Minimizing Protocol Processing in Multimedia Servers
– Implementation and Evaluation of Network Level Framing –

Pål Halvorsen[*], Thomas Plagemann, and Vera Goebel
Department of Informatics, University of Oslo, P.O. Box 1080, Blindern, N-0316 OSLO, Norway
{paalh, plageman, goebel}@ifi.uio.no

Abstract

Data servers for multimedia applications like News-on-Demand represent a severe bottleneck, because a potentially very high number of users concurrently retrieve data with high data rates. In the Intermediate Storage Node Concept (INSTANCE) project, we develop a 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 calculation in communication protocols are avoided by applying three orthogonal techniques: network level framing (NLF), zero-copy-one-copy memory architecture, and integrated error management. In this paper, we describe how to minimize the transport level protocol processing using NLF. In particular, we look at how NLF is implemented, and we present performance measurements indicating a large performance gain. The protocol execution is minimized to about 450 cycles per packet regardless of packet size, i.e., a reduction of about 87 % compared to 1 KB packets and more using larger packets. Consequently, the total server-side processing overhead is decreased by at least 50 %.

1 Introduction

In the last decade, there has been large growth in interest in the Internet and the World Wide Web (WWW). The number of users is rapidly increasing, and the same trend will probably continue in the future. At the same time, the availability of high performance personal computers and high-speed network services has increased the use of distributed multimedia applications like News-on-Demand (NoD), Video-on-Demand (VoD), internet protocol (IP) telephony, video conferencing, distributed games, digital libraries, and asynchronous interactive distance education. These kinds of applications have become very popular and will be an important part of the future, network-connected information society.

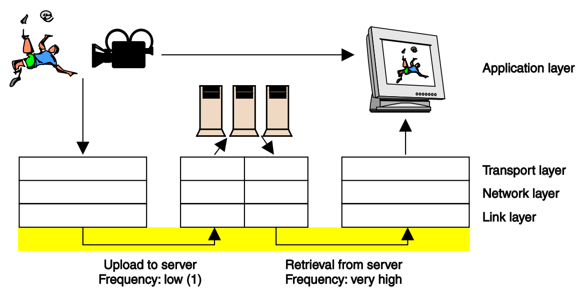
However, despite the rapid advances in hardware technology, operating systems and software in general are not improving at the same speed [16]. Due to this speed mismatch, traditional operating systems provide inadequate support for large scale MoD server applications. Providing services like playback of video and audio to a potentially large number of concurrent users for a rapidly growing class of I/O-intensive applications requires careful management of system resources. One of the main problems is transferring data from disk to network through the server's I/O data path, i.e., from disk to the buffer cache in the file system, from the buffer cache to the server application memory area, from the application to the communication system memory where network packets are generated, and from the communication system to the network card. In this data path, there are several factors that strongly limit the overall system throughput, e.g., disk I/O, bus speed, memory bandwidth, and network capacity. Each subsystem uses its own buffering mechanism, and applications often manage their own private I/O buffers. This leads to repeated cross domain transfers, which are expensive and cause high central processing unit (CPU) overhead. Furthermore, multiple buffering wastes physical memory, i.e., the amount of available memory is decreased, reducing the memory hit rate and increasing the number of disk accesses.

In the *Intermediate Storage Node Concept* (INSTANCE) project [18], 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. To avoid possible bottlenecks, the key idea of INSTANCE is to improve the server performance by combining three orthogonal techniques in a new architecture [10]:

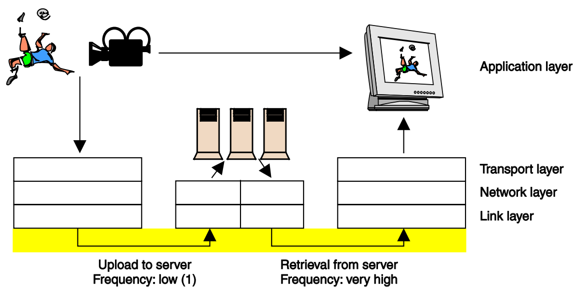
- A *zero-copy-one-copy memory architecture* removing all physical memory copy operations and sharing one single data element between all concurrent clients.
- An *integrated error management scheme* combining error management in both storage and communication system removing costly forward error correcting encoding operations.
- A *network level framing (NLF) mechanism* reducing communication protocol processing overhead.

This paper focuses on the implementation and evaluation 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 basic design is presented in [9], and in this paper, we provide implementation details and performance results.

[*]This research is sponsored by the Norwegian Research Council, the DITS program under contract number 119403/431



(A) Traditional server storage.



(B) NLF.

Figure 1. Traditional server storage versus NLF.

The rest of this paper is organized as follows: In Section 2, we describe the realization of the NLF mechanism, and Section 3 evaluates the mechanism with respect of performance. In Section 4, some related work is presented, and we summarize and conclude the paper in Section 5.

2 Network Level Framing

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., once for each client. Measurements described in [25] show that the sender latency is dominated by the transport level checksum operation, i.e., most of the time is consumed when reformatting data into network packets and calculating the checksum. This operation is repeated for each client and is wasteful, because an identical sequence of packets might be created each time – differing only in the destination IP address and port number fields. A logical approach to reduce this overhead is to create this sequence of packets once, store it on disk or in memory, and later transmit the prebuilt packets saving a lot of processor resources.

2.1 Basic Idea

To reduce this unnecessary workload in the communication system protocols, i.e., performing the same operations on the same data for each packet transmitted (Figure 1A), we regard the server as an *intermediate node* in the network where only the lower layers of the protocol stack are processed (Figure 1B). When new data is sent to the server for disk storage, only the lowest two protocol layers are executed, 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.

As shown in Figure 1B, we use the idea of asynchronous packet forwarding in the intermediate network nodes, and our multimedia storage server is considered as an intermediate storage node where the upper layer packets are stored on disk. The intention of NLF is to reduce the overhead of CPU intensive, data touching operations in the communication protocols like the checksum calculation and thereby increase the system performance. Since ARQ-based schemes are not suitable for our multicast and real-time environment [23], we use UDP as the transport level protocol to transmit data from the server to the clients.

2.2 Implementation

To be able to test and evaluate the NLF mechanism, we have designed and implemented a prototype of the NLF mechanism in NetBSD. The prototype is integrated with an in-kernel zero-copy data path between the storage system and the communication system [10] which means that no copy operations is performed in memory. Thus, data is placed directly in a memory area shared by the `buf` structures [15] used by the file system and the `mbuf` structures [26] used by the communication system. Since the application does not touch data, no pointer to the in-kernel data area is provided.

The NLF implementation differs slightly from the basic idea, because the header of the incoming packet uploading data on the server will be incorrect, i.e., both due to address fields and possibly using a reliable protocol like TCP for data upload to the server [9]. Thus, 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, i.e., after the UDP packet has been generated and the checksum has been calculated, but before handing it over to the IP protocol.

To allow flexibility with respect to packet sizes and protocols, we store the communication system meta-data (packet headers) in a separate meta-data file. This enables the use of different protocols and packet sizes without storing the data several times, i.e., only several meta-data files are necessary to hold the packet size dependent meta-data. Therefore, when a stream is opened, this meta-data file is retrieved from disk and stored in memory (if this is a large file, we might use some kind of sliding-window technique to keep in memory only the most relevant data). During data transmission, the correct packet header is retrieved from the meta-data file according to the offset of the data file and the size of each packet. The header is put into its own mbuf, and that mbuf's next-pointer is set to the mbuf chain containing the data.

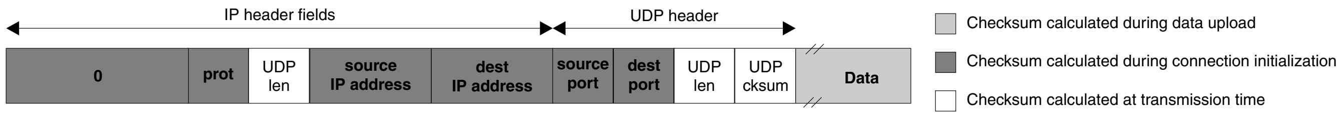

IP header fields
UDP header
0
prot
UDP len
source IP address
dest IP address
source port
dest port
UDP len
UDP cksum
Data
Checksum calculated during data upload
Checksum calculated during connection initialization
Checksum calculated at transmission time

Figure 2. Fields used for checksum computation in the `udpiphdr` structure and the data payload.

2.2.1 Version 1

To prefabricate the transport level packets, we use a special system call where a locally stored file is given to the packet prebuild routine, which generates the meta-data (packet headers) and stores them in a separate meta-data file. Furthermore, the data touching checksum operation is the main CPU-cycle consumer, so our first version of the NLF mechanism precalcuates the checksum over the packet payload only (application level data) and stores this on disk, i.e., the packet header is generated on-the-fly during transmission time. Compared to the traditional checksum procedure `in_cksum` [4] calculating the checksum over all the fields shown in Figure 2, our new on-line checksum procedure, named `in_QuickCksum`, calculate the checksum over the 28 B pseudo-header only (white and dark areas in Figure 2). If we assume packet sizes between 1 KB to 8 KB, `in_QuickCksum` is executed over 1.91 % (1 KB packets), 0.97 % (2 KB packets), 0.49 % (4 KB packets), and 0.24 % (8 KB packets) of UDP data. The time spent on checksum operations should therefore be reduced to nearly the same percentage (we must add some time for initializing the checksum function, etc., so the measured values will be slightly higher).

This design simplifies the prefabrication function and reduces the storage requirement of the meta-data file by 80 % compared to storing the whole packet header, but it is at the cost of calculating the checksum over the packet header at transmission time. Nevertheless, the performed prefabrication should still be sufficient to prove the advantages of the NLF concept.

2.2.2 Version 2

In the first version of the NLF mechanism, the packet header generation is the most time consuming operation in our modified UDP protocol. Thus, by either including the packet header in the NLF implementation (as in the basic design) or additionally using the idea of pregenerating header templates on connection setup [5, 21], the overhead of filling in header fields and calculating the header checksum can be reduced. From the generation of packet headers for a particular network connection, one can make some general observations. For example, in the 8 B UDP header, only the 2 B checksum field and 2 B packet length field will vary. Furthermore, in the generation of IP packets sent by a particular UDP connection, a 20 B header is added at the front of each packet. 14 B of this header will be the same for all IP packets, and the IP length, the unique identifier, and the checksum fields (6 B in total) will probably be different for each packet. In addition, the header might contain a variable number of options. However, most IP packets carry no options, and if they do, all packets transmitted through the same connection will likely carry the same options. Thus, most of the fields in the headers will be identical for all packets sent during a connection, with the exception of the checksum field and the packet length, so it will probably be preferable to only pregenerate a header template at stream initialization time instead of retrieving all this information from disk. The checksum for the header template could also be precalculated during stream initialization, minimizing the checksum operation in the function without storing the entire packet including header. If we precalculate the header template checksum, the template can, after performing the checksum operation (due to using the pseudo header in this operation), also include most of the IP protocol header fields, and the IP protocol processing will thereby also be minimized.

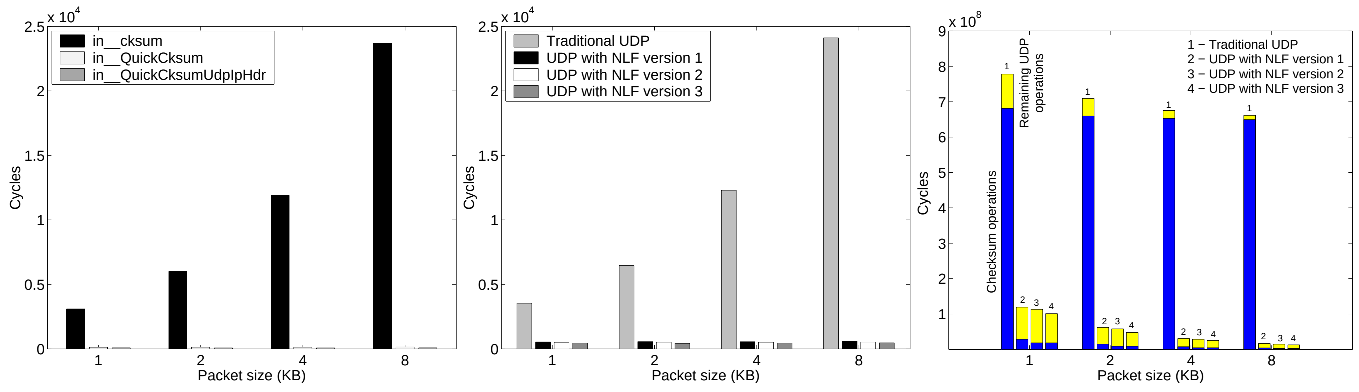
In addition to the checksum procedure in version 1, our second version of the NLF mechanism generates a header template at stream initialization time, precalculates the header checksum, and store both the checksum and a pointer to the template in the `inpcb` structure [26] for this connection, i.e., in the Internet protocol control block. This means that the checksum procedure is further reduced, i.e, only adding the (two) packet length fields, the header checksum, and the payload checksum. The on-line checksum procedure, called `in_QuickCksumUdpIpHdr`, is executed only over the white fields in Figure 2, i.e., corresponding 0.38 % (1 KB packets), 0.19 % (2 KB packets), 0.10 % (4 KB packets), and 0.05 % (8 KB packets) of UDP data. Additionally, the header template is copied into the header field in the mbuf, i.e., we do not fill in each field separately from the control block.

2.2.3 Version 3

We expected version 2 to consume a minimal amount of CPU cycles when supporting a traditional UDP service. However, our performance measurements (described next in Section 3) show that the operation of copying the template into the mbuf consumes more cycles compared to filling in the header fields at transmission time. Therefore, we combined version 1 and 2 into yet another protocol, version 3. This version uses the checksum facilities from version 1 and 2, but instead of using the header template, we fill in header fields during the transmission operation.

3 Performance Evaluation

We have performed the tests using a Dell Precision WorkStation 620 with a PentiumIII 933 MHz processor running



(A) Per packet checksum operations. (B) Per packet UDP protocol. (C) Total protocol versus checksum transmitting 225 MB.

Figure 3. Average processing overhead.

operation	size	maximum	minimum	average (cycles)	average ($\mu s^{\dagger}$)	standard deviation	Total‡	99% confidence interval	95% confidence interval
Traditional UDP	1 KB	72616	3202	3544.59	3.80	1765.95	777976456	[3279 - 4477]	[3306 - 3786]
	2 KB	68547	3218	6464.07	6.93	1852.11	709379968	[6054 - 7199]	[6096 - 6839]
	4 KB	70521	9246	12304.25	13.19	2217.50	675146716	[11746 - 26924]	[11816 - 12700]
	8 KB	75372	8906	24108.69	25.84	3057.94	661445954	[23168 - 49848]	[23286 - 24373]
UDP with NLF, version 1	1 KB	59021	421	544.10	0.58	785.60	119421191	[497 - 1067]	[497 - 670]
	2 KB	47088	437	565.09	0.61	483.71	62013914	[506 - 930]	[506 - 784]
	4 KB	33373	426	556.97	0.60	275.88	30561611	[487 - 972]	[493 - 832]
	8 KB	21644	442	598.05	0.64	226.90	16408082	[513 - 885]	[519 - 812]
UDP with NLF, version 2	1 KB	73605	304	515.98	0.55	735.65	113248822	[382 - 891]	[383 - 692]
	2 KB	52675	387	528.19	0.57	422.53	57964935	[396 - 806]	[408 - 716]
	4 KB	34353	369	522.58	0.56	282.76	28674532	[383 - 927]	[389 - 774]
	8 KB	32400	313	537.13	0.58	378.74	14736589	[383 - 852]	[395 - 723]
UDP with NLF, version 3	1 KB	60316	329	460.83	0.49	717.20	101144342	[380 - 965]	[381 - 607]
	2 KB	41473	297	433.41	0.46	376.04	47562868	[382 - 831]	[382 - 653]
	4 KB	22347	293	460.96	0.49	174.36	25293128	[381 - 787]	[382 - 656]
	8 KB	40485	358	471.30	0.51	388.26	12930708	[381 - 793]	[382 - 674]
`in_cksum`	1 KB	91828	2813	3103.73	3.33	1621.14	681217011	[2877 - 3581]	[2899 - 3314]
	2 KB	70387	2822	6010.18	6.44	1856.29	659568933	[5647 - 6498]	[5689 - 6253]
	4 KB	64954	8894	11899.73	12.75	2124.04	652949837	[11351 - 25414]	[11422 - 12160]
	8 KB	94041	8550	23674.59	25.37	3191.14	649536035	[22733 - 50358]	[22858 - 23908]
`in_QuickCksum`	1 KB	52229	106	130.55	0.14	705.17	28653746	[111 - 243]	[113 - 195]
	2 KB	46747	116	136.77	0.15	395.91	15009174	[121 - 257]	[122 - 245]
	4 KB	33788	107	138.66	0.15	341.33	7608332	[110 - 402]	[110 - 243]
	8 KB	16329	107	151.12	0.16	73.00	4146244	[110 - 264]	[110 - 243]
`in_QuickCksumUdpIpHdr`	1 KB	44880	66	84.25	0.09	134.18	18491626	[74 - 165]	[74 - 97]
	2 KB	46503	63	81.77	0.09	605.96	8973920	[66 - 149]	[66 - 133]
	4 KB	35232	64	82.48	0.09	287.80	4525627	[66 - 158]	[66 - 148]
	8 KB	16623	64	86.72	0.09	61.74	2379290	[66 - 158]	[66 - 158]

†Calculated using the cycle count and the 933 MHz clock frequency of the used CPU. ‡This column shows the total amount of cycles used for the whole 225 MB stream.

Table 1. Cycles per packet spent in the UDP protocol and on checksum operations.

NetBSD 1.5ALPHA2. To measure time (cycles) in the kernel, we have implemented a software probe using the Intel `RDTSC` instruction reading the processor cycle count and the `CPUID` instruction forcing every preceding instruction in the code to complete before allowing the program to continue. The overhead of executing this software probe is subtracted from the measured results. Finally, to see the total speed up in the amount of effective time used in the operating system kernel, i.e., the time the process really uses the CPU, we have measured the process' used kernel time. For these measurements we used `getrusage` which returns information about the resources utilized by the current process.

For each version of the protocol, we tested packet sizes of 1 KB, 2 KB, 4, KB, and 8 KB. We transmitted a 225 MB file, and we measured the time (in amount of cycles) to process each packet through the whole UDP protocol and the time to perform the checksum procedure. The per packet results is shown in detail in Table 1 and summarized in Figure 3. As we can see, our versions of the UDP protocol are faster than the traditional protocol. As the difference between version 1 and the traditional protocol shows, it is the checksum calculation over the application level data that is most time consuming in the traditional protocol, i.e., the processing costs vary according to packet size using the traditional UDP protocol and checksum. As we only compute the checksum over the packet headers in version 1, the overhead of processing the packet through the UDP protocol is approximately constant. The checksum procedure consumes about 140 cycles per packet. Precalculating the header checksum in version 2 further reduce the checksum operation by approximately 55 cycles, i.e., to about 85 cycles, compared to version 1. However, we also found that block-copying the 28 B header template takes more time compared to filling in each field in the header one-by-one[1], because the whole protocol performance gain is less than the checksum gain. We therefore implemented version 3 where we use the template header checksum, but we fill in the header fields at transmission time.

The overhead of the NLF approach is approximately constant whereas the traditional protocol overhead varies with the packet length. On average, the time to execute the checksum procedure is reduced by 97.29 % (1 KB packets), 98.65 % (2 KB packets), 99.31 % (4 KB packets), and 99.64 % (8 KB packets) when using NLF version 3 which is our fastest modified UDP protocol[2]. When comparing these experiments with the measurement of the whole UDP protocol, we observe that the results are almost identical. This is because the checksum operation is really the time consuming operation that is almost removed in NLF. However, please note that the experiments are run separately to minimize the impact of the probe, i.e., the peeks in checksum experiments might not correspond to the peeks in the respective experiment measuring the whole UDP protocol.

The relationship between the processing of the whole protocol and the checksum is shown in Figure 3C. This figure presents the total time used in the UDP protocol to transmit the 225 MB file and the amount of this time used for the checksum operation. The checksum overhead is, as also shown above, reduced using NLF, but the rest of the UDP protocol processing (displayed in the lightly-shaded area in the figure) is the same regardless of whether we use NLF (version 1 and 3) or traditional UDP, i.e., when using NLF version 2 it takes more time to block-copy the template.

Data touching operations like checksum calculation are addressed as one of the major time consuming operations in the end-systems [13]. For example, in a performance measurement described in [12], the processing overhead of data touching operations of the UDP/IP protocol stack is 60 % of the total software processing time. Thus, by minimizing the checksum overhead, we expect a processing speed-up of a factor two (already using the zero-copy data path). The measured effective CPU time, including stream initialization, is shown in Figure 4. In average we have a processing reduction of 53.10 % (1 KB packets), 54.65 % (2 KB packets), 60.90 % (4 KB packets), and 62.33 % (8 KB packets), i.e., the experienced speed-up factor is above two.

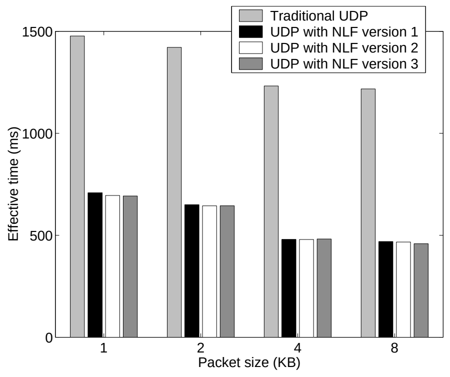


Figure 4. Effective time spent in the kernel.

We also measured the throughput using NLF, but combined with our zero-copy data path, we process data faster through the system than our Gbps network card can handle. Packets are dropped in the driver queue (using UDP). Nevertheless, assuming that the storage system and the network card are not bottlenecks, the consumed CPU time per stream indicate a throughput of about 3.65 Gbps using 8 KB packets, i.e., we can at least transmit data much faster than one Gbps.

4 Related Work

Even though the communication protocols' packet processing are not the main bottleneck in the I/O data path, improvements have been done to optimize the execution of the protocol stack. For example, ILP [6] reduces the costs related to the strict layering where data is touched in various layers and therefore data is moved between the CPU and memory many times. However, in our scenario, we have no data touching operations except transport level checksum calculation, which is also removed. In the next subsections, we therefore look at some related mechanisms that reduce the communication protocol processing overhead.

4.1 Header Templates

Clark et al. [5] have designed a better performance IP layer that uses the same observations made in Section 2.2.2 and creates a template of the header with the constant fields completed. Thus, transmitting a TCP packet on a particular connection involves calling the IP protocol with the template and the packet length. The template will then be block-copied into the space for the IP header where the non-constant fields are filled in. The idea of pregenerating header templates has also been used with TCP. Saltzer et al. [21] designed the TCP protocol to support remote Telnet login where the entire state including unsent data on the output side is stored as preformatted output packets. The Peregrine

[1]We also tested the performance of the copy procedure by implementing a hand-optimized assembly routine. However, the performance gain is minimal, i.e., the measurements show an average reduction of only 1 - 2 cycles per header template copy operation using the hand-optimized routine which indicate that the native copy operation is highly optimized.

[2]Since we must add time for initializing the checksum procedure, defer carries, and cast the value to an `u_int16_t`, this performance gain is as expected compared to the reduction in number of operations as outlined in Section 2.2.2.

remote procedure call (RPC) system [11] also uses pregenerated header templates over the RPC, IP, and ethernet headers. This reduces the cost of sending a packet to a few lines of code. However, as our measurements show, the block-copy operation is expensive, and the performance gain may be minimal (and may be even worse) using only header templates. The performance gain is depending on the amount of pre-executed operations, i.e., header templates comprising all the protocol headers in the protocol stack may be efficient as there are less updates in the IP header compared to the UDP header, and the ethernet header may not change at all for the period of the bind time [11].

4.2 Precalculating Checksums

Checksum calculations are known to be a time consuming operation. In [13], the overhead of checksum computations is removed by turning off checksums when it is redundant with the cyclic redundancy check (CRC) computed by most network adapters. However, removing the checksum is an unfortunate solution, because this may increase the probability of receiving damaged packets. Even though the link level CRC should catch more errors compared to the transport level checksum, corrupted packets are frequently not detected by the CRC [24]. This is due to transmission errors in the end-systems or in the intermediate nodes due to hardware failures or software bugs. Furthermore, some systems allow previously calculated checksums to be cached in memory. In [14], a similar approach to NLF is presented where video is preformatted and stored in the form of network packets. Precomputing and storing the headers is also proposed, and when the packets are scheduled to be transmitted, the destination address and port number are filled in, and the checksum is modified. However, the authors only describe an expected performance gain in the number of concurrent streams, because the number of instructions is greatly reduced. Furthermore, as the whole packet is stored as one unit, all streams must use one predefined packet size. In contrast, by storing the packet data in a separate meta-data file, our design enables several packet sizes without storing the data elements more than once. Moreover, the IO-Lite unified I/O buffering and caching system [17] is optimized to cache the computed checksum of a buffer aggregate, and if the same data is to be transmitted again, the cached checksum can be reused. In [20], caching of prebuilt transport level packets, in both end-system and network nodes, is used for load balancing and to reduce resource usage. This checksum caching 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 due to high data rates, and this type of caching will give no performance gain. However, caching in network nodes to enable other data distribution schemes, is an issue of future work.

4.3 On-Board Processing

Another approach to reduce communication system overhead is to use special hardware on the network adapters. Some early approaches include afterburner [7] and medusa [2] which remove data copying and perform checksum operations during programmed I/O data transfers to on-board memory. In the last few years, off-the-shelf network adapters come with on-board memory and CPUs, e.g., the 3Com Gigabit EtherLink card [1] based on the Tigon chip. These cards come with on-board firmware that may calculate the checksum(s) on the network adapter. Furthermore, there are several proposed solutions of how to move the whole protocol processing on-board the network adapter, e.g., [3, 8, 22], where performance issues like copying, checksumming, context switching, policing offloading, etc. are addressed. These new network adapters may therefore make the NLF mechanism unnecessary in some cases, but yet this hardware is not available for all platforms, and they have limited CPU power and amount of memory. Nevertheless, such hardware is very useful. It can replace NLF, but also be used together with NLF. Since the amount of CPU cycles are limited, there might be resource shortage if several streams perform checksum operations, encryption, and forward error correction. Doing NLF checksumming offline will therefore also save CPU cycles on the network adapter.

5 Conclusions

As advances in processor technology continue to outpace improvements in memory bandwidth and as networks support larger packets, proportionally less time is going to be spent on protocol processing and related operating system overhead. The I/O data path is the major component of communication costs and will remain so in the foreseeable future. Transferring data through the communication protocols also add costs, and using our zero-copy data path, the transport protocol processing is the most time consuming operation. In the INSTANCE project, we have designed and implemented three mechanisms removing potential bottlenecks like copy operations, redundant functionality, and communication protocol processing overhead in the operating system on the server side. This paper has especially concentrated on the NLF mechanism which almost eliminates the overhead imposed by the transport protocol.

Our results show that the server workload is significantly reduced using the INSTANCE mechanisms. The operation consuming most CPU cycles using our zero-copy data path, i.e., transport protocol processing, is minimized using NLF. The protocol overhead is reduced to about 450 cycles per packet regardless of packet size. The checksum processing overhead is of about 85 cycles, i.e., this corresponds to a reduction of about 97.29 % (1 KB packets), 98.65 % (2 KB packets), 99.31 % (4 KB packets), and 99.64 % (8 KB packets) when using NLF version 3 compared to the native UDP protocol processing. When combining NLF with the zero-copy data path, we are able to transmit data faster than the Gbps network card is able to handle. If we assume that the storage system and the network card are not bottlenecks, i.e., only the operating system processing is a bottleneck, the used CPU time indicate that we should be able to transmit data at a rate of 3.65 Gbps in a scenario using 8 KB packets and NLF version 3.

The performance experiments show that it takes more time to block-copy the template into the mbuf than filling in each field one-by-one. However, on different architectures, this might be different, but such tests are considered future work. Furthermore, as our observations show in Section 2.2.2, the IP and ethernet header will also have several identical fields during the lifetime of a connection. In the future, we will therefore also investigate the performance gain of including the IP and ethernet protocol processing into the NLF mechanism. Whether the NLF ideas will be ported to on-board processing is yet uncertain.

References

[1] 3com: *"3Com Gigabit EtherLink Server Network Interface Card (3C985B-SX) - Product Details"* http://www.3com.com/products/en_US/detail.jsp?tab=features&pathtype=purchase&sku=3C985B-SX, May 2001

[2] Banks, D., Prudence, M.:*"A High-Performance Network Architecture for a PA-RISC Workstation"*, IEEE Journal on Selected Areas in Communications, Vol. 11, No. 2, February 1993, pp. 191 - 202

[3] Beauduy, C., Bettati, R.: *"Protocols Aboard Network Interface Cards"*, Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems. Cambridge, MA, USA, November 1999

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

[5] 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

[6] Clark, D. D., Tennenhouse, D. L.: *"Architectural Considerations for a New Generation of Protocols"*, Proceedings of the ACM Symposium on Communications, Architectures and Protocols (SIGCOMM'90), Philadelphia, PA, USA, September 1990, pp. 200 - 208

[7] Dalton, C., Watson, G., Banks, D., Calamvokis, C., Edwards, A., Lumley, J.: *"Afterburner"*, IEEE Network, Vol. 7, No. 4, July 1993, pp. 36 - 43

[8] Dannowski, U., Haertig, H.: *"Policing Offloaded"*, Proceedings of 6th IEEE Real-Time Technology and Applications Symposium (RTAS 2000), Washington, DC, USA, May/June 2000, pp. 218 - 227

[9] Halvorsen, P., Plagemann, T., Goebel, V.: *"Network Level Framing in INSTANCE"*, Proceedings of the 6th International Workshop on Multimedia Information Systems 2000 (MIS 2000), Chicago, IL, USA, October 2000, pp. 82-91

[10] Halvorsen, P.: *"Improving I/O Performance of Multimedia Servers"*, PhD Thesis, Department of Informatics, University of Oslo, Norway, August 2001

[11] Johnson, D. B., Zwaenepoel, W.: *"The Peregrine High-Performance RPC System"*, Software – Practice & Experience, vol. 23, no. 2, February 1993, pp. 201 - 221

[12] 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

[13] 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

[14] Kumar, M.: *"Video-Server Designs for Supporting Very Large Numbers of Concurrent Users"*, IBM Journal of Research and Development, Vol. 42, No. 2., 1998, pp. 219 - 232

[15] McKusick, M. K., Bostic, K., Karels, M. J., Quarterman, J. S.: *"The Design and Implementation of the 4.4 BSD Operating System"*, Addison Wesley, 1996

[16] 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

[17] 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

[18] 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

[19] 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

[20] Race, N. J. P., Waddington, D. G., Sheperd, D.: *"A Dynamic RAM Cache for High Quality Distributed Video"*, Proceedings of the 7th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS'00), Enschede, The Netherlands, October 2000, pp. 26 - 39

[21] Saltzer, J. H., Clark, D. D., Romkey, J. L., Gramlich, W. C.: *"The Desktop Computer as a Network Participant"*, IEEE Journal on Selected Areas in Communications, Vol. SAC-3, No. 3, May 1985, pp. 468 - 478

[22] Shivam, P., Wyckoff, P., Panda, D.: *"EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing"*, to appear in Proceedings of the Supercomputing Conference (SC2001) Denver, CO, USA, November 2001

[23] Stevens, W. R.: *"UNIX Network Programming, Volume 1, Networking APIs: Sockets and XTI"*, 2nd edition, Prentice Hall, 1998

[24] Stone, J., Partridge, C.: *"When The CRC and TCP Checksum Disagree"*, Proceedings of the ACM conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'00), Stockholm, Sweden, August/September 2000, pp. 309-319

[25] Wolman, A., Voelker, G., Thekkath, C. A.: *"Latency Analysis of TCP on an ATM Network"*, Proceedings of the 1994 USENIX Winter Technical Conference, San Francisco, CA, USA, January 1994, pp. 167-179,

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