Vol. 40 No. 1 Transactions of Information Processing Society of Japan Jan. 1999

Regular Paper

Network Subsystem Architecture Alternatives for Distributed Real-time System

Takuro Kitayama,[†1] Tatsuo Nakajima,[†2] Shuichi Oikawa[†3] and Hideyuki Tokuda[†4]

In traditional operating systems such as UNIX, network protocol stacks reside in the kernel, and it processes the protocols in interrupt handlers. This strategy is suitable for time-sharing systems because of the high performance. On the other hand, this strategy cannot be applied to real-time distributed systems since packets are processed in a non-preemptable fashion. Thus, unbounded priority inversions that may violate timing constraints of packets occur. We have developed two user-level network subsystems for distributed real-time systems and show our network subsystems have many advantages over traditional network systems. This paper focuses on the effectiveness of user-level protocol stacks for distributed real-time systems and discusses how the protocol stacks affect the system structures. We present the implementation and performance evaluation results of our two different real-time protocol stacks, which are implemented on Real-Time Mach. The results show that respective architectures have different characteristics, and the selection of the architectures depends on the requirements of real-time applications. The paper can give a guideline that enables programmers to select suitable network subsystems for their application.

1. Introduction

Many user-level network subsystem architecture researches focus on their performance and flexibility. They are trying to obtain network level performance to applications over high speed networks such as ATM and giga-bit networks. Real-time communication, on the other hand, is becoming more important in many domains such as distributed multimedia systems, factory automation, and robotics. These applications require rigid timing constraints as well as high performance. However, few researchers addressed the advantage of user-level real-time network subsystem architectures.

Network subsystem architectures have strong impact over the tradeoff between their performance and timeliness of protocol stack execution. The past works[5),8),10),25)] have mainly focused on the performance but not the timeliness, and they have never been compared and discussed intensively. For example, although Maeda, *et al.*[14)] demonstrated the performance improvement of a network subsystem by introducing a user-level protocol library, they did not discuss its effect towards real-time systems. While Lee, *et al.*[13)] made the user-level protocol library more suitable for real-time systems by using real-time scheduling and synchronization, its performance improvement was not clear.

The objective of this paper is to show the impact of network subsystem architectures for real-time systems and discuss how the protocol stack subsystem architectures affect the system behavior. Our focuses are not only the performance, but also many aspects such as flexibility, security, and resource allocation policies.

We have implemented two types of network subsystem architectures to provide predictable communication for real-time systems on Real-Time Mach (RT-Mach) microkernel[28)]. One is Network Protocol Server (NPS)[18)], which implements protocol stacks in a user-level server, and the other is the combination of an extended packet filter implemented in the microkernel and a user-level socket library. Both use prioritized-IP (PIP), which can transfer priority information across a network by using the IP option field in an IP packet. Their extensive evaluation clarifies the characteristics of those architectures in terms of the performance and timeliness, and provides a guideline for programmers to choose the most suitable architectures for their applications.

In the remainder of this paper, we summarize the past works in Section 2. Then, we present

[†1] Keio Research Institute at SFC, Keio University
[†2] Japan Advanced Institute of Science and Technology
[†3] School of Computer Science, Carnegie Mellon University
[†4] Faculty of Environmental Information, Keio University

user-level network protocol stack models in Section 3. In Section 4, we describe the implementation of real-time network subsystems. The performance evaluation results of the network subsystems are shown in Section 5. We discuss the effect of network subsystems architecture in Section 6 and conclude in Section 7.

2. Past Work

Many modern operating systems have network subsystems implemented in user-space. It provides the flexibility, which enables to extend, modify, customize, and debug the protocol stacks easily and effectively. Coincidentally, the user-level network subsystems can be suitable for real-time environments. It provides high preemptability. Execution of protocol stack by lower priority thread can be preempted by other higher priority threads.

In this section, we summarize previous researches of user-level network subsystem architectures and user-level real-time network subsystem.

2.1 Protocol Stack in User-level Server

Mach3.0 is a microkernel-based operating system. It provides network protocol stacks in a server, which runs in user-space. UX[8] and 4.4 BSD Lite Server[9] are the OS personality servers, which provide network protocol stacks as well as other UNIX facilities. *x*-Kernel[10] is a protocol stack server. It provides a framework to implement network protocol stacks in user-space.

The dominant overheads of the user-level network server architecture are local IPC cost between servers and clients and synchronization cost. To eliminate the overhead of user-level protocol servers, extensible kernels, such as SPIN[1] and VINO[22], provide network protocol stacks in kernel-space without losing flexibility. The protocol stacks in these systems are dynamically loaded into and executes in kernel-space. The drawback of this architecture is the technique used to realize safeness of the system, such as using a type safe language and software fault isolation, to protect the system from the untrusted user code.

2.2 Protocol Stack in User-level Library

User-level protocol stack libraries[2],[7],[14],[25] reduce the overhead of IPC costs. It is linked with application programs, and executes the protocol stack in the same address space as the application programs.

Since the protocol stacks execute multiple user-space, mechanisms to demultiplex incoming packets are necessary. Packet filter[15],[17] is the mechanism to demultiplex packets from networks to appropriate user's address space.

Previous researches proposed hardware demultiplexing techniques such as U-Net[29] and Application Device Channels (ADC)[6]. In these systems, the network interface dispatches incoming packets, and application programs can directly access the network interfaces without operating system interactions.

2.3 Real-time Network Subsystems

Many real-time protocols[3],[24],[30] have been developed to support distributed real-time systems. These protocols, especially in capacity-based protocols, require complicated control message handling such as error handling and resource reservation. However, few researchers address the advantage of user-level network subsystem architectures in the context of real-time, although the architectures provide many benefits for implementing protocols effectively.

Druschel, *et al.*[5] proposed the Lazy Receiver Processing (LRP) technique, which delays protocol processing until the data is requested by an application when the network is under high load conditions. They expect that LRP is applicable to real-time networking when combined with real-time thread scheduling. They implemented the protocol stack in kernel-space, which lacks flexibility and preemptability. Tokuda, *et al.*[27] discussed the effect of preemptability of protocol stack execution. They increased preemptability and reduced unbounded priority inversion in ARTS kernel[26]. Lee, *et al.*[13] addressed the end-to-end predictability in distributed real-time and multimedia applications protocol processing, and provided preemptable protocol stack structure with the extension of the socket library[14], and coordinated with CPU reservation scheme on RT-Mach. However, they did not discuss the flexibility nor the performance penalty of real-time support. Network Protocol Server (NPS)[18] provides protocol stacks in a user-level server. It also provides a communication mechanism for distributed real-time environment by server architecture, and they discussed the advantage of implementing protocol stack in user-space for distributed real-time systems.

These real-time network subsystem researches are focused on their advantages such as preemptability and performance. However, precise comparison of user-level real-time network subsystem architectures from many aspects has not been made.

3. User-level Protocol Stack Model

In this section, we present these two user-level network protocol stack models and discuss their problems.

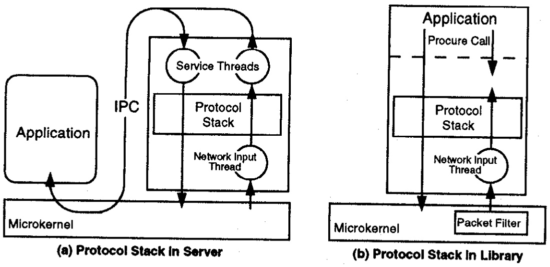
3.1 Protocol Stack in User-level Server

Figure 1 (a) illustrates the structure of protocol stack execution in a user-level server. There are two types of threads, service threads and a network input thread in typical servers on the microkernel. For sending packets, application threads send packets via IPC to service threads. After the data is passed, the service thread executes the protocol stack in the server, then send the packet to the microkernel. On receiving side, service threads receive requests from application, and wait until the receive buffer fills up. The network input thread, on the other hand, receives incoming packets from the microkernel, then it executes the protocol stack, and unblocks the waiting service threads. Then, the service threads send the data to application threads via IPC.

The cost of context switching and data passing is a main source of the protocol stack execution overhead in the server. Another source of overhead is the locking mechanism. Protocol servers use spin-lock mechanism for lock and unlock, while the in-kernel protocol stack uses *spl* to change processor priority level.

3.2 Protocol Stack in User-level Library

The original socket library, which implements protocol stacks in user-space, has been developed by Maeda, *et al.*[14] for high performance and flexibility. Figure 1 (b) illustrates the structure. In contrast to the execution of protocol stacks in a server, there is no service thread. When sending packets, application threads directly execute the protocol stack and send packets to the microkernel without communicating with other servers in the local machine or switching to other threads. For receiving incoming packets, on the other hand, a packet filter implemented in kernel space dispatches packets to the network input thread by sending IPC messages. After receiving message, the network input thread executes protocol stack in the user-level library. An application thread then receives the packets.



(a) Protocol Stack in Server (b) Protocol Stack in Library

Fig. 1 User-level protocol stack model.

This model eliminates the additional overhead of local communication and context switching, which can be found in a user-level server for sending and receiving packets. Unlike the server implementation of protocol stacks, send and receive calls are just procedure calls, not local IPC. So the cost of context switching and data passing between server and application is eliminated with this architecture.

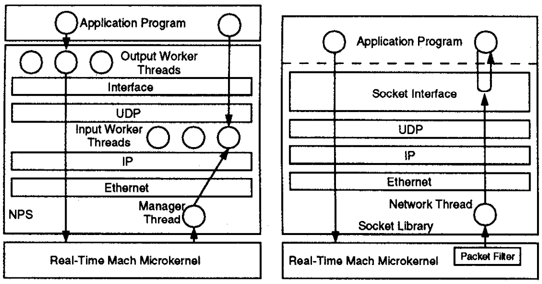
4. Real-time Protocol Stack Implementation

We have built two real-time network subsystems on RT-Mach. One is Network Protocol Server (NPS), which implements protocol stacks in a user-level server. The other implementation is an extension of the packet filter and user-level protocol library. Both protocol stacks support UDP/IP. The main reason, we choose UDP rather than TCP, is that TCP has much richer mechanisms, such as window control and retransmission, for reliable communication. Real-time system designers need to predict the worst case execution time, and these mechanisms make the behavior of real-time system unpredictable, since application programs cannot control the mechanisms. Furthermore, UDP is a light protocol stack compared to TCP. In this section, we describe these implementations.

4.1 Real-time Communication Server

Figure 2 shows the structure of NPS. It consists of two sets of worker threads, input worker threads and output worker threads. The output worker threads are the service threads for sending packets. They process requests from application and execute all layer of the protocol stack. The output worker threads inherit the priority of the applications, which sent the requests.

Unlike the network input thread, described in the previous section, the execution of protocol stacks for receiving packets is done by input



(a) Network Protocol Server Structure **(b) Real-Time Socket Library Structure**

Fig. 2 User-level protocol stack structure.

worker threads and a manager thread in NPS. The manager thread receives Ethernet packets from the microkernel and executes IP layer, then dispatches to an input worker thread for execution of UDP layer. Since Ethernet packets do not support the notion of priory, all packets should be scheduled at the same priority in IP layer. The manager thread is running at the highest priority in the NPS system for reducing unbounded priority inversion. The priorities of the input worker threads are assigned by the manager thread according to the priority, which is contained in incoming packets.

To transfer the priority information to the input worker threads across a network, Prioritized-IP (PIP)[18] was introduced. PIP is an extension of the IP protocol and provides the notion of priority. For compatibility with normal IP, PIP uses an IP option field to conation a priority, so that PIP packet can be received by normal IP software. As PIP options, there are three elements, priority, period, and deadline, since RT-Mach supports various scheduling policies, and each machine can select a suitable policy for the application. Threads in RT-Mach must have the priority, period, and deadline for their attributes to be scheduled with any scheduling policies.

In real-time environments, the spin-lock mechanism causes priority inversion problems, where a lower priority thread is executing a critical region when higher priority thread tries to obtain the lock. In order to solve this problem, *priority inheritance* scheme was developed[23]. A locking mechanism supporting this scheme should be used for real-time protocol stack. However, this mechanism needs to maintain thread queues of waiting threads and the thread executing the critical section needs to inherit the priority when a higher priority thread tries to acquire the same lock. Therefore, the cost of this locking mechanism is high.

4.2 Real-time Communication Library

Since the original socket library was designed to use C-threads package[4], Lee, *et al.*[13] extended the socket library to use real-time thread and real-time synchronization mechanisms provided by the RT-Mach microkernel.

We have extended the socket library for real-time communications[11]. This extension makes better use of sender priority transferred across the network using PIP, which is explained in the previous subsection. The packet filter in the microkernel was extended to check PIP options. The extended packet filter checks whether incoming packets have PIP option or not. If a packet has PIP option, the packet filter sends a RT-IPC[12] message containing the priority to the network thread when dispatching the packet. Then, the network thread can start executing the protocol stack at the priority of the request after receiving the message. After the execution of the protocol stack, the network thread sends a RT-IPC message to the receiver thread of an application program. By using RT-IPC, a receiver thread automatically inherits the priority of the sender thread.

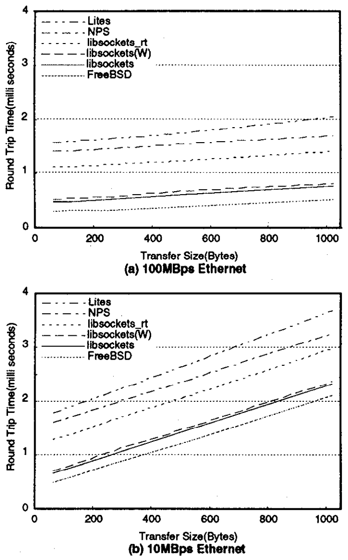
A problem is the locking mechanism, since multiple threads may use the same protocol stack library, and the protocol stack library itself is multi-threaded. The lock should be supported the priority inheritance protocol.

5. Performance Evaluation

We evaluated the performance of two different implementations of real-time communication subsystems. We used two IBM PC/AT compatible machines for the measurements. Each machine has 166 MHz Intel Pentium Processor, 32 Mega-bytes of memory, and DEC DE500 Ethernet interface card for both 10 Mbps and 100 Mbps Ethernet. We used the read time stamp count (RDTSC) instruction on the Pentium processor. The machines were disconnected from other machines on the network, so that there was no disturbance traffic. In this section, we will show the evaluation results of the network subsystems.

5.1 Basic Performance

We measured the round trip cost between two hosts with 100 Mbps and 10 Mbps Ethernet. The measurements were repeated 1000 times and the averages were taken. To transferred data between application programs running on two machines, we used shared buffer interface for the measurement of NPS, and *sendto()* and



(a) 100MBps Ethernet

(b) 10MBps Ethernet

Fig. 3 Round trip cost.

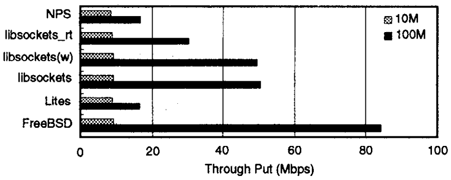


Fig. 4 Throughput.

recvfrom() system calls for the others. The result of the cost using 100 Mbps is shown in **Fig. 3** (a) and the result for 10 Mbps is shown in Fig. 3 (b).

Figure 4 illustrates the throughput on both 100 Mbps and 10 Mbps Ethernet. We used *ttcp* benchmark program for the measurement. The packet size is 1024 bytes which is the default value of the benchmark.

We evaluated real-time and non-real-time network subsystems for comparisons, and they are explained in **Table 1**. The difference between libsockets and libsockets(W) is that the network input thread of libsockets(W) is *wired* to a dedicated kernel thread while that of libsockets is not. This means that the scheduling operation between the network input thread and a service thread involve two kernel threads in libsockets(W).

In Fig. 3, the difference between 100 Mbps and 10 Mbps Ethernet is amplified by increasing the transfer data size. This is caused by the physical network performance. The overhead in Lites comes from the cost of context switching between Lites and the test program and the synchronization cost in Lites. The difference between libsockets and libsockets(W) is the difference between the cost of kernel threads and user threads context switching which is approximately 40 μs.

Table 1 Evaluated network subsystems.

Lites	4.4 BSD Lite Server
NPS	Network Protocol Server (NPS)
libsockets_rt	Real-time socket library
libsockets	Original socket library without threads *wired*
libsockets(W)	Original socket library with threads *wired*
FreeBSD	FreeBSD 2.2.1R

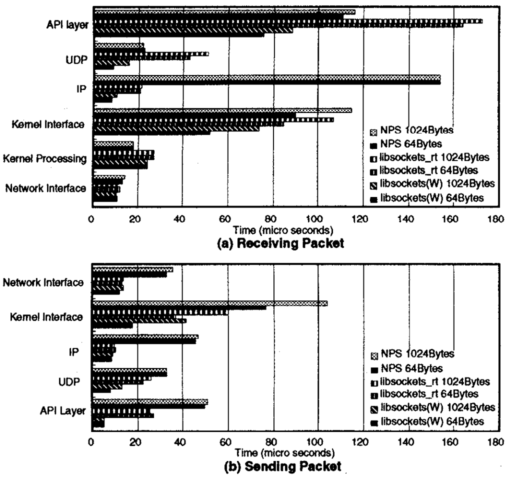
In Fig. 4, throughput of all protocol stacks is approximately 9 Mbps over 10 Mbps Ethernet, since the network capacity is saturated. Differences can be found over 100 Mbps Ethernet, and it reflects the result of the round trip cost shown in Fig. 3.

The round trip time of FreeBSD is approximately 175 μs faster than that of libsockets, 850 μs faster than libsockets_rt, and 1150 μs faster then NPS. These differences are constant and do not depend on the network speed or transfer size. This means that the overheads depend only on architectural difference of the network subsystems, and they affect the results of the throughput shown in Fig. 4.

The round trip of NPS is approximately 900 μs slower than libsockets, and libsockets_rt is about 600 μs slower. The throughput of NPS is 35 Mbps lower, and libsockets_rt is 20 Mbps lower than that of libsockets over 100 Mbps Ethernet. This is because that NPS and libsockets_rt use Real-Time Thread (*RT-Thread*), which needs kernel interaction when switching to another thread, while libsockets uses *C-Thread*, a user-level thread package. Another reason is the cost of synchronization mechanism. libsockets uses spin-lock mechanism, FreeBSD uses *spl* to change processor priority level, and NPS and libsockets_rt use real-time synchronization mechanism. These thread and synchronization costs cause the overhead of NPS and libsockets_rt, however, these real-time support facilities are important to solve the unbounded priority inversion problems. We analyzed the overheads in detail in the next subsection.

5.2 Micro Analysis

We measure the microscopic cost in each layer of the network subsystems to analyze the overhead of real-time support. **Figure 5** depicts the



(a) Receiving Packet

(b) Sending Packet

Fig. 5 Micro analysis.

result of the analysis. We measured the cost of sending and receiving 64 bytes and 1024 bytes of data over 100 Mbps Ethernet to see the effect of difference in transfer size.

There is an effect of the transfer data size when the data is moved between the kernel-space and user-space at kernel interface in Fig. 5. The overhead of data transfer is approximately 20 μs and this is due to the copying of the data. The execution of IP layer in NPS is approximately 140 μs slower than that of the others when receiving packets. In this figure, the cost of switching the manager thread to an input worker thread in NPS is included in IP layer. Switching the manager thread to a worker thread is expensive, since it needs to find a proper priority worker and priority hand-off to take place. We can eliminate this cost by using the packet filter to dispatch packets from microkernel to input worker threads. There is approximately 20 μs of overhead in the network interface when sending packets. The reason is that this version of NPS uses different kernel interface when sending packets to the kernel. This kernel interface involves additional overhead at the network interface layer execution.

The overheads of the real-time network subsystems can be found in variety of the layers in user-space execution. The reason is not only the cost of context switching or priority hand-off, because there is no such overhead necessary in the execution of the UDP and IP for sending packet and many other layers. Costs of the real-time synchronization are considered the main source of the overheads in each layer. In the next subsection, we will describe the effect of synchronization in detail.

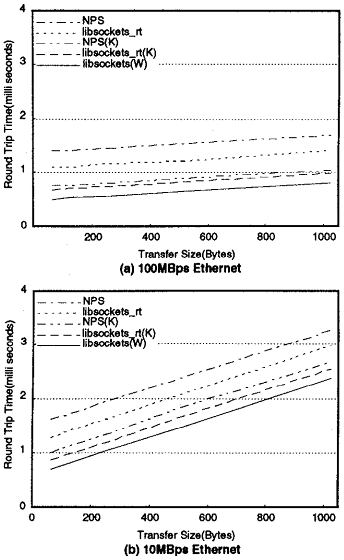
5.3 Performance with Kernelized Monitor

From the detailed evaluation of the real-time network subsystems, we found that the dominant cause of overhead in real-time network is the costs of real-time synchronization. Thus, we counted the number of synchronization calls to send and receive a packet. NPS acquires locks 9 times when sending a packet, and 13 times when receiving one. The protocol library acquires locks 4 times when sending one packet, 15 times when receiving one.

The cost of real-time synchronization is 15 μs for locking and unlocking where the cost of synchronization in C-thread is 0.8 μs. By using real-time synchronization, there are additional overheads of 312.4 μs in NPS and 269.8 μs in libsockets_rt for sending and receiving data[☆]. We measured the duration of each critical region and found that 80% of the critical regions are shorter than 3 μs, 17% are between 3 μs and 13 μs, and the rest of the critical regions are shorter than 30 μs. There is no relation between the duration of the critical regions and the packet size.

We implemented another version of NPS, which uses *kernelized monitor* instead of calling real-time synchronization. The kernelized monitor is a mechanism to protect critical regions. It disables and enables any context switching when entering and exiting a critical region while other real-time synchronizations allow switching to higher priority threads. Although, the kernelized monitor does not allow any preemption during a critical region, the critical regions in protocol stacks are short enough to be protected by the kernelized monitor. Most of the critical regions in the protocol stack benefit from kernelized monitor, since the duration of over 97% of the critical regions are shorter than the cost of real-time synchronization. The cost of kernelized monitor is 1.5 μs, since it just sets and resets a flag to disable and enable the context switch, while real-time synchronization needs to maintain a thread queue of waiting threads for the priority inheritance protocol. By replacing real-time synchronization with the kernelized monitor, additional overheads can expect to be reduced by 594 μs in NPS and 513 μs in libsockets_rt.

[☆] (9 + 13) [times] × 14.2 [μs] in NPS, (4 + 15) [times] × 14.2 [μs] in libsockets_rt.



(a) 100MBps Ethernet

(b) 10MBps Ethernet

Fig. 6 Round trip cost with kernelized monitor.

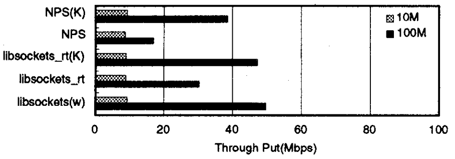
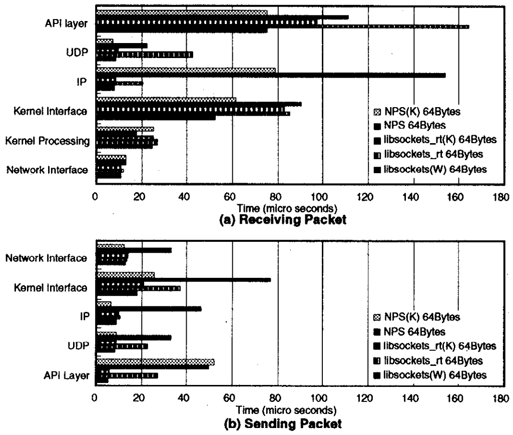


Fig. 7 Throughput with kernelized monitor.

We evaluated NPS, which uses kernelized monitor. We also modified libsockets_rt to use kernelized monitor. **Figures 6** (a) and (b) show the round trip time, **Fig. 7** shows the throughput, and the micro analysis is depicted in **Fig. 8**. In the figures, NPS(K) represents NPS with the kernelized monitor, and libsockets_rt(K) indicates user-level socket library using kernelized monitor.

In Fig. 6, NPS using the kernelized monitor is approximately 650 μs faster than that of using real-time synchronization. Replacing real-time synchronization, which occurs 44 times[☆] in one round trip with the same number of kernelized monitor, creates this performance improvement. The performance of libsockets_rt was also improved by approximately 450 μs using the kernelized monitor. In Fig. 7, it is shown that the kernelized monitor improves the throughput from 17 Mbps to 39 Mbps in NPS and 30 Mbps to 47 Mbps in libsockets_rt.

Performance difference still exits when sending data to NPS at API Layer in Fig. 8 (a). This overhead is approximately 40 μs. This is caused by local communication between NPS and the application, and context switching. When receiving data from NPS, in contrast, NPS is faster than the others, since NPS is optimized using continuation technique when sending data to the applications. This optimization counterbalances the cost of switching from NPS to applications.



(a) Receiving Packet

(b) Sending Packet

Fig. 8 Micro analysis with kernelized monitor.

The difference of round trip time between libsockets and NPS is approximately 300 μs using the kernelized monitor while 975 μs using real-time synchronization. And the difference between libsockets and libsockets_rt is reduced from 615 μs to 170 μs. The cost of real-time support is reduced by over 70% using the kernelized monitor.

5.4 Effect of Real-time Support

To show the effect of real-time support, we created a benchmark program. **Figure 9** illustrates the benchmark. The benchmark consists of two sets of client-server tasks. The first task set is a real-time task set, which we actually want to bound the behavior. The real-time client sends a request to the server, and the server consumes 10 ms of computation time, then sends back a reply to the client. Then, the client computes 5 ms. This activity is periodically executed every 50 ms. The other task sets are disturbance jobs of the real-time activity. A disturbance client sends a request, and the server consumes 25 ms of CPU time, then sends a reply back to the client. Then the client computes 25 ms. It is executed cyclically without sleeping. A disturbance job starts approximately 1 second after the start time of real-time activity. The number of the disturbance job is increased every 1 second to increase the system

[☆] (9 + 13) [times] × 2

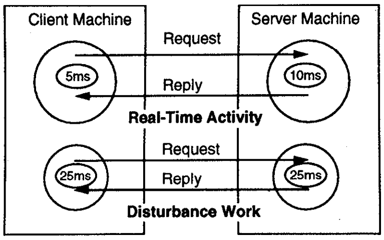
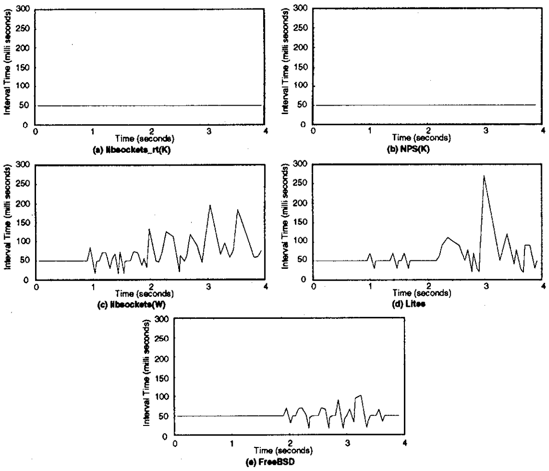


Fig. 9 Benchmark task set.



(a) libsockets_rt(K)
(b) NPS(K)
(c) libsockets(W)
(d) Lites
(e) FreeBSD

Fig. 10 Benchmark result.

load on both of the server and client machines.

For the measurement of real-time periodic activity of Lites, libsockets_rt, and NPS, we used periodic threads provided by RT-Mach. Since the original socket library libsockets is not compatible with periodic threads, we used high resolution timer[21], also provided by RT-Mach, to create periodic activities. Both client and server programs of the real-time activity have the highest priority, and the disturbance jobs have lower priority on RT-Mach. The scheduling policy was set to fixed priority. For the measurement of FreeBSD, we used *setitimer()* system call for periodic alarm signal. The priority of the real-time activity is bumped up by *setpriority()* system call.

We measured the interval time of the real-time activity and the results are shown in **Fig. 10**. In the figures, there is no jitter in the beginning in all of the cases, since there is no disturbance work running. In libsockets and Lites, jitter begins after the disturbance tasks start. The threads, which are executing protocol stack in Lites and the network thread in the socket library, have the same priority as the disturbance threads. Therefore, they compete for CPU resulting to jitters in the figures. The protocol stack of FreeBSD is kernelized and it can preempt the real-time activity when requests of the disturbance tasks come from the network, and the jitter occurs. The result shows that we can manage the interval time of the real-time activity in both libsockets_rt and NPS.

5.5 Evaluation Summary

RT-Mach provides many protocol stacks implemented with different architectures. The performance of Lites is the worst in our evaluation. The round trip time of Lites is three times longer than that of libsockets, and its throughput is 30% lower. However, if applications use many *select* system call, or *open* and *close* sockets frequently, Lites is better than libsockets *socket* are managed in Lites. libsockets needs to communicate with Lites when application use *select*, *close*, and *select* system calls for sockets. For example, the cost of creating and closing a socket is 2.2 ms in Lites, while libsockets takes 4.4 ms. For real-time system, RT-Mach provides libsockets_rt and NPS. The performance of libsockets_rt is better than that of NPS. The round trip time of libsockets_rt is 15% better than that of NPS, and the throughput of libsockets_rt is 20% better. However, the performance of network subsystems is only one criteria for selecting network subsystems. In the next section, we discuss the tradeoff between the server and library network subsystems form many aspects.

6. Discussion

In this section, we discuss the tradeoff of protocol stacks implemented with different architectures for real-time environment. We focus not only on performance, but also other aspects such as flexibility, resource management, and security.

6.1 Performance Tradeoff

Implementing a protocol stack in a user-level server is believed to be slower than implementing it in user-level library. The communication cost between a protocol server and application program actually exists, but the difference is very small. From our evaluation results, the overhead of NPS is approximately 10% higher than that of libsockets_rt. NPS and libsockets_rt currently use RT-Thread, which is provided by the RT-Mach microkernel. The context switching cost of RT-Thread is high compared with that of threads implemented in user-space. RTC-Thread[20] incorporates the performance of user-level threads and the functionalities of RT-Thread. By using RTC-Thread, the performance of both NPS and libsockets_rt can be improved. Moreover, by implementing a locking mechanism with RTC-Thread, it can reduce the lock and unlock costs without affecting other tasks in the system. This mechanism just disables the context switch to the threads within the same task, while the kernelized monitor disables switching to any threads in the machine. This locking mechanism may also reduce the security risk.

Further performance improvement idea for NPS is eliminating the cost of switching from the manager thread to an input work thread. We estimate this cost is approximately 70 μs from Fig. 8. It can be eliminated using the extended packet filter described in Section 4. The packet filter in the microkernel sends message directly to the worker thread with priority handoff. Then the cost of the manager thread can be eliminated, and the latency of NPS and libsockets_rt may become equivalent. The communication through put is almost proportional to the latency from our evaluation result. If the round trip time becomes short, then through put will be increased. Even in current implementation, the through put is the same, as long as we are using 10 Mbps Ethernet, which is the world's most popular local area network media. From our experiments, the performance may not be a major issue any more for distributed real-time system with proper optimized protocol stacks.

6.2 Server vs. Library for Real-time Network

Although the server and library architecture does not affect the performance, there are some other aspects such as flexibility, security, and resource allocation, which need to be discussed for building distributed real-time systems.

Both the server and library architecture provide flexible protocol stacks compared to in-kernel protocol stacks. However, if there are applications with various characteristics running on a machine, the library architecture provides better flexibility. Multiple protocols for special purpose may co-exist on a machine and provide different services by using library architecture. These protocols can be easily added, extended, adapted, and specially optimized for application demands without affecting other program running on the same machine. In addition, rapid prototyping and debugging favor the self-contained protocol stack library.

To coordinate with resource reservation such as CPU-reserves[16] and VM-reserves[19], the library architecture is more suitable than the server architecture, since the resources for the execution of protocol stacks are associated with threads or tasks. Using the server architecture, such resources are charged to the protocol server, but not to threads or tasks which use the protocol server. If the server architecture is used with the reservation schemes, the system designer must estimate all traffic going through the protocol server. Otherwise, higher priority threads of which execution time should be bounded may suffer from the unpredictable protocol execution of unestimated lower priority traffic. All such resources, however, are charged to the thread itself, by using protocol stack in library. The activities of lower priority threads may not affect the resources of higher priority threads.

Security needs to be considered for network subsystems if there are untrusted applications in systems. Thekkath, *et al.*[25] addressed the security issue of protocol stack libraries. They described two aspects to protection. First, only entities that are authorized to communicate with each other should be able to communicate. Second, entities should not be able to impersonate others. They solved the issues by using a trusted registry server and header matching of incoming and outgoing packets. In real-time systems, however, these protection schemes are not sufficient. For example, the abuse of the kernelized monitor may cause serious system failure. Any thread can disable preemption. Then, all threads in the system, except the thread in the kernelized monitor, are suspended. The kernelized monitor, however, needs to be exported to users, which use the library architecture for better performance. If we consider security, the server architecture is superior to the library architecture.

7. Conclusion

In this paper, we first present the user-level network stack models, and described two real-time network subsystems, which are implemented on RT-Mach. One is NPS which implements protocol stacks as a user-level server. The other is real-time socket library using the in-kernel packet filter. We evaluated the real-time network subsystems, and the results showed that there are many overheads to support real-time communication. From our detailed analysis of the network subsystems, we found that the dominant overheads are the synchronization cost. Using kernelized monitor improves the performance of the network subsystem. The performance of the library architecture is better than that of server architecture, however, the difference is very close even though their architectures are different. In addition, the both subsystems can eliminate unbounded priority inversion in real-time system.

From our evaluations and discussions, we concluded that the selection of the network subsystem architecture depends on application requirements. To build a distributed real-time system efficiently, consideration should be made for flexibility, resource management, security requirement, and implementation of applications. The library architecture is flexible than the server architecture, and it is easy to optimize to obtain better performance and coordinate with local resource reservation scheme. The server architecture provides better security over the library architecture, and easy to control network resource in user-space. Real-time system designer should select the architecture, which is suitable for the characteristics of their applications.

References

1) Bershad, B., Savage, S., Pardyak, P., Sirer, E., Fiuczynski, M., Becker, D., Chambers, C. and Eggers, S.: Extensibility, Safety and Performance in the SPIN Operating System, *Proc. 15th Symposium on Operating Systems Principles* (1995).

2) Braun, T. and Diot, C.: Protocol Implementation Using Integrated layer Processing, *Proc. SIGCOMM '96 Symposium* (1996).

3) Cheriton, D.R.: VMTP: Versatile Message Transaction Protocol Specification, RFC 1045, Stanford University (1988).

4) Cooper, E.C. and Drabes, R.P.: C threads, Technical Report CMU-CS-88-154, Carnegie Mellon University (1987).

5) Druschel, P. and Banga, G.: Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems, *Proc. USENIX 2nd Symposium on OS Design and Implementation (OSDI '96)* (1996).

6) Druschel, P., Peterson, L.L. and Davie, B.S.: Experiences with a High-Speed Network Adaptor: A Software Perspective, *Proc. SIGCOMM '94 Symposium* (1994).

7) Edwards, A. and Muir, S.: Experiences implementing a high performance TCP in user-space, *Proc.SIGCOMM '95 Symposium* (1995).

8) Golub, D., Dean, R., Forin, A. and Rashid, R.: Unix as an Application Program, *Proc.Summer USENIX Conference* (1990).

9) Helander, J.: Unix under Mach, The Lites Server, Master's Thesis, Helsinki University of Technology (1994).

10) Hutchinson, N.C. and Peterson, L.L.: The *x*-Kernel: An Architecture for Implementing Network Protocols, *IEEE Trans. Softw. Eng.*, Vol.17, No.1, pp.64–76 (1991).

11) Kitayama, T., Miyoshi, A., Saito, T. and Tokuda, H.: Real-Time Communication in Distributed Environment – Real-Time Packet Filter Approach, *Proc. 4th International Workshop on Real-Time Computing Systems and Applications* (1997).

12) Kitayama, T., Nakajima, T. and Tokuda, H.: RT-IPC: An IPC Extension for Real-Time Mach, *Proc. USENIX Symposium on Microkernel and Other Kernel Architectures* (1993).

13) Lee, C., Yoshida, K., Mercer, C. and Rajkumar, R.: Predictable Communication Protocol Processing in Real-Time Mach, *Proc. IEEE Real-time Technology and Applications Symposium* (1996).

14) Maeda, C. and Bershad, B.N.: Protocol Service Decomposition for High-Performance Networking, *Proc. 14th Symposium on Operating Systems Principles* (1993).

15) McCanne, S. and Jacobson, V.: The BSD Packet Filter: A New Architecture for User-level Packet Capture, *Proc. 1993 Winter USENIX Conference* (1993).

16) Mercer, C.W., Savage, S. and Tokuda, H.: Processor Capacity Reserves: An Abstraction for Managing Processor Usage, *Proc. Fourth Workshop on Workstation Operating Systems (WWOS-IV)* (1993).

17) Mogul, J., Rashid, R. and Accetta, M.: The Packet Filter: An Efficient Mechanism for User-Level Network Code, *Proc. 11th Symposium on Operating Systems Principles* (1987).

18) Nakajima, T. and Tokuda, H.: User-level Real-Time Network System on Real-Time Mach, *Proc. 4th International Workshop on Parallel and Distributed Real-Time System* (1996).

19) Nakajima, T. and Tezuka, H.: Virtual Memory Management for Interactive Continuous Media Applications, *Proc. IEEE International Conference on Multimedia Computing and Systems (ICMCS'97)* (1997).

20) Oikawa, S. and Tokuda, H.: Efficient Timing Management for User-Level Real-Time Threads, *Proc. 1995 IEEE Real-Time Technology and Applications Symposium* (1995).

21) Savage, S. and Tokuda, H.: RT-Mach Timers: Exporting Time to the User, *Proc. USENIX 3rd Mach Symposium* (1993).

22) Seltzer, M.I., Endo, Y., Small, C. and Smith, K.A.: Dealing With Disaster: Surviving Misbehaved Kernel Extensions, *Proc. USENIX 2nd Symposium on OS Design and Implementation* (*OSDI '96*) (1996).

23) Sha, L., Rajkumar, R. and Lehoczky, J.P.: Priority Inheritance Protocols: An Approach to Real-Time Synchronization, Technical Report CMU-CS-87-181, Carnegie Mellon University (1987).

24) Strayer, W.T., Dempsey, B.J. and Weaver, A.C.: *XTP: The Xpress Transfer Protocol*, Addison Wesley (1993).

25) Thekkath, C.A., Nguyen, T.D., Moy, E. and Lazowska, E.D.: Implementing Network Protocols at User Level, *Proc. SIGCOMM '93 Symposium* (1993).

26) Tokuda, H. and Mercer, C.W.: ARTS: A distributed real-time kernel, *ACM Operating Systems Review*, Vol.23, No.3 (1989).

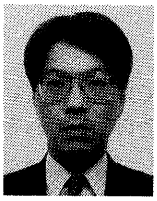
27) Tokuda, H., Mercer, C.W., Ishikawa, Y. and Marchok, T.E.: Priority inversions in real-time communication, *Proc. 10th IEEE Real-Time Systems Symposium* (1989).

28) Tokuda, H., Nakajima, T. and Rao, P.: Real-Time Mach: Towards a Predictable Real-Time System, *Proc. USENIX 1st Mach Workshop* (1990).

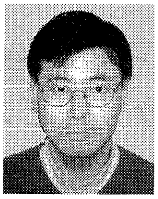
29) von Eicken, T., Basu, A. and Vogels, W.: U-Net: A User-Level Network Interface for Parallel and Distributed Computing, *Proc. 15th Symposium on Operating Systems Principles* (1995).

30) Zhang, L., Deering, S., Estrin, D., Shenker, S. and Zappala, D.: RSVP: A New Resource ReSerVation Protocol, *IEEE Network* (1993).

(Received May 8, 1998)
(Accepted October 2, 1998)

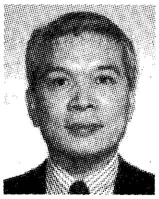


Takuro Kitayama is currently a research staff at Keio Research Institute at SFC, Keio University. His research interests are operating systems, realtime systems, and distributed system. He is a member of ACM, USENIX, and IPSJ.



Tatsuo Nakajima is an Associate Professor of Center for Information Science at Japan Advanced Institute of Science and Technology, where he is employed since 1993. He received his Ph.D. from Keio University in 1990 in distributed reliable computing. His research interests are operating systems for multimedia, high performance operating systems, reliable communications, and object-oriented languages. He is a member of ACM, USENIX, IPSJ, and Japan Society for Software Science and Technology.

Shuichi Oikawa is a Post Doctoral Fellow in Computer Science Department at Carnegie Mellon University. He received his Ph.D. from Keio University in 1996. His research interests are Operating Systems, Distributed Systems, Real-Time Systems, and Multimedia Systems. He is a member of ACM, IEEE, IPSJ, and Japan Society for Software Science and Technology.



Hideyuki Tokuda received his B.S. and M.S. degrees from Keio University in 1975 and 1977 and a Ph.D. degree in Computer Science from the University of Waterloo in 1983. He is currently an Executive Vice President and a Professor in the Faculty of Environmental Information, Keio University. His research interests include distributed real-time systems, multimedia systems, mobile systems, communication protocols, massively parallel/distributed systems, and embedded systems. He has created many operating systems and software tools such as Real-Time Mach, the ARTS Kernel, Shoshin, Scheduler 1-2-3, and ARM (Advanced Real-Time Monitor). He is a member of IEEE, ACM, IPSJ, and Japan Society for Software Science and Technology (JSSST). He is currently the chair of SIGOS in IPSJ and the executive board member of JSSST.