Implementation Method of High Speed Protocol as Transport Library

Yutaka Miyake Toshihiko Kato Kenji Suzuki

KDD R & D Laboratories
2-1-15 Ohara, Kamifukuoka
Saitama 356, JAPAN

Abstract

Along with the rapid progress of optical technologies, the transmission speed of LANs and public networks has been increased significantly. As a result, it becomes possible that computers distributed geographically communicate with each other in high throughput. However, the current protocols such as TCP/IP have some problems, especially in the performance through long distance and wide bandwidth networks. Therefore, it is required to use new protocols with new data transfer algorithms. In this paper, we describe an implementation method of a high speed transport protocol by a user level library which interfaces to UDP/IP. This method uses only functions commonly provided by UNIX operating systems, and therefore, allows a new protocol to be developed easily and to be ported to other UNIX workstations easily. Our library called the transport library realizes the coordination of buffer managements in the application and the library, and low overhead and prompt handling of receive and timer interrupts in order to achieve high performance. Our implementation results of a high speed transport protocol show that 32 Mbits/sec over ATM network whose effective transmission speed is 36 Mbits/sec, regardless the propagation delay from 0 to 200 msec. These values of throughput are better than those of the in-kernel TCP programs.

1 Introduction

Along with the rapid progress of optical technologies, the network transmission speed has been increased significantly. In the case of LAN (Local Area Network), many high speed networks such as FDDI, ATM LAN and Fiber Channel have been available in recent years. The transmission speed of public networks is also being increased with the introduction of high speed leased circuits and ATM networks. By interconnecting these high speed networks, it will be possible that computers distributed geographically communicate with each other in high throughput.

Currently most UNIX workstations use TCP as a transport protocol for reliable data transfer. However, there are some problems pointed out for TCP/IP protocols. For example, although RFC-1323 [1] defines the window scale option expanding window size in order to achieve achieve high throughput in long distance and wide bandwidth networks, there are still the following problems. Currently only a few TCP implementations support this option. The window scale option is not sufficient for achieving high throughput and an appropriate congestion avoidance algorithm is indispensable [2–4].

In order to solve problems of TCP, it is required to use new protocols with new data transfer algorithms. However, the introduction of new protocols brings another issue on how to implement them on UNIX workstations. This paper describes our approach to implement new protocols as a user level library which interfaces to UDP/IP. We have adopted this approach because of following reasons:

- The development and debugging are much easier compared with the in-kernel implementation.
- A data transfer algorithm can be tuned up for individual applications.
- The demultiplexing of received packets which is difficult at the user level is performed by UDP/IP implemented in the UNIX kernel.
- Porting to other UNIX based operating systems is much easier compared with the in-kernel implementation.

There have been several research activities on protocol implementation in the user level [5–7]. However, these research activities focus on the specific operating system such as Mach 3.0 [5, 6] and the specific hardware such as Afterburner [7]. In contrast with them, our approach is focusing on implementing new protocols using only functions commonly provided by UNIX based operating systems.

The implementation of new protocols in a user level library using only common UNIX functions has some issues which are not considered in the in-kernel implementation nor in the user level implementation under specific environments. They include the coordination of buffer managements in applications and libraries, and low overhead and prompt handling of receive interrupts and timer interrupts.

In this paper, we describe our implementation method of high speed transport protocol by a user level library, which we call transport library in this paper, on the UNIX workstations. This paper also describes the performance evaluation of the transport

0-8186-7216-1/95 $04.00 © 1995 IEEE

library. The next section describes the design principles of the transport library on the functionality of the protocol adopted and on the implementation techniques. Section 3 describes the key issues on the implementation. This section also covers both aspects on protocol functions and implementation techniques. Section 4 evaluates the throughput this library and compares it with in-kernel TCP. Section 5 shows an example of applying the transport library to FTP (file transfer protocol) and evaluates the throughput. Section 6 gives conclusions on our work.

2 Design Principles

Figure 1 shows the protocol configuration implemented based on our approach. The transport library is linked with an application program and provides a connection oriented reliable data transfer. It supports protocol mechanisms such as connection management, flow control, error control, buffer management and timer management. The UDP/IP protocols implemented in the UNIX kernel support the routing of packets and the demultiplexing of the received packets to destination processes according to port numbers.

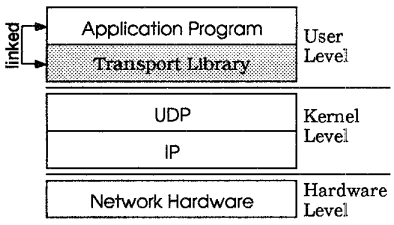

Figure 1: Protocol Configuration

The transport library is designed based on the following principles for the protocol functionality supported by the library and the protocol implementation techniques in the user level.

2.1 Principles on Protocol Functionality

In order to obtain high throughput bulk data transfer, we have selected the functionality of the protocol supported by the library based on XTP [8] with some modifications. The followings are the main features of our high speed transport protocol for reliable data transfer.

- Our protocol has an explicit connection establishment phase, which is different from XTP, and negotiates the application buffer size, the socket buffer size used in UDP and UDP ports during this phase.
- In order to separate the receiving of control and data packets, our protocol assigns different UDP sockets for these packet flows.
- Our protocol uses both the flow control mechanism and the rate control mechanism in a unified manner. The flow control is used to protect the receive buffer overflows and to avoid network congestion. The receiver determines *congestion window size*, the actual number of packets which the sender can transmit without an acknowledgment, according to the slow-start and congestion avoidance mechanism [2], and sends it to the sender in control packets.

  The rate control mechanism is used to suppress the speed of sending packets. This can avoid packet losses which may happen by the continuous packet sending according to large window size in the long distance and wide bandwidth network. The sending rate is changed dynamically according to the change of the congestion window size.
- In order to use our protocol in networks with various performance, we implements both the go-back-N and selective retransmission algorithms for error recovery.
- Our protocol provides two checksum fields in the header. One is used for header part, which is mandatory, and the other is for data part, which is optional. The use or non-use of data part checksum is indicated in the header on a packet by packet basis. The checksum algorithm of this protocol is the same as XTP [8].
- Some UNIX implementation limit the socket buffer size of UDP up to 64K bytes or 53K bytes. This limitation is often smaller than the required window size for long distance and wide bandwidth networks. Therefore, our protocol uses multiple UDP sockets for exchanging data packets when the limitation of socket buffer size is too small.

2.2 Principles on Implementation Techniques

In order to obtain high throughput using protocol software implemented as a user library, we have adopted the following implementation techniques.

- Application programs will use the buffers which they allocate for their own purposes. For example, an application which displays received data on the screen may allocate receive buffer on the frame memory. Therefore, we adopted the following principles for the application buffer management.
  - The transport library will not introduce its own buffers for sending and receiving data packets. The user data to be sent in application buffers will not be copied in the library for composing data packets. The user data in received data packets will be directly copied into application buffers.
  - The library needs to retain the sending data until they are acknowledged. This retention will be realized in the application buffers without burdening the application program with additional buffer management and without degrading the throughput.
- In the UNIX operating system, user programs cannot be interrupted when packets which they need to receive arrive. Therefore, the transport library needs to realize a polling mechanism for

the arrival of packet as often as possible. Especially, the control packets need to be detected and processed quickly for obtaining high throughput. The polling mechanism needs to be realized with low processing overhead.

- In the UNIX operating system, the time out handling has larger overhead in user programs than in-kernel programs. Therefore, the transport library needs to implement the timers with low processing overheads.

3 Key Issues on Implementation

3.1 Flow and Rate Control Mechanism

Our library realizes the flow control and rate control mechanisms in a unified manner based on the slow-start and congestion avoidance mechanism [2]. This mechanism is originally designed for the flow control, and changes the congestion window size dynamically along with detection of transmission errors to avoid network congestion. We use the congestion window size and round trip time to calculate the optimal transmission rate used in the rate control mechanism. Since the congestion window size is the maximum size of data that the sender can send during one round trip time (RTT), we calculate the transmission rate using the following equation in order to minimize the speed of sending packets.

$$\mathsf{TransmissionRate} = \frac{\mathsf{CongestionWindowSize}}{\mathsf{SmoothedRTT}}$$

In this equation, Smoothed RTT is the smoothed round trip time, and is calculated every RTT observation by the next equation used in TCP implementation [9].

$$\mathsf{SmoothedRTT} = \mathsf{SmoothedRTT} + \frac{1}{8}(\mathsf{RTT} - \mathsf{SmoothedRTT})$$

The rate control mechanism requires short period timing control. For example, when the data packet size is 8K bytes, the sender transmits data packets at every 2 msec. This value is much smaller than the other timers (refer to section 3.7). Therefore, we realized this mechanism not using our timer implementation but using the following procedure.

1. In the beginning of the `send()` function, the transport library will send the first data packet and remembers the time as the checkpoint time using `gettimeofday()`.
2. When the library is ready to send a data packet, it measures the sending throughput from the last checkpoint time.
3. If the sending throughput is smaller than Transmission Rate, the library sends a data packet immediately.
4. If the sending throughput is greater than Transmission Rate, the library waits using `select()` system call. In this system call, the library also waits for the arrival of control packets. If a control packet is received, it will process it and remember the received time as a new checkpoint time.
5. If there are any data to be sent in this `send()` function, then goes to step 1. Otherwise returns from `send()`.

3.2 Error Recovery

Error recovery mechanism in the transport library is based on the go-back-N and selective retransmission algorithms. Both algorithms are realized using a control packet which includes the receive sequence number, corresponding to the sequence number of the next in-sequence data packet, and the missing gap of the data packets.

When the receiver detects the packet loss or checksum error, it enters error recovery mode, transmits a control packet requesting the go-back-N or selective retransmission and waits for the requested data packet retransmitted.

When the go-back-N algorithm is selected, all the other data packets will be discarded in the error recovery mode. When the requested data packet is received, the receiver returns to the data transfer mode and start to receive data packets with the slow start and congestion avoidance mechanism.

When the selective retransmission algorithm is used, received data packets other than the requested packet are retained in the error recovery mode. When the requested data packet is received, the receiver will enter the data transfer mode and starts the slow start and congestion avoidance mechanism. By the very first control packet in the data transfer mode, it will request a selective retransmission of missing gap of data packets in the error recovery mode. The sender will retransmits the packets which is not received by the receiver.

Since the sender can respond the go-back-N and selective retransmission algorithms, the receiver can select both algorithms dynamically along with the performance of transmission line and so on.

3.3 Checksum

We have implemented the checksum for data part in the following way. UDP has 16-bit UDP checksum field in the header, whose use is optional and is determined by individual operating systems, and so the checksum for data part in the transport library is used only when the underlying UDP does not use the checksum. Since Host Requirements RFC [10] requires that UDP checksums must be enable by default, and the implementation of checksum within the data copy loops [11,12] is proposed, it is thought that most operating systems enable the checksum by default. Therefore, the data part checksum will not be used in many cases, which contributes obtaining high throughput.

3.4 Enlargement of Socket Buffer Size

Our protocol allocates the necessary send and receive socket buffers using the `setsockopt()` system call. However, some UNIX implementations have the limit on the socket buffer size. In this case, the transport library can handle multiple socket buffers as described in Fig. 2. The protocol negotiates the number of UDP sockets between communicating hosts during the connection establishment phase. If multiple sockets are selected, the sender distributes sending data

for each socket buffer and the receiver waits for the data using multiple UDP sockets.

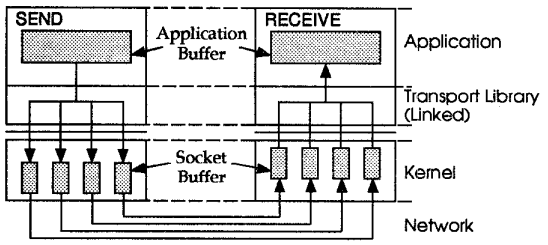

Figure 2: Use of Multiple Socket Buffer

3.5 Application Buffer Management

3.5.1 Avoiding Data Copying of Application Buffer

In the case of sending data, the application program calls the `send()` function with arguments of the pointer to an application buffer and its size. The function splits the data into an appropriate size, and sends the splited data with a header allocated by the transport library. In order to avoid user data copying, the header and the splited data are moved into kernel buffer in one `writev()` system call (see Fig. 3).

At the receiver side, the application program allocates an application buffer for receiving data, and calls the `receive()` function. The function waits for the arrival of data packets, and receives data using `readv()` system call. By this system call, the header part is put on the header buffer allocated by the transport library, and the user data is put on the application buffer (see Fig. 3).

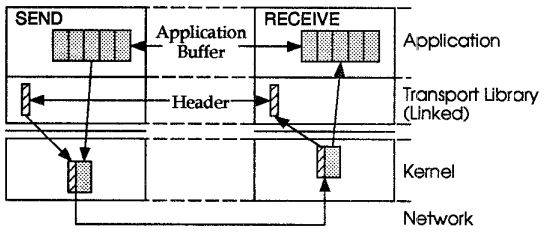

Figure 3: Header and User Data Handling without Data Copying

Received data are put on the application buffer in the receiving order. After it has been written into the application buffer, the sequence number is checked. If the receiver uses the go-back-N algorithm, the data with unexpected sequence number are discarded. If the receiver uses the selective retransmission, the received data with unexpected sequence number, i.e. stored in an incorrect position in the application buffer, are copied to the correct position.

3.5.2 Multiple Application Buffering

In order to allow the application program to manipulate the application buffer after the `send()` function is returned from the library, and in order not to degrade the throughput, we have implemented the following buffer management.

(1) When the application program sends application data in the application buffer, the `send()` function in the library sends the whole data in the application buffer. The `send()` function returns when the acknowledgment of the last data packet is received from the receiver. By this mechanism, the application program can manipulate the application buffer which it has sent as it likes. This process is depicted in Figure 4 (a).

(2) By using the above mechanism, there is a delay between the sending the last data packet and the receiving its acknowledgment. This might degrade the throughput especially in the long distance network. In order to avoid this problem, we introduce the multiple application buffering mechanism in the sending side (see Fig. 4 (b)).

- In the continuous data transfer, the application program gives multiple application buffers in the arguments of the first `send()` function call. The function sends the data in order from the first buffer to last buffer. When the acknowledgment of the last data packet in the first buffer is received, the `send()` function returns to application. The application program can update the first buffer.
- In the second `send()` function call, one application buffer is given in the arguments. The sending of data packet for the application buffers retained in the transport library is performed in the `send()` function. When the acknowledgment for the last data packet in the second buffer is received, the function returns and the application program can update the second buffer. (Note that the second buffer is different from the buffer given in the second `send()` function call.) For the third and later function call, the same procedure is performed.
- In the case of the last data sending, the application program calls the `send()` function with the argument of `EOF` flag. In this case, the function returns after receiving acknowledgment of the last data packet in the last application buffer.

3.6 Control Packet Handling

As described in section 2, the transport library needs to process received control packets promptly. The library checks the arrival of control packets at the following points.

1. The transport library provides an I/O demultiplexing function. The application program will call this function, in stead of `select()` system call, before accessing to I/O interface. When this function is called, the transport library checks the arrival of control packets. If any control packets are arrived, the library reads and processes them.
2. As described in this section, the transport library checks the arrival of control packet during the `send()` function. If the control packet is arrived, the protocol library reads and processes

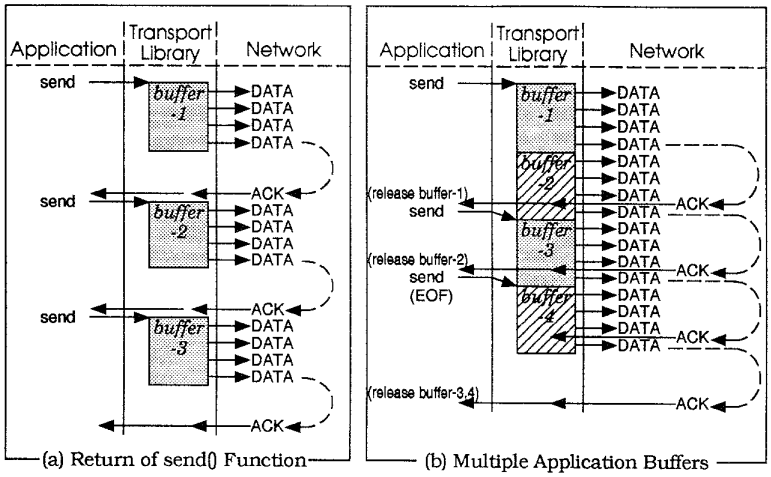

Figure 4: Buffer Management for Application Buffers

this packet, and the transmission rate is adjusted for the rate control mechanism.

3. The received control packets are also checked and processed in the `receive()` function which perform user data receiving for a requested application buffer.

3.7 Timers

The transport library uses `setitimer()` and `SIGALARM` signal to realize a timer facilities.

When the sender and receiver are communicating each other, the transport library sets the interval timer with 500 msec (this value is changeable from the application). The timers used in the protocol, such as persist timer, connection timer, retransmission timer, are registered in the expire list, and checked every 500 msec whether they are expired or not.

Since the transport library uses `SIGALARM` signal, the application cannot use related functions, such as `setitimer()`, `sleep()`, `usleep()`, `fork()`. Therefore, the library provides similar functions for application programs.

4 Performance Evaluation

4.1 System Configuration

Figure 5 depicts the system configuration used for the performance evaluation. We use an ATM network and a delay/error generator. The ATM switch used has two kinds of interfaces, one is TAXI (140Mbits/sec) and the other is DS-3 (45Mbits/sec). The TAXI interfaces are connected with workstations, and DS-3 interfaces are connected with the delay/error generator. The workstations are connected by PVC through the ATM switch and the delay/error generator.

The workstations used in this evaluation are SPARCserver 670MP (SPARC CPU 40MHz × 2) and SPARCstation 20 (SuperSPARC CPU 60MHZ × 2). The operating systems are SunOS 4.1.2 on SPARCserver 670MP and Solaris 2.3 (SunOS 5.3) on SPARCstation 20 respectively. In this evaluation, data is

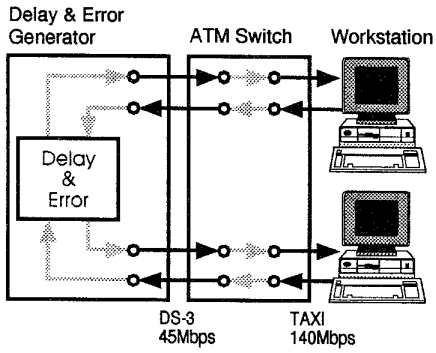

Figure 5: System Configuration for Evaluation

transmitted from SPARCserver 670MP to SPARCstation 20 basically.

In order to measure throughput, we use programs that perform memory copies between two workstations. Parameters for the transport library, such as send/receive buffer size, number of UDP sockets, number of application buffers, checksum on/off, error recovery (selective retransmission/go-back-N), rate control on/off, etc., can be changed with command arguments. In this evaluation, the throughput of TCP in the UNIX kernel is also measured. A free software called `ttcp` is used for this purpose. TCP program used here does not have the window scale option [1], and so maximum window size is 65535 bytes.

In this configuration, the bottleneck is the DS-3. Although the accurate transmission speed of DS-3 is 44.736 Mbits/sec, the maximum transmission speed of data carried by ATM payload is 36.864 Mbits/sec.

4.2 Results of Performance Evaluation

4.2.1 Effect of Transmission Delay and Number of Application Buffers

We have measured the throughput between two workstations using the transport library and `ttcp` with var-

ious delay. Figure 6 depicts this result. Round trip time (RTT) on this figure is the inserted delay by the delay/error generator. The parameters used in this evaluation is shown in Table 1. Buffer size for TCP is 52 Kbytes, since this value is the maximum size on SunOS 4.1.2. In Figure 6, "TCP" shows the performance of TCP derived from ttcp, and the others shows the performance of the transport library. In the case of the transport library, we measure 4 type of parameters changing the number of application buffers.

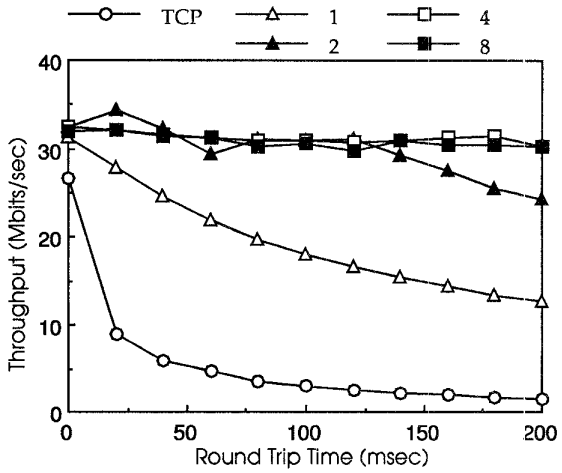

Figure 6: Relationship between Round Trip Time and Throughput

Number of Application Buffer	1, 2, 4, 8
Number of UDP Sockets	4
Checksum	OFF
Error Recovery	Selective Retransmission
Rate Control	OFF

Table 1: Parameters of the Transport Library for Evaluation

In the case of no delay inserted, the throughput of TCP is 26.78 Mbits/sec, and that of the transport library is about 32 Mbits/sec. The throughput of TCP is effected by the round trip time extremely. This is caused by the limitation of the maximum window size for TCP. On the other hand, when the number of application buffers is four or eight, the throughput of the transport library is kept the same level regardless of RTT increasing. In the case that the number of application buffers is one, the transport library has to wait for the acknowledgment of the previous buffer before it sends the contents of next buffer. Therefore, the throughput deceases gradually along with RTT increase. In the case of two buffers, when the round trip time is larger than 120 msec, the throughput decreases gradually. This corresponds to the calculated RTT using throughput of 32 Mbits/sec and buffer size of 512 Kbytes, by the equation $(512\ Kbytes \times 8\ bits/byte) \div 32\ Mbits/sec = 128\ msec$.

The above results indicate the followings:

- The transport library can achieve higher throughput than TCP implemented in kernel.
- Multiple application buffering works effectively especially for long distance and wide bandwidth network.

4.2.2 Rate Control

It is thought that the rate control mechanism is useful for the communication from the high performance computer to the low performance computer, since it prevents overrun of received packets at the receiving side. Therefore, we choose the SPARCstation 20 as sending side, and the SPARCserver 670MP as receiving side. This evaluation uses Ethernet (10Mbits/sec) as a communication network. Parameters for the transport library is as the same as the Table 1 except number of application buffer and rate control. In this evaluation, number of application buffer is also four.

Throughput comparison between the rate control ON and OFF is shown in Table 2. Performance improvement by the rate control is about 7 % in this case. Number of Retransmission requests for dropped packets is retained low level compared with no rate control.

4.2.3 Error Control

Figure 7 shows the relation ship between round trip time and throughput when transmission errors exist. We have inserted the random bit error rate of 1.0×10^{-8}. Parameters for the transport library is as the same as the Table 1 except number of application buffer and error recovery algorithm. In this evaluation, number of application buffer is four, and both the go-back-N and selective retransmission error recovery algorithms are used.

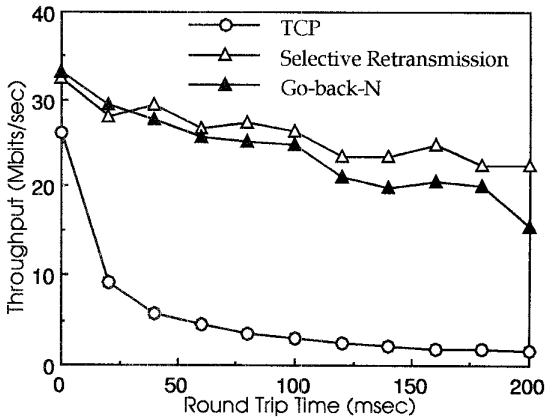

Figure 7: Relationship between Round Trip Time and Throughput with Transmission Errors

Protocol	Throughput (Mbits/sec)	Retransmission Requests (Times)
Transport Library with Rate Control ON	8.92	60
Transport Library with Rate Control OFF	8.31	164
TCP (ttcp)	8.09	

Table 2: Effectiveness of Rate Control

In the case of small RTT, the go-back-N algorithm is a little faster then the selective retransmission, since the handling go-back-N is simpler than the other. However, the selective retransmission algorithm works effectively on the long distance networks.

5 Application Example – FTP

This section describes an example use of the transport library. We have chosen FTP (File Transfer Protocol) [13] as an example, and modified FTP programs to include the transport library.

5.1 FTP programs

When we use FTP for file transfer, we need two kind of programs, a client program and a server program. Figure 8 shows a model of FTP client and server programs [9]. Generally, the client program have user interface, and is invoked by user. User commands the client program to retrieve a file list, get files from server, put files to server, and so on. The server program waits client requests, and performs that requests.

There are two TCP connections between FTP client and server. One connection is a control connection for transmitting commands and replies between clients and servers. The other connection is a data connection created at each time a file is transferred. We modify the data connection part of FTP client and server programs, which will affect the data transfer performance very much.

The programs used for this modification are ncftp and wu-ftpd, both of which are well known free software, and have changed the data connection part of these programs. There are no features that are deleted in this modification.

5.2 Performance Comparison

Figure 9 shows the throughput of file transfer using original ftp program and modified ftp program. The parameters used for the evaluation is shown in Table 3. The configuration for evaluation is the same as that described in section 4. In this graph, plot of ftp corresponds to the throughput using OS bundled ftp, and plot of tplftp corresponds to the throughput using modified ftp.

Maximum throughput using original ftp is about 16Mbits/sec at no inserted delay. The throughput decrease dramatically along with the increasing of inserted delay. As we mentioned in section 4, the maximum window size of TCP is limited under 65535 bytes, and this prevents the throughput. On the other hands, ftp programs which uses the transport library keep high throughput. Moreover, the throughput of tplftp always overcomes that of the original ftp. This result

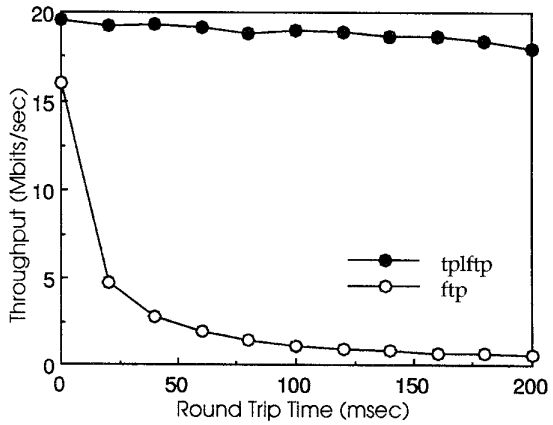

Figure 9: Throughput Comparison between Original and Modified FTP Programs

Application buffer size	512000 (bytes)
Number of application buffers	2
Number of UDP sockets	2
Error control	Selective Retransmission
Use of checksum	Checksum OFF

Table 3: Parameters of Transport Library for FTP Evaluation

shows that the transport library works effectively even if this is embedded in the applications.

6 Conclusions

In this paper, we have shown an implementation method of a new high speed transport protocol by a user level library which interfaces to UDP/IP in the UNIX kernel. This method allows a new protocol and a new data transfer algorithm to be developed easily and realizes the performance which is higher than or at least compatible with the in-kernel implementation.

This paper described our implementation results of a high speed transport protocol based on XTP with some modification. The protocol is designed to achieve high throughput even in long distance and wide bandwidth networks and the features include the combined flow control and rate control mechanisms, support of the go-back-N and selective retransmission error recovery algorithm and use of multiple UDP socket buffers. This paper also described the implementation techniques such as the coordination of buffer manage-

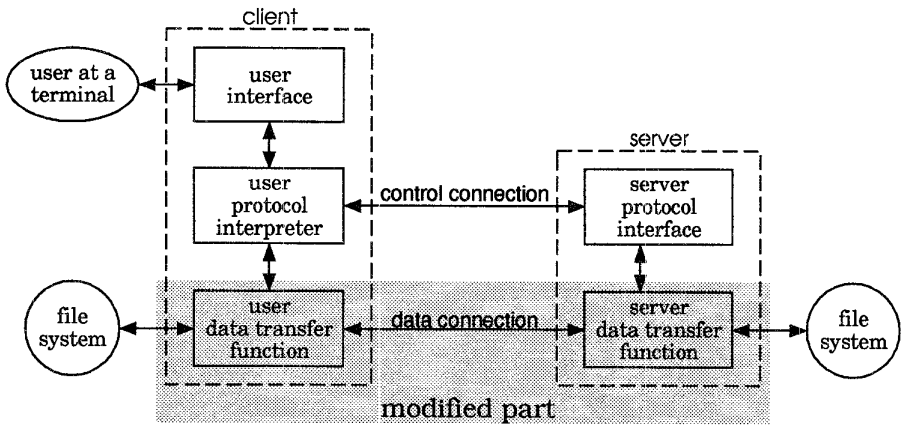

Figure 8: Modification of FTP Client and Server Programs

ments in the application and the library using multiple application buffering, and low overhead and prompt handling of receive and timer interrupts.

As the result of the performance evaluation, the transport library achieved 32 Mbits/sec over ATM network whose effective transmission speed is 36 Mbits/sec. This throughput was kept in the same level when the round trip time increased from 0 msec to 200 msec. These results is better than the in-kernel TCP program.

This paper also showed how to apply the transport library to existing application programs using FTP. We modified free software of FTP to include our library and the evaluation result showed the modified FTP program obtain higher throughput than the original program.

Our method uses only functions commonly provided by UNIX operating systems, and therefore, the implemented library is easily ported to other UNIX workstations. In our implementation, SUN OS 4.1.2 and Solaris 2.3 are used as UNIX based operating systems. Only the difference between two libraries for these operating systems is only a part related with the signal function, and the difference of source code is less than 100 steps, while the total size is about 8 Ksteps.

Acknowledgments

The authors wish to thank Dr. Y. Urano, Director of KDD R & D Laboratories for the continuous encouragement of this study.

References

[1] D. Borman, R. Braden, V. Jacobson, "TCP Extensions for High Performance," RFC-1323, May 1992.

[2] V. Jacobson, "Congestion Avoidance and Control," *Proceedings of the ACM SIGCOMM Conference*, pp. 314–329, August 1988.

[3] L. S. Brakmo, S. W. O'Malley and L. L. Peterson,"TCP Vegas: New Techniques for Congestion Detection and Avoidance," *Proceedings of the ACM SIGCOMM Conference*, August 1994.

[4] L. S. Brakmo and L. L. Peterson, "Performance Problems in BSD4.4 TCP," *ftp://cs.arizona.edu/xkernel/Papers/tcp_problems.ps*, November 1994.

[5] C. Maeda and B. N. Bershad, "Protocol Service Decomposition for High-Performance Networking," 14th ACM Symposium on Operating Principles, December 1993.

[6] C. A. Thekkath, T. D. Nguyen, E. Moy and E. D. Lazowska, "Implementing Network Protocols at User Level," *IEEE Trans. on Networking*, Vol. 1, pp. 554-565, October 1993.

[7] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Calamvokis and C. Dalton, "User-space protocols deliver high performance to applications on a low-cost Gb/s LAN," *Proceedings of the ACM SIGCOMM Conference*, pp. 14-23, August 1994.

[8] "XTP Protocol Definition Revision 3.6," *Protocol Engines Inc.*, PEI 92-10, 11 January 1992.

[9] W. R. Stevens, "TCP/IP Illustrated, Volume 1," Addison Wesley, ISBN 0-201-63346-9, February 1994.

[10] R. Braden, "Requirements for Internet Hosts – Communication Layers," RFC-1122, October 1989.

[11] V. Jacobson, "Some Design Issues for High-speed Networks," *Networkshop '93*, November 1993.

[12] C. Partridge and S. Pink, "A Faster UDP," *IEEE Trans. on Networking*, vol. 1, pp. 429-440, August 1993.

[13] J. Postel and J. Reynolds, "FILE TRANSFER PROTOCOL (FTP)," RFC-959, October 1985.