A Portable Communication System for Video-on-Demand Applications using the Existing Infrastructure

Yasushi NEGISHI, Kiyokuni KAWACHIYA and Kazuya TAGO
IBM Research, Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato, Kanagawa 242, Japan

Abstract

Video on demand (VOD) systems are becoming popular. They have special requirements for receiving VOD data steadily, but common general communication protocols cannot meet their requirements. Several new protocols have been designed to fulfill this service requirement, but it takes a long time for a protocol to be supported from end to end.

We propose a VOD communication system that is implemented as an user-level library on top of a common existing protocol. This protocol is easy to port, and the level of connectivity is almost the same as in the underlying existing protocol. We describe the important points related to its implementation, such as the low overhead of implementation and the flow control mechanism. We implemented a prototype system, measured its performance, and concluded that implementing a communication system as a user-level library enables us to implement a VOD portable communication system with a practical level of performance.

1 Introduction

The performance of communication media is improving rapidly, and it has become economically feasible to access multimedia data on remote nodes via communication networks. As a result, a large number of video on demand (VOD) systems are now being built. A new type of communication service is required to implement such systems, because multimedia data need to be delivered to clients within a limited delay period. Existing communication protocols such as TCP/IP do not ensure that data are delivered within the maximum delay period. An explicit quality of service (QOS) control is needed for VOD applications, and several new protocols, including ST-II [1], have been designed to fulfill this service requirement.

On the other hand, popularizing a protocol takes much time and requires much effort. A new protocol needs to be implemented for all the routers and gateway machines on the path from the server to the client. Implementing the protocol even on one node is not easy, because it is implemented as a kernel module. As a result, it takes a long time for a protocol to be supported from end to end. However, the problem is alleviated if we can implement a new protocol as a user program. Such a program would use an existing widely used protocol as the lower half of the protocol stack, while the upper half of the protocol stack would implement an application specific service such as QOS control at user level.

It is not clear whether it is actually feasible to build a user-level protocol handler for multimedia applications, because placing a protocol handler on the user level tends to increase the overhead, and the user program cannot control the details of allocating system resources. We have implemented a communication system with a practical level of performance as a user-level library by employing the following techniques: (1) designing a new low-overhead API that makes it possible to pass data to the communication system without copying, (2) implementing a protocol that guarantees the data supply to applications by feeding back the amount of data remaining in the receive buffer, and (3) implementing a protocol stack as a coroutine of the application program in order to reduce the context-switching overhead.

This communication system was initially implemented on a Unix workstation, and later ported to a personal computer (PC) running DOS. We implemented a system that transfers MPEG-1 video data on a Unix workstation to PCs through a LAN. This communication system's target application is VOD; it should be noted, however, that is not restricted to long-running videos such as movies, but it includes the all applications that transfer video data from storage media and display them. In this paper, we describe the necessary functions of a communication mechanism for VOD applications in Section 2, issues for implementing a communication system over an existing protocol in Section 3, implementation and evaluation of a prototype system in Section 4, an application using the proposed communication system in Section 5, and related technologies in Section 6.

2 Communication Systems for VOD Applications

VOD systems transfer video data from a server's disk to multiple clients via a network. Figure 1 shows a system with a single client. The control program of the server gets data from the file system, and passes them to the communication system. The control program of the client periodically supplies video data to the decoding system.

As one of the QOS controls, VOD communication systems as a whole should guarantee a steady receiving pace. If the client control program's receive buffer is empty when a request arrives from the decoding system, the client control program cannot pass any data, and the display becomes disordered.

0743-166X/96 $5.00

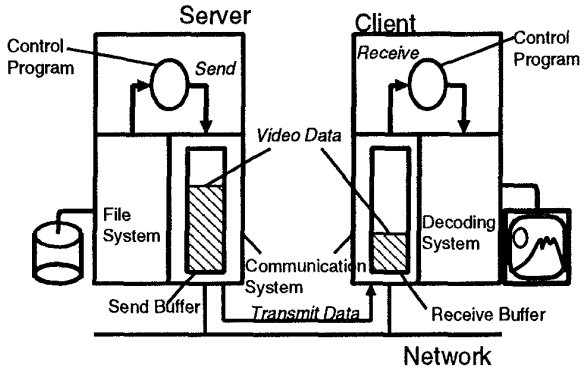


Figure 1: VOD System

The communication system as a whole should guarantee that the probability of the receive buffer's being empty when a request arrives is lower than a constant value. We will call this function the "guarantee of a steady receiving pace." The term doesn't imply a 100 % assurance that every request for data will be met, but keeps the probability above a specific value.

When there is a sufficient margin of communication media bandwidth and processor power, it is possible for an existing communication system without the above function to transfer video data. The purpose of a VOD communication system is to increase the number of clients that a system can support by explicitly guaranteeing a steady receiving pace.

We can take advantage of the following features of VOD communication:

Feature 1. The required bandwidth is almost constant, and can be predicted in advance. For MPEG-1 video data, for example, it is 1.5 Mbps.

Feature 2. Communication errors are not fatal, but should be minimized.

Feature 3. VOD application users can tolerate a longer communication delay than interactive users.

To guarantee a steady receiving pace, we can consider the following two approaches:

Approach 1. Implement a system that ensures the allocation of communication resources, such as communication media, processor power, and memory area for buffering.

Approach 2. Implement a system that monitors the volume of data in the receive buffer and keeps it above a specified value by sending requests to the server node. For example, the system might have a feedback control that requests the server node to send data faster.

It is possible to follow either one of these approaches or a combination of them.

To support more clients, it is also important to implement a system that requires less processor power, and memory by reducing the overhead of protocol handling.

2.1 General Communication Systems

The greatest advantage of a general communication system is high connectivity. For example, TCP/IP and UDP/IP are available on almost all operating systems and communication media, especially among workstations, and all the nodes using them can communicate with one another. Moreover, a system using TCP/IP and/or UDP/IP on a LAN can be connected to a WAN with little modification.

On the other hand, the performance goals of these general communication protocols are to maximize the throughput and to minimize the response time, not to guarantee a steady receiving pace. For example, although the TCP/IP protocol has a flow control mechanism, its purpose is not to guarantee a steady receiving pace, but to prevent overflows of the receive buffer caused by transfer of more data than the receive buffer can accommodate. TCP/IP does not guarantee a specific volume of residual data in receive buffer, and if a communication error occurs, retransmission is prompted by the sender's timeout. Thus, in TCP/IP the number of clients that one server and one network can support is small, because a lack of QOS causes imbalance of the data flows, even if on an average there is enough throughput for each connection.

2.2 Communication Systems for VOD

Some VOD communication systems, such as HeiTS [2], LAN Server Ultimedia, and StarWorks, are used in practical VOD systems. The ST-II protocol, used by HeiTS, has a function for reserving necessary resources along a path of communication. LAN Server Ultimedia transfers VOD data preferentially by using the priority function of Token Ring. StarWorks has a function for assigning processing power to VOD communication handling. They have a light implementation, achieved by optimizing their implementation, increasing the size of communication packets, and decreasing the number of acknowledgment packets.

However a part of each of these communication systems is implemented in the OS kernel. The in-kernel implementation improves the performance, because it allows necessary resources to be accessed directly. However, it is not efficient to port these communication systems to existing operating systems for the following reasons:

- It is necessary to understand the details of the target operating system.
- It is necessary to redesign the manner of implementation according to the structure of each target operating system.
- It takes much more time to write an in-kernel program and to test it than for conventional out-of-kernel programs.

3 A Communication System on Top of an Existing Protocol

We consider a portable application-specific communication system, which is realized as a user-level library on top of an existing protocol. This section describes a VOD-specific communication mechanism on top of UDP/IP. The approach of using an existing protocol

has two major advantages: First, the communication system becomes more portable, and the level of connectivity is almost the same as in the underlying existing protocol (UDP/IP). Second, existing infrastructure elements such as network equipment, network-management tools, and know-how can be used with the new system, and we do not have to wait for the infrastructure to be established.

A feedback mechanism is used to guarantee a steady receiving pace. The receiver monitors the remaining volume of data in the receive buffer, and notifies the sender regularly of the result. This corresponds to the second approach described in Section 1. This approach (with a large buffer) is suitable for VOD systems, in which communication delays are a relatively small problem.

To realize a communication system as a user-level library, we must resolve several issues, such as reduction of the overhead. We are trying to do so by taking advantage of the characteristics of the VOD applications.

3.1 Problems with Out-of-Kernel Communication Systems

Generally speaking, the coexistence of in-kernel and out-of-kernel communication systems implies the following performance disadvantages:

1. Extra data copying between the communication systems may be necessary. According to Kay and Pasquale [3], data copying accounts for 30% of the total communication.
2. The UDP/IP protocol has a data buffer in the kernel, and the out-of-kernel VOD communication system also has its own buffer outside the kernel. If the out-of-kernel communication system does not pay attention to the in-kernel buffer, packets may be lost owing to overflow of the in-kernel buffer. In UDP/IP, most packet losses are caused by buffer overflow rather than by physical transfer errors.
3. The out-of-kernel (user-level) communication system cannot control its own processor scheduling. Moreover, if the communication system consists of multiple user processes, an extra overhead for context switching occurs.

3.2 Approaches to Solving the Problems

Figure 2 shows the structure of our VOD-specific communication system.

To solve the problems described in the previous subsection, we adopted the following methods:

1. A new API that enables copyless data exchange between the user program and the communication system is introduced.
2. The out-of-kernel communication system notifies the sender of the necessary amount of data taking account of both the in-kernel and out-of-kernel buffer usage.
3. In VOD applications, communication occurs continuously. By making use of this calling pattern, the out-of-kernel communication system can be implemented as a coroutine of the user application program. The communication processing is performed all at one time when the control is passed to the coroutine by requests such as send or receive.

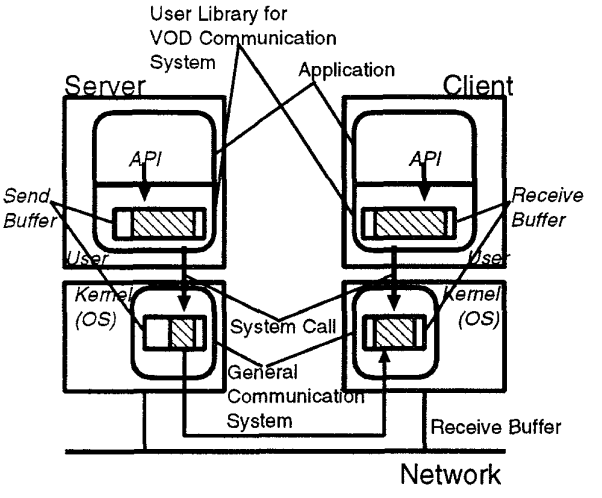


Figure 2: Structure of the VOD-Specific Communication System

The rest of this section describes the copyless API, the flow-control mechanism, and the coroutine implementation of our VOD-specific communication system.

3.3 A New API Suitable for VOD Systems

For reliable communication, the sender must keep the transmitted data for retransmission until successful data transfer is confirmed. Therefore, the data are normally copied from a buffer in the user program to a buffer inside the communication system.

To reduce this data-copying overhead, the communication system directly uses the data buffer of the user program. But in this method, some mechanism is needed to notify the user program when the transfer is complete. The simplest mechanism is to block the user program until the completion of the transfer; in this case, however, the user program cannot run during the transmission, even if the processor is available.

To avoid the user program being blocked, we can consider another mechanism, whereby, a shared buffer pool is provided between the user program and the communication system. The data are placed in the buffer which has a structure specified by the communication system, and a synchronization mechanism is provided to notify the user program when the transfer is complete. This mechanism allows the user program to run in parallel with the data transfer. But the data structure of the shared buffer — for example, a page boundary or buffer-link structure — is restricted, and an extra copy of the data to be transmitted may be needed in order to match this data structure. The API must also be changed to support explicit synchronization.

To avoid this problem, we designed our communication system so that the shared buffer area is managed

Functions provided by the user program

Buffer allocation	`int BUFALLOC(char **bufp, int *sizep, int *bufidp)`
Buffer free	`int BUFFREE(int bufid)`

API functions provided by the communication system

Buffer send	`int BUFSEND(char *buf, int size, int bufid)`
Buffer receive	`int BUFRECV(char **bufp, int *sizep, int *bufidp)`

Table 1: Functions Used by the VOD-Specific API

by functions in the user program. This mechanism enables the user program to use a convenient data structure and synchronization mechanism.f The buffer management functions in the user program are registered with the communication system, and upcalled from the communication system when it needs buffer management.

Table 1 shows the functions used in the new API.

`BUFALLOC()` and `BUFFREE()` are buffer management functions provided by the user program. They may be upcalled from the communication system.

`BUFALLOC()` allocates a buffer whose the size is `*sizep`, and returns its address in `*bufp`. If -1 is specified as `*sizep`, an appropriately sized buffer is allocated. The real size and ID of the buffer are returned in `*sizep` and `*bufidp`. The return value is the size of the allocatable buffer area.

The `BUFFREE()` frees a buffer whose ID is `bufid`. The synchronization of transfer completion is achieved by the upcall of this function. The return value is the size of allocatable buffer area.

These functions allow a user program to manage buffers that are convenient for the program. The sender and receiver may use different kinds of buffer management.

`BUFSEND()` and `BUFRECV()` are API functions for data communication provided by the communication system.

`BUFSEND()` sends data in the buffer `buf` whose size is `size`. This function returns before the transmission is complete. The user program will be notified of the completion through the upcall of `BUFFREE()` with the buffer ID `bufid`.

`BUFRECV()` returns a buffer that contains the received data. The buffer's address and size are returned in `*bufp` and `*sizep`. The communication system allocates receive buffers in advance through the upcall of `BUFALLOC()`, and the data are directly received in the buffer.

The following is a sample flow of the communication in these functions.

Sender:

1. When the connection is set up, the user program registers `BUFFREE()` with the communication system.
2. The user program calls `BUFALLOC()` to allocate a buffer, and places data in it.
3. The user program calls `BUFSEND()` to request transmission of data.
4. The communication system records the request, starts the transfer, and returns the control to the user program.
5. When the transfer is completed, the communication system upcalls `BUFFREE()`.
6. This upcall informs the user program that the transfer is complete.

Receiver:

1. When the connection is set up, the user program registers `BUFALLOC()` with the communication system.
2. The communication system calls `BUFALLOC()` to allocate a buffer, and waits for data.
3. The communication system directly receives data in the buffer.
4. The user program calls `BUFRECV()` to request reception of data[1].
5. The communication system waits until the buffer is filled, and returns the buffer to the user program.
6. The user program uses data in the buffer, and calls `BUFFREE()` to free the buffer.

The buffer management is opened to the user program, and therefore this mechanism is very flexible and extensible.

3.4 Flow Control

Flow control, with feedback to the sender of the volume of data remaining in the receive buffer, is necessary for guaranteeing a steady receiving pace. In this case, the out-of-kernel communication system should control the flow while taking account of the amount of space in the in-kernel buffer. Packets are lost on the receiver side when more data are sent at one time than the in-kernel buffer can accommodate.

The receiver side's out-of-kernel communication system controls the in-kernel buffer size by using a function such as `setsockopt()` in Unix, and controls the volume of data sent in order to prevent more data being sent than can be accommodated in the smallest space in in-kernel and out-of-kernel buffers.

Figure 3 shows an example of the flow control. The size of the space in the in-kernel buffer is 3, and that in the out-of-kernel buffer is 2. The system should control the sender in order to prevent transfer of more data than 2, which is the smallest space in the in-kernel and out-of-kernel buffers.

3.5 Coroutine Implementation

Introduction of a communication server process for protocol processing inevitably brings several overheads, such as data copying and context switching. In the case of VOD, applications issue periodically send and receive requests. Therefore, the communication system can handle necessary communication processing all at one time when it is called. By using this

[1]The order of steps 2, 3, and 4 in the receiver may be shuffled according to the timing of the receive request.

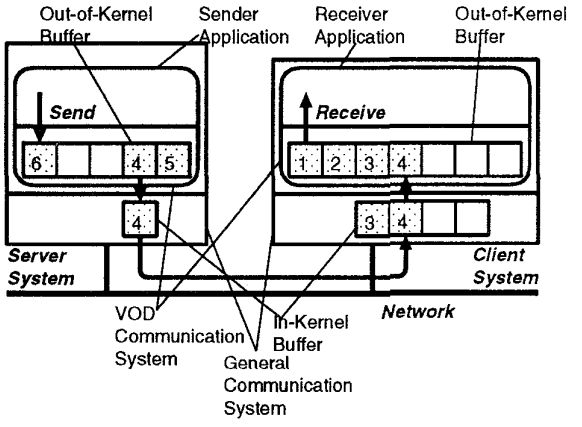


Figure 3: Management of Out-of-Kernel and In-Kernel Buffers

calling pattern of VOD, we implement a communication system without any context switching overhead. This increases the response time for send/receive requests, but it does not become a problem if the system avoids block operations and previously sets a limit on the maximum number of packets that can be handled all at one time. In this case, the system handles multiple packets all at one time. The overhead for sending or receiving acknowledgment packets can be decreased by using a protocol that allows one acknowledgment packet to be sent for multiple data packets.

4 Implementation and Evaluation of Prototype System

We implemented a prototype of the proposed VOD communication system, called the Loosely Continuous Media Communication (LCMC) system, on top of UDP/IP protocol, and evaluated its portability and performance.

4.1 Implementation of LCMC

LCMC's program consists of about 2,500 lines written in C++. It was implemented as a user-level library by using the socket library for the UDP/IP protocol. At first, LCMC was implemented on AIX, which is IBM's derived version of Unix, and then it was ported to DOS. This subsection focuses on the flow control mechanism of LCMC.

LCMC packet was implemented as a payload of the UDP/IP packet. The sender-side communication system controls the flow according to the field on the header of received LCMC packets, which indicates the volume of data that should be sent. At the beginning of a communication, it blocks the receiver program until the out-of-kernel receive buffer is filled with valid data.

The sender application program creates a buffer and passes it to LCMC. LCMC uses the buffer as an out-of-kernel send buffer, and sends the data according to the feedback field.

There is a limit on the size of the out-of-kernel send buffer. If the sender application program requests to transmit a buffer, and the out-of-kernel send buffer exceeds this limits, the program is blocked until the size of the buffer is below the limit. This blocking action thus controls the output volume of the sender program.

4.2 Portability of LCMC

To port the LCMC program from AIX to DOS, we changed it in just seven places. Five of these changes were necessitated by the differences between the TCP/IP socket API in AIX and that in DOS. Another was necessitated by differences in the compiler, and the last was necessitated by difference in the maximum buffer size for communication. We took about seven days to port the LCMC program from AIX to DOS.

4.3 Performance of LCMC

As we explained in Section 2, the objective of VOD communication systems is to maximize the number of clients supported by one server and communication medium. This means that we can evaluate the performance of a VOD communication system according to the number of clients that can be supported. To measure the number, we developed the virtual VOD system, shown by Figure 4, and measured its performance.

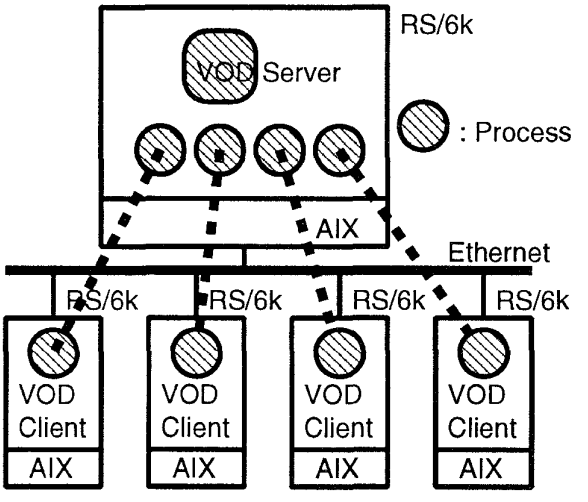


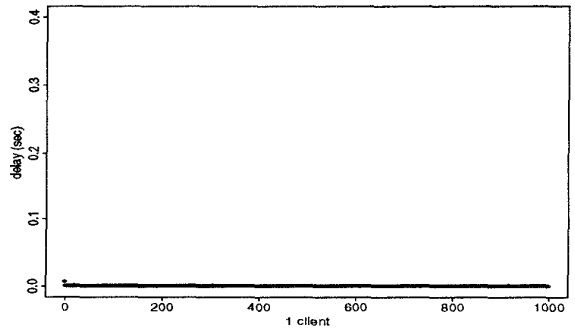
Figure 4: Virtual VOD System for Measuring Performance

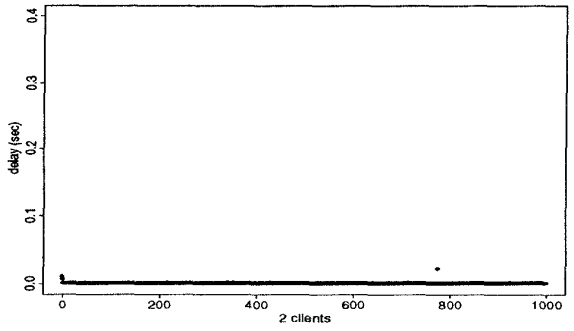
We measured the time between a receive request and a data-supply time in this system by using the real-time clock of the RS/6000. We used two protocols, TCP/IP and LCMC, and took measurements with client numbers from 1 to 4 for each protocol.

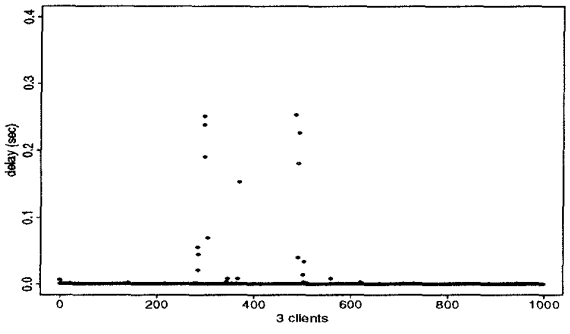
The unit of data transfer is 8,000 bytes. The transfer rate is 1.5 Mbps, which is an MPEG-1 standard rate. Therefore the interval time of the transfer is 42.7 msec. The number of transfers per client is 1000.

Figures 5 and 6 show the results for the TCP/IP and LCMC protocols. In these figures, the X-axis represents the number of the transfers and the Y-axis represents the period between the time of a read request and that of the data supply.

In the case of TCP/IP, the configuration with two clients has two points for which the delay period is







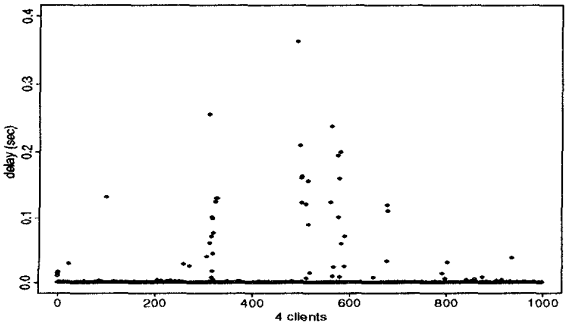
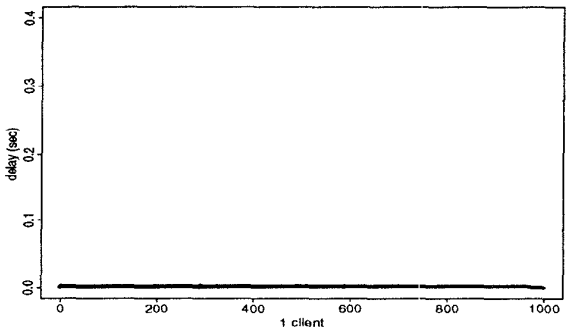
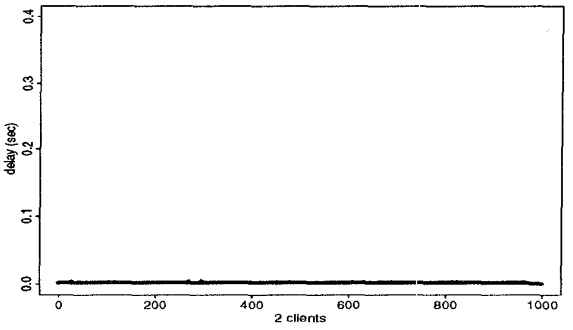
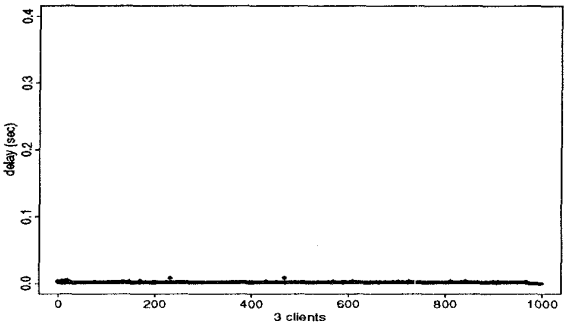


Figure 5: TCP/IP







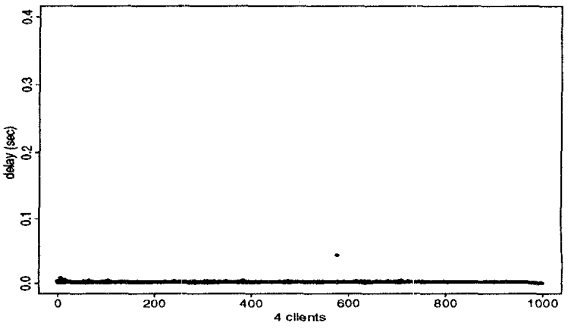


Figure 6: LCMC

more than 10 msec. The configurations with three and four clients have many points for which the delay period is more than 10 msec, and the delay periods for some points are more than 200 msec. This means that it is difficult to use the TCP/IP system for real applications in configurations with three or more clients.

On the other hand, in the case of LCMC, the configurations with two or three clients have no points for which the delay period are more than 10 msec. The configuration with four clients has one such point, but the delay period for the point is only about 40 msec. This means that it is possible to use the LCMC system for real applications in the configurations with at least four clients. The configuration for four clients uses more than 60% of the bandwidth of the communication medium. We think that this system's performance is adequate, because there are some other sources of overheads, such as other communications, headers of media packets.

We show another graph based on the same experiment, in which the X-axis represents the number of clients, and the Y-axis represents the percentage of points for which the delay periods are less than 10-msec threshold. What may be considered as a reasonable threshold period depends on the application; we fix the time at 10 msec only in this paper. As shown in Figure 7, the percentage in TCP/IP becomes smaller as the number of clients grows. In LCMC, on the other hand, it remains almost the same as the number of clients grows.

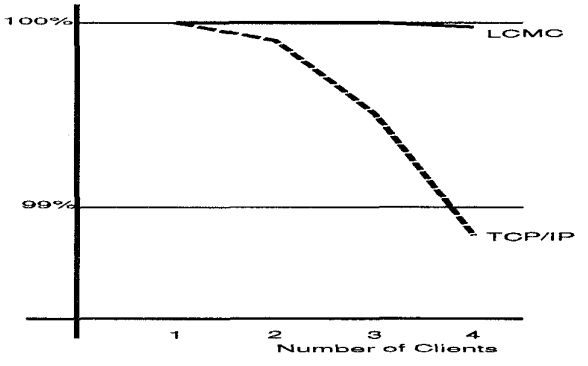


Figure 7: Percentage of Points for which the Delay Periods are Less than threshold

This difference between these two communication systems stems from the difference between their flow-control mechanisms. In the case of TCP/IP, there are 60 points for which the delay period is more than 10 msec. For 58 of these points, the reason for this is that data do not arrive before the receive requests. For the other two, the reason is that the user program is not dispatched by the scheduler even if data arrive before the receive requests. Because the performance of TCP/IP communication systems is high enough to occupy the entire Ethernet bandwidth, the fact that data do not arrive before the receive requests is the fault of TCP/IP's flow control. TCP/IP has no way of requesting the sender to transfer data. On the other hand, LCMC does so according to the amount of data remaining in the receive buffer. This control makes it possible to support more clients.

5 An Application Using the Proposed Communication System

To show the effectiveness of LCMC in a practical environment, we implemented an experimental VOD system (Figure 8). This system consists of PCs as clients and a Unix workstation as a VOD server, connected by a Token Ring network.

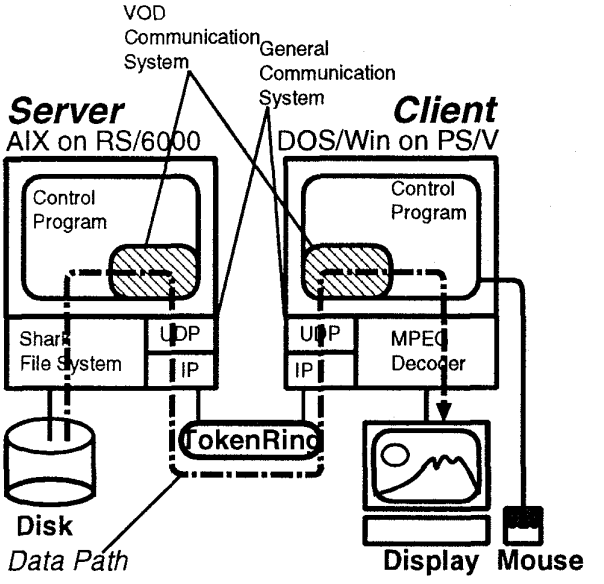


Figure 8: Sample Application of LCMC

The server's file system for storing video data is the Shark file system [4], developed by IBM's Almaden Research Center, which has a function for guaranteeing the data transfer rate of each file access. The video coding format is MPEG-1, the data is displayed by using a generally available hardware device attached to the client PC.

This prototype system consists of one server and three clients, which can display different pictures in parallel.

6 Related Technologies

We describe technologies related to our proposed system in this section.

6.1 Communication System Implementation Out of Kernel

There have been some attempts to implement a protocol by combining in-kernel and out-of-kernel communication systems. These have investigated new system calls for implementing a protocol by using the smallest possible in-kernel communication system. Among the most important research being done is

that of Anderson [5]. Its objective is to implement a lightweight scheduling mechanism for implementing real-time functions. Anderson's approach is to implement a system that consists of two schedulers: an in-kernel one and an out-of-kernel one. This approach focuses on investigating the role of each scheduler and the interface between them.

x-kernel [6] supplies an integrated interface to implement a protocol at the user-level. Thekkath's group implements a TCP/IP protocol handler as a user-level library [7].

The protocol handlers of these systems are implemented at the user-level; however it is necessary to modify and/or extend their kernels.

The objective of all of these projects is to implement a portable communication system. Our objective is the same, but we use an existing communication system with no modification. On the other hand, the above projects attempt to change a communication system in order to make the user-level implementation more portable, so it is difficult to apply their results to existing systems without modifying the systems.

6.2 Mechanism to Assign Resources

The proposed communication system does not have a function that guarantees necessary resources for communication. Consequently, it cannot guarantee a steady receiving pace when there are too many resource requests. Some mechanism is preferred for preventing the excessive use of resources, in order to reserve these resources in advance.

There are two ways of implementing a system for assigning resources. The first way is to implement the system as a part of a protocol handler, such as LAN Server Ultimedia, or StarWorks. The second is to implement it as an independent system for reserving resources for communication. The resource allocation should not depend on a specific protocol, and should be done by an independent system for resource allocation, because the resource allocation for communication is closely linked to the overall system management.

Network Resource Reservation (NRR) [8] is one such independent system, with a function for reserving the bandwidth of a communication medium such as ATM or Token Ring by using the SNMP protocol. The proposed system can be used with the NRR system to prevent delays caused by excessive use of resources.

Besides these resource reservation systems, some attempts are being made to implement a resource management system for ensuring that each packet arrives by the deadline. These systems do not require a feedback mechanism, because each packet's timely arrival is guaranteed. However, these systems are too precise to guarantee a steady receiving pace for VOD applications. We think a combination of the resource reservation system and the proposed system would be easier to implement and more portable.

6.3 Overhead of an In-Kernel Communication System

A VOD system on WAN will in many cases use the ATM as its communication medium. If a broadband communication medium such as ATM is used, the in-kernel communication system itself will become a communication bottleneck. In the case of the proposed communication system, about 83% of the CPU processing time for communication is used by the in-kernel communication system to support UDP/IP protocols.

Recent researches [9] [10] has shown that the overhead of TCP/IP and UDP/IP communication can be largely eliminated. We are trying to eliminate the overhead at at minimal cost by re-implementing the IP and UDP protocols as a single module that creates a short-cut for sending and receiving data. This system does not change the protocol and the system call interfaces, so there are no effects outside of the communication system except that of the performance.

We implemented a short-cut for the UDP/IP protocol under AIX on the RS/6000, using 100 Mbps ATM (Figure 9). This short-cut improves the UDP/IP communication performance by 20%, and the improved performance is almost equal to that in case where a user program uses the device driver directly. We think that the performance of UDP/IP communication can be improved to the same level as that in which the device driver is used directly without protocols.

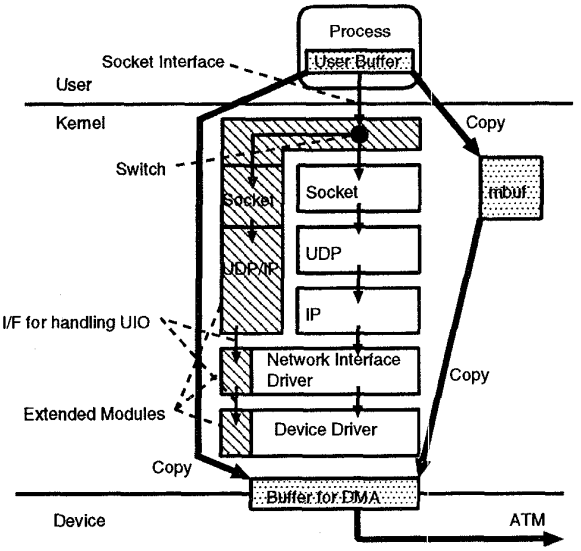


Figure 9: Performance Improvement of a General Communication System

This improvement is not necessary for each client, because each client handles only one video stream. But it is useful for the server system, because this handles many video streams for its clients.

6.4 Native ATM API

Native ATM API is an API for using the ATM's function directly without any protocols. The standard for this API is being established by the ATM Forum. When VOD systems do not use any communication medium except ATM, it is possible to implement a protocol for VOD systems on this API. In this case, the ATM guarantees its own bandwidth by using this API, without using the resource management system described in Section 6.2, and the overhead of the com-

munication system is small because it uses a communication medium directly.

The objective of the system using the Native ATM API is to use a communication medium's characteristics to maximize the communication performance. On the other hand, the objective of our proposed system is not to depend on any communication medium. The both positions of systems are clear. As a result of the experiments we described above, we think that our proposed system based on UDP/IP can achieve an adequate performance without losing and that it can be used for a wider range of applications than Native ATM API.

7 Conclusion

We have proposed a VOD communication system that is implemented as a user-level library on top of a common existing protocol. This protocol is easy to port. We have described the important points related to its implementation, such as the low overhead of implementation and the flow control mechanism.

We implemented a prototype system, and confirmed that our proposed communication system can support at least four clients on Ethernet, whereas a TCP/IP system can support only two clients. We also confirmed that our system has adequate portability, because we were able to port the system from AIX to DOS with only seven modifications. Finally, we conclude that implementing a communication system as a user-level library enables us to implement a portable VOD communication system with a practical level of performance.

In future, we will conduct a more general investigation of optimal way of implementing communication systems for specific applications.

References

[1] Casner, S., Lynn, C., Park, P., Schroder, K., and Topolcic, C., "Experimental Internet Stream Protocol, Version 2 (ST-II)," *RFC 1190*, Oct. 1990.

[2] Hehmann, D. R., Herrtwich, R. G., Schulz W., Schutt, T., and Steinmetz, R., "Implementing HeiTS: Architecture and Implementation Strategy of the Heidelberg High-Speed Transport System," *Proceedings of the 2nd International Workshop on Network and Operating System Support for Digital Audio and Video*, Springer-Verlag, 1991.

[3] Kay, J., and Pasquale, J., "The Importance of Non-Data Touching Processing Overheads in TCP/IP," *Proceedings of SIGCOMM '93*, ACM, Sep. 1993.

[4] Haskin, R., "The Shark Continuous-Media File Server," *Proceedings of IEEE COMPCON*, Spring '93.

[5] Anderson, D., "Metascheduling for Continuous Media," *Transactions on Computer Systems*, ACM, Aug. 1993.

[6] Peterson, L., Hutchinson, N., O'Mally, S., and Rao.H, "The x-kernel: A Platform for Accessing Internet Resources," *IEEE COMPUTER*, May 1990.

[7] Thekkath, C., Eguyen, T., and Lazowska, E., "Implementing Network Protocols at User Level," *Proceedings of SIGCOMM '93*, ACM, Sep. 1993.

[8] IBM Manual, "Network Resource Reservation User's Guide, Version 1.0," GC30-9654-00, Sep. 1994.

[9] Clark. D., Jacobson, V., Romkey, J., and Salwen, H., "An Analysis of TCP Processing Overhead," *IEEE Communications Magazine*, Jun. 1989.

[10] Kay, J., and Pasquale, J., "Measurement, Analysis, and Improvement of UDP/IP Throughput for DECstation 5000," *1993 Winter USENIX*, Jan. 1993.

[11] "Native ATM API Service Description Draft Specification," ATM Forum.