PAPER

Asynchronous UDP

Chuck YOO[†], ***Regular Member***, **Hyun-Wook JIN**[†], ***and*** **Soon-Cheol KWON**[††], ***Nonmembers***

SUMMARY Network bandwidth has rapidly increased, and high-speed networks have come into wide use, but overheads in legacy network protocols prevent the bandwidth of networks from being fully utilized. Even UDP, which is far lighter than TCP, has been a bottleneck on high-speed networks due to its overhead. This overhead mainly occurs from per-byte overhead such as data copy and checksum. Previous works have tried to minimize the per-byte overhead but are not easily applicable because of their constraints. The goal of this paper is to investigate how to fully utilize the bandwidth of high-speed networks. We focus on eliminating data copy because other major per-byte overhead, such as checksum, can be minimized through hardware. This paper introduces a new concept called Asynchronous UDP and shows that it eliminates data copy completely. We implement Asynchronous UDP on Linux with ATM and present the experiment results. The experiments show that Asynchronous UDP is much faster than the existing highly optimized UDP by 133% over ATM. In addition to the performance improvement, additional advantages of Asynchronous UDP include: (1) It does not have constraints that previous attempts had, such as copy-on-write and page-alignment; (2) It uses much less CPU cycles (up to 1/3) so that the resources are available for more connections and/or other useful computations; (3) It gives more flexibility and parallelism to applications because applications do not have to wait for the completion of network I/O but can decide when to check the completion.

key words: *asynchronous UDP, UDP, zero-copy, ATM, high-speed network*

1. Introduction

High-speed networks such as ATM (Asynchronous Transfer Mode) and gigabit LAN have appeared with the progress of network technology, and the rapid increase of data size, mainly due to the volume of multimedia data, makes high-speed networks indispensable for many applications. Many people have tried to integrate legacy network protocols, such as TCP/IP, into high-speed networks. However, they soon realized that the increased throughput was not as high as expected, because the processing cost of legacy network protocols is still too high to fully utilize high-speed network bandwidth [1]–[3]. As legacy network protocols are the foundation of the Internet and its applications, it is very important to refine legacy protocols for high-speed networks as well as to develop new protocols for new networks.

The main goal of this paper is to investigate how to fully utilize the bandwidth of high-speed networks using legacy protocols, and we choose UDP for the following reasons. First, it is much simpler and more efficient than TCP so that it is a convenient vehicle for our purpose. Second, recent advances in network link technology, such as optical fiber cable, makes packet loss very unlikely so that UDP is gaining higher popularity than ever before. Third, multimedia protocols, like RTP (Real-time Transport Protocol), are based on UDP because in most cases recovering from lost packets (e.g. lost audio packet) does not help improve the quality of multimedia.

Although UDP is very light and lean, its overhead can accumulate to become a bottleneck for high-speed networks. To be concrete, overheads of legacy protocols including UDP are classified into two categories: per-packet overhead and per-byte overhead [4], [5]. Per-packet overhead is the cost required to generate each packet. For example, it contains the cost to build packet headers and increases in proportion to the number of packets rather than the length of each packet. Per-packet overhead can be reduced by larger MTU. On the contrary, per-byte overhead increases in proportion to the length of data contained in each packet. Several studies [6]–[8] show that per-byte overhead is more important than per-packet overhead because per-packet overhead is rather constant and packet size tends to become large in order to achieve better network utilization.

Two major components of per-byte overhead are data copy and checksum calculation. The latest trend is to use hardware to speed up the checksum calculation, and many network interfaces already have such hardware. Now data copy remains the single most expensive operation in per-byte overhead. Therefore, eliminating data copy is crucial in achieving full utilization of high-speed network.

We propose a new concept called Asynchronous UDP that eliminates data copy in UDP. To embody and evaluate the proposed concept, we implement Asynchronous UDP on protocol stacks in Linux and experiment its performance over ATM. We compare Asynchronous UDP with traditional UDP to show the difference in mechanism and performance improvements

Manuscript received November 20, 2000.
Manuscript revised March 16, 2001.

[†]The authors are with the Department of Computer Science and Engineering, Korea University, Seoul, 136-701 Korea.
[††]The author is with Nanum Technologies, Seoul, 100-180 Korea.

in network utilization. We also discuss the advantages of Asynchronous UDP over previous attempts.

This paper is organized as follows. Section 2 explains the mechanism of traditional UDP and its overhead. Section 3 surveys previous attempts to avoid the overhead. In Sect. 4, we introduce Asynchronous UDP and describe its mechanism. Section 5 details the design and implementation of Asynchronous UDP on Linux, and the results are shown in Sect. 6. Section 7 analyzes Asynchronous UDP in comparison with previous solutions. Section 8 concludes the paper.

2. Background

2.1 Mechanism of Traditional Send/Receive

Basically, the operating system on a host provides the means for applications to send and receive data through networks, like the `send()/recv()` system call. Protocols, including UDP, accomplish `send()recv()` in the following way. Applications issue `send()` for sending data and assume that the data is sent successfully whenever `send()` returns. But in reality, it cannot always be assumed for various reasons. One reason is: when data is about to be placed in a network buffer in the end in order to be sent, the network buffer may not be available due to its limited size. Since it is not desirable for the application issued `send()` to be blocked until the network buffer is finally available, the kernel copies the target data in the user buffer into a buffer allocated in the kernel area (Fig. 1①). Then, the kernel finishes the system call by returning to the application (Fig. 1②). Therefore, the kernel cannot guarantee that the data is actually sent upon the return of the system call. When network buffer becomes available, the kernel takes the data out and copies it into the network buffer and sets the kernel buffer free (Figs. 1③,④). From the application's view, the successful return from `send()` system call means that the user buffer containing the data can be released or modified without any problem because the application can assume that data has been already sent out.

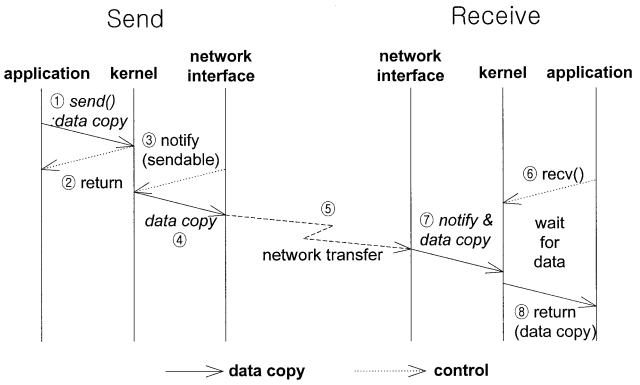


Fig. 1 Mechanism of traditional send/receive.

On the other hand, data received by the network interface is primarily stored in the network buffer. Due to the possibility of successive reception of new data and the limited size of the network buffer, the network interface informs the kernel via interrupt that data has been received, and the kernel immediately allocates the kernel buffer to save the data and releases the network buffer as soon as possible (Fig. 1⑦). Since applications do not know when the data will arrive, the time when the application issues `recv()` (Fig. 1⑥) can be either before the data is actually received or after the incoming data is already kept in kernel buffer. In the case that the data is already received, `recv()` copies the received data from the kernel buffer to the user buffer and returns to the application immediately (Fig. 1⑧). But in the other case, `recv()` system call should sleep until the data actually arrives. When the data actually arrives, after copying the data to the kernel buffer and to the user buffer, the kernel wakes up the application and `recv()` finally returns to the application. Like the send side, the data copy between the kernel and the user buffer is a well-known protocol bottleneck inside the kernel.

2.2 Data Copy

The kernel buffer plays the role of safe area between the network buffer and the user buffer: preventing data from being lost or overflowed for `recv()`, and allowing the user buffer to be freed upon the return of `send()`. However, the kernel buffer makes data copy between buffers inevitable: between the user buffer and the kernel buffer and between kernel buffer and network buffer. Data copy has become a primary source of the per-byte overhead. In order to reduce this overhead, recent network interfaces have adopted the DMA (Direct Memory Access) technique, which enables data to be copied without intervention of the CPU. But DMA can work only between the network interface and the kernel buffer. Data copy between the kernel buffer and the user buffer still depends on CPU, and this copy occupies 30% of the total packet processing cost in 8 kB UDP packets [4]. For this reason, the data copy performed by kernel internal layers has been identified as one of major bottlenecks in high-speed networking.

3. Related Works

The efforts to reduce data copy overhead can be classified largely into two categories: hardware and software solutions. Hardware solutions need assistance from some kind of network interface hardware. **Afterburner** [9], for instance, is a specially designed network interface that has the ability to process protocols. It contains a large size of network buffer within itself, enabling a host to copy the user buffer directly into the network buffer without going through kernel buffer.

This solution achieved very high performance improvement (about 49%). However, as the hardware has to be properly programmed for individual protocols to be used, its utility is constrained.

Page remapping [10] is one of the software solutions. The main idea is that, instead of copying data, the kernel remaps the user buffer's physical memory pages into the kernel buffer's virtual pages where data would otherwise be copied. As a result, one physical page is mapped to two (or more) virtual memory pages for the user and kernel buffers. Just remapping the virtual page is far more handy and simpler than expensive data copy. But a constraint of this technique is the possibility of copy-on-write. If an application tries to change the contents of user buffer upon the return of `send()`, the kernel has to allocate new physical pages and copy the contents of the user buffer to the newly allocated page so as to preserve the contents. This copy-on-write has as much (or more) cost as data copy in the traditional mechanism. When copy-on-write occurs frequently, the merit of **page remapping** disappears. An additional constraint is that the user buffer should be aligned in the page boundary.

Another similar work is **mmbuf** (multimedia buffer) [11] of the MARS Project. The main idea of **mmbuf** is based on **mbuf** of 4.3BSD. It is designed to read files from the disk and send them directly without going through the user buffer, and data is copied to **mmbuf** only once. It is highly effective in the case of transferring raw files in Web servers, FTP, and NFS.

Copy emulation [12] is more advanced and effective than the above solutions. It is similar to **page remapping** but proposes a new network sub-layer that is transparent to the upper protocol layer. It does not require that the user buffer has to be page-aligned. However, copy-on-write may still occur, and its design highly depends on virtual memory systems.

There are researches focused on performance enhancement of UDP itself. Among these works, research by Craig Partridge and Stephen Pink is very notable [5]. Copying data and calculating checksum require two separate loops of reading memory of the data size. By merging these two loops into one, memory is read once and per-byte overhead is significantly reduced. Their method attempts to eliminate costs for checksum calculation while preserving the traditional mechanism of using kernel buffer. This optimization is already included in Linux, which is the basis of our implementation in this paper.

4. Overview of Asynchronous UDP

As we pointed out earlier, data copy overhead mainly occurs because the kernel buffer always participates in sending or receiving data. The necessity of kernel buffers exists because the traditional `send()/recv()` requires a "synchronous" mechanism. By synchronous mechanism, we mean that the application can assume that data have been sent/received when `send()/recv()` returns. This characteristic is very convenient for applications because it does not need to check many details such as whether network interface is available for communication. But it is the source of overhead leading to redundant data copy.

This paper introduces a new concept that changes the synchronous mechanism and shows how it can eliminate data copy by completely bypassing the kernel buffer. We call it Asynchronous UDP. The key idea of Asynchronous UDP is not to assume that the data in the user buffer is actually sent/received when `send()/recv()` returns.

4.1 Send in Asynchronous UDP

In Asynchronous UDP, even if `send()` returns, the user buffer is still unsafe. The steps of sending data are as follows:

① Upon `send()`, the kernel builds a packet header, records the location of the user buffer, and returns immediately to the application without copying data into kernel buffer.
② When network interface is ready for transmission, the data is copied directly from the user buffer into the network buffer (Fig. 2).
③ The kernel sets a flag when the data in the user buffer is actually sent out.

Note that `send()` in Asynchronous UDP keeps the address of the user buffer instead of data copy. The application should protect the data in the user buffer until step (3) is done. Then the question is how the application comes to know when the user buffer can be freed.

There are two methods that the application can do. One method is to receive a signal from the kernel. However, current signal semantics has no way to identify which user buffer is relevant to the signal. The other method is to poll the status of the flag in step (3) above—the application checks the kernel to deter-

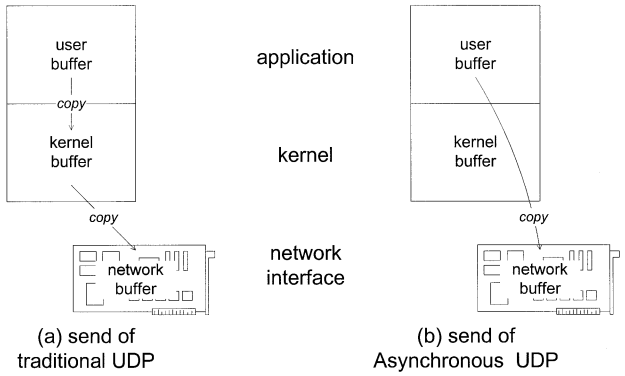


Fig. 2 Comparison of send mechanism between traditional UDP and Asynchronous UDP.

```
while ( !is_end_of_data() ) {
     fillbuffer (user_buffer, len);
     send (fd, user_buffer, len, UDP_ASYNC);
     while ( !udpwait (user_buffer) );
}
```

Fig. 3 Example of Asynchronous UDP send.

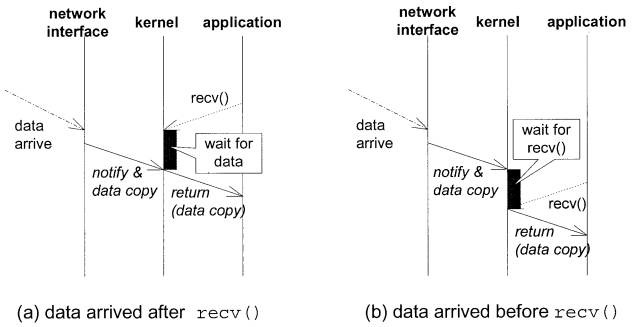


Fig. 4 Two possible cases for `recv()`.

mine whether the data in the user buffer has been sent. Polling seems to be a dully way, but it gives the application an explicit way to check the status at its convenience.

Figure 3 illustrates a simple (although not efficient) example code of the polling method, where `fillbuffer()` is a function that fills the user buffer with data, the `UDP_ASYNC` flag indicates that the data should be sent with Asynchronous UDP, and `udpwait()` is a new system call for polling.

4.2 Receive in Asynchronous UDP

The receive side of Asynchronous UDP is a little more complicated than the send side. The time when the application requests `recv()` can be either before or after data arrives (see Figs. 4(a) and (b)).

In Fig. 4(a), the traditional `recv()` should sleep because it cannot proceed any further. When the anticipated data finally arrives, the traditional UDP copies the data to the kernel buffer and invokes the suspended `recv()` to copy the data in the kernel buffer into the user buffer.

Asynchronous UDP has a difference: `recv()` never sleeps. More detail steps of `recv()` are as follows:

① Upon `recv()`, the kernel records the location of user buffer and returns immediately. Since `recv()` does not sleep, the application can continue to do other computation until data arrives.
② When data arrives, kernel first finds the corresponding user buffer.
③ The data is copied directly to that user buffer, without going through kernel buffer.
④ Kernel sets a flag when the user buffer is filled.

For the application to determine when the user buffer is filled by `recv()`, similar methods such as signaling or polling can be used.

If Fig. 4(b) happens, i.e. data arrives before `recv()`, the data has to be copied to the kernel buffer because there is no user buffer available. When `recv()` is called, the data in the kernel buffer is copied to the user buffer. Asynchronous UDP does not kick in with Fig. 4(b) because the kernel needs to save the data.

4.3 Comparison with Traditional UDP

In addition to the elimination of data copy, Asynchronous UDP has other advantages over the traditional UDP. First, Asynchronous UDP allows user code to express parallelism of computation and communication. Since Asynchronous UDP returns without waiting for the completion of network I/O—especially `recv()`, the application can do computation while the network I/O is in progress. This characteristic makes Asynchronous UDP suitable for user-level thread packages without the kernel thread because all the user-level threads have to stop if one of them issues a blocking I/O [13], [14]. Second, Asynchronous UDP gives more flexibility to applications. After issuing Asynchronous UDP calls, an application can check the completion when it is convenient. The application does not have to pay attention to when the network I/O is done.

A possible confusion is that Asynchronous UDP seems very similar to non-blocking I/O in the traditional UDP. But the non-blocking I/O attempts to initiate I/O and if the network interface is not ready, it immediately returns with a failure code. Application should retry again later. On the contrary, Asynchronous UDP does not forget the request so that application needs not to retry even though the network interface is not available.

5. Implementation of Asynchronous UDP

5.1 Environment

Although Asynchronous UDP proposed in this paper is independent of physical network interfaces and protocols, the idea is currently embodied in the following configuration for the implementation purpose.

- Linux kernel source version 2.0.27
- ATM on Linux release 0.28 (pre-alpha)
- ENI ATM driver for Efficient Networks ATM adapter

ATM on Linux is a software package that provides the features of ATM on Linux [15]. Because the basic unit of ATM is a cell that is far smaller than typical 'packets' and because cell-level fragmentation is not suitable for high-speed communication, the package includes AAL (ATM Adaptation Layer) for splitting data into cells very efficiently [16]. Also, because ATM is connection-oriented, it also converts IP addresses to ATM internal addresses and virtual channel numbers

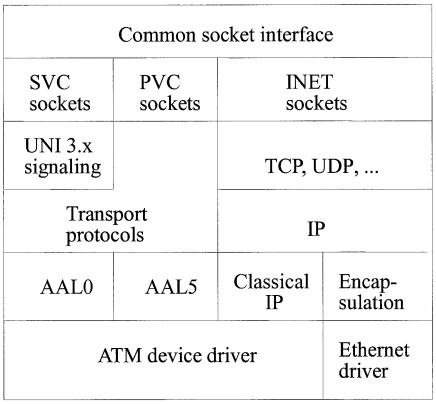


Fig. 5 Network layers of ATM on Linux.

and vice-versa.

Figure 5 shows the overall layers of ATM on Linux [15]. ATM on Linux provides several socket interfaces for programmability (e.g. SVC and PVC). Our implementation uses the INET interface because it is the most popular.

When an application sends an UDP packet, the IP over ATM (Classical IP) layer on Linux handles the packet, determines which AAL it needs to go through, and forwards it to the ATM device driver. The ATM device driver delivers the packet into the network interface's buffer, and finally the network interface sends the contents of the buffer via a physical medium.

When the network interface receives cells or AAL packets composed of cells, it saves them in the network interface and invokes an interrupt immediately. The interrupt handler allocates the kernel buffer and copies packets there. Then the kernel dispatches each packet to the corresponding protocol layer according to its header. Each protocol maintains private queues for received packets, and if the user application already issued `recv()`, the packet is immediately delivered to the user buffer, which the application allocated. Otherwise, packets should wait in the queue until they are taken out.

We turned off the checksum computation because the ATM adapters in the experiment are connected by optical fiber so that the link does not generate an error. Therefore, both traditional UDP and Asynchronous UDP did not perform the checksum computation in the experiment.

5.2 Asynchronous Send

Figure 6 shows the main difference of the send procedure between the traditional UDP (Fig. 6(a)) and Asynchronous UDP (Fig. 6(b)). Asynchronous UDP just keeps the pointer of the user buffer (labeled *data pointer* in Fig. 6(b)), without copying data to the kernel buffer. A real data transfer toward the network buffer is performed by DMA using the *data pointer* when network

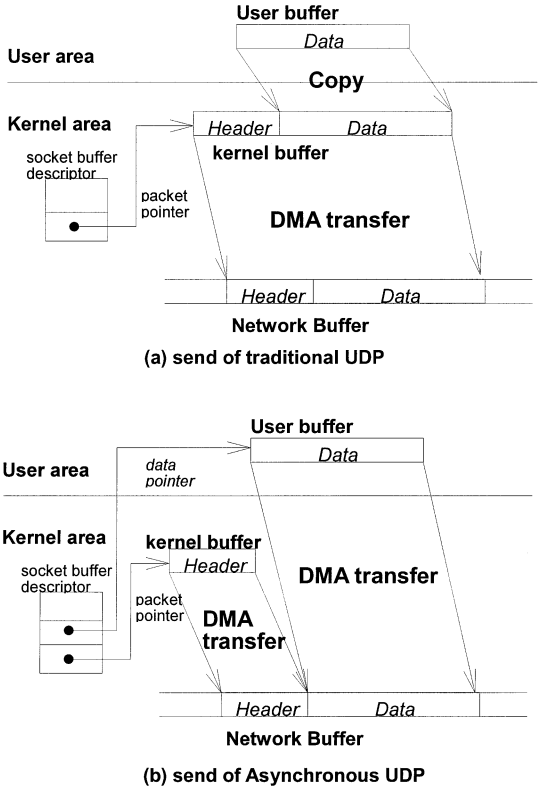


Fig. 6 `send()` of traditional UDP and Asynchronous UDP.

interface is ready to send data. As a result, Asynchronous UDP eliminates data copy from user buffer to kernel buffer. In addition, Asynchronous UDP has other advantages. First, kernel buffers do not need to be allocated for data so that less memory is required to send data. Second, because CPU is not involved (for data copy), the number of CPU cycles is reduced in the send procedure. Applications can send more UDP data and/or do something else while data is being sent because DMA runs in parallel with CPU.

5.3 Asynchronous Receive

Asynchronous UDP acts differently depending on when packets arrive. In order to distinguish between the two cases mentioned in Sect. 4.2, the interrupt handler first examines the packet header and checks whether it is an UDP packet. Then it reads which UDP port would accept this packet and checks whether there is any application issued a `recv()` for that UDP packet. In order for the interrupt handler to find the appropriate application and user buffer, we add a new internal data structure that records all information about the user buffer (named *async recv descriptor* in Fig. 7(b)) for each `recv()`. This data structure contains several fields such as a data pointer for the user buffer and a process identification that the user buffer belongs to. The interrupt handler searches this data structure to determine which application issued `recv()` and where data should

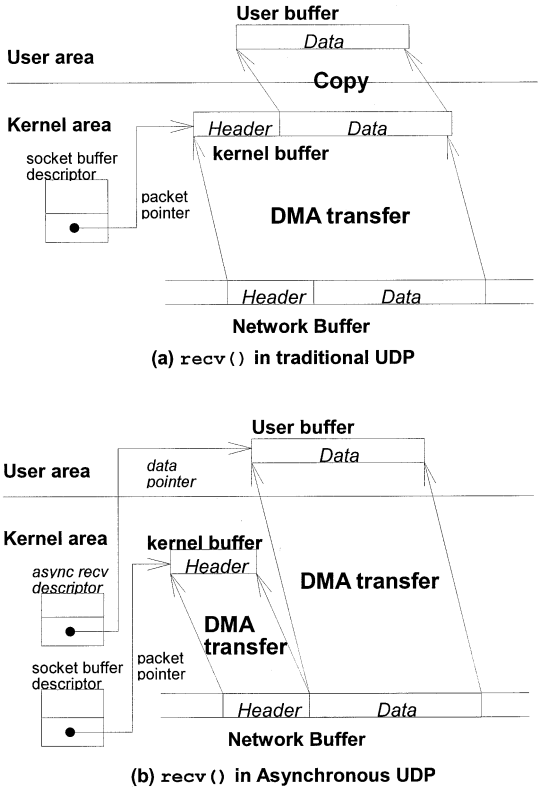


Fig. 7 `recv()` of traditional UDP and Asynchronous UDP.

be copied. If the interrupt handler succeeds in finding the corresponding user buffer, it moves the packet from the network buffer directly to the user buffer via DMA as Fig. 7(b) shows. This procedure eliminates data copy involving CPU and merely keeps the header of the received packet in the kernel buffer. Thus Asynchronous UDP `recv()` requires a much smaller size of kernel buffer per packet. But if the interrupt handler fails to find the corresponding user buffer, it means that the application has not issued `recv()` and Asynchronous UDP acts exactly the same as the traditional UDP, allocating kernel buffer and copying data there. This prevents data from being lost and enables the network interface to be ready for incoming packets.

When a host has many network-active applications, it may be time-consuming to search *async recv descriptor* for each packet. Since the search is done inside the interrupt handler, it should be done as quickly as possible. Thus, we use a hash table to manage *async recv descriptor* efficiently.

6. Performance Evaluation

To evaluate the performance of Asynchronous UDP, we use three cases of machine configurations. Because the data copy is affected by the CPU speed, each case has different speeds for the sender and receiver, as seen in Table 1.

The performance is measured using **ttcp**, which is

Table 1 Machine configurations.

Case	Case 1		Case 2		Case 3	
	Sender	Receiver	Sender	Receiver	Sender	Receiver
CPU	Intel Pentium III 450MHz	Intel Pentium 120MHz	Intel Pentium III 450MHz	Intel Pentium III 450MHz	Intel Pentium 120MHz	Intel Pentium III 450MHz
RAM	128MB	32MB	128MB	128MB	32MB	128MB

Table 2 Sender and receiver of each experiment.

Experiment	Experiment A		Experiment B		Experiment C	
	Sender	Receiver	Sender	Receiver	Sender	Receiver
Traditional UDP	•	•		•		
Asynchronous UDP			•		•	•

a well-known benchmark program. Because **ttcp** does not have options for Asynchronous UDP, we modified it to support Asynchronous UDP. We measured how much time it would take to transfer packets from sender to receiver, varying the data size from 32 B to 8 kB. The throughput is measured in three different types of experiments for each case as in Table 2. Experiment A is the basis for performance comparison.

The throughput in the experiments is measured with pure data, excluding ATM cell header, header for AAL, and header for UDP/IP. Therefore, we compute the ATM physical data bandwidth excluding header overhead as follows. Although the theoretical throughput of ATM is 155 Mbps, each cell has a 5 B header. Therefore, the practical maximum throughput of ATM is 155 * (48/53) = 140 Mbps. For 8 kB data, $\lceil 8192/48 \rceil = 171$ cells should be transferred. The overhead of AAL5 is an 8 B header and an 8 B trailer for each packet, and the size of UDP/IP header is 28 B. To transfer an 8 kB UDP packet through ATM, $\lceil (8192 + 8 + 8 + 28)/48 \rceil = 172$ cells should be transferred. This means that, when transferring an 8 kB UDP packet, the physical data bandwidth is 140 * (171/172) = 139 Mbps.

Another performance measure we used is the number of blocking `recv()` in Experiments A and B. It is to see how many times the receiver sleeps.

For the performance measurements, Efficient Network's ENI-155p-MF-C ATM adapters are installed in both sender and receiver, and they are connected directly with optical-fiber cable (called back-to-back connection) in order to exclude the effect of ATM switches.

Figure 8 and Table 3 show the experiment results of Case 1. The first noticeable result is that the results of Experiment C are far better than Experiments A and B over the wide range of data sizes. The greatest

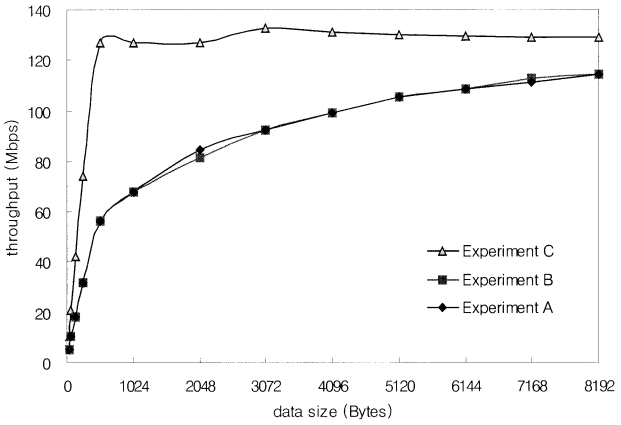


Fig. 8 Comparison between Experiments A, B, and C in Case 1.

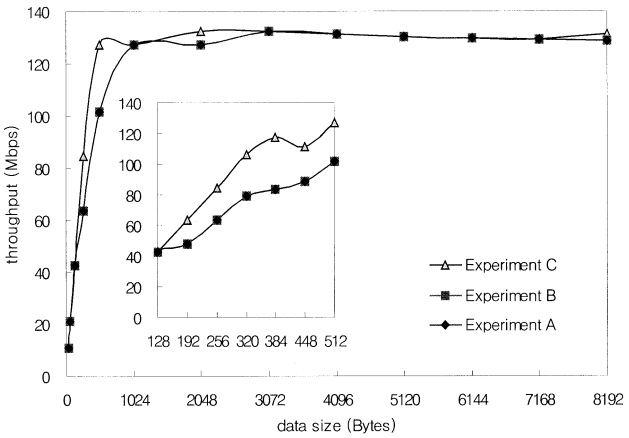


Fig. 9 Comparison between Experiments A, B, and C in Case 2.

Table 3 Improvement rate of Experiments B and C over A in Case 1.

Bytes	Experiment A (Mbps)	Experiment B (%)	Experiment C (%)
32	5	0	100
64	11	0	100
128	18	0	133
256	32	0	133
512	56	0	125
1024	68	0	88
2048	85	-4	50
3072	92	0	44
4096	99	0	32
5120	106	0	23
6144	109	0	19
7168	111	2	16
8192	114	0	13

Table 4 Improvement rate of Experiments B and C over A in Case 2.

Bytes	Experiment A (Mbps)	Experiment B (%)	Experiment C (%)
32	11	0	0
64	21	0	0
128	42	0	0
256	63	0	33
512	102	0	25
1024	127	0	0
2048	127	0	4
3072	132	0	0
4096	131	0	0
5120	130	0	0
6144	130	0	0
7168	129	0	0
8192	129	0	2

improvement over traditional UDP (Experiment A) is 133% at 128 B and 256 B, and the other numbers in Table 3 are also very respectable because Linux used in the measurements employs a highly optimized UDP. The peak throughput is 132 Mbps at 3 kB. Furthermore, in terms of bandwidth utilization, Experiment C reaches the physical data bandwidth very quickly even at small data sizes. 98% of ATM's physical data bandwidth is utilized at 512 B, and a similar utilization is maintained for larger data sizes. This implies that the per-packet overhead is negligible at data sizes larger than 512 B because Experiment C achieves full bandwidth utilization. On the other hand, the throughput of Experiments A and B increases quite slowly.

Another advantage of Asynchronous UDP we found is low CPU consumption. The receiver of Experiment C requires only 1/3 of the CPU cycles compared with Experiments A and B. Therefore, even lower CPU machines (120 MHz) can achieve full utilization of ATM physical bandwidth.

Experiments A and B have almost the same throughput. The blocking count shows that the receiver of both Experiments A and B does not sleep, and this means that the receiver is the bottleneck. This explains why Experiments A and B have the same performance.

The experimental results of Case 2 are summarized in Fig. 9 and Table 4. Although Experiment C shows a 40% improvement over A and B with small data sizes, as shown in the inset graph of Fig. 9, basically all three types of experiments show similar throughputs. Through investigating the root cause, we realized that the physical data bandwidth of ATM has been reached in all three experiments. To see the difference, we run Experiments A, B, C with Myrinet [17] in the same configuration. Myrinet is a quite popular LAN technology that provides 1.2 Gbps in one direction. Figure 10 shows the results. It proves that the results of Fig. 9 are due to the lack of bandwidth, and when a faster network is used, Experiment C with Asynchronous UDP is much better than A and B. Its throughput increases continuously with data size, and the improvement ratio is 51% at 8 kB.

Figure 11 and Table 5 show the results of Case 3, which present that Experiment C is better than Experiment A up to 3 kB, and the improvement ratio is 36%

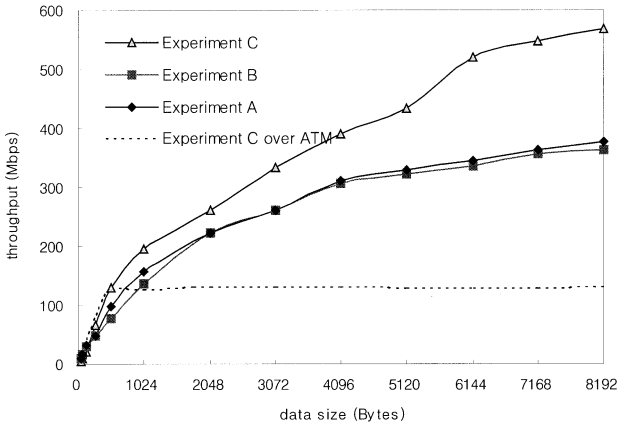


Fig. 10 Throughput comparison of Case 2 over Myrinet.

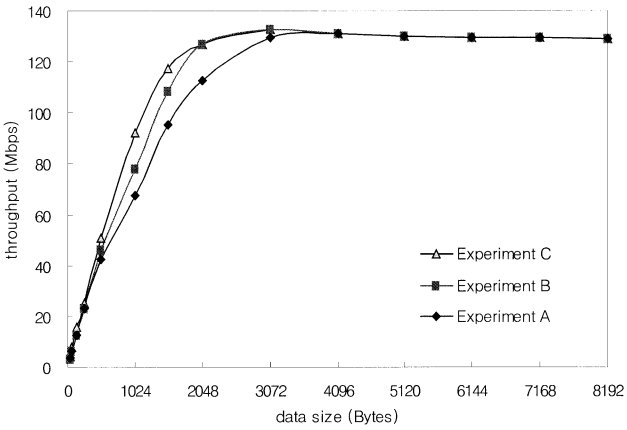


Fig. 11 Comparison between Experiments A, B, and C in Case 3.

Table 5 Improvement rate of Experiments B and C over A in Case 3.

Bytes	Experiment A (Mbps)	Experiment B (%)	Experiment C (%)
32	4	-9	14
64	6	0	25
128	13	0	25
256	23	0	10
512	42	9	20
1024	68	16	36
2048	113	13	13
3072	130	2	2
4096	131	0	0
5120	130	0	0
6144	130	0	0
7168	129	0	0
8192	129	0	0

at 1 kB. For larger data sizes, the throughput is not distinguishable. The reason is again the full utilization of the physical data bandwidth as observed in Case 2. We refer readers to [18] for proof that Asynchronous UDP can achieve a higher throughput with Myrinet in Case 3.

We also found that the receivers of Experiments A and B block 100%, but the receiver of Experiment B sleeps for a shorter time than that of Experiment A. This means that a sender using Asynchronous UDP sends data faster, which leads to improvement of the throughput by 16% at 1 kB.

7. Comparison

In addition to performance advantages, Asynchronous UDP has other advantages compared to the others. First, it is a pure software solution contrary to **Afterburner**. Second, copy-on-write does not occur, and buffers in Asynchronous UDP need not to be aligned by page as opposed to **page remapping**.

Copy Emulation is very similar to Asynchronous UDP, especially where it keeps the pointer of buffers to avoid data copy and that it does fragmentation and header building based on this pointer. It tries to remove overheads while preserving the semantics of traditional send/receive and provides good transparency for higher level protocols. But the possibility of copy-on-write still remains, and it is questionable that it is suitable for high speed networks because it makes network layers complicated. **Copy Emulation** is also highly dependent on virtual memory systems. In comparison, Asynchronous UDP adds a new path without disturbing the network layer and device driver so that the implementation is clean and modular. Also, it does not depend on virtual memory systems at all. Thus, Asynchronous UDP has better architecture than copy emulation.

The study by Craig Partridge and Stephen Pink shows a high performance increase by merging the loop for data copy and checksum calculation. Linux kernel 2.0.27, the base of our implementation in this paper, already includes this technique. Therefore, Asynchronous UDP achieves an even greater performance boost on top of their optimization.

8. Conclusions

This paper focuses on how to utilize the full bandwidth of high-speed networks. Our approach is to minimize the per-byte overhead by eliminating data copy in the protocol layers. We have proposed Asynchronous UDP and implemented it on Linux over ATM. While the data in the traditional UDP is copied through the network, kernel, and user buffers, Asynchronous UDP bypasses the kernel buffer and transfers the data directly between the network buffer and user buffer without any data copy. Through extensive experiments, we show that Asynchronous UDP not only utilizes 98% of ATM's bandwidth but also can achieve even higher throughput with faster networks such as Myrinet. Compared with traditional UDP, Asynchronous UDP is faster by 133%. This performance improvement is very notable because

the Linux networking code used in the experiments already contains many performance optimizations.

In addition to the performance improvement, Asynchronous UDP has other advantages. First, it uses only 1/3 of the CPU cycles of traditional UDP. This makes CPU available for more network connections and/or other useful computations. Second, it allows more flexibility and parallelism in applications. This is possible because an application does not have to wait for the completion of network I/O but chooses when to check the completion. Third, it is a software solution that does not require special hardware support, and it can be easily applied to existing applications because it can be integrated with existing socket API. Finally, it does not have constraints such as page-alignment or copy-on-write. Based on these characteristics, we believe that Asynchronous UDP is a viable communication primitive based on the legacy protocol for high-speed networks.

Acknowledgement

This research was supported in part by Science and Technology Policy Institute of Korea under contract 97-NF-03-01-A-01, Electronics and Telecommunications Research Institute of Korea under contract 2000-187, and University Software Research Center Supporting Project from Korea Ministry Information and Communication.

References

[1] S. Dharanikota, K. Maly, and C.M. Overstreet, "Performance evaluation of TCP(UDP)/IP over ATM networks," Technical Reports TR-94-23, Computer Science Dept, Old Dominion University, Norfolk, 1994.

[2] G.J. Armitage and K.M. Adams, "How inefficient is IP over ATM anyway," IEEE Network, Jan./Feb. 1995.

[3] Y.A. Fouquet, R.D. Schneeman, D.E. Cypher, and A. Mink, "ATM performance measurement: Throughput bottlenecks and technology barriers," Report and Discussion on the IEEE ComSoc TCGN Gigabit Networking Workshop 1995, April 1995.

[4] D.D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An analysis of TCP processing overhead," IEEE Commun. Mag., vol.27, no.6, pp.23–29, June 1989.

[5] C. Partridge and S. Pink, "A faster UDP," IEEE/ACM Trans. Networking, vol.1, no.4, pp.429–440, Aug. 1993.

[6] J. Kay and J. Pasquale, "Profiling and reducing processing overheads in TCP/IP," IEEE/ACM Trans. Networking, vol.4, no.6, pp.817–828, Dec. 1996.

[7] J. Kay and J. Pasquale, "The importance of nondata touching processing overheads in TCP/IP," Proc. ACM SIGCOMM'93, San Francisco, Sept. 1993.

[8] H.-W. Jin and C. Yoo, "Latency analysis of UDP and BPI on myrinet," Proc. 18th IEEE International Performance, Computing, and Communication Conference (IPCCC'99), pp.185–191, Feb. 1999.

[9] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley, "Afterburner," IEEE Network, vol.7, no.4, pp.36–43, July 1993.

[10] H.K. Jerry Chu and H.-K. Jerry, "Zero-copy TCP in Solaris," 1996 Winter USENIX, 1996.

[11] M.M. Buddhikot, X.J. Chen, D. Wu, and G. Parulkar, "Enhancements to 4.4 BSD UNIX for networked multimedia in project MARS," Technical Report, WUCS-97-38, 1997.

[12] J.C. Brustoloni and P. Steenkiste, "Copy emulation in checksummed, multiple-packet communication," IEEE Infocom'97, April 1997.

[13] A. Buck and R. Coyne, "An experimental implementation of draft POSIX asynchronous I/O," Proc. 1991 Winter USENIX Conf., pp.289–306, 1991.

[14] C. Yoo, "Comparative analysis of asynchronous I/O in multithreaded UNIX," Software-Practice and Experience, vol.26, no.9, pp.987–997, Sept. 1996.

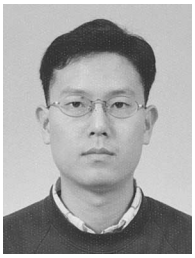
[15] W. Almesberger, "ATM on Linux," 3rd International Linux Kongress 1996, March 1996.

[16] ITU-T Recommendation I.363. "B-ISDN ATM adaptation layer (AAL) specification," ITU, March 1993.

[17] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W.-K. Su, "Myrinet—A Gigabit-per-second local-area network," IEEE Micro, vol.15, no.1, pp.29–36, Feb. 1995.

[18] C. Yoo and H.-W. Jin, "Gigabit network protocol for Linux," Proc. ITRC Korea Forum 2001, May 2001.

Chuck Yoo received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea and the M.S. and Ph.D. in computer science in University of Michigan. He worked as a researcher in Sun Microsystems Lab. from 1990 to 1995. He is now an Associate Professor in Department of Computer Science and Engineering, Korea University, Seoul, Korea. His research interests include high performance network, multimedia streaming, and operating systems. He served as a member of the organizing committee for NOSSDAV 2001.



Hyun-Wook Jin received the B.S. and M.S. degrees in computer science from Korea University, Seoul, Korea, in 1997 and 1999, respectively. He is currently a Ph.D. candidate at Korea University, Seoul, Korea. His current interests are in the protocol design for high-speed networks and wireless networks.

Soon-Cheol Kwon received the B.S. and M.S. degree in computer science from Korea University, Seoul, Korea in 1996 and 1998, respectively. He is now working for Nanum Technologies, Seoul, Korea as researcher. His research interests include high performance network and distributed application.