INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An Experimental User Level Implementation of TCP.

T. Braun, C. Diot, A. Hoglander, V. Roca

N° 2650

September 1995

PROGRAMME 1

Architectures parallèles, bases de données, réseaux et systèmes distribués

Rapport de recherche

1995

ISSN 0249-6399

An Experimental User Level Implementation of TCP.

T. Braun, C. Diot, A. Hoglander, V. Roca*

Programme 1: Architectures parallèles, bases de données,
réseaux et systèmes distribués

Projet Rodeo

Rapport de recherche n°2650- September 1995

20 pages

Abstract: With high speed networks supporting data rates of a few hundreds Mbps and distributed applications demanding high performance, existing communication system and protocol architectures are under discussion. In this report, we describe an implementation of the TCP/IP stack with TCP [1] in user space and IP in kernel space. The goal by placing TCP in user space is to obtain a flexible transport protocol which can in future allows integrated communication subsystems based on Application Level Framing (ALF) and Integrated Layer Processing (ILP) [2]. ALF and ILP are expected to achieve more efficient data transfer, and more application control in the communication process.

Key-words: Protocol Implementation, ALF, ILP, TCP, User level. High Performance.

(Résumé : tsvp)

* Email: tbraun@sophia.inria.fr; christophe.diot@sophia.inria.fr
Anna Hoglander is currently working at Telia Promotor AB; Islandsgatan 2,
S-751 42 UPPSALA, Sweden. e,mail: Anna.Hoglander@upp.promotor.telia.se
Vincent Roca is a researcher at LGI-IMAG; 24 Avenue Felix Viallet;
38031 Grenoble cedex (France); e.mail: vincent.roca@imag.fr

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
Unité de recherche INRIA Sophia Antipolis : 2004, route de Lucioles - B.P. 93 - 06902 Sophia Antipolis cedex **(France)**
Téléphone : +33 93 65 77 77 - Télécopie : (33) 93 65 77 65 - Télex : 970 050 F

Établissement public national à caractère scientifique et technologique - Décret N° 85.831 du 2 août 1985

Une implémentation expérimentale de TCP dans l'espace utilisateur d'une machine UNIX

Résumé : Avec l'avènement des réseaux hauts débits et des applications multimédia de nouvelle génération, les systèmes de communication conçus il y à une vingtaine d›années, ainsi que leur architecture, doivent être ré-étudiés. Dans ce rapport, nous décrivons une implémentation des protocoles TCP et IP avec TCP implémenté dans l'espace utilisateur du système UNIX. Placer TCP dans l'espace utilisateur plutôt que dans l'espace noyau confère une plus grande flexibilité au système de communication et permet d'étudier de nouveaux concepts d'implantation hautes performances tels TIC (ILP) et DTA (ALF).

Mots-clé : Hautes performances, TIC (ILP), DTA (ALF), Espace utilisateur UNIX.

1 Introduction

The existing standard protocols have been defined in the seventies when the transmission medium was the communication bottleneck. Today, with the emergence of high speed networks, the bottleneck has moved to higher layer protocols. New applications with time constraints only amplify this phenomenon. Current protocols and the associated layered architecture are not efficient in this new application environment. Furthermore, they do not provide services to support multimedia application requirements. Communication system architectures need to be revised. To evaluate new architectures and new protocols on UNIX workstations, flexible communication environments are required.

Traditionally, the communication software has been implemented within the kernel because of performance and security reasons. However, classic kernel implementation of TCP or UDP over IP are not flexible enough. This paper proposes a communication subsystem where the end to end communication protocols are implemented in user space and only IP remains in the kernel space for efficiency reasons. The end-to-end communication protocols are linked as a library to the application process. Such an environment is easy to modify and allows a large range of experiments with new communication architectures [3][4][5][6] and new implementation techniques such as ALF (application layer framing) and ILP (integrated layer processing) [2].

ILP is an implementation concept allowing to manipulate data in a single integrated loop. ILP experiments[7] have been performed on the user-level implementation described in the following. ALF is a precondition for applying ILP and means that the application breaks the data into suitable aggregates called application data units (ADUs), and the lower levels preserve these frame boundaries as they process the data. Another advantage of the fact that an ADU is the logical unit of *all* protocol functions is that a ADU can be processed independent of other ADUs.

The user level TCP implementation is based on the BSD TCP/IP stack. TCP has been divided in two parts: the demultiplexing function that remains in the kernel, and the rest of the protocol that has been moved to the user space. The user level TCP is also designed to be able to communicate with any other implementation of TCP.

The report is organized as follows: Section 2 describes the architecture of the user level implementation. Tasks left in the kernel are justified. Section 3 presents the implementation of TCP in user space (called TCPU). Section 4 explains some details of the TCPU implementations concerning timer and buffer management. A performance evaluation and a comparison with a kernel implementation conclude the paper.

2 Implementation Architecture

This Section illustrates the architecture to support the TCP implementation in user space. The platform in Figure 1 contains two TCP implementations: the experimental one running in user space, and the BSD implementation in kernel space. The purpose of this platform is to give the application programmer the choice of his communication strategy and to preserve the already existing environment for application programs using the classic TCP. Interoperabilty between the two different TCP implementations can be achieved by slight modifications of the IP implementation.

The original TCP kernel implementation has been split into two parts: TCPU and TCPK. TCPK resides in the kernel and provides demultiplexing for incoming packets. The user space TCP implementation (TCPU) is realized as a library, which must be linked to an application process. TCPU (cf. Section 3) is running in the user space and provides a TCP connection-oriented transport service. Instead of running TCPU over TCPK it would also be possible to run it over UDP. However, in this case, TCPU is not compatible with a pure kernel TCP. The following subsections describe the required modifications of the kernel.

2.1 Demultiplexing TCP packets

A packet demultiplexing function must know the IP addresses from the IP-header and the port numbers from the TCP header in order to map this information to a local socket number. With this information, the demultiplexing module can deliver the packet through the socket interface to the correct user process. Demultiplexing should be done in the kernel in order to deliver packets directly to the correct application process. Furthermore, demultiplexing in the kernel guarantees security, i.e. it prevents unauthorized packet reception. Another reason for demultiplexing on a low level is the possibility to guarantee quality-of-service on a per-application basis [8][9].

Our implementation with demultiplexing in the kernel, is close to a solution using TCPU over UDP without UDP checksum. One advantage of our approach is to avoid the UDP header in the communication stack. The more important advantage is that the TCPK solution can be connected transparently with an kernel implementation of TCP. This would not be possible with a solution using UDP.

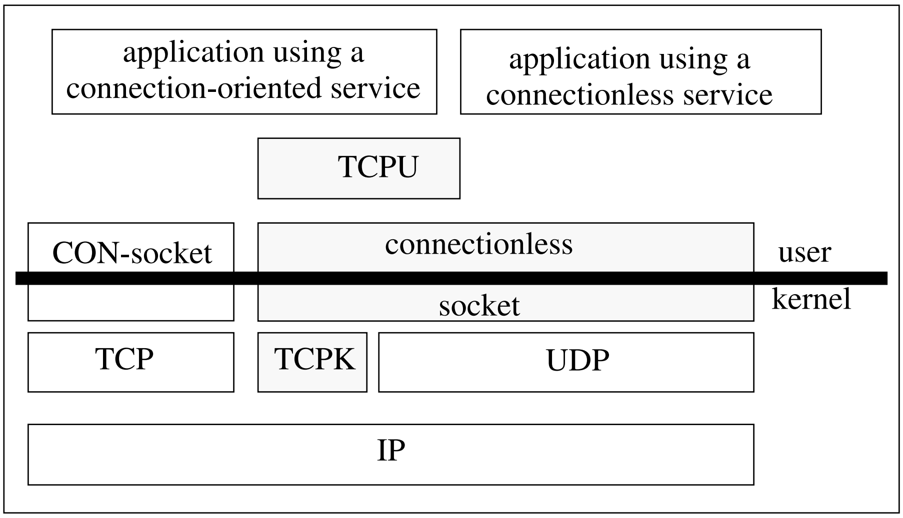

FIGURE 1. Configuration of TCPU and TCPK in kernel space.

Another solution for demultiplexing would be to pass directly received packets from IP to TCPU, i.e. to run TCPU on top of a raw IP socket interface. This approach would require a special demultiplexing process between the kernel and each application process running in user space and would cause significantly more context switches, because a received packet must be processed by at least two user processes instead of one with a kernel demultiplexing solution.

2.2 Socket interface between TCPU and TCPK

TCPK is defined as a new module between the socket interface and the IP protocol and it is responsible for demultiplexing of incoming packets. Connections are identified in TCPK by their port number and address. For outgoing packets, TCPK sends datagram packets to the connected host. TCPK is implemented as a UDP-like protocol, i.e. a protocol belonging to the AF-INET family and having the SOCK_DGRAM type.

The division and implementation of TCP in two parts did not require much changes of the already existing code in the kernel. The decision to split TCP has partly been based on the fact that existing code could be reused. Furthermore, the socket system call could be used without changing any code in the socket layer. From the user level point of view the application program interfaces (APIs) defined for BSD sockets can be used. The only difference for calling the socket for the first time, e.g. using the socket system call, is that a special identifier for the TCPK protocol must be delivered as the protocol parameter of the socket call as described in Section 3.2 .

2.3 IP implementation extensions

IP remains in the kernel space because IP performs only routing functions without any data manipulations. One reason for the user level implementation was the desired support of integrated layer processing. Because all data manipulations to be integrated are on the transport level or above, there is no need to implement IP in user space. Furthermore, efficient and secure demultiplexing should be implemented within the kernel for performance reasons.

The new platform provides two TCP implementations. Traditonal applications can further use the kernel TCP, while new applications based on the ALF/ILP concept can use the user level TCP to take advantage of the ILP support and the ability to configure the TCP user level implementation dependent on the application requirements [10].

The kernel supports the existence of two TCP user level implementations by defining a protocol identifier for TCPK. This architecture is local and only the modified machine will know that two TCP implementations are existing. A remote entity connected to the user level TCP does not know that there are two different TCP implementations.

This transparency leads to an inconsistency for a received packet in the IP layer, because the IP layer cannot decide only using the protocol identifier if the packet has to be directed to the user level TCP. The problem can be solved by testing not only the protocol identifier, but also the local port number. IP code must be modified as shown in Figure 2. IP uses a table containing all port numbers belonging to the user level TCP implementation. If the test succeeds for an incoming packet then the TCPK input function is called, otherwise, the input function called is the one defined by the protocol identifier. A remote entity that wants to address the user level implementation just sends a packet with the protocol identifier set to the classic TCP identifier.

```
                    /* switch out to protocols's input routing */
if(ip->ip.p == IPPROTO_TCP)
      if (tcpk_lookup_table(local port number) == true)
          (*inetsw[ip_protox[IPPROTO_TCPK]].pr_input) (m, ifp);
          goto next;
(*inetsw[ip_protox[ip->ip_p]].pr_input) (m, ifp);
goto next;
```

FIGURE 2. Code modification in the IP layer (bold corresponds to changes).

3 TCP implementation in user space.

3.1 TCPU

3.1.1. Connection establishment

The connection establishment procedure consists of two steps: First, a connection is established between TCPU and TCPK. TCPU must initiate a *socket* system call, specifying the type of communication protocol desired, which is TCPK in the case of the user level implementation. The *socket* system call is followed by a *bind* system call, binding the local address and the local port number in TCPK. This local connection establishment between TCPU and TCPK layer is followed by the classical TCP connection establishment between two remote entities. When both sides agree to establish a connection, each side does a *connect* system call to bind the remote address within TCPK. It facilitates data transmission of subsequent packets by TCPU without giving explicitly the remote address and port number when using a system call. That means that the *write* instead of the *sendto* system call can be used.

3.1.2. Data transfer

After establishing a connection data transfer can start. In the user level implementation, the changes of the input and output functions are minimal.

According to TCP definition, "a sending TCP is allowed to collect data from the sending user and to send that data in segments at its own convenience, until the push function is signalled; then it must send all unsent messages" [1]. The user level implementation follows the same strategy. The application passes data to TCPU through a dedicated interface. Data is stored in a buffer before calling the TCPU output function. This output function sends the stored data as its own convenience. When TCPK receives a datagram packet from TCPU, it passes the packet directly to IP. IP sends packets defined by the underlying network, e.g. the maximum transmission unit (MTU) of Ethernet is 1500 bytes.

On the receiver side, TCPU receives data from the socket interface. According to the TCP protocol definition, TCPU can collect data from several packets before delivering them to the application. This collecting strategy is suitable for a kernel TCP because it avoids to spend too much time passing the user/kernel boundary with small amounts of data. In the case of user level implementation, it becomes more time consuming to buffer packets in the TCPU layer before delivering them to the final application. Therefore, header prediction [11] checks if a packet is in order, and if so it delivers the packet directly to the application, reducing data copying (see Section 4). If the packet doesn't pass the header prediction, intermediate data copying is necessary. This happens if the header has an option field such as for TCP extensions for high performance [12] or transactions [13].

3.1.3. Connection termination

When an application demands to close a connection, the sending side must send all data stored in the buffer before finally closing the connection. After all data is sent, the sender initiates to close the connection. The connection release procedure has not been modified in the user level implementation.

3.2 TCPU application programming interface

The interaction between the application and TCPU is done through a dedicated interface (see Figure 3). TCPU is linked to the application process as a library.

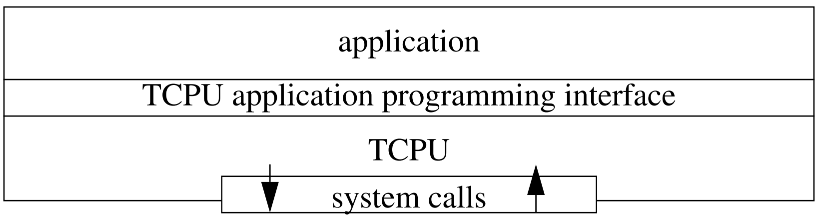

FIGURE 3. A user process using TCPU

The interface between the application and TCPU plays an important role in the user level implementation. The interface consists of the following functions and is very close to the BSD socket interface.

`int socket_tcpu (int domain, type, protocol)`

opens a socket connection to TCPK by a *socket* system call. I also creates and initiates a transmission control block (TCB), which is a data structure storing all necessary information about the connection. Two buffers are allocated: one for sending data and one for receiving data. The function only supports the AF_INET domain and SOCK_STREAM type connections. To select the TCPK protocol below TCPU, TCPK is required as the protocol parameter. otherwise, UDP is used. A socket descriptor is returned finally.

`int bind_tcpu (int socket; struct sockaddr_in *addr; int addrlen)`

binds the local socket address in the TCB and even in the TCPK to the socket by a *bind* system call.

`int accept_tcpu (int socket; sockaddr_in *addr; int *addrlen)`

If there is a connection request, this function opens a connection. It is executed until the connection is established or refused. If a connection is established, it stores the remote address and process number in TCB. It saves also these information in TCPK by a *connect* system call. This function is used by the server side. The function combines the *listen* and the *accept* functions of the BSD socket interface and returns a socket descriptor.

`int connect_tcpu (int socket; sockaddr_in *addr; int addrlen)`

sends a connection request to a known address until the connection is established or refused. If the connection is established, the remote address is stored in the TCB. This information is also stored by a *connect* system call, in TCPK. This function is mainly used by clients.

`int close_tcpu (int socket)`

closes a connection using a three-way handshake. Before returning to the application the function close the connection between the TCPU and the TCPK by a *close* system call.

int send_tcpu (int socket; char *buffer; int length, flags)

By this function the application can pass data to TCPU. The data will be stored in a retransmission buffer. The algorithm of the function is given in Figure 4.

```
send_tcpu(.., length, ...){
...
length = number of bytes passed from application
do
{
    store as much data as possible in retransmission buffer
    if (there was space)
        tcpu_output()
    if (the complete message is sent)
        if (there is an acknowledgment)
            tcpu_input ()
    else
        wait for an acknowledgment
} while (the complete messages is not sent)
return (length)}
```

FIGURE 4. Algorithm for sending a packet.

According to the description one can see that the *send_tcpu* function will be blocked if the retransmission buffer is full. This blocks the called application until there is again buffer space. Otherwise, the passed data would be lost. The function returns the number of bytes sent.

The TF_NODELAY flag must be set to send short messages. Otherwise, it might be that the delivered data are sent after one of the followings *send_tcpu* calls, which can cause an undefinitive long delay.

int recv_tcpu (int socket; char *msg; int length, flags)

By this function the application receives data from TCPU. In our implementation this function decides when to send an acknowledgment. The description in Figure 5 shows that *recv_tcpu* will be blocked when TCPU is not longer receiving packets. This must be the case because the application should not use this function if the connection is closing. Normally, the function returns the number of bytes received. The function returns 0, if no data are received within a certain interval.

```
recv_tcpu(){
...
do
{
     if (packet received)
         if (packet in order)
             deliver_data_to_user
             return(number of bytes received)
         else
             send acknowledgment
     else if (timeout)
         timer management
         send acknowledgment
} while (no data are delivered to the user)}
```

FIGURE 5. Algorithm for receiving a packet.

4 Implementation Details

This section presents implementation details, to overcome various performance problems of the user level environments. These optimizations mainly concern buffer management and timer management.

4.1 Buffer management

With high speed networks, the bottleneck of data transfer has moved from the network to the upper layers. One reason for latency is due to data copying between different layers. For kernel TCP/IP implementations, data copying is performed between two different system levels. For an incoming packet, the first copy is performed from the device driver to mbufs. The second copy is performed from mbufs to the application buffer. Similar copy operations are required for an outgoing packet.

The user level architecture introduces two buffer levels: one at the interface between the application and TCPU (***retransmission buffer***), and another one between TCPU and TCPK (***socket buffer***). The size of the retransmission buffer must be smaller than or equal to the size of the socket buffer. Otherwise, TCPU could deliver packets to TCPK even when the socket buffer is full. In that case, packets would be lost and retransmitted.

The user level implementation intends to provide an environment allowing to reduce data copying operations, e.g. by applying ILP. The optimization of buffer management tries to avoid data copying in the user space. The optimization mainly concerns incoming packets and is based on the fact that TCPU uses header prediction and the system call *readv*. This system call copies the header and data parts into two different memory areas. However, this approach is only successful, if the header size is known in advance. The header is copied into an internal TCPU buffer, while the data is copied to a buffer allocated by the application. That means that, if the header passes the header prediction [11][14], then the data can be directly passed to the final application buffer. The optimization is that there is no intermediate copying performed in TCPU. This optimization is only possible for a successful header prediction. If the header carries an option, or if the packet is out of order, this optimization is not possible, and the received packet must be copied in an internal TCPU buffer for further processing. Note that with this optimization the user-level implementations achieves the same amount of copy operations as the TCP kernel implementation.

For outgoing packets, buffer management is a little bit more difficult to optimize. This is due to the fact that data delivered by the application must be stored in an internal buffer for retransmission purposes. This copy procedure can be integrated with other higher level data manipulations into a single ILP loop [7].

4.2 Timer handling

Packets can be lost due to errors or loss, e.g. caused by network congestions. TCP uses retransmissions managed by a timer to ensure reliable data transfer. TCP needs, therefore, an efficient timer management [15].

Two clocks running with different frequencies are used to implement the TCP timers in a BSD implementation. The clock running with 5 Hz (fast timer) is used for acknowledgment generation. Each timeout causes to test whether an acknowledgment has to be sent or not. The clock running with the lower frequency (2 Hz, slow timer) is used for implementing the the most TCP timers, e.g. for the connection establishment timer, the retransmission timer, and others.

Unfortunately, kernel level primitives cannot be used with the same efficiency at the user-level. Kernel TCP implementations are based on the mentioned clocks (slow and fast timer) expiring periodically. A single context switch allows to control all timers of all TCP connections. This is not possible with user-level implementations since all applications are running in different address spaces. Using kernel level primitives in user-level implementations would cause much more interrupts and context switches with a negative performance impact.

In the user level implementation, the timer management moved to the interface between the application and TCPU. This user level timer manager is based on the system call *select*, which is called by the interface functions in `tcpu_usrreq.c`. This strategy avoids interrupts and consequently time consuming context switches. Because of efficiency and simplicity reasons, the fast timer, which controls sending of acknowlegements, has not been implemented. Acknowledgments could be exchanged after every received packet instead. This, however, decreases the throughput dramatically. The selected solution is that the receiver receives data PDUs until there are no more packets in the input buffer. After that, the receiver sends an acknowledgment to the sender. The receiver also sends an acknowledgment immediately if a packet is out of order. The sender can then retransmit lost or delayed PDUs.

5 Performance evaluation

Throughput measurements have been performed in loopback mode on different platforms using a socket buffer size of 16 Kbytes. The comparison between the BSD kernel implementation and the user level implementation (Figure 6 and Figure 7) should be taken with care, because the user level implementation is not as well tuned as the BSD TCP implementation.

First of all, packets destinated to TCPU are not coalesced into contiguous messages. This requires that TCPU collects and processes an input function for each packet, leading to that the user/kernel boundary is crossed for every received packet. Each packet also leads to a process scheduling. In the same manner the user/kernel boundary is crossed more times when the application wants to send data of a size larger than 1460 bytes over Ethernet. The reason for this is that TCPU cannot send packets larger than the underlying network permits (1500 bytes including 40 bytes of the TCP/IP header without options). For example, a packet of size 7300 bytes will cross the user/kernel boundary five times. An application using the kernel TCP can send all data in one send system call, therefore crossing the boundary only once. Retransmission and acknowledgments will increase the number of crossings of the user/

kernel boundary. Another overhead is due to the intermediate copy for retransmission purposes, which is not required in kernel implementations.

The timer management implementation is also a complex task, which has been realized using the *select* system call. This solution makes the interface between the application and the TCPU more complex and causes extra code execution for each packet. It also brings the client into an idle position, waiting for an acknowledgment.

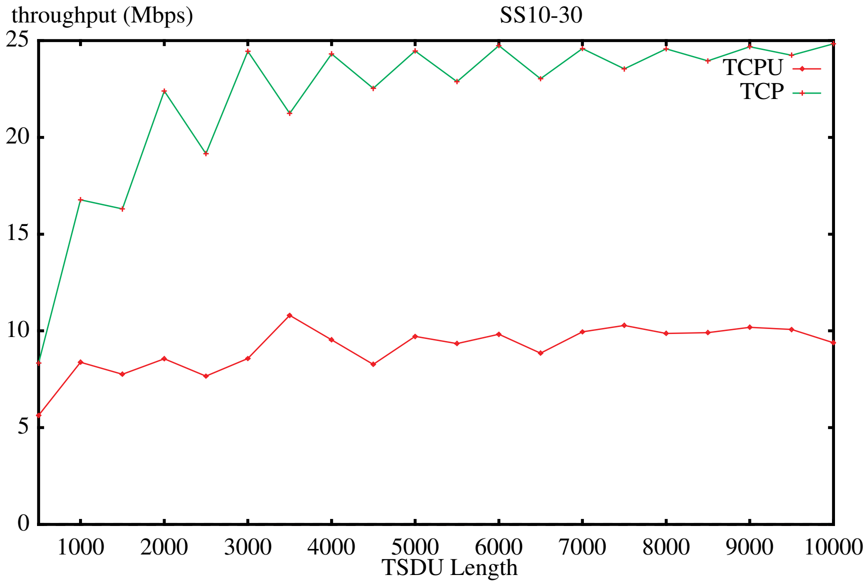

FIGURE 6. Performance comparison between BSD and user level TCP on a SUN SPARCstation10-30 with SUNOS 4.1.3

6 Conclusion

This report describes an implementation of TCP in the user space of UNIX workstations. The performance obtained with the user level implementation are still far from a kernel implementation, which is much more optimized than the user level implementation. However, as shown in [7], advanced implementation techniques such as ILP can help to close the performance gap between user-level and kernel protocol implementations.

Besides this feasibility aspect, the user level implementation was intended to provide a platform to study new implementation techniques and architectures. Kernel implementations are not flexible enough to study new concepts like ILP or ALF. Moreover, giving more control on data transmission to the application requires that end-to-end mechanisms are easily accessible to the application, i.e. that they are located in the same area. Protocol modification, enhancement, and debugging is also easier in user space than in kernel space.

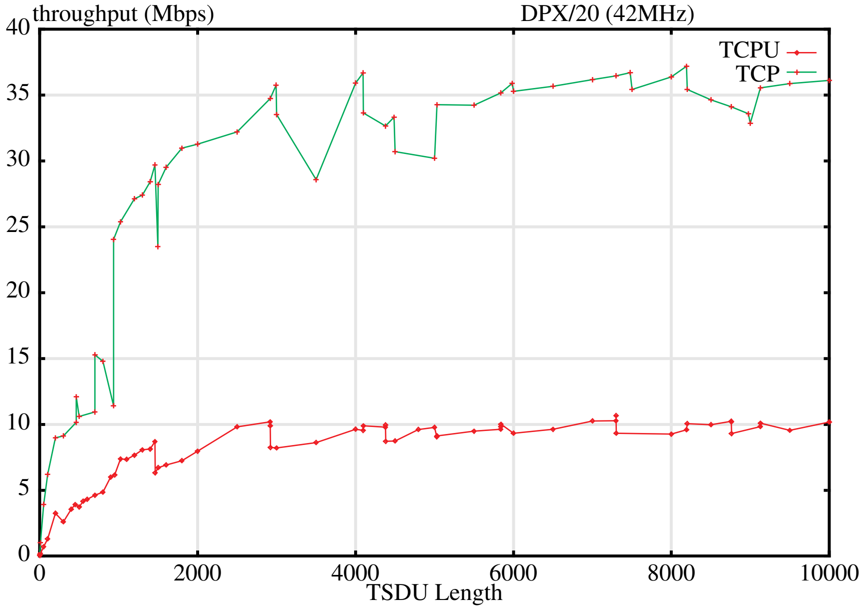

FIGURE 7. Performance comparison between BSD and user level TCP on a DPX/20 (42MHz POWER) with AIX 3.2.5

However, putting the transport protocol into user space also causes various problems. First of all, crossing the user/kernel boundary is required more frequently, introducing overhead on latency and throughput. With TCP in user space, all PDUs (including control PDUs like acknowledgments) must cross this boundary. That causes that the user/kernel boundary is crossed more often, but with smaller packets than with using TCP in kernel space. Extra data copying due to TCP retransmissions also cause overhead. This problem could be solved if data could be copied directly to a buffer, which is accessible by the kernel as well as the application. For receiving data, additional data copying occurs when a packet contains an option field, or if the packet is out of order.

Timer management is an important task in a user level implementation. It is more generally the problem of operating system primitive calls in the user space. UNIX systems make a large use of interrupts. When such an interrupt is done in the user space, processes are switched, and it is not possible to control exactly when the interrupted process will return for execution. Applied to timer management, each timer start, stop, and time-out introduce important overheads due to context switching. There is no optimal solution of this problem in UNIX. The solution proposed in this paper also relies on UNIX system call (through the *select* primitive), but it is not very efficient.

A solution to user level implementations could be to add multi-threading capabilities to the TCP user. Using threads could solve the problem of using just one process for both the application and TCP. To solve these limitations it is required to use multiple threads. One of them could permanently listen for packet arrival in order to react rapidly to asynchronous events. Another one could implement the timer routines. Synchronization between these threads wouldn't be expensive since they share the same global address space. User level implementation, as well as ALF and ILP, would strongly benefit from an enhanced operating system architecture, with system capabilities offered at the user level with minimum overhead.

7 References

[1] J. Postel: Transmission Control Protocol, DARPA Internet Program Protocol Specification, Request For Comments 793, September 1981

[2] D. D. Clark and L. Tennenhouse: Architectural Consideration for a New Generation of Protocols, In Proc. ACM SIGCOMM '90, pp. 200-208, Philadelphia, September 1990.

[3] C. Maeda and B. N. Bershad. Networking performance for microkernels. Technical report, Carnegie Mellon University, March 1992. ftp://mach.cs.cmu.edu/doc/mach3/netperf.ps

[4] C. Papadopoulos and G. M. Parulkar. Experimental Evaluation of SunOs IPC and TCP/IP Protocol Implementation

[5] C. Maeda and B. N. Bershad:. Protocol service decomposition for high-performance networking. Technical report, Carnegie Mellon University,1993 FTPable from mach.cs.cmu.edu in /doc/mach3/.

[6] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska: Implementing Network Protocols at user Level. In proceedings of SIGCOMM '93, September 1993

[7] T. Braun and C. Diot: Protocol Implementation Using Integrated Layer Processing, ACM SIGCOMM '95, August 30 - September 1, 1995, Cambridge, MA.

[8] C. Feldmeier: A Framework Architectural Concepts for High-Speed Communication Systems, IEEE Journal of Selected Areas in Communications, 11(3), April 1993

[9] V. Roca and C. Diot: A high performance Streams-based architecture for communication subsystems, Proceedings of Protocol for High Speed Networks, August 1994

[10] C. Diot, I. Chrisment, and A. Richards: Application Level Framing and Automated Implementation, 6th IFIP International Conference On High Performance Networking, HPN'95, Palma de Mallorca, Spain, September 11-15, 1995

[11] V. Jacobson, 4BSD TCP Header prediction, Computer Communication Review, 20(2):13-15, April 1990.

[12] V. Jacobson, R. Braden, and D. Borman: TCP Extensions for High Performance, Request for Comments 1323, May 1992

[13] R. Braden: T/TCP - TCP Extensions for Transactions Functional Specification, Request for Comments: 1644, July 1994

[14] D. D. Clark, V.Jacobsson, J. Romkey, and H. Salwen: An Analysis of TCP processing Overhead, IEEE Communications, 27(6):23-29, June 1989.

[15] G. Varghese and T. Lauck: Hashed and hierarchical Timuing Wheels: Data structures for the Efficient Implementation of a Timer Facility, Eleventh ACM Symposium on Operating Systems Principles, Austin, Texas, November 1987.

The code of the TCP user-level implementation isavailable on anonymous ftp server zenon.inria.fr/rodeo/tcpuser.

Unité de recherche INRIA Lorraine, technopôle de Nancy-Brabois, 615 rue du jardin botanique, BP 101, 54600 VILLERS-LÈS-NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, domaine de Voluceau, Rocquencourt, BP 105, LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur

Inria, Domaine de Voluceau, Rocquencourt, BP 105 LE CHESNAY Cedex (France)

ISSN 0249-6399