger than 1 MTU, the performance degraded by two orders of magnitude, pushing the values way off the top of the chart, so that any comparison was meaningless.

LAMP shows consistently better latency than both UDP and TCP. Furthermore, TCP has two unexpected spikes of extremely bad performance for message sizes around 3260 and 4587 bytes. Using a least-squares linear fit, the per-byte and per-message coefficients for the latency curve are:

	per-byte	per-message
TCP (Nagle disabled)	0.120	36.724
UDP	0.096	28.536
LAMP	0.092	23.419
optimal	0.082	2.256

Both the per-message and per-byte costs are smaller (better) for LAMP than for either UDP or TCP.

5.3 Evaluation

The limited benchmark testing described here may say as much about the implementations as about the basic designs. However, in the situations tested, LAMP performs better than TCP. More important, LAMP's performance is more predictable and more uniform than TCP's. Latency tests show that TCP should really never be used with the Nagle algorithm enabled, since there are situations where a small difference in message size can cause two orders of magnitude difference in response. However, blast tests show that when TCP's Nagle algorithm is disabled, throughput exhibits wild swings and consistently poorer performance than for either LAMP or TCP with Nagle enabled. Use of TCP therefore requires the user to be intimately familiar with his or her application's use of the network transport in order to avoid surprises. On the other hand, LAMP's throughput and latency vary smoothly with the size of the messages, avoiding unpleasant surprises for slight differences.

6 CONCLUSION

This paper presents an overview of the design of LAMP, a transport protocol that provides reliable, in-order, message-oriented service on LANs. It incorporates many of the features described in the literature as beneficial for high-speed networks. Initial performance tests demonstrate that it has higher throughput and lower latency, and is more predictable, than TCP in most situations. Further testing is planned to evaluate the effect of operational parameters, such as acknowledgement thresholds and buffer size, and to fine-tune them. We are also reorganizing LAMP's fragment reassembly to increase pipelining on long messages.

A project is currently underway to determine more precisely where processing time is consumed by protocol stacks in Linux. One item being looked at is the extra copying necessitated by the way Linux drivers "consume" network buffers. Although changing this would require changing the operating system, it should be possible to do it in a backward compatible manner that would allow new code to take advantage of it without reimplementing existing protocols or drivers. We also need to study the effects of LAN congestion on LAMP performance. Although hooks were designed into LAMP so that rate flow control could be used, it was never implemented. LAMP's effectiveness in real applications also needs to be investigated.

REFERENCES

[1] B. T. Doshi and P. K. Johri. Communication protocols for high-speed packet networks. *Computer Networks and ISDN Systems*, 24:243–273, 1992.

[2] W. Doeringer, D. Dykeman, M. Kaiserwerth, B. Meister, H. Rudin, and R. Williamson. A survey of light-weight transport protocols for high-speed networks. *IEEE Transactions on Communications*, 38(11):2025–2039, November 1990.

[3] Sandia National Laboratories, California, USA. *SandiaXTP, the Sandia Implementation of the Xpress Transport Protocol*, http://www.ca.sandia.gov/xtp/SandiaXTP.

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

[5] M. Allman and A. Falk. On the effective evaluation of tcp. *Computer Communication Review*, 29(5):59–70, October 1999.