Network Level Framing in INSTANCE

Pål Halvorsen, Thomas Plagemann
UniK, University of Oslo
P.O. Box 70, N-2027 KJELLER, Norway
{paalh, plageman}@unik.no

Vera Goebel
Department of Informatics, University of Oslo
P.O. Box 1080, Blindern, N-0316 OSLO, Norway
goebel@ifi.uio.no

Abstract

Internet services like the world-wide web and applications like News-on-Demand have become very popular over the last years. The number of users, as well as the amount of multimedia data downloaded by each user from servers in the Internet, is rapidly increasing. In this context, the potentially (very) high number of concurrent users that retrieve data from servers represents a generic problem. In the Intermediate Storage Node Concept (INSTANCE) project, we develop a radically new architecture for Media-on-Demand servers that maximizes the number of concurrent clients a single server can support. Traditional bottlenecks, like copy operations, multiple copies of the same data element in main memory, and checksum calculations in communication protocols are avoided by applying three orthogonal techniques: zero-copy-one-copy memory architecture, integrated error management, and network level framing. In this paper, we describe the design of the network level framing concept, which enables us to reduce the server workload by reducing the number of operations performed by the communication system at transmission time.

1 Introduction

There has been a tremendous growth in the use of multimedia Internet services, and in particular, applications like News-on-Demand (NoD) have become very popular. Today, contemporary mid-price personal computers are capable of handling the load that such multimedia applications impose on the client system, but in the Media-on-Demand (MoD) servers, the potentially (very) high number of concurrent users retrieving data represents a generic problem. In MoD servers in general, commodity operating systems represent the major performance bottleneck, because *operating systems are not getting faster as fast as hardware* [8]. To support multiple concurrent users each retrieving a high data rate multimedia stream, the operating system and server architecture must therefore be improved and optimized.

In the *Intermediate Storage Node Concept* (INSTANCE) project [12], we concentrate on developing a new architecture for single servers that makes optimal use of a given set of resources, i.e., maximize the number of concurrent clients a single server can support. Thus, the task of reading data from disk and transmitting it through the network to remote clients with minimal overhead is our challenge and aim. Our target application domain comprises MoD-like applications in the smaller scale, like MoD for entertainment in hotels or airplanes, and in the medium scale, like MoD provided by city-wide cable companies or pay-per-view companies. Whether the INSTANCE approach is also beneficial for large scale systems like NoD services from CNN or BBC is subject to further research.

To avoid possible bottlenecks, the key idea of INSTANCE is to improve the server performance by combining the following three orthogonal techniques in a radically new architecture [4]:

- *Zero-copy-one-copy memory architecture*: Memory copy operations have been identified as a bottleneck in high data rate systems. Several zero-copy architectures removing physical data copying have been designed to optimize resource usage and performance using shared memory, page remapping, or a combination of both [13]. These approaches reduce resource consumption of individual clients, but concurrent clients requesting the same data require each its own set of resources. Traditional broadcast or multicast is an easy way of dealing with per client resource allocation, but startup delay might be a problem. To minimize the startup delay, a couple of broadcast partitioning schemes are proposed [13]. Still, zero-copy and delay-minimized broadcasting only reduce the per-data or per-client resource usage, respectively. To optimize both, we integrate both mechanisms to have no physical in-memory copy operations and to have only one shared copy of a data element in memory.

- *Integrated error management scheme*: When transferring data from disk in a server through the network to remote clients, the correctness of the information is checked multiple times. In INSTANCE, this redundancy is removed by integrating the error management in a disk array and a forward error correction (FEC) scheme in the communication system [5]. Opposed to the traditional data read from a RAID system [11] where the parity information is only read when a disk error occurs, we retrieve also the redundant error recovery data. All data is passed to the communication system, where the parity information from the RAID system is re-used as FEC information for the original data. Thus, by using the same parity information in both subsystems, the FEC encoder can be removed from the server communication system, and both memory and CPU resources are made available for other tasks.

- *Network level framing (NLF) mechanism*: Each time a client retrieves data from a server, the data is processed through the communication system protocols executing the same operations on the same data element several times, i.e., for each client. To reduce the workload of processing all the communication protocols for each packet transmitted to a client (Figure 1), we regard the server as an *intermediate node* in the network where only the lower layers of the protocol stack are processed (Figure 2). When new data is sent to the server for disk storage, only the lowest two protocol layers are handled, and the resulting transport protocol packets are stored on disk. When data is requested by remote clients, the transport level packets are retrieved from disk, the destination port number and IP address are filled in, and the checksum is updated (only the new part of the checksum, i.e., over the new addresses, is calculated). Thus, the end-to-end protocol which performs the most costly operations in the communication system, especially the transport level checksum, are almost completely eliminated.

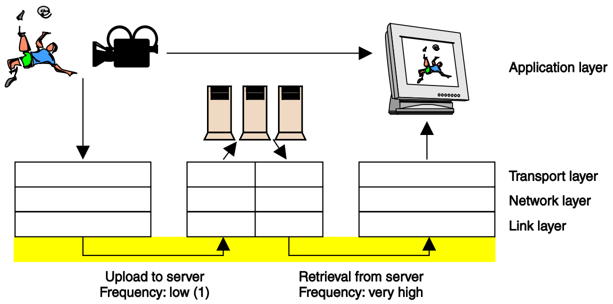

Figure 1: Traditional server storage.

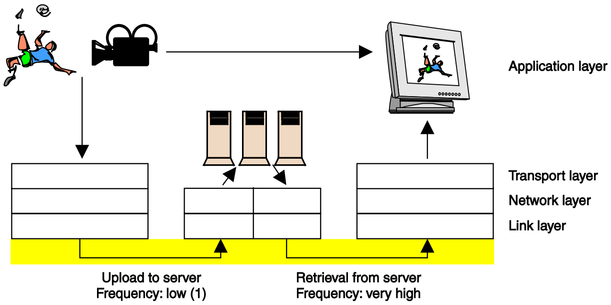

Figure 2: NLF.

This paper focuses on the design of the NLF concept, which enables us to reduce the server workload by reducing the number of operations performed by the communication system at transmission time.

The rest of this paper is organized as follows: In Section 2, we describe our design of the INSTANCE server prototype and especially the NLF mechanism under influence of the integrated error management scheme. Section 3 outlines our expected performance gain. We summarize and conclude the paper in Section 4.

2 Prototype Design

In this section, we present our MoD server prototype, which will be implemented in NetBSD, and especially the NLF mechanism. NLF reduces the overhead of CPU intensive, data touching operations in the communication protocols like the checksum calculation. Since retransmissions are not suitable for our multicast and real-time environment [15], we use UDP as the transport level protocol to transmit data from the server to the clients. Furthermore, our three techniques to improve performance in an MoD server can be used alone or combined in a system. The memory architecture has no direct implications on the NLF mechanism, but the integrated error management and the NLF schemes should be closely integrated if both techniques are used. This is because our recovery scheme uses fixed size blocks, each to be transmitted as a UDP packet, and we also transmit the calculated parity information to the remote clients. In the following subsections, we look at the integration of the recovery scheme and NLF. Then we describe how packets are stored on disk and how stored packets are retrieved from the storage system and transmitted over the network.

2.1 Integrated Error Management and NLF

Our integrated error management scheme [5] reuses parity information (calculated using a Cauchy-based Reed-Solomon erasure code) from a disk array for recovery of lost or damaged packets in a FEC scheme. However, the storage parity information does not detect random bit errors for example introduced by noise. Thus, to detect damaged packets, we need an additional scheme for this purpose, and to avoid extra overhead and redundant data, we use the UDP checksum [1]. When using both the integrated error recovery scheme and NLF, the two mechanisms must be carefully merged together, because the storage recovery scheme uses fixed size blocks, and the size of the packets (or size of a group of packets stored within a single disk block) must therefore fit into the disk block size boundaries. The performance analysis in [5] shows that the correcting scheme performs best using data blocks of 1 KB, 2 KB, 4 KB, or 8 KB, and we will therefore use one of these packet sizes. However, these sizes are too small to make efficient use of today's disks. In [14], disk requests of 64 KB or larger are suggested. Therefore, we perform (multiple) disk requests with the maximum size in NetBSD (64 KB) and afterwards split these data in several packets depending on the used packet size.

2.2 NLF Overview

As shown in Figure 2, we use the idea of asynchronous packet forwarding in the intermediate network nodes. Thus, we consider our multimedia storage server as an intermediate storage node where the upper layer packets, i.e., UDP packets, are stored on disk.

2.2.1 When to Store Packets

If data is arriving from a remote site as depicted in Figure 2 using UDP, the most efficient approach regarding overhead to store the transport level packets is to collect incoming UDP packets and put them directly on disk without transport protocol processing. However, this is not a suitable solution, because the arriving packet header will be partly invalid. The source address must be updated with the server address, the destination fields will be unknown until transmission time, and the checksum must be accordingly updated. Furthermore, UDP is an unreliable protocol where there might be losses. A reliable transport protocol like TCP must therefore be used to assure that all data is transmitted correctly to the server. This is shown as phase 1 in Figure 3. Finally, in our integrated error management scheme (possibly using a different block size than the arriving packet size), we transmit the redundant parity data calculated by the storage system, i.e., we must also generate packets from the recovery data. Consequently, we cannot store incoming packets.

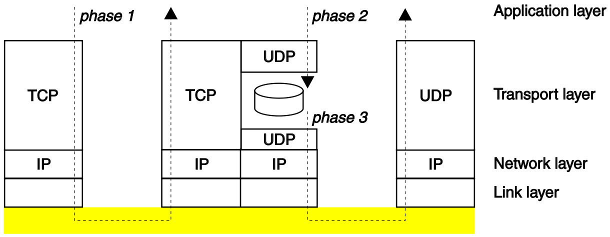

Figure 3: NLF in detail.

To be able to prefabricate UDP packets of correct size and with a correct, partly completed header, we store outgoing packets processed through the transport level protocol (phase 2 in Figure 3), i.e., after the UDP packet has been generated and the checksum has been calculated, but before handing it over to the IP protocol.

2.2.2 Splitting the UDP Protocol

To preprocess data through the UDP layer and store packets on disk, we split the traditional BSD `udp_output()` procedure [16], in accordance with phase 2 and 3 in Figure 3, into two distinct functions as depicted in Figure 4: (1) `udp_PreOut()`, where the UDP header is generated, the checksum is calculated, and the packet is written to disk, and (2) `udp_QuickOut()`, in which the remaining fields are filled in, the checksum is updated with the checksum difference of the new header fields, and the datagram is handed over to the IP protocol.

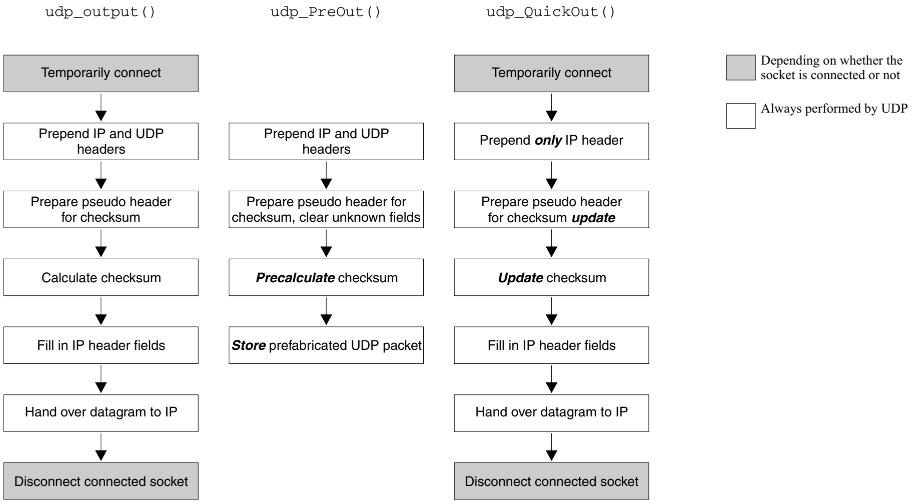
udp_output()
udp_PreOut()
udp_QuickOut()
Temporarily connect
Prepend IP and UDP headers
Prepare pseudo header for checksum
Calculate checksum
Fill in IP header fields
Hand over datagram to IP
Disconnect connected socket
Prepend IP and UDP headers
Prepare pseudo header for checksum, clear unknown fields
Precalculate checksum
Store prefabricated UDP packet
Temporarily connect
Prepend only IP header
Prepare pseudo header for checksum update
Update checksum
Fill in IP header fields
Hand over datagram to IP
Disconnect connected socket
Depending on whether the socket is connected or not
Always performed by UDP

Figure 4: `udp_output()`, `udp_PreOut()`, and `udp_QuickOut()`.

By looking at Figure 4, one might wrongly assume that the same (or even more) amount of work is performed in our `udp_PreOut()` and `udp_QuickOut()` functions compared to the traditional `udp_output()`. It is important to note, however, that in our MoD scenario where the multimedia data is retrieved multiple times, our approach will cause considerably less work. The `udp_PreOut()` is executed once per UDP datagram, and then the output is stored on disk. For each client, only the `udp_QuickOut()` is executed where several operations, especially the time consuming checksum calculation (see Section 2.5.2), is considerably simplified.

2.3 Storing Packets

The integrated error recovery and NLF write operation is shown in Figure 5. The data is handed over to the storage system where write operations are done à la RAID level 4/5 using the information data to generate a set of redundant parity data for recovery. Each of the codeword symbols, i.e., recovery scheme blocks, is processed through `udp_PreOut()` to prefabricate and store the UDP packet, i.e., the output of CPU intensive operations like the checksum calculation is stored in the packet header. Finally, to optimize storage system performance, several codeword symbols are concatenated to form a disk block, and the newly formed blocks are written to disk.

2.4 Retrieving and Transmitting Packets

Since the server application only will retrieve data from the storage system and then forward the data to the communication system without performing any operations on the data itself, the data retrieval and transmission operations should be performed using some zero-copy, in-kernel data path for example supported by the MARS stream API [2]. This will reduce system call overhead and data movement operations. Furthermore, to reuse the stored parity data for FEC in the communication system, read operations are performed in a RAID level 0 fashion as sketched in Figure 6. The whole UDP packet is retrieved from disk, containing either information or parity data, and is processed through the `udp_QuickOut()` where only a quick assembly of the packet is performed, i.e., the following four steps are performed: (1) the missing values, i.e., source port number, destination port number, and destination IP address, are filled in the appropriate fields of the UDP packet, (2) a checksum over these

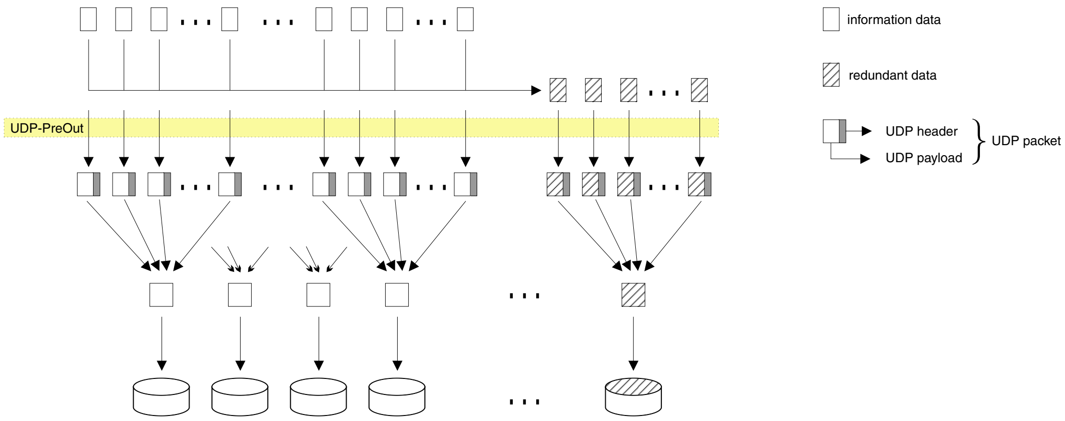

Figure 5: Writing data to disk.

three fields is calculated, (3) the value of the UDP checksum field is updated by adding the checksum of the three new field values to the precalculated UDP checksum, and (4) the UDP packet is handed over to the IP protocol. Consequently, the server side UDP protocol operations are reduced to a minimum at transmission time.

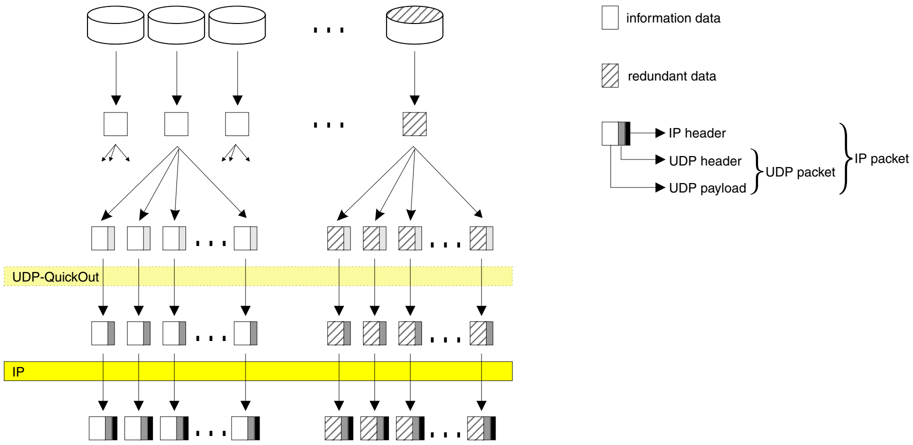

Figure 6: Retrieving data from disk.

2.5 Checksum Operations

Originally, the UDP checksum is calculated over the three areas displayed in Figure 7: a 12 B pseudo header containing fields from the IP header, the 8 B UDP header, and the UDP data. These values are contained in the `udpiphdr` structure, as depicted in Figure 8, which holds the UDP and IP headers. The pseudo header values are inserted into the IP header fields. The `udpiphdr` will be placed in front of the first mbuf structure to enable prepending of lower-layer headers in front [16], and the traditional `in_cksum()` is called with a pointer to the first mbuf containing the UDP packet to calculate the 16 bit transport level checksum.

The checksum procedure is in our NLF mechanism divided into two parts. In the checksum procedure in `udp_PreOut()`, we precalculate the checksum over the known header fields and the UDP payload, and the checksum procedure in `udp_QuickOut()` updates the stored checksum value with

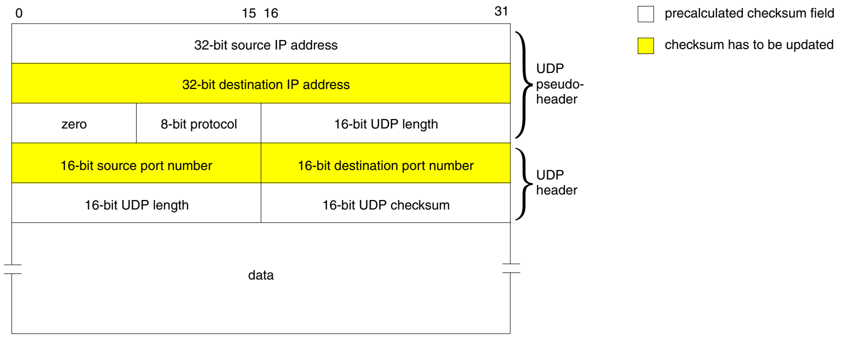
0
15 16
31
precalculated checksum field
checksum has to be updated
32-bit source IP address
32-bit destination IP address
zero
8-bit protocol
16-bit UDP length
UDP pseudo-header
16-bit source port number
16-bit destination port number
16-bit UDP length
16-bit UDP checksum
UDP header
data

Figure 7: Used fields for UDP checksum computation.

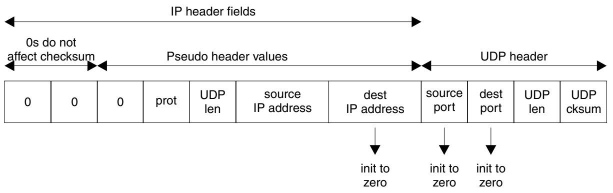

Figure 8: `udpiphdr` structure with the fields used for checksum.

the checksum of the new fields (shaded in Figure 7) in the pseudo and UDP headers. The two checksum operations are described next.

2.5.1 Precalculating Checksum

The `udp_PreOut()` function generates UDP packets with (precalculated) checksums (see Figure 5), but to use the prefabricated packet for transmission from server to client, we have to fill in the corresponding source and destination IP addresses and port numbers. However, most of these values are unknown when storing the data. The source IP address will be the server's IP address, but the source port number will be allocated dynamically at transmission time, and the client's destination IP address and destination port number are also unknown. Therefore, these unknown fields are initialized to zero (see Figure 8) before the checksum is calculated using the traditional `in_cksum()`, because these fields will then not affect the checksum.

2.5.2 Updating Checksum

In order to transmit data from server to clients, the prefabricated UDP packets are read from disk and then processed through the UDP/IP communication protocols using `udp_QuickOut()` instead of the traditional `udp_output()` at the transport layer. Since the `udpiphdr` structure is placed in front of the first mbuf in the chain of mbufs holding the UDP packet data, the checksum procedure can be simplified as shown in Figure 9. We only have to update the stored checksum with the new port numbers and destination IP address, and because these fields were initialized to zero before calculating the stored checksum, the value of these fields are just added to the stored checksum value. We do not need to check whether there are more mbufs containing data or whether there are words spanning between the mbufs. Finally, after the checksum has been updated, the `udp_QuickOut()` proceeds the protocol processing as the traditional `udp_output()`.

```
int in_QuickCksum(struct mbuf *m)
{
  u_short *w;
  int sum;

  /* Some more declarations */
  ....

  /* return a pointer to the data associated with an mbuf,
     and cast the pointer to the specified type, i.e., u_short */

  w = mtod(m, u_short *);

  sum = ~w[13]; /* Stored checksum value */
  sum += w[11]; /* Destination port */
  sum += w[10]; /* Source port */
  sum += w[9];  /* Destination IP address, last part */
  sum += w[8];  /* Destination IP address, first part */

  REDUCE;
  return(~sum & 0xffff);
}
```

Figure 9: Checksum calculation procedure used in `udp_QuickOut()`.

2.6 NLF and Other Fast Media Delivery Approaches

In the last decade, there has been performed a lot of work in the area of fast delivery of multimedia data. Disk arrays and parallel disks in general are used to increase the overall storage system throughput, whereas different data placement techniques are suggested to increase the performance of a single disk. To avoid or minimize disk accesses, several caching and prefetching algorithms are proposed, and memory copy operations are eliminated by sharing a memory region between processes or transferring the memory page to a new process. Additionally, access to a resource might be ensured or the overall throughput increased by adapted scheduling algorithms and allocation mechanisms. In [13], several of these mechanisms are discussed and summarized.

However, to the best of our knowledge, there has been no other research in the area of prefabricating network packets to reduce the communication protocol execution overhead. The IO-Lite unified I/O buffering and caching system [9] enables checksum caching, but this is only efficient if the data is accessed and transmitted frequently and not paged out. For example, in the case of HDTV data delivery, most data will be paged out, and checksum caching will give in no performance gain. Nevertheless, our NLF mechanism should be seen as a supplement rather than an alternative to other fast media delivery solutions, because it is an orthogonal technique which further increases the throughput and reduces the resource consumption per stream.

3 Expected Performance Gain

By integrating the error management schemes in the storage and the communication systems (reusing storage parity information for FEC in the network), we do not have to handle retransmission overhead in a multicast scenario on the server side, and the time consuming execution of the complex parity encoding algorithms are omitted. Thus, as our experimental evaluation shows [5], all the network error recovery operations are performed with a sufficient throughput to present data from a high data rate multimedia stream at the client side. The only server side costs, depending on the required recovery capability, are an increased requirement of buffer size and bandwidth.

Data touching operations like checksum calculation are addressed as one of the major time consumers in the end-systems [3, 7]. For example, in a performance measurement described in [6], the processing overhead of data touching operations of the UDP/IP protocol stack is 60 % of the total software processing time. In [10], 370 μs were used per KB to process the packets through TCP/IP calculating the checksum. To reduce this data touching cost, we store the UDP packet on disk including output from the data touching, CPU intensive checksum calculation. The gain of our checksum precomputation depends on the packet size. In our scenario using packet sizes of 1 KB, 2 KB, 4 KB, and 8 KB and updating only the source port (16 bit), destination port (16 bit), and destination IP address (32 bit), the on-line checksum procedure is executed over only 0.77 %, 0.39 %, 0.19 %, and 0.10 % of UDP data, respectively. Furthermore, as we only calculate the checksum over known position fields, all located in the same mbuf, no checks for byte swapping or odd numbers of bytes in an mbuf are needed. Thus, if 60 % of the UDP/IP processing time is spent on checksum calculation [6], we might expect a processing speed-up of factor two by prefabricating the UDP packets and eliminating most of the data touching overhead as we propose in our MoD server, i.e., the CPU will be available for other tasks. However, to see an approximately gain in throughput, we did a simple experiment measuring the maximum throughput in a gigabit environment, using *netperf*, between two machines using 32 KB send and receive buffers, a 1 Gbps network, and a 350 MHz processor. The result is shown in Table 1. The throughput improvement of our approach should be nearly the same[1] as the difference between the UDP measurements with and without checksum. Thus, by using NLF, we could expect a throughput gain when sending data from the server of about 10% compared to the traditional UDP protocol.

Packet size	**without checksum (Mbps)**	**with checksum (Mbps)**	**Throughput improvement removing checksum (%)**
4 KB	∼420	∼385	∼9.1
8 KB	∼450	∼405	∼11.1

Table 1: Maximum measured UDP throughput.

4 Conclusions

The INSTANCE server architecture is based on three orthogonal techniques: zero-copy-one-copy memory architecture, NLF, and integrated error management. In this paper, we presented our design of the NLF mechanism, and in particular, we described how packets are stored on disk and later retrieved for transmission to remote clients. We expect that our light-weight, transmission time UDP protocol, i.e., the `udp_QuickOut()` function, will reduce the communication protocol processing overhead making resources available for other tasks or more concurrent users.

At the time of writing, the NLF design is finished, and we are about to start implementing our design in NetBSD. To summarize our preliminary experiments, the NLF mechanism will increase the throughput of about 10% and most of the CPU time spent on calculating the checksum will be available for other streams or additional tasks. Furthermore, the design of the other two INSTANCE techniques, i.e., zero-copy-one-copy memory architecture and integrated storage and communication system error management, are also finished. Our ongoing work in INSTANCE is concerned with the implementation of these two techniques in our prototype. We are currently implementing different zero-copy architectures to evaluate and determine which are best suited for our purposes. Moreover, we have experimentally evaluated several possible recovery schemes for the integrated error management scheme, and we found the Cauchy-based Reed-Solomon erasure correcting code suitable [5].

[1] It will not be exactly the same, because we still have to update the checksum. The checksum is not completely removed, but only reduced to a minimum.

References

[1] Braden, R., Borman, D., Partridge, C.: *“Computing the Internet Checksum”*, RFC 1071 (Updated by RFC 1141 and 1624), September 1988

[2] Buddhikot, M. M: *“Project MARS: Scalable, High Performance, Web Based Multimedia-on-Demand (MOD) Services and Servers”*, PhD Thesis, Sever Institute of Technology, Department of Computer Science, Washington University, St. Louis, MO, USA, August 1998

[3] Clark, D. D., Jacobson, V., Romkey, J., Salwen, H.: *“An Analysis of TCP Processing Overheads”*, IEEE Communication Magazine, Vol. 27, No. 2, June 1989, pp. 23 - 29

[4] Halvorsen, P., Plagemann, T., Goebel, V.: *“The INSTANCE Project: Operating System Enhancements to Support Multimedia Servers”* (poster), WWW-site for the 17th ACM Symposium on Operating Systems Principles (SOSP’99), Kiawah Island, SC, USA, December 1999

[5] Halvorsen, P., Plagemann, T., Goebel, V.: *“Integrated Storage and Communication System Error Management in a Media-on-Demand Server”*, Technical Report I-2000.018-R, UniK, March 2000

[6] Kay, J., Pasquale, J.: *“The Importance of Non-Data Touching Processing Overheads in TCP/IP”*, Proceedings of the ACM Conference on Communications Architectures, Protocols and Applications (SIGCOMM’93), San Francisco, CA, USA, September 1993, pp. 259-268

[7] Kay, J., Pasquale, J.: *“Profiling and Reducing Processing Overheads in TCP/IP”*, IEEE/ACM Transactions on Networking, Vol. 4, No. 6, December 1996, pp. 817-828

[8] Ousterhout, J. K.: *“Why Aren’t Operating Systems Getting Faster As Fast As Hardware?”*, Proceedings of the 1990 USENIX Summer Conference, Anaheim, CA, USA, June 1990, pp. 247-256

[9] Pai, V. S., Druschel, P., Zwaenepoel, W.: *“IO-Lite: A Unified I/O Buffering and Caching System”*, Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI’99), New Orleans, LA, USA, February 1999, pp. 15 - 28

[10] Papadopoulos, C., Parulkar, G. M: *“Experimental Evaluation of SUNOS IPC and TCP/IP Protocol Implementation”*, IEEE/ACM Transactions on Networking, Vol. 1, No. 2, April 1993, pp. 199–216

[11] Patterson, D. A., Gibson, G., Katz, R. H.: *“A Case for Redundant Arrays of Inexpensive Disks (RAID)”*, Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, IL, USA, June 1988, pp. 109 - 116

[12] Plagemann, T., Goebel, V.: *“INSTANCE: The Intermediate Storage Node Concept”*, Proceedings of the 3rd Asian Computing Science Conference (ASIAN’97), Kathmandu, Nepal, December 1997, pp. 151-165

[13] Plagemann, T., Goebel, V., Halvorsen, P., Anshus, O.: *“Operating System Support for Multimedia Systems”*, The Computer Communications Journal, Elsevier, Vol. 23, No. 3, February 2000, pp. 267-289

[14] Reddy, A. L. N., Wyllie, J. C.: *“I/O Issues in a Multimedia System”*, IEEE Computer, Vol. 27, No. 3. March 1994, pp. 69-74

[15] Stevens, W. R.: *“UNIX Network Programming, Volume 1, Networking APIs: Socets and XTI”*, 2nd edition, Prentice Hall, 1998

[16] Wright, G. R., Stevens, W. R.: *“TCP/IP Illustrated, Volume 2 - The Implementation”*, Addison-Wesley, 1995