Improving I/O Performance of Multimedia Servers

by

Pål Halvorsen

A thesis submitted to Department of Informatics,
Faculty of Mathematics and Natural Sciences, University of Oslo, Norway
in partial fulfillment of the requirements for the degree of
"doctor scientiarum"

August 1, 2001

Abstract

In the last years, there has been a tremendous growth in the use of Internet services. In particular, the world-wide web and applications like News- and Video-on-Demand have become very popular. Thus, the number of users, as well as the amount of data each user downloads from servers in the Internet, is rapidly increasing. The usage of multimedia data to represent information in a user-friendly way is one important reason for these two developments. Today, contemporary mid-price personal computers are capable of handling the load that such multimedia applications impose on the client system. However, the potentially (very) high number of concurrent users that download data from Media-on-Demand (MoD) servers represents a generic problem for this kind of client-server applications.

In MoD servers, the data retrieval operations represent a severe bottleneck, because the clients concurrently retrieve data with high data rates. We have developed a new architecture for MoD servers that maximizes the number of concurrent clients that 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: (1) the zero-copy-one-copy memory architecture removes all in-memory copy operations and shares a single data element between all concurrent clients; (2) the network level framing mechanism precalculates the transport level checksum and thereby removes most of the communication protocol execution overhead; and (3) the integrated error management scheme removes the redundant error management functionality, i.e., eliminating the parity data encoding costs in a forward error correction scenario.

Our performance measurements show that a lot of resources within the server are freed for other tasks, i.e., enabling more concurrent clients, when using our proposed improvements for streamed multimedia data. The broadcasting scheme eliminates identical data elements in memory while keeping the start-up delay at a minimum, i.e., an unlimited number of users may retrieve data from the number of streams broadcasted from our server. Furthermore, we achieve throughputs of 1 Gbps (limited by the network card) using our zero-copy data path. The amount of CPU time is reduced by approximately 35 %. The communication protocol processing overhead is almost eliminated with network level framing which reduces the checksum procedure by at least 95 % and gives a total server speed-up of a factor of two. Finally, our integrated error management performs the parity data encoding operation off-line, and the parity information is retrieved together with the application data from the storage system at transmission time. This means that the potential encoding bottleneck is eliminated. If this operation would be performed at transmission time, our measurements show a maximum throughput of 25 Mbps in a gigabit environment. Thus, our server supports a high number of broadcasted streams to an unlimited number of clients, and the number of concurrent streams is increased by reducing the resource usage for each stream.

Acknowledgments

The road towards a doctoral degree has been experienced as long and winding and has been fraught with frustration, despair, pain, and *joy*. The support I have received during these years have made this journey rewarding, and I will hereby try to thank and acknowledge the people who helped me and shared this memorable time. I would never have been able to complete this challenging task without their encouragement.

First and foremost, I would like to thank my advisers Professor Dr. Thomas Plagemann and Professor Dr. Vera Goebel for their constant support, inspiration, enthusiasm, and guidance. They have provided an instructive environment which taught me a lot. They helped me present my ideas in a way that is clear, concise, and readable, and they always had time for discussions.

Many thanks to my fellow students for their friendship during this time. Especially, I would like to acknowledge Ketil Lund – my next door neighbor at UniK. We have had many valuable conversations and discussions. Ketil also pulled me out of the office for lunch every day (which I usually forgot if he was not around).

I also wish to thank Dr. Denise Ecklund and Dr. Carsten Griwodz for doing the job of proof reading my whole thesis together with Thomas and Vera. Ketil Lund, Tom Kristensen, and Åge Kvalnes also read selected chapters. They all did an excellent job and gave me a lot of constructive feedback. Additionally, I thank the committee for taking time to read my thesis and to learn about my work.

Being a PhD student at UniK (Center for Technology at Kjeller), University of Oslo, has been a stimulating experience. It is a very nice environment to work in, and I appreciate all the assistance I have had from Nina, Hellfrid, Kathy, Ivar, Anja, Gerd, and Kristin. Thanks also to Trond and the drift group for providing a lot of assistance with the equipment I used in my work.

Last and most important, I would like to thank my family to whom this thesis is dedicated. My wife Ann Kristin has been a constant source of support and encouragement, and her love has improved the quality of life. My son Aleksander moved my mind to something completely different than the PhD work, and he brings me great joy and happiness. Finally, I would like to thank my parents and my brother. I would not be where I am today without the support of my family.

Pål Halvorsen
UniK - Center for Technology at Kjeller
Department of Informatics
University of Oslo
August 1, 2001

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Distributed multimedia systems and applications presenting audio and video data are still in their infancy, but they already play an important role and will be one of the cornerstones of the future information society. A Media-on-Demand (MoD) server stores multimedia data like video and audio and offers playback services to remote clients. Thus, the data retrieval operations represent a severe bottleneck, because multiple clients concurrently retrieve data with high data rates. In this thesis, we aim to optimize the input/output (I/O) data path of multimedia servers and thereby improve their I/O performance.

1.1 Motivation and Background

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 increasing rapidly, 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.

Seen from the users' point of view, these applications can support two types of communications: *synchronous* and *asynchronous*. Synchronous communication is often called real-time communication, because information is exchanged between users in real-time, i.e., all users using the distributed application can be geographically separated, but must use the application at the same time. Asynchronous communication allows indirect information exchange, i.e., users can be separated in time and space. The information is thus often stored on a server, and the information is then retrieved asynchronously by a client or information consumer.

However, despite the rapid advances in hardware technology, operating systems and software in general are not improving at the same speed [109]. 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.

1.2 The INSTANCE Project

The work presented in this thesis is performed in the context of the *Intermediate Storage Node Concept* (INSTANCE) project[1] [70, 128]. In this project, we focus on an MoD application scenario as shown in Figure 1.1 where data is asynchronously transmitted between information provider and information consumer. This means that we put an MoD server with a persistent storage device somewhere between the data capture devices and the client systems that will request the data asynchronously. Our MoD server should be applicable for small and medium scale companies like city-wide cable companies or pay-per-view companies. Whether our approach is beneficial for large scale, world-wide systems is subject to future research.

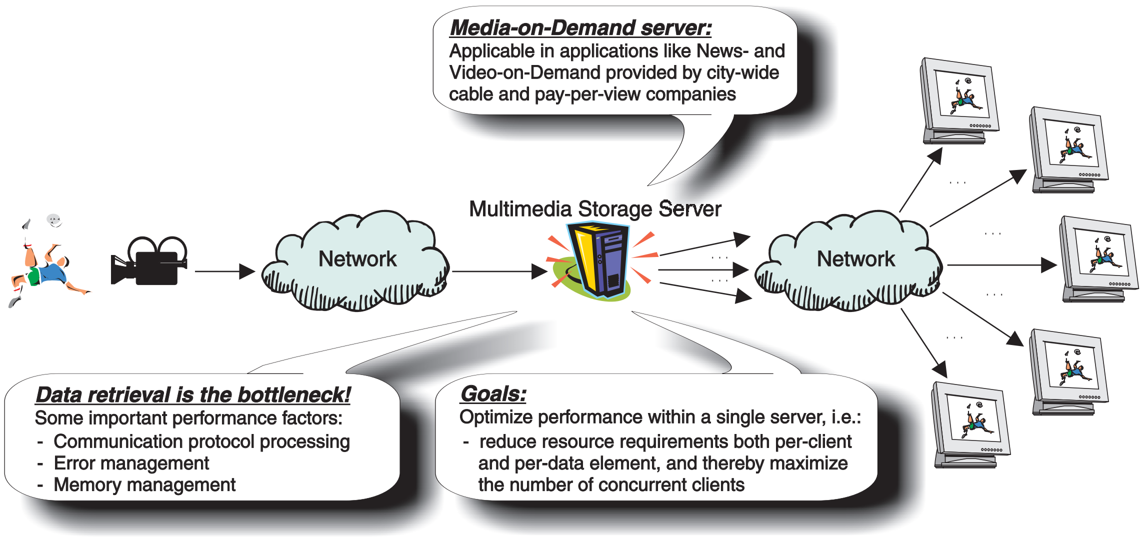


Figure 1.1: Application scenario.

The operation of fetching data from disk and sending it out through the communication system to the network is the main bottleneck for the MoD server in a multi-user scenario. The objective of our project is to minimize the overhead in the common case operation of storage nodes, i.e., the data retrieval. We identify and eliminate the possible bottlenecks of the existing operating systems by improving the design of traditional server-based systems. However, the potentially (very) high number of concurrent users that download data from the MoD servers represents a generic problem for this kind of client-server applications. It is a well known fact, that *operating systems are not getting faster as fast as hardware* [109], and that commodity operating systems represent the major performance bottleneck in MoD servers. The crucial issues that contribute to this situation include copy operations, context switches, multiple copies of the same data element in main memory, and checksum calculation in communication protocols [129]. There are basically three orthogonal approaches for this problem:

1. Develop an architecture for a single server that makes optimal use of a given set of resources, i.e., maximize the number of concurrent clients a single server can support.

[1] This research is funded by the Norwegian Research Council, the Distributed IT Systems program under contract number 119403/431.

2. Combine multiple single servers, e.g., in a server farm or cluster, to increase the amount of resources and thereby scale up the number of concurrent users.

3. Make use of multiple single storage nodes in the distribution infrastructure as proxies.

We concentrate on the first approach developing a new server where we make optimal use of a given set of resources. We especially look at the bottlenecks pointed out above, and thereby try to minimize resource requirements both per-client and per-data-element. By reducing the resource requirements, we hope to increase the number of clients supported concurrently by our MoD server. 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. In the next section, we therefore look at some of the limitations of traditional I/O systems.

1.3 Limitations of Traditional I/O Systems

Retrieving data from disk in a server, transmitting it through the server operating system, and onto the network for reception at a remote client is a costly operation, especially in a multi-user scenario where each client requests a high data rate stream. Thus, moving data between different protection domains along the I/O-pipeline and processing them through the different subsystems and communication protocols is very time consuming. Some of the bottlenecks are:

- Data is retrieved from disk and copied several times between different memory address-spaces. This copying is not a hardware constraint, but it is imposed by the system's software structure and its interfaces.

- Different parts of the end-system, i.e., the server and client machines, may add processing costs. These overheads include data touching operations, like checksum calculations and encryption, and non-data touching operations, such as network buffer manipulation, protocol specific processing, operating system functions, and error checking.

- There is redundancy in the functionality of different subsystems. Buffer management is performed by both the file system and the communication system, and error management is performed by both a disk array controller and the communication system.

- A lot of concurrent users might request the same data, i.e, one single data element is processed by the same subsystem several times.

Thus, the traditional I/O system is an end-system bottleneck for supporting high bandwidth, networked multimedia applications. Next, we state some claims based on system analysis and literature work, and we give a short overview of our contributions, before we end the chapter with the thesis outline.

1.4 Claims

This thesis focuses on mechanisms to increase the I/O performance of an MoD server running on a personal computer (PC) or a workstation. To do so, we have looked at the system functionality and how it works to find potential bottlenecks. Based on this analysis and some literature work, we state the following claims for a server system with concurrent users in the context of an MoD application:

Claim 1: *Error management is a potential, but removable, bottleneck.*
Error management should be applied to deliver a correct set of data to the clients in order to have a reliable service. Existing retransmission-based schemes usually require a large amount of buffer space to hold unacknowledged data in memory in case of retransmissions, and in the case

of multi-user scenarios, the processing of large amounts of feedback information is very resource consuming. On the other hand, forward error correction (FEC) based schemes require costly parity encoding operations for each transmitted stream on the server side. Thus, both schemes impose large overheads on the server.

Claim 2: *Memory copy operations are a potential, but removable, bottleneck.*
Different buffering mechanisms and separate address spaces for different subsystems are used by the operating system, for instance, to support different functionality and to achieve protection and security. Regardless of the various benefits, moving data from the kernel memory region to the application buffer in user space, and vice versa, are expensive operations for various reasons. For example, the bandwidth of main memory is limited, and every copy operation is effected by this. Additionally, a lot of CPU cycles are consumed for every copy operation, and data copy operations affect the CPU cache(s).

Claim 3: *Concurrent clients represent a potential, but removable, bottleneck.*
Traditionally, each concurrent client requires its own set of the system resources, and when providing popular services accessed by a (very) large number of users, there might be shortage in the server capacity to provide services to all clients. On an overloaded machine, we can experience client reneging due to long startup delays waiting for available resources or service refusal by the server.

Claim 4: *Transport level checksum operations are a potential, but removable, bottleneck.*
The transport level communication protocol ensures end-to-end data delivery. However, 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. Data touching operations making data flow through the CPU constitute most of the processing time sending data from disk to network in a server machine.

Claim 5: *By removing the mentioned bottlenecks, the operating system is no longer a critical component.*
Traditional memory management, communication protocol processing, and error management represent some of the most time consuming and resource exhausting elements within the server side operating system. Removing these operations, the operating system will be able to process data faster through the system than today's standard off-the-shelf hardware devices can handle, e.g., faster than a disk can deliver data to the operating system and faster than the network card can receive and forward to the network.

1.5 Contributions

The key idea of this thesis is to satisfy and prove the claims above by avoiding the mentioned bottlenecks. Thereby we improve the server performance and reduce the resource requirement both per-data-element and per-client. This is achieved by combining the following three orthogonal techniques in a new architecture:

- *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, i.e., the error management functionality is redundant. In INSTANCE, we integrate the error management in a disk array and a FEC scheme in the communication system. Opposed to the traditional data read from a redundant array of inexpensive disks (RAID) system where the parity information is only read when a disk error occurs, we retrieve also the redundant error recovery data. All data is passed over to the communication system, where the parity information from the RAID system is reused 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.

- *Zero-copy-one-copy memory architecture*: Memory copy operations have been identified as a bottleneck in high data rate systems. Several zero-copy architectures [129] removing physical data copying have been designed to optimize resource usage and performance using shared memory, page remapping, or a combination of both. These approaches reduce resource consumption of individual clients, but concurrent clients requesting the same data require their own set of resources. Traditional broadcast or multicast is an easy way of dealing with per client resource allocation, but start-up delay might be a problem. To minimize the start-up delay, a couple of broadcast partitioning schemes are proposed [129]. 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.

- *Network level framing (NLF) mechanism*: To reduce communication protocol processing workload, we regard the server as an *intermediate node* in the network where only the lower layers of the protocol stack are processed. When new data is sent to the server to be stored on disk, our idea is to only process the packet through the lowest three protocol layers and store the resulting transport protocol packet on disk. When data is requested by remote clients, the transport level packet is retrieved from disk, the destination port number and IP address are filled in, and the checksum is updated (only the new part for the checksum, i.e., the new addresses, is calculated). Thus, the end-to-end protocols which perform the most costly operations in the communication system, especially the transport level checksum, are almost completely eliminated in our MoD server.

As one can see, our solutions do not consist of only new mechanisms. Some are already well known in the field of computer science. However, there are no existing designs integrating the mechanisms as we propose today, and we have combined and tested them in one integrated system which, to the best of our knowledge, is unique.

To see the performance gain of our mechanisms, we have implemented a prototype in NetBSD[2]. The integrated error management frees the storage node from all resource intensive error management tasks by using precomputed parity data. Furthermore, the FEC decoding time at the client is relative to the number of network errors, and a worst-case simulation shows that the code is capable of decoding a 3.5 Mbps video in Digital Versatile Disk (DVD) quality on several machines and architectures. The removal of copy operations reduces the amount of needed memory and CPU resources, and minimizes the time to transmit data. We show that the zero-copy data path increases the number of concurrent clients that can be supported, and the NLF mechanism further reduces the resource requirement per-stream. Combined with our in-kernel data path, the kernel time, i.e., the amount of time spent on the CPU by the operating system kernel itself, is reduced by 66.18 - 75.95 % depending on the transport level packet size.

In summary, we show that the INSTANCE approach can improve the I/O performance of a single storage node at least by factor of two. Our architecture, which combines these three techniques, appears to be a unique and leading solution to the I/O performance problem in MoD scenarios.

1.6 Outline

The rest of this thesis describes the design, implementation, and evaluation of our proposed mechanisms to improve performance of an MoD server, and is organized as follows:

[2]NetBSD is a derivative of 4.4 Berkeley Software Distribution (BSD) operating system and is chosen because the source code is freely available, the kernel is continuously under development, there are a high number of users, and the BSD source code tree is well documented (for example in [100]).

- Chapter 2 gives a brief overview of multimedia applications and some essential background on multimedia systems.
- In Chapter 3, we provide some background information and a short state-of-the-art summary of existing mechanisms dealing with the bottlenecks addressed in our project, i.e., error management, memory management, and communication protocol processing.
- The design of our mechanisms is presented in Chapter 4 where we use some of the previous designs as a starting point. We also address implementation issues and show how we have realized the concept in NetBSD.
- Chapter 5 describes the performance measurements carried out on our implemented mechanisms. We first look at each mechanism by itself, before we analyze the total server speed-up using a combination of the proposed system enhancements.
- In Chapter 6, we summarize and conclude our results and outline directions for future research.
- For the interested readers, we include two appendixes. They contain details about the testbed and the implementation details of the measurement probe, and we present the results from some initial performance measurements used to verify bottlenecks and to estimate performance gained by removing the bottleneck. Thus, the appendixes are a supplement and should be used for further information.
- Finally, there is an appendix defining prefixes and abbreviations used in this thesis.

In Chapters 3, 4, and 5, we have chosen to structure the content in accordance to the three mechanisms, i.e., one section for each mechanism, and at the end of each chapter, we look at how different parts of the system can be integrated.

Chapter 2

Multimedia Systems

The explosive growth and popularity of the WWW has greatly enlarged the number of users, and it has also provided a glimpse of the computer systems and dominate applications of the future. Image-based content has become an integral part of the user interface, but the use of multimedia data like audio and video is still in its infancy [146]. In the future, multimedia applications will be an important part of the information society spanning from e-commerce via interactive games and on-Demand services to distance learning. This revolution in multimedia applications will, however, not become a reality unless the supporting technology develops at the same speed.

In this chapter, we give a short outline of typical multimedia application requirements and how to support such services. Next, we look at different types of multimedia applications, and then we describe the requirements a multimedia stream imposes on a system in Section 2.2. Section 2.3 describes typical access patterns. In Section 2.4, we look at the multimedia environment in particular. Section 2.5 points at some research issues, and we summarize the chapter in Section 2.6.

2.1 Multimedia Applications

Multimedia applications exhibit a wide variety of characteristics, and can be classified based on several different criteria. For instance, such applications can be either *synchronous*, such as videoconferencing, or *asynchronous*, like NoD or VoD. The asynchronous applications can be further classified according to the degree of possible user interaction, leading to a range of different application classes [28, 146]. At the one end, we have the non-interactive applications like broadcast video in a pay-per-view service. This application is used by broadcasting companies, as a substitute for the ordinary video player, and for the end-users no interaction is possible. All playout control is performed by the server. MoD applications are a slightly more complex application type where a potentially large number of viewers retrieve a video over a large geographically dispersed area. Near-VoD applications allow the users to start watching at specific times by batching requests to minimize the number of concurrent streams transmitted from the server. A more advanced application is true-VoD that allows the users to start watching at any time and to perform arbitrary video cassette recorder (VCR) interactions, i.e., the server will perform separate retrieval and transmissions for each of these requests. The most advanced multimedia applications, with respect to user interaction, are the applications presenting complex multimedia objects, for example stored in a multimedia database system. Such applications include applications like Learning-on-Demand (LoD), distributed games, and virtual reality, and they are characterized by a high degree of user interaction. In games and virtual reality, the user interactions may vary between all users, i.e., the viewing of images is not sequential, and any adjacent picture may be viewed next [146]. LoD interactions include VCR-type operations and navigating between different (parts of) presentations and through complex composite multimedia documents composed of several different data streams [98].

In this thesis, we have primarily looked at MoD-like applications transmitting video and/or audio on demand as shown in Figure 1.1. Data is uploaded to a multimedia storage server and later requested and retrieved by several concurrent clients. Today, there exists several such applications. For example, broadcasting companies like NRK [199], TV2 [205], BBC [178], and CNN [180] offer NoD services with audio and video on-line, but they are best effort services, and data is presented at a poor quality. It is these kind of applications we aim to improve. Hence, in the following subsections, we will concentrate on the requirements these applications impose an a multimedia server and briefly describe how to support such services.

2.2 Multimedia Requirements

Due to real-time delivery requirements, multimedia applications are fundamentally different from traditional text-based applications. The multimedia objects have a high data rate and require much storage space. Some examples of these requirements are [151]:

- Audio in pulse code modulation coded compact disk quality uses 16 bit samples at 44.1 KHz with two stereo channels. This gives a bandwidth requirement of 1.35 Mbps and a storage requirement of 607.50 MB for one hour of audio.

- Video in the European phase alternating line standard uses 24 bits per pixel and 25 frames per second. Assuming no compression and a resolution of 640×480 pixels, this gives a bandwidth requirement of 175.78 Mbps and a storage requirement of 77.25 GB for one hour of video. Using high-definition television this requirement must be increased by a factor 5.33. However, most playback applications use a compression codec like moving picture expert group (MPEG). MPEG-1 strives for a data rate of about 1.2 Mbps whereas MPEG-2 is targeted for bit streams up to 40 Mbps. The current DVD standard uses MPEG-2 and has an average video bit rate of 3.5 Mbps and a maximum bit rate of 9.8 Mbps. After system overhead, the maximum rate of combined elementary streams (audio + video + sub-picture) is 10.08 Mbps [184, 194]. A one hour video in DVD format gives a storage requirement of about 1.54 GB assuming average bit rate.

To see the total impact of these requirements for a typical MoD application, consider for example an NoD server presenting the latest news from CNN or BBC. Assume data is presented in DVD quality with an average bandwidth requirement of 3.5 Mbps for video [184, 194] and that the most popular clip is about 3 minutes long, i.e., 78 MB video data. Suppose 1000 concurrent clients continuously retrieve data from this news clip until the clip is replaced with more up-to-date information. Imagine that the server reads and transmits 64 KB of data each time which means that this process is repeated seven times each second for every clients. If all data is held in memory for each client, 192 KB (64 KB in each subsystem, i.e., file system, application, and communication system) of memory is needed per client, which is 187.5 MB totally for all concurrent clients. Some of the data in memory may be shared in the file system buffer cache, but on the other hand, data may be queued in the communication system disabling buffer reuse making our assumptions valid. In addition to the memory requirement itself, each process of reading and sending data requires two kernel accesses (system calls), one buffer cache lookup (and a possible disk access), two copy operations, processing data through the communication protocols, and queuing each packet in the limited-sized output queue for the network card. Thus, there are several possible bottlenecks running a multi-user MoD server. When adding more clients and additional video-clips, the server workload will increase even more. Later in this thesis, we will go back to this example and see how performance and resource requirements can be optimized in this application scenario.

As we can see, storing multimedia data requires large storage space. Furthermore, data must be delivered to the user's display at its playback rate without glitches. To provide such services to a potentially large number of concurrent users for a growing class of I/O-intensive applications, Quality-of-Service (QoS) support is necessary. In this context, "QoS is a quantitative and qualitative specification of an application's requirement, which a multimedia system should satisfy in order to achieve desired application quality"[1] [95]. Based on this definition there are two aspects: applications or users specify QoS requirements, and systems provide QoS guarantees. For example:

- The throughput offered per stream must be at least the data consumption rate of the transmitted data (unless data is buffered). This is typically 3.5 Mbps for DVD quality video data. This means that all components in the data path from server to client must support the data rate of the transmitted data.

- The start-up latency must be small to avoid clients reneging, i.e., all components in the data path must use a minimum of time to process data through the system.

- To have a hiccup free presentation, there must be no jitter in the data stream. For example, a person detects a skew in the data delivery of $\pm$ 80 ms in lip synchronization, and data arrival differences between two stereo audio streams above $\pm$ 11 μs are detected [151]. Jitter between blocks within a single stream is detected likewise. Thus, to minimize the time difference in data delivery, access to resources along the data path must be guaranteed.

- The failure of a system component to deliver the requested data may lead to loss of client goodwill. Thus, reliability and availability must be ensured by replication of data in the distributed environment using a back-up system and by error correcting schemes to reconstruct any damaged data during transmission.

Even though we concentrate on improving performance, the QoS requirements are still important in our research, because they describe the requirements in terms of needed resources. However, providing resource reservations and access guarantees are beyond the scope of our work.

2.3 MoD Data Access Patterns

In most MoD applications, the access frequency of the multimedia files follows the Zipf distribution [146]. This means that a small set of the available data files are viewed by a large number of the concurrent users, i.e., some multimedia objects have a very high access frequency whereas others are less often accessed. The degradation in access frequency is often dependent on the age of the object, e.g., the top ten video rentals (in a VCR rental store or a VoD scenario) are often the newest movies, or dependent on the time of day, e.g., children's programs. Nevertheless, there are usually some data objects that are more popular than others which means that the access patterns can be used for optimization, because the same operations will be performed on the same data for all concurrent clients retrieving the same data.

Furthermore, in contrast to complex applications, like LoD where several data streams might be retrieved and synchronized per client, each client accesses one data file at a time in VoD or NoD systems. Additionally, the playout is usually from start to end without major requirements for seek, rewind, and forward operations. Although VCR operations may occur, they are few, and more importantly, they are read-only [66]. This simplifies the mechanisms needed to enhance system performance, because data is never updated (by remote clients), and we therefore restrict our investigations to non-interactive, read-only access patterns.

2.4 The Distributed Multimedia Environment

In spite of the complexity of multimedia applications, the multimedia server environment can be built from only three main components [146], i.e., the client, the network, and the server. Each component

[1] There is no generally accepted definition of QoS at the moment, and a lot of other definitions exists, e.g., [80], [165]. The important property of the term is that QoS defines characteristics that influence the user's perceived quality of an application.

consists of different subcomponents and subsystems. Figure 1.1 shows a scenario with one single server transmitting many concurrent streams from the server to different, geographically separated users. Thus, all the components along the data path must support the requested service.

The client represents the physical devices at which users playout the multimedia data, and most contemporary mid-price personal computers are capable of handling the load a multimedia application imposes on the system. The network connects the end-systems and is responsible for reliable and efficient transportation of data and control signals. The network support for high data rate multimedia streams is still not sufficient, at least not in the Internet, if the data is to be transmitted over large distances. However, the networks are improving and broadband services offering bit rates up to 10 Mbps are available to some end-users in Norway today [179]. Finally, the multimedia server itself is responsible for storing and retrieving data from the storage system and sending it to the remote clients via the network.

At the server side, there are a lot of issues to be dealt with, because, as opposed to the client system handling only one stream (or very few streams), the server deals with a large number of clients at the same time. Each client requires a high data rate, real-time service, and a challenge is therefore to support multiple users on a given set of hardware.

To increase the capacity of a single multimedia server, there are basically three approaches as described in Section 1.2: (1) optimize performance within one machine, (2) combine multiple machines into a server farm, and (3) use proxies. The two first approaches may give a server model which has

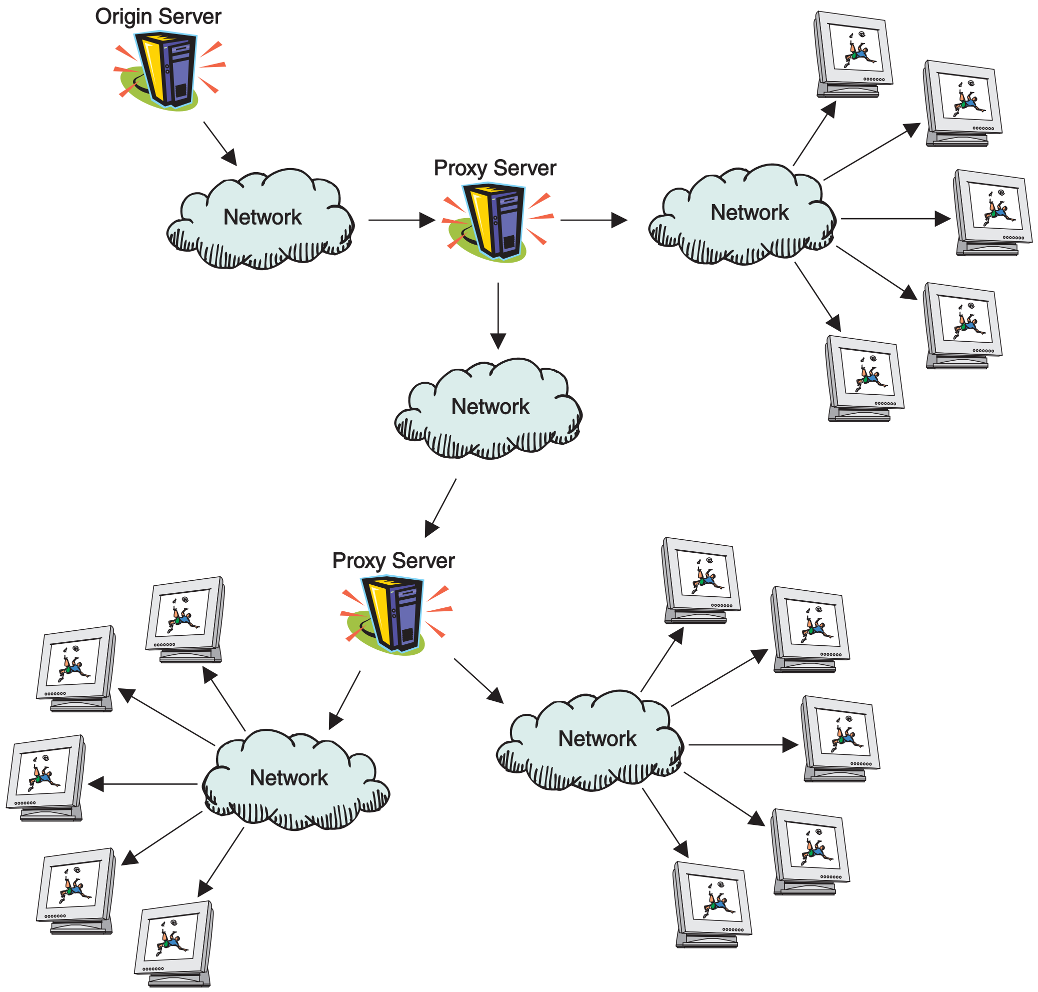


Figure 2.1: Distributed server environment.

a single point of failure. The single machine may crash, or the network connection to the server farm may break. Therefore, proxies are often used to increase the reliability and the scalability of MoD systems, i.e., use multiple machines in the distribution infrastructure (see Figure 2.1). Data is replicated and cached somewhere in the distribution chain between information provider and information consumer, i.e., the proxy offloads the origin servers by serving clients directly from the proxy with the cached data. This reduces the problem of startup latency, jitter, and data corruption caused by many hops during data transmissions over long geographical distances and increases reliability and accessibility.

2.5 Research Issues

Designing, implementing, and running a multimedia server supporting video and audio presentations is a challenging task, because of the high resource requirement and the timing constraints. Hence, there are a lot of challenges and bottlenecks to be solved before such a system can be supported in a wide-area or global context. For example, the whole system including network and end-systems must support reservations and access guarantees, and appropriate mechanisms to ensure the reservations are not violated and the timing requirements are fulfilled must be provided. The network capacity must be increased to allow multiple concurrent users to retrieve and playout a high-data rate video from a remote server, and the routers must deal with congestion with respect to real-time and non-real-time traffic. Furthermore, proxies save time and resources when transmitting data over longer distances, but there are several issues determining where to put a proxy and which data elements to cache. Should the caches cooperate or be independent of each other, and how is data uploaded to a proxy? Which request should be routed to a particular machine depending on the current (and future expected) load? In the case of overloading the network and the intermediate nodes, appropriate adaptation mechanisms should be applied, and we have to address the issue of transport control protocol (TCP) friendliness, i.e., a TCP-friendly stream is supposed to back off in a similar way as TCP in case of congestion. Finally, within the end-systems (server and client machines), the challenges are many. Assured services are required and the server capacity should be increased with respect to the number of concurrent users. All the different subsystems like file system, server application, and communication system process data differently resulting in memory copy operations and replication of data in each subsystem and per-client. Device scheduling algorithms are different depending on hardware characteristics. For instance, a disk must take into consideration the seek time to move the head, rotational latency to position the head, and transmission time to read/write the data. These operations depend on the data placement on disk when determining the order of the disk operations to optimize efficiency. Additionally, the request deadlines are also important to avoid jitter and thereby poor playouts. As each stream requires a large set of resources, the number of concurrent clients that simultaneously can be serviced by one single machine is strongly limited by the given set of hardware and the way the resources are used by the system. Thus, all resources in a multi-user system can be scarce, especially in a multimedia system, and attempts should be made to try to minimize the amount of resources a single stream needs. At present, most of the work in this area has focused on small parts of the system itself. However, there are still unsolved questions and a large challenge is to integrate appropriate mechanisms for all subsystems into one integrated multimedia system.

The research area of MoD application support is huge, and in addition to the above mentioned areas, general topics like pricing, security, content search, etc. are important. However, in this thesis, we focus on the server and strive to optimize the common case operation, i.e., the task of reading data from the storage system and transmitting it through the network to remote clients with minimal overhead. Consequently, we use the data access characteristics to analyze and improve the whole data path through the server end-system. However, even though our focus is on the server machine, we will in later chapters also briefly discuss topics that relate to the network and the client, because all these components are closely coupled. The mechanisms used to support efficient and reliable services are therefore influenced by the characteristics of all devices along the data path.

2.6 Summary

In this chapter, we have provided background information describing various issues in the design of an MoD server. In particular, we described the requirements of presenting multimedia data in an MoD scenario, what this imposes on a system, and possible solutions to deal with high resource requirements and scalability problems. As mentioned above, we concentrate on developing a new architecture for single server machines to increase the I/O performance. However, proxies are confronted with the same scalability problem as MoD servers: a successful caching strategy can lead to the situation that a high number of concurrent clients has to be served by a single proxy. Servers and proxies do not only share the scalability problem, they also use exactly the same operating system mechanisms to retrieve stored multimedia data and transmit it to receivers. Our mechanisms, originally designed for the origin server, are therefore also suitable in the proxy operating system, because the same hardware and software is used [146].

Much research has been done in the area of resource reservations, access guarantees, and optimization of resource usage and performance. The next chapter gives a brief overview of existing work in the areas of reducing resource requirements and minimizing the processing overhead of transmitting data from disk to network in a multimedia server. To provide resource reservations and access guarantees by appropriate scheduling algorithms is beyond the scope of our work and is therefore not presented in this thesis.

Chapter 3

State-of-the-Art and Related Work

The objective of this thesis is to support asynchronous communication between information provider and information consumer in server-based systems supporting MoD-like services. We identify and eliminate possible bottlenecks in traditional systems that consist of off-the-shelf components with special focus on the three following factors: (1) memory management, (2) communication protocol processing, and (3) redundant functionality, i.e., buffer and error management.

To understand how the system works, to find the possible bottlenecks, and to see possible, existing solutions dealing with the three identified bottlenecks pointed out above, we provide some background information and a short state-of-the-art summary of existing mechanisms in this chapter. Section 3.1 describes error management mechanisms and background related to this area. Section 3.2 gives an overview of traditional memory management in (UNIX-like) operating systems and optimization solutions reducing copy operations and memory usage. In Section 3.3, we describe designs that improve the communication protocol processing overhead like packet generation and checksum operations, and we discuss and conclude the chapter in Section 3.4.

3.1 Error Management

When transferring data from a remote server in a distributed environment, several components along the data path may fail and introduce errors. To offer an “error-free” service, error detection and correcting schemes are used in several subsystems, i.e., in the storage system and the communication system. Different components cause errors due to different reasons, and the number of errors vary depending on the reliability of the component and the transmission media. Thus, the correctness of the information is checked multiple times, and the functionality is redundant.

An error free system cannot be guaranteed, i.e., we can only improve the chances of correct transmission. If too many bits within the block are corrupted, the receiver might receive a legal, but incorrect, data block and not be aware of the fact that an error has occurred [65, 137]. Nevertheless, to ensure that the information received at the client has a high probability of being the same as that transmitted, there must be some way for the receiver to deduce at a high probability if the received information contains errors. In this section, we describe different error models and error-handling schemes in both the storage system and the communication system, and we look at the possibility of integrating the redundant error recovery functionality of the different subsystems.

3.1.1 Error Models

When deciding which error management scheme to use in a system, we have to look at the kind of errors occurring in the environment where data is transmitted. In the next subsections, we have gathered some device characteristics and have looked at some works describing loss and data corruption measurements to get an overview of typical error rates.

3.1.1.1 Error Patterns in Storage Systems

The primary medium for long-term on-line storage of data is the magnetic disk [77, 145], because it survives power failures and system crashes when data already is correctly written to disk, and because it is faster than other persistent storage devices like tape devices. Although disk failures are much less frequent than other system errors, the disk devices themselves may sometimes fail, and thus, data is lost. For example, the top-end small computer scalable interface (SCSI) disk drives of Seagate (Barracuda and Cheetah disks) [202] and Western-Digital Enterprise disks [207] have a mean time between failure of at least 10^6 hours. Furthermore, the Seagate disks have a seek error for every $10^7 - 10^8$ bits read and a recoverable error rate of 1 in $10^{10} - 10^{11}$ bits accessed. The Seagate and Western Digital disks have an unrecoverable error rate[1] of 1 in $10^{14} - 10^{15}$ bits accessed and $10^{13} - 10^{14}$ bits accessed, respectively.

However, when retrieving multimedia data for several different streams each requiring a high bandwidth, and as the gap between processor and I/O performance increases in general, the magnetic disk does not provide the required data rate [77]. Therefore, disks combined in an array increasing performance and improving reliability by striping data among several disks, are often used to store data for delay sensitive applications on high performance storage servers. However, each disk in the disk array may introduce the disk errors described above, and as the number of disks increases, the mean time to disk failure also increases. If a disk has a probability p of failure, an n-disk disk array system has a probability $n \times p$ of failure using the same disks.

3.1.1.2 Error Patterns in Communication Systems

Transmitting a data element across a network can introduce various kinds of errors. Mainly, there are two kinds, i.e., bit errors and packet loss, which are both described in the two next sections.

Bit Errors All the various transmission media have their own typical error source [68], and a transmitted bit stream may therefore be damaged and corrupted by different sources inverting the bits. Any signal carried on a transmission medium is affected by:

- *Attenuation* reducing the signal. As the signal propagates along the transmission medium, its amplitude decreases making it harder to decode the received signal (especially if each signal carries more than one bit).
- *Noise* changing the transmitted value. For example, errors are generated when unpredictable electrical signals from other electrical equipment interfere with the transmitted electrical signal on wired electrical transmission lines and disturbances like bad weather scramble the transmission of wireless channels via satellite, radio, or terrestrial microwave links.
- *Bandwidth* (or the *transmission speed*) where the number of bits affected by noise increase with higher bandwidth.
- *Distortion* caused by the varying signal receiving delay. Two adjacent signals can interfere giving wrong values when one signal arrives late or early.

[1] The exact interpretation of what is meant by an uncorrectable error is somewhat unclear, e.g., does the read operation actually generate errors or do the errors occur during write operations and become evident when the data is read? Nevertheless, disk manufacturers generally agree that reading data from a disk is very unlikely to cause permanent and uncorrectable errors. Most non-recoverable errors are generated because data is incorrectly written or because the magnetic media is gradually damaged as it ages [35].

The number of bit errors introduced into the bit stream, depends on several different parameters like the underlying network technology and the length of the network. Wired networks are today usually very reliable. For example, the 10BASE-T (Ethernet) allows a bit error rate of 1 in 10^8, but typical performance is 1 in 10^{12} or better [175]. 100BASE-T and 1000BASE-T both perform with less than one erroneous bit in 10^{10} [185]. Wireless networks are more prone to noise than wired networks, and bit errors are therefore more common. Some frequencies are more susceptible to atmospheric disturbances such as rain attenuation, and mobile receivers may be more vulnerable to distortion and shadowing, e.g., blockage by buildings. Typical bit error rates in satellite networks are in the order of 1 error per 10^7 bits or less [4].

Packet Loss In addition to the bit errors caused during the transmission, whole packets may be lost, e.g., due to *congestion* in the routers, and this is today the overwhelming cause of loss [127]. The queuing buffer in an intermediate node or at the receiver might be full when receiving a packet, and it is therefore not able to store more packets resulting in an overflow. Furthermore, a router that has failed causes the other routers in the current network topology to alter the routing-path mid-stream, i.e., packets might be lost during the update of the routing tables where the congestion delay usually scales from hundreds of ms to several seconds and even up to several minutes [123]. In [19], MPEG video is transmitted over the Internet using the user datagram protocol (UDP) over IP. The average packet loss rates vary between 3% to 13.5% with greater loss rates during periods of high network activity. Packet loss also varies with the packet size where larger packets exceeding the maximum transfer unit (MTU) size have a higher loss rate, because if one fragment is lost the whole IP packet is lost. If no buffering or other means are taken to re-order out-of-order packets before decoding, further loss can be experienced. On average, 1.7% to 15.4% of the packets arrive out-of-order. However, a majority of the late arrivals are delayed by one packet only, i.e., a small buffer might solve out-of-order arrival due to network jitter or varying delays in different network routes. Moreover, measurements of packet loss transmitting (low rate, up to 64 Kbps) audio data between southern France and London is presented in [17]. Consecutive losses up to 30 packets are reported both for unicast and multicast connections, but the vast majority is a loss of one or two consecutive packets. Furthermore, in a study described in [170], the packet loss in an MBone multicast network, transmitting small packets of audio data, is measured. On average, the losses in the backbone links are small compared to the loss seen by a receiver at each site where there is usually a significant amount of small burst (consecutive) and isolated single losses. Only a few long bursts, i.e., spanning over 100 packets, of congestion loss were observed. As we can see, some packets are often discarded transmitting data to a remote client, and the number of packet loss is highly dependent on several different parameters like the capacity of the networks and the nodes, the number of hops (intermediate nodes), and time of day. However, there is no well defined model describing how packets are lost, and although the above described results vary somewhat, the broad conclusion is that some receivers will experience packet loss [127]. Furthermore, as the Internet grows larger, it will be harder to measure and characterize its behavior [124], because the network changes quickly, and the amount of heterogeneity increases.

3.1.2 Error Detection and Correction Mechanisms

As we can see from Section 3.1.1, different subsystems have different error models, and thus, different error management schemes are applied according to the kind of errors present in the subsystem. Next, we present some traditional error management codes, and then we look at typical recovery schemes for the storage system and communication system using either some of the traditional codes or new codes adapted to special error models.

3.1.2.1 Traditional Codes

We have seen that errors in a computer system might occur due to different reasons, and to improve the accuracy of a data transmission, certain types of coding schemes are used. These schemes vary depending on the kind of errors that appear. In this thesis, we differentiate between *erasures* where the exact position of the damaged data is known and *errors*[2] which are corrupted data with unknown position. Since the position of damaged data is known in erasures, they are easier to correct than errors. Furthermore, in error management mechanisms, a frame normally consists of some data bits (or groups of bits called *symbols*, i.e., a symbol is a fixed size data element used in the coding operations as a single unit) and some redundant check bits (or symbols). The combined message unit containing both information data and redundant data is called a *codeword*.

In general, two basic ways of dealing with corrupted data are developed [159], i.e., error detection and error correction. The former strategy includes redundant information in the codeword to only allow the receiver to deduce that an error has occurred, and the latter strategy also includes enough information to find the location of the error and to correct it. The error detection and error correction properties of a code depend on its *Hamming distance* [68, 159], i.e., the number of bit positions in which two code words differ. In general, the more redundancy there is in the code word, the more errors can be detected and corrected. In particular, to detect *d* errors, a code with Hamming distance *d + 1* is needed, and to correct *d* errors, a Hamming distance of *2d + 1* is needed in the code.

To ensure that the information received at the client has a high probability of being the same as that transmitted, there must be some way for the receiver to deduce at a high probability if the received information contains errors. Several error detection and/or correction schemes suitable for different error models exist and are described in the literature (see for example [35, 65, 68, 137, 145, 159]). Various important properties of some of these codes are shown in Table 3.1, and below, a short overview is given for these codes[3].

	Systematic†	**Memoryless**‡	**Reported use**
Block	Yes	Yes	Storage and Communication system
Convolutional	No	No	Communication system
Turbo	Yes	Yes	Communication system
Tornado	Yes	Yes	Communication system

†The encoder produces a codeword that explicitly contains the information block

‡The code operates on a fixed length block of information bits

Table 3.1: Traditional error recovery codes.

A *block code* [65, 68, 137, 159] is an error detection/correction code that operates on a fixed length block of information bits. The bit stream is broken up into k-bit blocks, and (n - k) check bits are added producing an n-bit codeword. An often used type of cyclic block code, i.e., codes where every cyclic shift of a codeword is also a codeword, is the *Reed-Solomon* code [130, 137], which works on the non-binary symbols $0, 1, \ldots, p^{m-1}$ drawn from the Galois field, $GF(p^m)$, which is a finite field. The code parameters are often given in the form of "(n, k) over $GF(2^{\mathrm{m}})$", where m is the code symbol in bits, $n = 2^{\mathrm{m}} - 1$ is the maximum block size in number of code symbols, and k is the number of data symbols per block ($k < n$). The error correcting ability of the code then depends on n - k, i.e., the number of parity symbols in the block. In pure erasure correcting mode, the decoder can correct up to n - k symbol erasures per block, and in the pure error correcting mode, the decoder can recover from up to (n - k)/2 symbol errors. If we have both errors and erasures, the decoder can restore the data block if

[2]The term error will also be used when we have damaged data in general whether the location is known or not.

[3]The basic mathematical principles and elements of coding theory are out of the scope of this thesis and are therefore not described here. For further information, see for example [65, 137].

(2 × #errors + #erasures) < (n - k). A special class of Reed-Solomon codes are *Reed-Solomon Erasure* codes where only erasures, not errors, can be corrected. This speeds up the coding process, and an (n, k) erasure code guarantees that a successful receipt of any k of n symbols enables a complete reconstruction of the source data [29, 136].

Block codes are memoryless codes as each output codeword depends only on the current n-bit message block being encoded. In contrast, with a *convolutional code* [182, 68, 137], the continuous bit stream is operated upon as a sliding window of data bits to produce an encoded, continuous stream of output bits. Each bit in the output sequence is dependent not only on the previous bit, but also on the previous sequence of source bits, thus implying some form of memory. A convolutional coder consists of a k-stage shift register where k is also the constraint length of the coder. Input data bits are shifted into a shift register one bit at a time, and the convolution operation is performed using one or more exclusive or (XOR) gates. For each bit shift, the encoder produces v alternating XOR sums as output bits, and a coder putting out v bits per input bit is called a rate 1/v coder. An example of a rate 1/2 3-stage shift register encoder is displayed in Figure 3.1 where the bits shifted into the SR_1 and SR_3 registers are used to produce output bit c_1, and the bits shifted into the SR_2 and SR_3 registers produce output bit c_2. The aim of the decoder is to determine the most likely output sequence given the transmitted bit stream, which may have errors. The decoding procedure is equivalent to comparing the received bit sequence with all possible sequences that may be obtained with the respective encoder and then selecting the sequence closest to the received sequence. Such a decoding solution works well for short codewords, but as the length of the codeword increases, the number of possible bit sequences increases exponentially [182]. The most used solution for identifying the correct sequence and at the same time minimizing the number of comparisons is the *Viterbi (maximum-likelihood) decoding algorithm* [182, 68, 137].

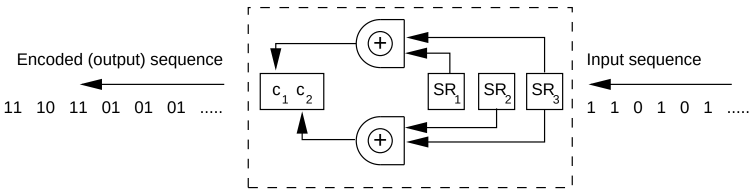


Figure 3.1: Example of a rate 1/2 3-bit shift convolutional encoder.

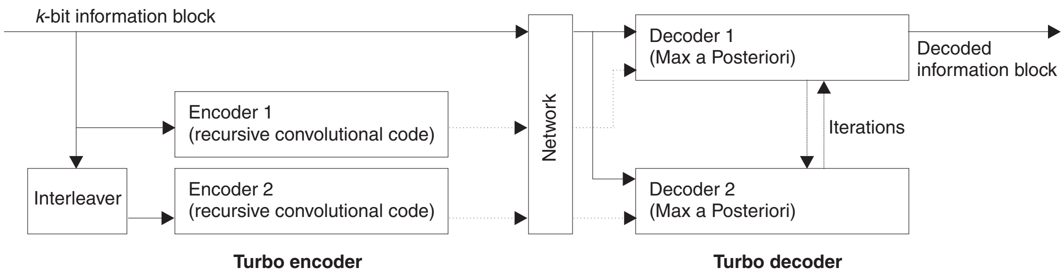


Figure 3.2: Turbo codes [193].

Convolutional codes are again used in *turbo codes* [46, 193] (see Figure 3.2). A turbo code uses two convolutional codes in parallel and data interleaving to increase correction capability and performance compared to traditional convolutional codes. Turbo codes can also be regarded as a block code, because it takes an information block and generates two sets of parity data (horizontal and vertical) using convolutional codes. Likewise, the *Tornado* codes [29, 96, 97] are linear-time erasure codes for increasing performance of erasure correcting block codes like Reed-Solomon Erasure. The code has n message bits and βn check bits and is defined by arranging these bits in a bipartite graph which has n left nodes and βn right nodes corresponding to the message bits and check bits, respectively (see Figure 3.3). This means that the graph defines the mapping from message bits to parity bits. However, this code is not able to recover from all losses regardless of their location. This problem is solved by cascading the code which means more redundant data. However, the encoding and decoding speed of a Tornado code is reported to be faster than other previous software-based schemes and should be an appropriate solution for high bandwidth, real-time applications [96].

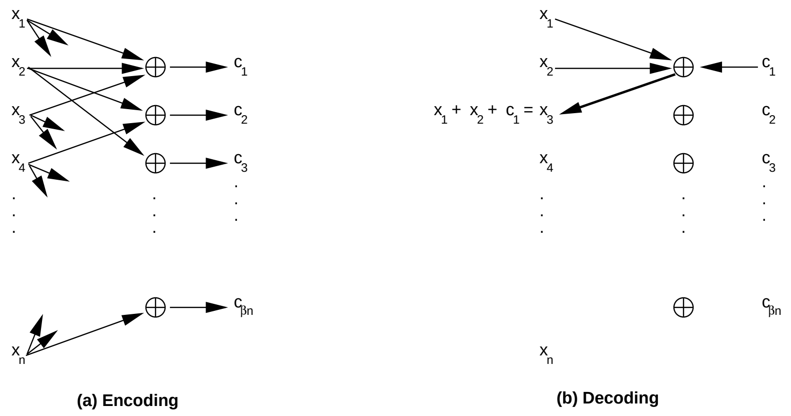


Figure 3.3: Tornado coding and decoding [96].

3.1.2.2 Storage Systems Recovery

Even though errors in a disk-based storage system are very infrequent, the disk controller attaches checksums to each sector derived from the data that is written to detect errors during read operations retrieving data from disks. When data is read, the checksum is computed again and compared to the stored value. If an error occurs, the controller will retry to read data several times until the data block is correctly read, but if the error continues to appear, the controller will signal a read failure [145]. Thus, transient errors, e.g., caused by dust on the head, and seek errors can often be corrected by the controller re-reading the data, but permanent errors, e.g., physically damaged disk blocks, will cause a data loss if no other error recovery mechanism is used.

As mentioned earlier, one single disk cannot deliver data fast enough for a high data rate server, and if the single disk gets corrupted, data will get lost. This has lead to I/O systems that increase performance and reliability through disk parallelism using techniques like data striping. A variety of disk-organizations have been proposed where the different levels of RAID organizations [35, 119, 145] are often used to address both the performance and the reliability issues. The obvious solution to restore data if a disk fails is to introduce redundant data for error recovery. Adding error correcting information can be done in several ways, and in the case of RAID, the various levels manage errors differently (see also Table 3.2):

- *RAID level 0* is only a general disk array and has no redundancy, i.e., errors will not be corrected, and we lose data in case of an error.
- *RAID level 1* uses the simplest form of error correction. Each disk is duplicated which means that the storage system has two copies of every data element. Thus, if one disk fails, the data element is read from the other disk, i.e., no other redundancy is used. This approach is also called *mirroring*.
- *RAID level 2 - 6* make use of error correcting codes to recover from different kinds of errors. Parity information is computed, often by using an XOR calculation, to detect and correct errors. The main difference between these RAID levels is whether the information is bit (level 2/3) or

block (level 4/5/6) interleaved and if all parity information is located on one (or more) predefined parity disk(s) (level 2/3/4) or distributed (level 5/6) among all the disks. Level 6 adds additional redundant information to recover from two errors at the same time.

	Parity unit	Parity location	Recovery capability
Level 0	None	None	None
Level 1	Mirroring	Disk duplicating	One
Level 2	Bit	Predefined disk	One
Level 3	Bit	Predefined disk	One
Level 4	Block	Predefined disk	One
Level 5	Block	Distributed	One
Level 6	Block	Distributed	Two

Table 3.2: RAID levels.

If an error is detected reading a disk block, the disk controller first tries to re-read the disk block as described above in a single disk scenario, but if the block is not correctly read, the error is corrected according to the error correction scheme in the used RAID level. Since the controller knows which disk failed or gave an error, the XOR parity and the remaining data from the other disks can be used to determine bit values of the lost data.

In today's RAID solutions, an XOR error correction scheme is usually applied for reliability, but, in addition to the standard error mechanisms in the RAID levels, some other error management schemes are proposed for disk arrays in general or as improvements of the RAID mechanisms. For example, the *EVENODD* [12] double disk failure scheme uses a two-dimensional XOR parity calculation which shows good results regarding redundant information overhead and performance, and in [77], the problem of designing erasure-correcting binary linear codes protecting against disk failures in large arrays is addressed. Several two-dimensional erasure-correcting solutions are presented capable of correcting multiple disk failures. However, due to the good reliability of a disk array capable of recovering from two disk failures, there is no need for x-disk-correcting disk array where $x \geq 3$. Reed-Solomon RAID systems are described in [130] where m disk failures can be recovered by applying m redundant disks.

Work is also performed with the aim of increasing performance of the error correction scheme. For example, the performance of write operations, where both application and parity data might be read, updated, and written back to disk, is increased in [103, 154]. Moreover, decreasing the time to recover from a failure or the ability to recover without taking the system down are issues addressed in [11, 75, 76]. However, in this first step towards an integrated error management scheme, only the coding scheme itself and its correcting capability are of interest to us. Thus, increased *storage* performance of an integrated error scheme will be a future research issue.

3.1.2.3 Communication Systems Recovery

As we can see from Section 3.1.1.2, there are a lot of sources of errors in the communication system, and loss caused by congestion in the intermediate switches (erasures) and impulse noise (errors) need to be dealt with differently [99]. This is because noise is of short duration and should have no lasting effect on error rate, i.e., only the bits affected by the noise may be damaged. Congestion, on the other hand, may last for a while making the problem even worse. However, irrespective of the kind of errors that occur, a mechanism is needed to obtain a copy of the (hopefully) correct information, and there are basically two approaches for achieving this [10, 17, 68]:

- *Feedback error control*, also called *automatic repeat request* (ARQ), in which information is added to enable the receiver to detect an error but not the location. A retransmission control scheme is used to request another copy of the required data where each retransmission adds at least one round-trip time (RTT) of latency.

- *Forward error control* where additional information is included so the receiver cannot only detect an error, but also determine where the error is located. The error is then corrected by inverting the corrupted bits, or the packet is reconstructed entirely in the case of packet loss. FEC trades bandwidth for latency to improve packet damage/loss rate, i.e., by transmitting more redundant information.

In communication systems, error control is often performed by error detection followed by a retransmission, i.e., feedback error control, because it is usually more efficient. We need more redundant data to be able to correct a corrupted data element rather than only detect the error, and to minimize the amount of data to transfer to a remote client, only an error detection scheme is usually applied requesting a retransmission if an error is detected. Furthermore, wired networks have improved regarding error rate making the redundant error correction information even more unnecessary, because fewer retransmissions will be requested. Due to both bit errors and packet loss, it is on average more efficient to use feedback error control and then retransmit the package in case of an error compared to transmitting (possibly unneeded) error correcting data.

The Internet checksum feature was chosen to detect errors in a packet by the Internet community in the late 70's after experiments on the ARPANET [155]. The checksum is included in packets so that errors encountered during transmission may be detected [20]. The checksum is computed by treating each 16-bit field as an integer and adding them all together using 1's-complement arithmetic. The checksum is then the 1's-complement (inverse) of the sum. This checksum will catch any burst error of 15 bits or less, and all 16 bits bursts except those which replace one 1's-complement zero with another, i.e., 16 adjacent one bits replace 16 zero bits or vice versa.

However, the Internet checksum feature does not detect all errors, and retransmissions are not always preferred (see Section 4.1.1), e.g., in delay sensitive applications like real-time multimedia applications. Therefore, the attractiveness of FEC has changed during the past decades, and several different codes have been used. Convolutional codes with Viterbi decoding have been the de facto standard for over 30 years, and it is a good scheme for correcting randomly occurring errors. It has been a preferred method for achieving FEC in data communication systems [68], and it is used in many real-world systems [137], especially in satellites and space communications where it is efficient for short constraint lengths ($k \leq 7$) codes [182]. There exist several hardware implementations of turbo codes [46, 193] for satellite and wireless links [177, 206]. Furthermore, the Internet Engineering Task Force (IETF) recently standardized a particular scheme [126] for audio data where a low quality copy of the data is sent as redundant data in subsequent packet(s) [15]. Finally, Reed-Solomon codes are also used for FEC in communication systems where for example simplified Reed-Solomon codes correcting only erasures are used in asynchronous transfer mode (ATM) networks [6, 10, 99, 150] to reduce the damage caused due to congestion loss in broadband wide area networks and real-time environments.

A major difficulty when using FEC is to choose the right amount of redundant data in changing network condition scenarios, and it is usually not sufficient to only consider the mean packet loss ratio as for example outlined in Section 3.1.1.2. The nature of the loss process has an impact on the recovery technique performance where different users in a multi-user environment experience different kinds of losses. Furthermore, a FEC scheme is only able to correct a certain number of errors, and no FEC methods can identify all errors. In general, codes like the convolutional code are primarily used to reduce the error probability of a link to a more acceptable level. A typical reduction with a rate 1/2 convolutional code is between 10^2 and 10^3. Hence, assuming an ARQ error control procedure is also being used, the overall link efficiency is much improved [68]. In scenarios where a higher level of reliability is required than a single FEC scheme can assure, a *hybrid ARQ* scheme [32, 67] might be appropriate combining FEC and ARQ. Type I hybrid ARQ is like a traditional FEC scheme, but if data is not recoverable, a retransmission is executed. Type II hybrid ARQ is like a traditional ARQ scheme, but parity data is first sent when a retransmission is required, and when data is not recoverable, the packet itself is retransmitted. Both hybrid ARQ types are currently not considered, because they impose higher CPU time and memory demands onto the server than pure FEC to assure a degree of reliability that is higher than needed in our current scenario. However, the suitability of these hybrid schemes in large scale systems, e.g. NoD over the Internet, is subject to future research.

3.2 Memory Management

In a traditional server based system, there are several different subsystems within the operating system which all are running in their own protected domain (address space). Executing a request from a remote client, involving data transfers from disk to network adapter, makes the data path through these different subsystems critical. Due to different protection mechanisms, data is usually copied or moved at a high cost from domain to domain to allow the different subsystems to manipulate the data where each subsystem might add processing costs. This also means that there must be redundancy in the different subsystems where, for example, mechanisms for buffer management must be performed in both the file system and the communication system. Finally, the different subsystems are not adapted to each other to guarantee services and optimize server performance, e.g., the buffer structures used in the file system cannot be used when processing data through the communication protocols. Thus, in an MoD server supporting a large number of users, the transport of very large amounts of data through the I/O pipeline in the end-system will make a lot of cross-domain data transfers, and each subsystem therefore represents a potential bottleneck [128].

Consequently, designing a high performance MoD server requires additional investigation to address the above mentioned problems. To achieve an optimal utilization of system resources and to be able to support a maximum number of clients, support for QoS should be added to all system components. However, lately, quite a lot of work has been done to optimize the performance of operating systems in general. Several systems have attempted to reduce data touching overhead when performing I/O operations. Data touching overheads include those operations that require processing of data within a given buffer such as checksumming or copying data from one buffer to another [161]. Previous efforts of improving performance include for example restructuring operating system software to minimize data movement and copying between different protection domains. Data-touching overhead is most significant for large I/O transfers [84], and thus, reducing these costs can have considerable benefits for operations like video and audio transfers over high speed networks.

In this section, we first take a look at the traditional I/O architecture of current UNIX systems, and then we describe different models of transferring data. Furthermore, we look at some "new" approaches for reducing memory copying, and we describe some work minimizing either the per-client or per-data-element resource usage.

3.2.1 Traditional I/O Architecture

A typical layered architecture of the I/O-pipeline of current UNIX operating systems is depicted in Figure 3.4. The *storage I/O-system* is responsible for storage and retrieval of digital data, e.g., video and audio, on/from persistent storage media, and the *data (file) management system* provides file system services executing requests from the application retrieving data from the storage I/O-system and manages buffering. The *application management system* handles requests from clients and passes data to the communication protocols. *End-to-end communication protocols* provide services like error control, flow control, and other QoS related services, and the *host-network interface* transfers data between the end-systems and the network, i.e., in most cases the host-network interface provides services in the data link layer and the physical layer.

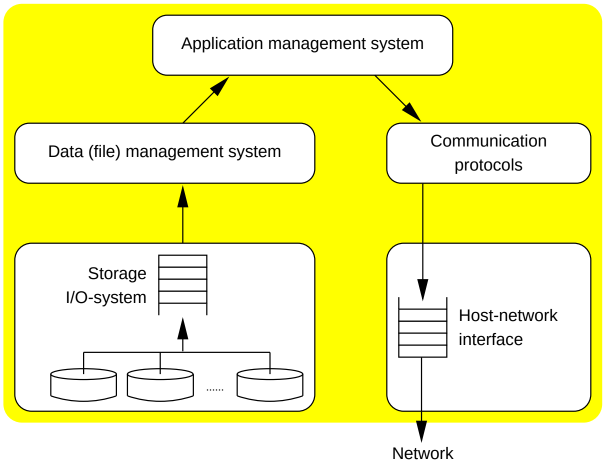


Figure 3.4: Critical data path in server-based systems.

Common user-level applications access the file and network I/O system services through the `read()` and `write()` system calls for data transfers, respectively. These system calls work on descriptors. For file I/O, the `open()` system call produces a descriptor by opening a file, and the `socket()` system call performs the same task in the case of network I/O. The `close()` system call is used to deallocate any descriptor [92].

3.2.1.1 Buffer Management in the Application

No support is provided in UNIX systems for buffering and caching at the user level. Applications are expected to provide their own buffering and caching mechanisms, and I/O data is usually copied between the operating system and the application buffers during read and write operations.

3.2.1.2 Buffer Management in the File System

Storage devices are accessed through the file system interface which in turn consists of several layers. The top layers, i.e., *virtual file system* layer of the Network File System [158] and the *local file system* layer of the local operating system, provide file system services like data organization and file naming. The bottom layer, controlling the storage devices, is the *device driver*. This layer allows accesses in terms of fixed sized blocks that are an integral multiple of the smallest possible block on the disk (disk sector).

Within the operating system, there are a lot of I/O operations, and if every data request results in real disk accesses, the CPU would spend most of its time waiting for I/O to complete. However, in addition to managing the memory that buffers data being transferred from disk, the *buffer cache* avoids expensive disk accesses by acting as a cache of recently used disk blocks. In a typical UNIX system, over 85% of the implied disk operations are served by the buffer cache where the requested blocks already reside in the buffer cache [92]. Whenever a read request is performed, the buffer cache attempts to find the requested data in the cache. If it is successful, a disk transfer is avoided and the buffer is returned. Otherwise, a new buffer is allocated (or if none is available, old buffers are deallocated and freed for reuse) and passed to the device driver which copies the data into the buffer.

All data transfers between applications and storage devices through the file system interface use the buffer cache, and both the data and the control path goes through the described layers: data is first retrieved from disk by the device driver, copied into the buffer cache, and subsequently copied from the buffer cache into the application buffer. Thus, the traditional UNIX file I/O system is a bottleneck for supporting high bandwidth data transfers in networked multimedia applications. It is designed for traditional text and binary file accesses, and in this case, the buffer cache and the memory buffer (mbuf) facilities work efficiently. The performance gains due to minimized disk traffic makes the cost of keeping data in the buffer cache and the extra cost of copying data from cache to application buffer is negligible. However, multimedia data like audio and video, as used in an MoD scenario, do not benefit from the caching provided by the file system buffer cache [36]: (1) they need a large amount of memory space, and (2) the retrieved data is relevant only for a small amount of time. Due to a limited amount of buffer space, the multimedia data is often replaced before it can be reused. Consequently, only processes reading the same data in a short time interval are able to use the cached data. Therefore, retrieving data through the buffer cache usually does not provide any benefits in a high bandwidth, networked multimedia application. For example, NetBSD usually uses between 5-10 % of the memory for the buffer cache. On our 256 MB test machine (see Section 5.1 and A.2), 13168 KB memory is originally used for caching disk data. Streaming one single MPEG-2 video file to two concurrent clients at an average DVD bit-rate of 3.5 Mbps [184, 194], the cache can only hold about 30 (29.4) seconds of video data, i.e., if the system is to benefit from caching, there must be no longer that 30 seconds between the starting points of these two clients. Adding more streams and concurrent clients will make the caching effect even less. This also means that complex data replacement techniques (paging algorithms) in the buffer cache for multimedia applications, like for example *distance* [174], *least/most relevant for presentation (L/MRP)* [104], *QoS-L/MRP* [69], *generalized interval caching* [49], and *SHR* [83], will have no effect on our MoD server, because we will have no caching effect and are actually not using, i.e., bypassing, the buffer cache in the proposed data path.

The poor performance of demand paging due to disk access speeds and the necessity of prefetching data from disk to memory to support continuous playback of time-dependent data types is described several places in literature, e.g., in [69, 104, 107, 160]. Prefetching is a mechanism to preload data from slow, high-latency storage devices such as disks to fast, low-latency storage like main memory. This reduces the response time of a data read request dramatically and increase the disk I/O bandwidth. Applications with a lot of interactions may benefit from mechanisms using access patterns of linked data structures [141], statistical access trees [93], or automatic hint generation [34]. In the case sequential operations as in our MoD scenario, some kind of a simple read-ahead mechanism [8, 31, 69, 104, 120, 173] will be efficient.

3.2.1.3 Buffer Management in the Communication System

Data is sent to the network through the communication system. This subsystem consists of three sub-layers: the *socket* layer which provides an interface to the application, the *protocol* layer performing communication protocol specific operations, and the *network interface* layer managing the network hardware interface.

The requirements placed on the memory management scheme by interprocess communication (IPC) and network protocols tend to be very different from those of other parts of the operating system like the file system [92]. Therefore, a special purpose memory management facility called the *memory buffer* (mbuf) is used which is designed for both fixed sized data structures such as sockets and protocol state blocks as well as for variable sized structures like communication protocol packets. Mbufs are usually 128B long[4] with 112B reserved for header and application data storage. For larger messages, data may be associated with an mbuf by referencing an external page. An offset and a length field in the mbuf header allows routines to trim (packet header/trailer) data efficiently which provides flexibility to add data to and delete data from a message in the communication system. Furthermore, the provision for mapping pages into an mbuf avoids physical memory copy operations between mbufs. Data is moved

[4]In NetBSD version 1.5 and later, the size of an mbuf is 256 B to allow more data in one single mbuf [196]. This is because the traditional size of 128 B can make problems in the communication systems using maximum packet header sizes, i.e., both TCP and IP headers of 64 B results in header data spanning several 128 B mbufs.

by remapping page table entries associated with the physical pages. In addition, the same page may be mapped to multiple mbufs to share the same data block avoiding several identical physical copies.

However, even if no physical copying is necessary between mbufs, two copy operations are needed to transfer data along the data path between application and network interface. Data is first copied from the user level buffers to the kernel level mbuf facility, and then, the data is further copied into buffers on the network interface card.

3.2.2 Models of Data Transfer

Data transfers between protection domains usually require moving (or copying) data from one memory location to another, and the system performance is greatly affected by implementation issues like whether to use a physical or virtual data transfer:

- A *physical* transfer involves moving data in physical memory where each byte of data is moved from the source domain's physical memory to the destination domain's physical memory. These transfers promote flexibility, because the transfer granularity size is a byte. Thus, we can transfer data from source to destination of any size to any location. However, this flexibility is overshadowed by the high overhead in space and time. First, more memory space is needed to store each copy of the data. Second, a physical transfer is very time consuming. Two memory accesses are needed to read and write, and each memory-access instruction takes more time than instructions that do not access memory[5]. This overhead is insignificant for small data transfers, but when transferring a large video object through several domains, i.e., several copy operations, this additional workload degrades the system performance by increasing delay and decreasing throughput.

- A *virtual* transfer involves moving data in virtual memory. Instead of moving data in physical memory, only the addresses to the data elements in physical memory are mapped into the destination domain's address space (virtual memory). The transfer granularity size is the physical page. Since entire pages are the actual units of transfer, the data's destination address must have the same relative offset from page boundary as the source address. Furthermore, the destination space must be large enough to store the number of transferred pages which usually needs more space than the data itself (fragmentation), i.e., unless the data size is exactly a multiple of pages. However, when large amounts of data are transferred between different protection domains, the reduced memory space requirement and the performance gain using virtual transfers instead of physical copying overcome the poor flexibility and the risk of fragmentation.

Using either physical or virtual data transfers, there are three different models for transferring data between different protection domains as shown in Figure 3.5 [121]. These are the copy model, the move model, and the share model which are described further below:

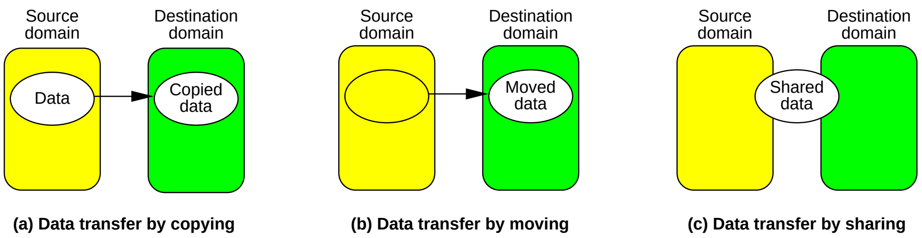


Figure 3.5: Different models for data transfer between protection domains.

[5]This gap is getting even larger as for example the RISC architectures continue to allow CPU speed to scale much faster than memory speed [121].

- The *copy* model (Figure 3.5a) copies all the data from one domain to another, i.e., an exact copy resides in both the source and the destination domain. Generally, physical transfers apply to the copy model, but in the case of read-only data[6], virtual transfers between domains could also be used, i.e., a virtual copy maps the transferred pages into the destination address space and do not affect the mappings in the source address space.

- In the *move* model (Figure 3.5b), data is removed from the source domain and placed in the destination domain. This model avoids maintaining several identical copies of the same data in memory at a time, but in cases when the source later needs to access the data again, e.g., by handling a retransmission request, data must be fetched back into the source domain. Physical moves do not make much sense in the move model since erasing data in the source domain introduces additional costs. A virtual move is used where the transferred pages are mapped into the destination address space and unmapped from the source.

- The *share* model (Figure 3.5c) makes the transferred data visible to both the source and the target domain, i.e., regions in both the source and the destination domains' address space are mapped to the same physical pages. They might both access the same data, and modifications made by a process in one domain will be visible to other processes in other domains. This model has the advantage of virtual copying, because the source does not lose access to the data. Thus, no data copying is required which may increase system performance. However, modified transferred data may cause problems in other domains. Since these modifications are asynchronously seen from the process point of view, explicit synchronization mechanisms might be necessary. This will in turn make programming more complex and can propagate errors across domains.

Which mechanism to use depends on the amount of data to transfer, the cost of copying and remapping, and the functionality to be obtained. In large multimedia systems where large amounts of data are transferred throughout the system, physical copying will take up too much system resources and should therefore be avoided, i.e., even virtual remapping with copy-on-write (CoW) which performs a physical transfer if the data is modified is not appropriate. Thus, since data still may be manipulated in several subsystems, virtual page remapping often has move rather than copy semantics. However, this requires another data transfer in the case that the sender needs to further access the data. Measurements shown in [53], indicate that page remapping alone is not fast enough to support the bandwidth of a high-speed network adapter. Finally, shared virtual memory avoids data transfers and its associated costs altogether, but its use may compromise protection and security between sharing protection domains. Thus, a combination of shared memory and virtual remapping may give the efficiency of shared memory and the flexibility and protection of virtual remapping.

3.2.3 Reducing the Number of Memory Copy Operations

Traditionally, there are several different possible data transfers and copy operations within an end-system as shown in Figure 3.6. These often involve several different components. Using the disk-to-network data path as an example, a data object is first transferred from disk to main memory (A). The data object is then managed by the many subsystems within the operating system designed with different objectives, running in their own domain (either in user or kernel space), and therefore, managing their buffers differently. Due to different buffer representations and protection mechanisms, data is usually copied, at a high cost, from domain to domain ((B), (C), or (D)) to allow the different subsystems to manipulate the data. Finally, the data object is transferred to the network interface (E). In addition to all these data transfers, the data object is loaded into the cache (F) and CPU registers (G) when the data object is manipulated or copied.

[6] If a process modifies the data, a new physical copy must be made. However, such copy-on-write mechanisms are often complex compared to simple physical copying schemes.

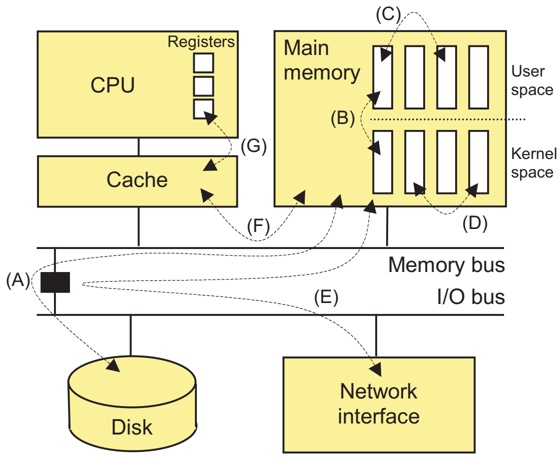


Figure 3.6: Data transfers and copy operations.

Figure 3.6 clearly identifies the reason for the poor performance of the traditional I/O system. Data is copied several times between different memory address spaces which also causes several context switches. Both, copy operations and context switches represent major performance bottlenecks. Furthermore, the different subsystems, e.g., file system and communication system, are not integrated. Thus, they include redundant functionality like buffer management, and several identical copies of a data object might be stored in main memory, which in turn reduces the effective size of the physical memory. Finally, when concurrent users request the same data, the different subsystems might have to perform the same operations on the same data several times.

Many designs have been proposed to improve I/O performance in operating systems (see [129] for a state-of-the-art overview), and the concept of avoiding unnecessary physical data copying is not new. In this section, we take a closer look at existing designs and implementations for optimizing throughput and reducing the number of data transfers along the I/O pipeline. We distinguish between three types of copy operations: memory-CPU, memory-device, and memory-memory. Solutions or mechanisms improving performance for these types of copy operations have been developed for both general purpose and application specific systems, and below, we take a closer look at some of these designs.

3.2.3.1 Memory-CPU Transfers

Data manipulations, like encryption, compression, error detection and correction, and presentation conversion, are time consuming and are often part of different, distinct program modules or communication protocol layers, which typically access data independently of each other. Consequently, each data manipulation may require access to uncached data resulting in loading the data from memory to a CPU register, manipulating it, and possibly storing it back to memory. Thus, these repeated memory-CPU data transfers, denoted (F) and (G) in Figure 3.6, can have large impacts on the achieved I/O bandwidth. To decrease the number of memory-CPU data transfers, integrated layer processing (ILP) [2, 39] performs all data manipulation steps, e.g., calculating error detection checksums, executing encryption schemes, transforming data for presentation, and moving data between address spaces, in one or two integrated processing loops instead of performing them stepwise as in most systems. Thus, data will be moved between memory and the CPU only one time.

Memory-CPU transfer optimizations like ILP increase efficiency if there are a lot of data touching operations being performed. However, in our MoD scenario, we try to eliminate all data touching operations. ILP will have no effect on performance in our system and will therefore not be considered for the data copy avoidance mechanism design.

3.2.3.2 Memory-Device Transfers

Along the I/O pipeline, data is transferred between hardware devices, such as disks or network adapters, and applications' physical memory. This is often done via an intermediate subsystem, like the file system or the communication system, adding an extra memory copy. A mechanism to transfer data without multiple copying is *direct I/O*, which in some form is available in several current commodity operating systems, e.g., Solaris and Windows NT. The most used techniques are direct memory access (DMA) and programmed I/O (PIO) [52]:

- DMA implies that the I/O adapters transfer data directly from and to main memory without involving the CPU. Transfer rates close to the limits of the main memory and the I/O bus can be achieved by transferring large blocks of data in a single bus transaction. The data transfer can proceed concurrently with the CPU activity as long as the data needed by the CPU resides in the cache, i.e., main memory accesses may induce processor stalls during heavy DMA traffic. However, DMA increases complexity in the device adapters and caches are often not coherent with respect to DMA transfers.

- PIO requires the CPU to transfer every word of data between main memory and the I/O adapter in a programmed loop. Thus, two bus transfers are required: (1) when data is streamed from memory to the CPU and (2) when transferring the data from the CPU to the device adapter (or vice versa). The CPU is occupied during the transfer, and often, only a fraction of the peak I/O bandwidth is achieved.

Due to high transfer rates, DMA is often used for device/memory data transfers. However, despite the reduced bandwidth, PIO can sometimes be preferable over DMA. If data manipulations, e.g., checksum calculations, can be integrated with the PIO data transfer, one can save one memory access, and after a programmed data movement, the data may still reside in the cache, reducing further memory traffic.

3.2.3.3 Memory-Memory Transfers

Direct I/O is typically used when transferring data between main memory and a hardware device as described above. However, data transfers between different process address spaces are necessary due to protection and are done through well-defined channels, like pipes, sockets, files, and special devices, giving each process full control of its own data [100]. For example, the data handled by an application running on top of a monolithic operating system kernel, must cross the user-kernel boundary. Moreover, additional user processes and the trend in operating system design towards a kernelized system structure may introduce additional domain boundaries into the I/O data path [52]. Thus, in the future, it will be even more important to reduce the costs of data transfers between different protection domains. Nevertheless, such physical copying is slow (see Section B.2) compared to the CPU speed and requires at least two system calls per transaction, i.e., one on the sender and one on the receiver side.

Several general cross-domain data copy avoidance architectures have been suggested. These architectures try to minimize or eliminate all (B), (C), and (D) copy operations depicted in Figure 3.6 to reduce the data transfer costs and thereby increase I/O bandwidth and throughput. We have divided the existing mechanisms into three groups: (1) general purpose mechanisms supporting all operating system services, (2) networking mechanisms reducing copy operations in the application-to-network data path, and (3) application-specific streaming mechanisms transferring data as fast as possible through the system:

Universal Mechanisms: Already in 1972, *Tenex* [14], a page-based time-sharing system for the PDP-10 system, was one of the first systems to use virtual copying, i.e., several pointers in virtual memory to one physical page. The pager permits pages to be shared for write as well as read references. A CoW facility allows users to share address spaces and to obtain private copies of pages which are changed. *Accent* [61, 138] generalized the concepts of Tenex by integrating virtual memory management and IPC in such a way that large data transfers could use memory mapping techniques rather than physical data copying. Virtual copying is achieved by sharing the transferred pages among the sending and the receiving domain, and it delays the copying until one of the sharing domains attempts to update the shared data unit, i.e., only when a process modifies the page, a copy is placed in this process' own address space (using CoW). Furthermore, the V distributed system [37] supports page remapping using move rather than copy semantics. During the last decade, several systems have been designed which use virtual memory remapping techniques to transfer data between protection domains without requiring several physical data copies. An interprocess data transfer occurs simply by changing the ownership of a memory region from one process to another. Additionally, there exist several other general purpose mechanisms supporting a zero-copy data path between disk and network adapter like the DASH IPC mechanism [163], Container Shipping [7], IO-Lite [110, 111, 112], Genie [23, 24, 25, 26], the universal continuous media I/O system [41, 43], and UVM virtual memory system [42, 44, 45] which use some kind of page remapping, data sharing, or a combination. Some of these mechanisms are further described in Table 3.3A.

Networking Mechanisms: In the case of application-to-disk transfers, direct I/O can often be applied since the traditional UNIX-like file system usually does not touch the data itself. However, in the case of application-network transfers, the communication system must generate packets, calculate checksums, etc., making it harder to avoid the data transfer through the communication system. Nevertheless, there are several attempts to avoid data touching and copying operations by transferring data directly between application buffers and network interface, i.e., omitting physical data transfer through kernel space [9, 18, 40, 47, 51, 54, 87, 149, 162, 171] and reducing the traditional (B)(E) data path in Figure 3.6 to only (E). Since these mechanisms consider only the application-to-network data path, these mechanisms are not outlined in detail in Table 3.3, but several different designs are proposed: (1) Page remapping is used to virtually transfer data to the kernel; (2) PIO is used to transfer data directly to the network card, and the data touching operations are performed as the data is transferred through the CPU; (3) outboard protocol processors are used, and data is transmitted directly to the network card where the packets are generated; and (4) user level communication systems combined with DMA where the packets are generated in user space, and the packets are copied directly to the network card.

Streaming Mechanisms: In addition to mechanisms removing copy operations in all kinds of I/O, some mechanisms have been designed to create a fast in-kernel data path from one device to another, e.g., the disk-to-network adapter data path. These mechanisms do not transfer data between user and kernel space, but keep the data within the kernel and only transfer it between different kernel subsystems. This means that target applications comprise data storage servers for applications that do not manipulate data in any way, i.e., no data touching operations are performed by the application. Examples of such mechanisms are the `splice()` system call [57, 58, 59], the Multimedia Mbuf (MMBUF) mechanism [28], the `stream()` system call [101, 102], and the `sendfile()` system call [186]. The in-kernel data path is created by transferring data by reference, data sharing, or copying data directly between descriptors. Some of these mechanisms are further described in Table 3.3B.

From these three classes of mechanisms, the streaming mechanisms are most appropriate in our scenario, because they are designed for purposes similar to our desired data-path. The networking mechanisms fail to support file system services, because only the application-to-network data path is considered (some of these mechanisms might though be extended). Furthermore, techniques that rely on device-specific characteristics such as programmable DMA or outboard protocol processors cannot provide uniform, device independent copy-free I/O, because these mechanisms cannot transfer data that is already in memory [122], i.e., these mechanisms are more suitable for applications where the user-level process manipulates or generates data itself like in a live-video broadcast. The universal mechanisms support the disk-to-network data path, but the cost of mapping pages into user-space should still be omitted, because the server application does not need data access as no data touching operations are performed. Virtual memory remapping is not a simple technique, and it must be used with care to achieve high performance. Measurements described in [45] show that on data chunks smaller than 4 KB the overhead of mapping pages to another process might be more expensive than copying the data, i.e., although data remapping is usually always faster than data copying, remapping also consumes time. This time comes from kernel virtual memory bookkeeping and from side effects (such as translation-lookaside buffer flushes) of address space changes [122]. Thus, the streaming mechanisms are most convenient, and we have therefore summarized the most important properties from some of the streaming approaches in Table 3.3B.

Table 3.3: Copy avoidance techniques.

(A) Universal mechanisms

	Designed to	OS	Semantic	How
DASH IPC	Support an efficient message passing mechanism for IPC.	DASH kernel for Sun 3/50	Move	Page remapping. The message header containing pointers to possible non-contiguous data pages is physically copied whereas the data pages are remapped.
Container Shipping	Data access and transfer patterns of I/O pipelines, increase IPC.	DEC OSF/1 v2.0 UNIX	Move	Virtual memory page remapping. Pallets (contiguous virtual memory pages) are wrapped in containers (ordered set of pallets). Containers are shipped between domains, and pallets get unloaded (mapped) only if data are accessed; further gains in performance by selective mapping of data.
IO-Lite	Unify all buffering in a system where all processes share a single physical copy of the data.	Digital UNIX v3.2C	Share Copy (CoW)	Shared memory and virtual memory page remapping. Buffering in all subsystems integrated and a single physical copy of the data is shared safely and concurrently. I/O data is stored in immutable buffers whose location in memory never change. Access control and protection through access control lists.
Genie I/O System (emulated copy)	Improving data passing efficiency while preserving copy semantics.	NetBSD	Copy	Selective transient virtual memory mappings. Genie inputs or outputs data to or from emulated shared buffers in-place. Data is shared by managing reference counters, and a page is only deallocated if there are no processes referencing the page. Copy semantic is preserved using input alignment and transient output CoW.
UVM	Replace the Mach based 4.4BSD virtual memory system.	NetBSD	Move/-Share/-Copy¶	UVM provides three new virtual memory based data movement mechanisms: page loanout, page transfer, and map entry passing. The memory can be shared, copied (CoW), and donated (moved). Page loanout and page transfer allow processes to lend out and receive pages of memory, i.e., providing a page-level granularity data movement. Map entry passing allows a process to exchange chunks of its virtual address space, i.e., providing mapping-level data movement.

¶All semantics are supported.

(B) Stream mechanisms

	Designed to	OS	Semantic	How
MMBUF	Remove all physical in-memory copying when transferring data from disk to network. Specially designed and optimized for data transfers from disk to network in an MoD server.	NetBSD	Move/-Share◇	Includes both the buffer cache structure and the mbuf structure in the mmbuf structure. The mmbuf structure is used by both the buffer cache and the mbuf facility. For a stream process, a ring buffer of mmbuf chains are allocated, and two pointers for read and send are maintained in the buffer ring. These are used to read data from disk and to send data to the network interface.

continues on next page

continued from previous page

	Designed to	OS	Semantic	How
`splice()`	Copying between all[†] I/O objects should be minimized. General data transfers between both hardware and software I/O objects	Ultrix 4.2A	Move/-Share⋆	Removes the traditional I/O interfaces, kernel buffer sharing, and page remapping. An in-kernel data path is established where data is moved asynchronously between I/O objects specified by file descriptors. Physical block numbers are remapped from the source's buffer header to the destination's buffer header.
`stream()`	Remove all[‡] physical copying between I/O devices. Designed to support efficient kernel streaming, i.e., direct data transfers between devices without any virtual memory boundary crossings.	Roadrunner	Move	Transfers by reference. A new architecture requiring changes to all I/O systems establishes an in-kernel data path where data is moved between I/O objects specified by file descriptors. Uniform buffer representation for all subsystems allows data transfers with or without transformations, e.g., adding and removing network packet headers.
`sendfile()`	Minimize copy operations in a stream scenario. Used to improve performance in WWW-servers and -caches that transmit a file directly without modifications.	Linux FreeBSD HP-UX ++	Move/-Share⋆	This call copies a certain amount of data between one file descriptor and another file descriptor or socket. After sendfile finishes, the offset is updated.

[◇]The mmbuf structure keeps data pointers to the memory area for both the file system and the communication system (share). However, the access is sequential. The file system writes data into the buffer, and then the communication system reads and transmits the data.

⋆The literature does not say anything about the semantic, but it is either move or share (or somewhere in-between).

[†]The reported implementation supports splices between two file descriptors, between two sockets descriptors, and between a socket descriptor and a frame buffer.

[‡]Depending on the input and output block, i.e., if the block size is variable or fixed and if the input block size is smaller or larger than the output block size, a copy might be required.

3.2.4 Memory Allocation and Increasing Server Capacity by Sharing Resources

Usually, upon process creation, a virtual address space is allocated which contains the data of the process. Physical memory is then allocated and assigned to a process and then mapped into the virtual address space of the process according to available resources and a global or local allocation scheme. This approach is also called *user-centered allocation*. Each process has its own share of the resources. However, traditional memory allocation on a per client (process) basis suffers from a linear increase of required memory with the number of processes.

In order to better utilize the available memory, several systems use so-called *data-centered allocation* where memory is allocated to data objects rather than to a single process. Thus, the data is seen as a resource principal. This enables more cost-efficient data-sharing techniques in the context of multimedia systems [63, 89]:

- *Batching* starts the video transmission when several clients request the same movie and allows several clients to share the same data stream.
- *Periodic services* (or *enhanced pay-per-view*) is a batching policy which assigns each clip a retrieval period where several clients can start at the beginning of each period to view the same movie and to share resources. Such systems are often referred to as near VoD systems.
- *Buffering* (or *bridging*) caches data between consecutive clients omitting new disk requests for the same data.
- *Stream merging* (or *adaptive piggybacking*) displays the same video clip at different speeds to allow clients to catch up with each other and then share the same stream.
- *Content insertion* is a variation of stream merging, but rather than adjusting the display rate, new content, e.g., commercials, is inserted to align the consecutive playouts temporally.

These data-sharing techniques are used in several systems. For example, a per movie memory allocation scheme, i.e., a variant of the buffering scheme, for VoD applications is described in [140]. All buffers are shared among the clients watching the same movie and work like a sliding window on the contiguous data. When the first client has consumed nearly all the data in the buffer, it starts to refresh the oldest buffers with new data. Furthermore, batching multiple requests for the same movie into one logical channel, i.e., broadcast one single stream to multiple concurrent clients, can significantly reduce the number of streams required by a multimedia server. However, if the startup delay is large, the probability of client reneging the playback request increases. In [48], the *staggered broadcasting* scheme is proposed where a video broadcast is continuously retransmitted over k distinct channels at equal time intervals. The request withdraw rate is reduced by setting short batching intervals for popular videos and longer intervals for videos that are less popular. Refinements of this scheme are for example based on client-side buffering. *Stream tapping* [33], *patching* [79], and *gleaning* [66] reduce latency by initially starting a separate stream while tapping into an existing stream at the same time. The data is played out from the initial stream until data from the tapped stream, which is retrieved at the same time and buffered at the client, can be used. Periodic services are used in *pyramid broadcasting* [164]. The data is split into partitions of growing size, because the consumption rate of one partition is assumed to be slower than the downloading rate of the subsequent partition. Each partition is then broadcast in short intervals on separate channels. A client does not send a request to the server, but instead it tunes into the channel transmitting the required data. The data is cached on the receiver side, and during the playout of a partition, the next partition is downloaded. This pyramid scheme has a substantial storage requirement at the client-side. To reduce the client-side buffering and still have low server bandwidth and startup latency, variations of the pyramid schemes, like *permutation-based pyramid broadcasting* [3], *skyscraper broadcasting* [78], *dynamic skyscraper broadcasting* [55], *greedy disk-conserving broadcasting* [62], *harmonic broadcasting* [82], *cautious-* and *quasi-harmonic broadcasting* [116], *polyharmonic broadcasting* [117], and *polyharmonic broadcasting with partial preloading* and *mayan temple broadcasting* [118], have been proposed using more efficient and complex schemes to break up the data stream. For example, the skyscraper policy uses the same broadcasting idea. However, to avoid very large partitions at the end of a movie, and thus, to reduce the client buffer requirement, the partitioning is changed such that not every partition increases in size, but only every nth partition. Harmonic broadcasting reduces the total bandwidth requirement, by dividing the stream into x equally-sized streams, and data from all streams is received simultaneously. Nevertheless, performance evaluations show that the data-centered allocation schemes scale much better with the number of users compared to user-centered allocation [164]. The total buffer space required is reduced, and the average response time is minimized by using a small partition size at the beginning of a movie.

The *memory reservation per storage device* mechanism [64] allocates a fixed, small number of memory buffers per storage device in a server-push VoD server using a cycle-based scheduler. In the simplest case, only two buffers of identical size are allocated per storage device. These buffers work cooperatively, and during each cycle, the buffers change task as data is received from disk. That is, data from one process is read into the first buffer, and when all the data is loaded into the buffer, the system starts to transmit the information to the client. At the same time, the disk starts to load data from the next client into the other buffer. In this way, the buffers change task from receiving disk data to transmitting data to the network until all clients are served. Admission control adjusts the number of concurrent users to prevent data loss when the buffers switch and ensures the maintenance of all client services.

In [106], the traditional allocation and page-wiring mechanism in Real-Time Mach is changed. To avoid privileged users monopolizing memory usage by wiring an unlimited number of pages, only real-time threads are allowed to wire pages, though, only within their limited amount of allocated memory. This means that if more pages are needed, a request has to be sent to the reservation system. Thus, pages may be wired in a secure way, and the reservation system controls the amount of memory allocated to each process.

In our scenario, we try to optimize the data delivery for the most popular movies in a VoD application or the hottest news clips in an NoD application. Which broadcast protocol to use, depends on the application characteristics and the preferences of the system designer. In our system, we want few concurrent channels per video clip, low download rate for each segment, short startup latency, low receiving rate at the client, and minimized buffering requirement at the receiver side. However, several of these goals are contradictory, and we must find a compromise where we find the best trade-offs regarding our server design. The client requirements are not that important since the client receives data from one video-clip (possibly at several concurrent channels) at a time, and a modern PC is able to handle such a workload. Looking at the broadcast protocols above, some kind of periodic service, like pyramid-, skyscraper-, or harmonic broadcasting, will give a good result where the per-client resource usage is minimized, and the server workload will be approximately constant. Mechanisms like staggered broadcasting might have a long startup latency, and stream tapping allows clients to start their own initial streams.

Finally, if we should also support all movies in a VoD database, we might apply a broadcasting protocol for the top video clips only. If several concurrent clients access a media file not supported by the periodic broadcasting scheme, resource requirement can still be reduced applying a caching (or data replacement) strategy like *viewer enrollment window* [140], *distance* [174], *generalized interval caching* [49], *SHR* [83], or *L/MRP* [104, 69] or some kind of stream tapping based scheme [33, 79, 66] for less popular clips. However, this is subject of future work.

3.3 Communication Protocol Processing

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 [9]. However, transferring data through the communication protocols does add costs, even though they are marginal compared to the I/O data path. For example, 235 and 61 instructions are executed to send a packet in the Berkeley implementation of TCP and IP, respectively [38]. The cost of operations on the data itself like checksum calculation dominate header and protocol processing and the related operating system overheads [38, 39, 109]. Nevertheless, 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. ILP [2, 39] 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. In [60], several ways of increasing communication system speed are described. Most interesting in the context of this thesis are the ideas of reducing the number of operations to get more efficient algorithms and to eliminate unneeded or replicated functionality. In the sections below, we take a closer look at some approaches to reduce the overhead of processing data through the protocol stack.

3.3.1 Packet Generation

From the generation of packet headers for a particular network connection, one can make some general observations. For example, in the generation of IP packets sent by a particular TCP 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 a TCP connection will likely carry the same options. In the Berkeley implementation of UNIX, some of these observations are used to associate a template with the IP and TCP headers for each connection with a few of the fixed fields filled in, and Clark et al. [38] have designed a better performance IP layer that 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. [142] 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. This reduces the cost of sending a packet to a few lines of code.

3.3.2 Checksum Caching

Checksum calculations are known to be a time consuming operation. In [85], 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 adapter, and some systems allow previously calculated checksums to be cached in memory. In [90, 91], 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. The IO-Lite unified I/O buffering and caching system [112] 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 [134], caching of prebuilt transport level packets, in both end-system and network nodes, is used for load balancing and to reduce resource usage.

3.4 Discussion and Conclusions

The previous sections present a summary of the much significant works 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. Moving data between different protection domains along the I/O-pipeline and processing them through the different subsystems and communication protocols is very time consuming. Here, we summarize some of the limitations of traditional I/O mechanisms when they are applied on networked multimedia systems:

- Data is retrieved from disk and copied several times between different memory address-spaces. This copying is not a hardware constraint, but it is imposed by the system's software structure and its interfaces. A prime example is UNIX-based systems which require the physical copying of data between different protected domains, e.g., user and kernel space [92]. Physical copying is detrimental to the performance of operating system and system-related software [121]. Copying and processing data in different subsystems also introduces a lot of context switches which decrease the utilization of the CPU. Some copy performance measurements are presented in Section B.2.

- Different parts of the end-system may add processing costs. These overheads include data touching operations, like checksum calculations and encryption, and non-data touching operations, such as network buffer manipulation, protocol specific processing (e.g., generating packets and setting header fields), operating system functions, and error checking. In an MoD system, for instance, large messages or packets should be used to achieve maximal throughput. However, data touching operations' overhead dominate the total processing time[7], and this overhead scales approximately

[7]Non-data touching operations' overhead is roughly constant and only indirectly affected by message size, i.e., overhead increases with the number of allocated buffers or fragments into which a message is divided. Thus, the processing time for large messages is dominated by data touching overhead [84].

linearly with the message size. For example, using large messages, checksum calculation and data movement operations can consume about 70% of the total processing overhead time [84].

- There is redundancy in the functionality of different subsystems meaning that different subsystems perform the same operations or functions. In the data management system and the communication protocols, for example, buffer management and error handling are often performed by both subsystems. This lack of integration requires that several identical copies of a data object are stored in main memory where the effective size of physical memory is reduced by this duplication which in turn may affect the hit rate increasing the number of disk accesses.

- A lot of concurrent users might request the same data which means that data could be processed by the same subsystem several times. This introduces unnecessary overhead and could be avoided. For example, the network subsystem is forced to recompute the checksum for the network packets each time a data object is transmitted. Removing the checksum as proposed in [85] 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 [156]. This is due to transmission errors in the end-systems or in the intermediate nodes due to hardware failures or software bugs.

- The subsystems are not adapted to each other to guarantee services and optimize server performance, e.g., QoS management, size of storage blocks, and protocol data units. In order to be able to support the requested QoS, all the system components have to support the same QoS. This means that if one component is not able to support the requested service, it will become a bottleneck. Thus, the whole system will fail to provide a service with the requested quality.

Because of its ignorance of these issues, the traditional I/O system is an end-system bottleneck for supporting high bandwidth, networked multimedia applications. Much work has been done in an attempt to eliminate some of these bottlenecks, but little has been done to integrate these solutions into one system, investigate how they work together, and propose a cohesive set of mechanisms that provide a more complete and efficient solution. We have addressed three of these limitations in this thesis. In the next chapter, we describe the design, implementation, and integration of our mechanisms, and we further evaluate the related work described so far with respect to the mechanisms we have designed for our MoD storage server.

Chapter 4

Design and Implementation

The overall goal of our research is to maximize the number of concurrent clients a single MoD server machine can support using standard off-the-shelf components. This is done by making optimal use of a given set of resources, i.e., reducing the resource requirement both per-client and per-data-element. We have designed and implemented a fast in-kernel data path for the common case operation in a storage server, i.e., retrieval of data from the storage system and sending it to receiver(s) over the network, where all in-memory copy operations are removed. Furthermore, a single data element is shared by all concurrent users using a broadcasting scheme which also minimizes the start-up delay. Finally, to reduce the number of consumed CPU cycles, we remove as many instructions as possible for each read and send operation by removing redundant error management functionality and by prefabricating transport level packet information.

In this chapter, we describe the requirements, design, and implementation of our mechanisms tuning the I/O performance in an intermediate multimedia storage server. We assume that our server machine has a disk array for persistent data storage, and we use the traditional Internet protocol suite [131, 132, 133, 153] for transmitting data, i.e., we use UDP over IP (IPv4). Section 4.1 describes the integrated error management scheme [72] where storage system parity data is reused for FEC recovery data in the communication system. In Section 4.2, we show how we reduce resource usage by optimizing memory management, and how our NLF mechanism [71] reduces communication protocol processing is presented in Section 4.3. Finally, we give an overview of the integrated server in Section 4.4 and conclude with a discussion in Section 4.5.

4.1 Integrated Storage and Communication System Error Management

When sending data from disk in a server to a remote client in a distributed environment, several components along the data path may fail and introduce errors. Error detection and correction schemes are used in several subsystems, i.e., in the storage system and the communication system, so the correctness of the information is checked multiple times as shown in Figure 4.1. A RAID system generates redundant parity information to be able to recover from a disk crash and restore the lost data. When data is sent to the communication system in the FEC case, another error mechanism is applied where a new set of redundant parity information is computed to either detect or correct a possible network data corruption.

Figure 4.2 shows our approach to integrate the different error recovery mechanisms transmitting data using FEC over UDP. As opposed to the traditional data read from a RAID system where the parity information is only read when a disk error occurs, we want also to read the redundant error recovery data and transmit this data to the remote client for use in the FEC scheme in the communication system. Thus, instead of reading only the original data, we also read the parity data by simulating a RAID level 0 read operation. All the data retrieved from the storage system is sent to the communication system where the FEC encoder, which performs CPU expensive operations, can now be omitted. Finally, the data is sent

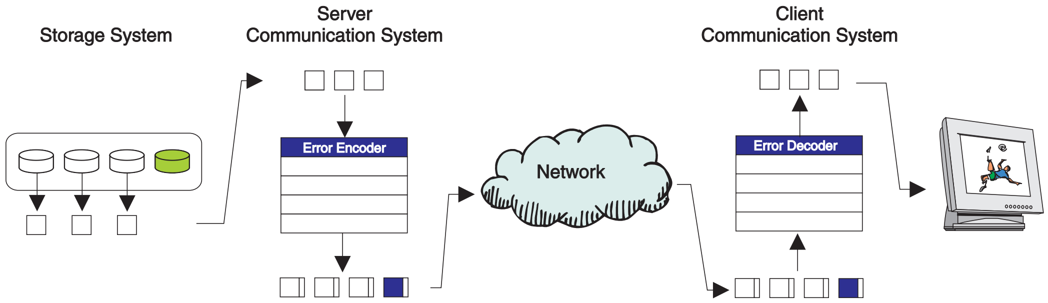


Figure 4.1: Traditional error management.

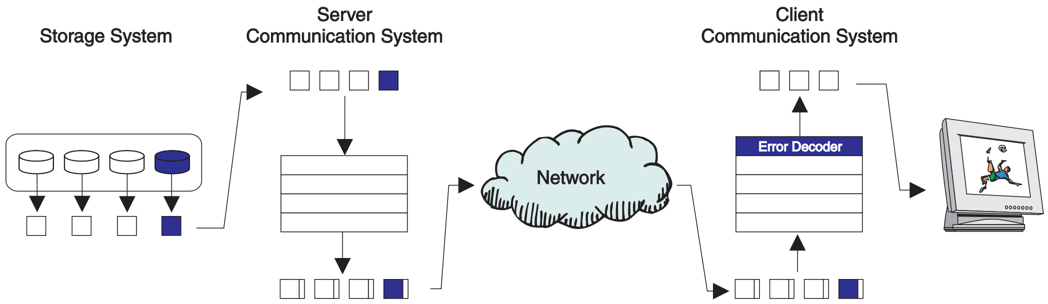


Figure 4.2: Integrated error management.

to the client where the communication system has an identical FEC decoder as in the storage system on the server side.

In the following subsections, we look at our integrated error management scheme [72] including the possible benefits and drawbacks of such a mechanism, the correcting capability requirements of a suitable mechanism, and possible solutions supporting the requirements from both storage and communication system.

4.1.1 Application of an Integrated Error Management Scheme

The attractiveness of FEC has changed during the past decades. In traditional networks, ARQ is used almost exclusively, because variable capacity telephone lines made FEC inefficient and unreliable, and because the complexity of FEC algorithms made the execution of these procedures too slow and expensive. However, significant changes in communication technology for switched networks has reduced the number of errors, making congestion the dominant source of errors, i.e., thus, enabling the use of more efficient codes [99]. Nevertheless, both ARQ and FEC have their advantages and disadvantages. ARQ is generally used in unicast protocols, because it is very efficient, simple to implement, and does not require any processing of the data stream. On the other hand, it performs badly due to high buffer requirements and long delays when the number of errors increases. This means that its biggest drawback is the need for a feedback channel and the time required to recover missing packets. Another disadvantage of ARQ-based schemes is the overhead, including both CPU cycles and buffer space, required to keep track of the potentially large number of outstanding messages [10]. FEC, however, also has to transmit redundant data for error recovery, which depending on the correcting capability may greatly exceed the amount of redundant information using checksums in ARQ. In other words, FEC trades bandwidth for latency to improve packet corruption rate. Thus, an important question is whether a FEC-based integrated error management scheme is appropriate, using the redundant data from a disk array for error recovery in the storage system and for FEC in the communication system. Next, we discuss some of these factors further in the context of an MoD scenario.

4.1.1.1 Loss Tolerance

One might argue that some data loss in a multimedia presentation, like the loss of a video frame, will not degrade the perceptual quality of the playout, because the duration of each data element is often very short [136], e.g., a video frame in a 30 fps presentation is displayed for 33.33 ms. The data loss may be concealed for example by substituting a black frame for a lost frame as in [30], or just displaying the previous frame a bit longer. However, in the case of a coded data stream with internal dependencies, an error can destroy data used to predict the next data element(s). For example, in an MPEG coded video [151], where we frames decoded with respect to both previous and future frames (P- and B-frames) , data loss may propagate through several frames. This means that if the error occurs in a reference frame (I- or P-frame), it will spawn until the next intra frame (I-frame) is presented resulting in a very corrupted presentation due to the inter-frame prediction [19, 30]. Thus, some kind of error recovery is necessary to prevent the error from propagating and degrading the quality of the multimedia presentation for a longer period.

4.1.1.2 Multicast and Broadcast Scenarios

Unlike ARQ which has to retransmit data which is lost or corrupted in the network, FEC can tolerate some amount of loss due to client self-reconstruction on damaged data. In multicast connections, FEC can therefore have a big effect on server-side error management overhead since the global packet loss will increase with the number of receivers and links. For small multicast groups, the gain of using FEC instead of ARQ is enlarged quickly as the number of receivers increase and the number of shared links decrease. For large multicast groups, the gain is even larger and independent of the number of remote clients [108]. Using ARQ as the end-to-end error control scheme in such an environment performs poorly, does not scale well, and requires a lot of buffer space [136]. ARQ scales poorly for multicast protocols and large groups, because the sender might have to deal with exceedingly growing number of acknowledgments (ACKs) or negative ACK, and the chance for packet losses grows as the size of the group grows. Another problem with ARQ in multicast protocols is that it requires a precise feedback from the receivers in order to decide which packets to retransmit. FEC-based multicast protocols scale much better for large groups than ARQ-based protocols, since packet losses will be handled at the receiver and will not occupy the multicast line. Thus, FEC is a good solution for broadband communications where data is multicast over wide area networks to a large group of clients [29, 50, 99, 136]. In such a system, performance is significantly degraded by using ARQ for error control [96, 167], because the transmitter complexity is proportional to the number of receivers, the latency may increase with the number of retransmissions, and throughput of the server is reduced as the transmitter is busy sending retransmissions.

Even though ARQ usually is considered unusable in multicast and broadcast scenarios, there exists work improving the TCP implosion and exposure problems. In [115], a retransmission-based error control scheme is presented dealing with implosion and exposure, and at the same time has a low recovery latency suitable for delay sensitive data transmissions. The scheme typically recovers from errors in one RTT or less by adding a replier to each router, i.e., the replier acts as a feedback and retransmission server for the sub-trees connected to the router.

4.1.1.3 Real-Time Services

For real-time data services, or time sensitive and delay critical services in general, ARQ is not suitable [6, 15, 19, 29, 96, 105, 127, 146]. A large drawback of ARQ is the need of a feedback channel and the time required recovering missing packets. This is usually an important limitation when there are time sensitive applications involved, e.g., a retransmission of a lost video frame may be wasted due to the high bandwidth requirement, and the retransmitted video frame will probably be old (the playout of the video has passed the point where the lost frame should be displayed) when arriving at the client site. Likewise, low latency is necessary for applications supporting human interactions [10], like a video editing bench, process control, and remote sensing, in which the delay of retransmission will make the result unusable in worst case. Consequently, FEC may be an appropriate error recovery scheme for time sensitive, high bandwidth applications providing reliability without increasing end-to-end latency.

Nevertheless, there exists work extending TCP for continuous media streams. In [114], an enhanced selective-repeat retransmission scheme is presented. The performance measurements show that ARQ can successfully be used if the RTT is not to large and some playout buffering is employed.

4.1.1.4 Network Characteristics

The potential of FEC to recover from losses is highly dependent on the loss behavior of the network [10]. A wired network is more reliable and less vulnerable to bit errors due to noise than wireless networks, and heavily loaded nodes are more exposed to packet loss due to congestion. This means that if the network has a low error probability like many of today's wired networks, the transmission of redundant data for error recovery in FEC may be unnecessary, and ARQ should be used giving a better overall performance. However, the skewed arrival of packets sent long distances over a variety of heterogeneous networks is a global feature that seems to be harder to control on a local basis, i.e., by the server. It therefore makes sense to protect traffic sent through such networks against losses by adding a moderate level of redundancy using some kind of FEC [13]. Furthermore, as we now approach a more mobile world where all kinds of networks are used, and the users (soon) may use different kinds of receiver devices ranging from a traditional PC or workstation to a cellular phone or a personal digital assistant, the gain of using FEC will increase. In fact, for high packet loss ratios, as is the case with noisy wireless channels, the throughput of FEC-based systems tends to be higher than the throughput using ARQ [6].

4.1.1.5 Heterogeneous Environments

A possible drawback of FEC in addition to the increased bandwidth is that, in heterogeneous scenarios, i.e., those with receivers of different capabilities and different requirements with respect to error recovery ability, conventional FEC is not optimal. Receivers that do not need the improved probability of an error free connection or better quality of the multimedia playout are forced to receive and process the data stream consisting of user data and redundant data [50], and even an "optimal" FEC scheme cannot provide guaranteed quality given the best-effort service of today's current Internet.

4.1.1.6 Resource Requirements

In order to see the resource requirements using TCP, we give a simple example assuming a data transmission between Oslo and Tromsø, i.e., a distance of about 1600 km. To be able to retransmit data, TCP must hold data for at least one RTT. If we assume that data is transmitted at the speed of light (299792458 meters per second), and that we neither have delays in the intermediate nodes nor in the end-systems processing data, managing timers, and sending acknowledgments, the time between sending the data and receiving a confirmation is minimum 10.67 ms, i.e., propagation delay only. In this case, TCP must hold at least 4.78 KB of data assuming a 3.5 Mbps bit rate. TCP uses a default transmission window of 16 KB (`TCP_SENDSPACE`) which in this case holds. However, our assumption that RTT equals the propagation delay only does not hold. Along the transmission path there will be several delays: driver queue at sender side, processing and queues in intermediate nodes, and processing (for example checksum) and queue at client side. Likewise, the acknowledgment packet returned to the sender must travel the same way back experiencing the same delays, i.e., the real RTT will be higher than in our example giving a higher buffer requirement, and using `ping(8)`, the average RTT was measured to be 22 ms to *www.cs.uit.no*.

The example described above shows a best-case scenario where there are no losses and data only need to be buffered for one RTT. However, if a packet is lost, the amount of data to be buffered is the amount of data consumed from the data element is transmitted until the retransmission timer timeouts.

The retransmission timeout (RTO) value may vary for each connection. Initially, it is recommended to set the RTO to three seconds [21, 125] (NetBSD uses six seconds), and the three seconds value seems to be appropriate based on the measurements in [5], i.e., slightly more than 1 % had an average RTT over three seconds. During the transmission, the RTT is measured, and the RTO is updated according to the measured value as described in [125, 153]. However, it is recommended to at least have a RTO of one second [125], and in case of a time out, the time between retransmissions is doubled (exponential backoff). Still, a lower RTO is possible, but in NetBSD, values below 500 ms are not meaningful, because the TCP protocol timeout routine is called only every 500 ms. Thus, during start-up we might have to hold as much as six seconds of data and at least 500 ms during the connection in NetBSD, i.e., 2.6 MB and 224 KB, respectively.

Our examples assume that the send window can be arbitrary dynamically changed. However, if the used window size is static, the maximum throughput is given by $\frac{window\_size}{RTT}$. Without the window scale option, the maximum size is 64 KB (`TCP_MAXWIN`), but with the window scale option, the maximum size is 64 KB $\times$ 2^n where n is maximum 14 (`TCP_MAX_WINSHIFT`). Assuming the (unrealistic) RTT above and maximum, non-scaling window, we can send at 4.7 Mbps which is slightly above the average DVD data rate.

Using FEC, we do not need any retransmission buffer, but we do require n % more buffer space and bandwidth to transmit n % parity data used for error recovery at the client side. Additionally, the FEC coding operations consume CPU cycles. Thus, both ARQ and FEC schemes require resources, and which scheme to use for a unicast stream is basically a trade-off between latency and bandwidth, respectively. In the case of multicast streams, one should also consider scaling properties.

4.1.1.7 Summary

To avoid losses, some kind of error management should be applied since an error might propagate through several video frames. Both ARQ and FEC have their advantages and disadvantages, but the conclusion from the discussion above is that ARQ is usually not suitable in an MoD scenario due to long latency of a retransmission, high overhead processing ACKs (especially in multicast transmissions), and large buffer space requirement.

Despite its disadvantages, FEC can improve the multimedia playout quality even on networks with high or highly varying loss rates. Recent results also suggest that network error control schemes using FEC are good candidates for decreasing the impact of packet loss on the media quality [15]. Using a FEC-based integrated error management mechanism combining disk array recovery with FEC in our scenario, i.e., transmitting high bandwidth multimedia data to a large number of users in a heterogeneous environment, therefore seems to be a good solution. The latency introduced by retransmission-based error recovery schemes will be too high for applications with latency constraints, and a lot of buffer space and processing overhead is required to handle the outstanding packets. Thus, FEC is usable, because almost no buffering is required at the server side, there is no need for a retransmission channel, it scales well in a multicast scenario, and the operation of error recovery is handed over to the client rather than assigning this job to a heavily loaded server.

4.1.2 Correction Scheme Requirements

In the previous section, we argued for why FEC is a good solution for communication system error recovery in a real-time, multicast MoD scenario. However, a FEC scheme imposes a substantial encoding overhead in a multi-user server. In our integrated error management scheme, this overhead is removed. A challenge then is to find a code that suits both the storage and the communication system, and the integrated error management scheme must have the following properties in a cost efficient, high performance multimedia server:

- *Errors/erasure correcting capability:* The coding scheme must be able to correct both bursts of bit errors, e.g., introduced by some kind of noise during the data transmission over the network, and large groups of erasures, e.g., due to a packet loss, a damaged disk block, or a disk failure. The number of burst bit errors depends on the underlying network media where wired networks are getting more and more reliable, but wireless networks will still be a source of introducing errors. Moreover, to be able to recover from a disk failure or a packet loss, a set of codewords and its error correcting information must be striped over several disks and distributed over several packets. Thus, the size of the erasure group, i.e., erasure correcting capability, is determined by the striping units in the RAID system and the network packet size in the communication system (or the size of the striping units and the network packet size must be suited to fit the correcting capability of the error management scheme).

- *Throughput (performance):* The error decoding algorithm must be efficient and fast to be able to decode multimedia data in time for playout. For example, audio in telephony and CD-quality require about 16 Kbps and 1.4 Mbps respectively, and a video stream's requirement ranges from approximately 1.2 Mbps for MPEG, to 20 Mbps and more than 1 Gbps for compressed and uncompressed high definition television (HDTV) respectively [105, 151]. The current MPEG-2 DVD standard has an average bit rate of 3.5 Mbps depending on the length, amount of audio, etc., and the maximum bit rate is 9.8 Mbps (10.08 Mbps for audio, video, and sub-pictures) [184, 194]. This requirement is not that important for the encoding algorithm, because the encoding operation is done once, and the coded error correction data is stored on and retrieved from the disks.

- *Amount of redundant data:* Since the error correction information will be transmitted to the remote client together with the original data, a higher bandwidth is needed. As the bandwidth requirement of multimedia streams already is very high and as increased bandwidth introduces more noise raising the probability of an error, we should find a scheme which minimizes the amount of redundant information. Nevertheless, this might be a contradictory requirement compared to the correcting requirement above, because the correcting capability is often dependent on the amount of redundant error correcting information.

- *Applicable within a single file:* We want to transmit both the original information and the stored redundant correcting data, but we do not want to transmit data from other files. This means that correcting schemes which calculate parity information based on disk setup regardless of which files the disk blocks (or striping units) belong to, i.e., the redundant information may span several different files, are not suitable to serve our purpose.

- *Decoding costs dependent on loss rate:* The cost of the decoding function should be dependent on the amount of damaged data, i.e., if no data is corrupted, the decoding function only checks if all the data is received, and no decoding is necessary. Thus, clients with a reliable network connection should experience nearly no overhead.

- *Systematic code:* An appropriate recovery code must be systematic in that the codeword itself explicitly contains the information block, i.e., the parity data is not interleaved into the original data block. This is because in an error-free environment, where parity data should not be transmitted, a non-systematic code would still have to transmit all data. Likewise, if a client on the server itself would like to perform a data playout, no network errors will occur, but a non-systematic code must run the data through the decoder.

- *Memoryless code:* The code must be memoryless, operating on fixed sized blocks, because the storage system (disks) stores data using predefined block sizes.

4.1.3 Error Model in Our MoD Scenario

In Section 3.1.1, we described typical data corruption and loss rates. Disk failures and bit errors are quite rare, but they do occur. The dominant cause of data loss is congestion. The measurements described in Section 3.1.1, show that consecutive losses up to 30 packets are average, and the total loss rate is somewhere between 0 - 15 %. With these numbers as a basis, we decided to configure our prototype such that it should be able to correct 12.5 % of loss or data corruption. With an eight disk RAID system, we tailor the size of the codeword, which is the amount of data subject to reconstruction, to contain 256 symbols, i.e., 224 for application data and 32 for parity data. A symbol is a single data element within the codeword. The symbol size is a configurable parameter, and each symbol is transmitted as a single packet, i.e., packet size equals symbol size[1]. In summary, using this error model and the described codeword configuration, we are able to correct one disk failure in the eight-disk disk array and any 32 out of 256 packets in the communication system.

4.1.4 Finding a Suitable Correction Scheme

To find a recovery code that meets our requirements, we have evaluated the existing non-commercial codes[2] described in Section 3.1.2.1 with respect to the requirements described in Section 4.1.2. Thus, a suitable code must meet the demanding requirements of both subsystems. There must exist a possible system configuration, i.e., striping layout, network packet size, and amount of redundant correction information, that gives adequate performance (decoding must be done in time for a hiccup free playout on the client side) and that corrects the varying number of errors and erasures that occur.

First, since we have blocks of fixed size in the storage system, a block code rather than a convolutional code is preferred. This means that also the IETF FEC scheme [126] used for audio data in [15, 17] and video data in [16] sending a low quality copy of the data as redundant data in subsequent packet(s) is not appropriate in our integrated solution. During a disk recovery operation we would like to be able to restore the data with the original quality which is not possible with this scheme unless the redundant data is an exact copy of the original data, i.e., similar to mirroring. However, transmitting two identical copies for error management is not suitable for example due to bandwidth requirements.

Second, the traditional RAID recovery mechanisms are not applicable in the communication system. In RAID level 1 (mirroring), all the data would be sent twice giving 100 % redundancy. RAID level 2 - 5 recovery using simple XOR-based reconstruction is not appropriate, because this error management scheme cannot handle the loss of consecutive packets due to network congestion, i.e., only one out of eight packets might be lost or corrupted in this scenario. Furthermore, the idea of multi-dimensional parity calculations used in some disk array configurations to correct more than one disk failure, like RAID level 6 and as proposed in [12, 77], cannot be applied since the error correction information corrects disk blocks spanning several different files. Since we want to reuse the redundant information for FEC in the communication system where only one file is transferred to a remote client, parity codes spanning only one file are appropriate.

Thus, suitable codes comprise traditional block codes only, because these codes can easily be configured in a suitable way. Turbo codes can be viewed as block codes, but they use convolutional codes for generating parity and are therefore not evaluated. Tornado codes are very similar to Reed-Solomon Erasure codes, but require slightly more redundant data and will therefore not suit the parity data requirement in the storage system. In the following subsections, we evaluate some Reed-Solomon-like codes. We have performed several experiments on a 167 MHz Sun UltraSparc1 to test coding speed. We

[1] The difference between symbol and packet might be a bit confusing, but a symbol is used in the context of the error management mechanism and packet in the context of the data transmissions. Nevertheless, each symbol is sent as an individual packet, i.e, they have equal sizes.

[2] Designing and implementing a new code is beyond the scope of our research. Our primary goal is to demonstrate that an integrated error management scheme will save a lot of processing resources on the server and that it can be designed using existing codes.

used *Quantify*[3] to estimate the number of used CPU cycles, the time to execute the code, and the coding throughput. In the following subsections, we present the conclusions from these tests, and we outline a possible solution selecting the best coding scheme for our requirements.

4.1.4.1 Reed-Solomon Codes

The easiest and least complex solution of integrating the storage and communication system data recovery management would be to use a correction scheme which corrects both errors and erasures. Traditional Reed-Solomon codes have these correcting capabilities, and we have tested this kind of code[4]. However, this code fails to support some of the requirements above. Conventional FEC schemes using Reed-Solomon have a uniform error correction capability which can result in poor bandwidth utilization due to a complex algebraic decoding operation, and the amount of redundancy is doubled compared to traditional storage system (RAID) recovery schemes. Since we do not know whether the data is damaged or not, the decoding function must always be performed on the whole codeword, i.e., client side cost is not dependent on the loss ratio. Additionally, the overall coding performance will usually (depending on hardware) not be efficient enough for a multimedia presentation. Our tests show in a best-case (no errors in the data stream) and a worst-case (12.5 % loss) scenario decoding throughput of 3.6 Mbps and 1.7 Mbps, respectively. Thus, we do not consider pure Reed-Solomon as suitable for our integrated error management scheme.

4.1.4.2 Reed-Solomon Erasure Codes

In current wired networks, the probability of getting a bit-error is very small, and congestion losses are usually the dominant form of data loss. The network can often be modeled as a well-behaved erasure channel where the unit of loss is a packet. Thus, by only using erasure correction, recovery can be greatly simplified [6], and correcting power can be approximately doubled [10].

If we assume that bit errors are negligible, we can apply only an erasure code like the ones used in [6, 99]. We have evaluated such a solution where the erasure correcting code is used as the storage system failure recovery mechanism and as the erasure correcting FEC scheme in the communication system. There exist several possible implementations of such codes, e.g., Reed-Solomon erasure codes based on Cauchy matrices [13][5] and Vandermonde matrices [136] over finite fields. Since the Cauchy-based code is reported faster than the Vandermonde-based code [29], we have performed tests on this code using the same scenario and the same correcting scheme as described above.

The encoding time is approximately constant when generating the parity data, but the decoding time may vary, i.e., increasing with the amount of loss [13]. The decoding function first checks how many of the original packets are received, and if all have arrived correctly no further decoding is necessary. The decoding measurement is therefore a worst case scenario within the limits of the code, i.e., we lose 32 of the packets containing real data. Our performance measurements show that on the 167 MHz Sun Ultrasparc1, we can achieve decoding (and encoding) throughputs above 6 Mbps if the packet size is above 512 B, and in a more reliable network where the amount of loss is reduced, the code will perform even better.

Thus, if we assume that bit errors are negligible, this is an suitable code. However, one of our goals in designing a multimedia server is to support presentation of data to a wide range of remote receivers. Thus, data will also be transmitted over more error-prone networks, like wireless networks, suffering

[3]Quantify estimates the needed CPU cycles and gives an upper limit of the performance, because the estimated times are calculated using the cycle count divided by the CPU clock frequency, i.e., assuming the program has the CPU alone. To compute the execution times in ms and the throughput in Mbps, Quantify assumes an UltraSparc with clock rate of 168 MHz. Please note that these results exclude Quantify overhead and also possible memory effects, i.e., Quantify subtracts the overhead of using resources itself.

[4]We used the Reed-Solomon code implemented by Phil Karn available at http://imailab-www.iis.u-tokyo.ac.jp/∼robert/rs.tar.

[5]The Reed-Solomon Erasure code described in [13] is available at http://www.icsi.berkeley.edu/∼luby/cauchy.tar.uu.

from data destruction caused by different kinds of noise. This means that we also must take bit errors into consideration when designing our system. Pure Reed-Solomon erasure codes fail to support network topologies more vulnerable to unknown errors and are consequently not suited for recovering this type of errors in the communication system.

4.1.4.3 Combined Checksum and Erasure Codes

In the previous subsections, we have showed that error correcting schemes have insufficient performance, and the decoding cost is independent of the amount of corruption. Schemes that correct only erasures fail to reconstruct damaged data due to bit-errors. However, one possible solution is to still view congestion (packet) loss as the dominant data corruption source. Then, we combine the erasure correcting scheme running on large blocks spanning multiple packets described above with some kind of error detection (or correction) scheme running on small blocks within a single packet. Thus, we get an inner and an outer mechanism of error management. This means that a fast efficient inner error detection is applied within each packet to detect (or correct) bit errors introduced by noise, and an erasure correcting scheme is applied to a larger information block spanning several packets capable to correct a large number of erasures (see Figure 4.3). If an error is detected, but cannot be corrected, by the inner mechanism, or the whole packet is lost, it is marked as an erasure where the outer recovery scheme is responsible for restoring the original data block. In such an integrated recovery mechanism, the outer erasure correcting scheme will also work as the storage system recovery mechanism and correct failures due to bad disk blocks or disk crashes.

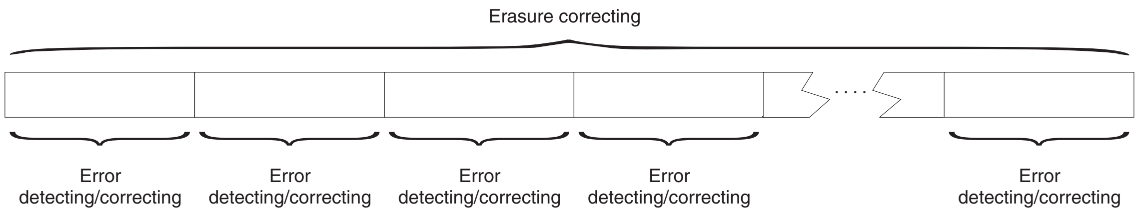


Figure 4.3: Inner and outer error management mechanism.

In such a design, we have two alternatives. The inner mechanism can be either detecting or correcting. If we have an inner correcting scheme, i.e., using a code similar to the Reed-Solomon code tested in Section 4.1.4.1, the performance would still be a bottleneck, and we need twice as much redundant data compared to a pure Reed-Solomon scheme. Thus, the inner scheme should only be error detecting, marking the packet as corrupted, and letting the outer erasure scheme correct the damage. This kind of scheme requires less redundant data and performs better as error detecting is faster and less demanding compared to error correcting. Furthermore, the easiest solution in this approach is to use the traditional 16-bit transport level checksum [20, 68, 159] used in traditional protocols like UDP and TCP for error detection, because nearly no additional redundant data is added compared to the Reed-Solomon erasure codes, and the checksum calculation operation is already optimized for performance on different architectures.

Next, we describe our prototype design and implementation. We use the Reed-Solomon erasure code based on Cauchy matrices as the storage system error recovery mechanism and outer error management scheme in the communication system. Within each packet, we use the traditional transport level checksum for error detection, and if a packet is damaged, it is marked, and we leave it up to the erasure code to correct the error.

4.1.5 Integrated Error Management Prototype

In this section, we present a prototype for the integrated error management scheme using the Cauchy based Reed-Solomon Erasure code. We first describe a general scheme, and then we present our prototype scheme.

4.1.5.1 General Scheme

The Cauchy based Reed-Solomon erasure code [13] is very flexible regarding configuration. Generally, this code can be configured as (n,m) over $GF(2^L)$ where n is the total number of symbols in a codeword, m is the number of information symbols, and L determines the size of the finite field. The number of parity symbols is given by $n - m$, and if s denotes the number of segments within a symbol, the size of each symbol in bytes is given by $4 \times s \times L$. n, m, L, and s can be arbitrarily configured in Z^+ as long as $n, m < 2^{L-1}$. Since the code also is to be used in a disk environment, the symbol size should be a power-of-two to fit into a disk block, i.e., the size of one or multiple symbols should equal the block size. n should be chosen to be $\frac{\#disks \times disk\_block\_size}{symbol\_size}$, and m should be chosen to be $n - \frac{\#parity\_disks \times disk\_block\_size}{symbol\_size}$ if the error models have corresponding loss behaviors in the storage system and the communication system. If the network error model indicates a higher loss rate, we should decrease m to allow more parity symbols in each codeword, i.e., we will also have parity blocks on other disk blocks than the parity disk, or increase the number of disks in the disk array and have more than one parity disk.

As we can see, there are several possible schemes that can be configured using this code. Almost any error model can be handled, but the performance of the required configuration should be tested. Next, we describe the prototype scheme where we have tested the code given an eight-disk disk array and the error model described in Section 4.1.3, i.e., bit errors are rare and the dominant cause of loss is congestion.

4.1.5.2 Example Scheme

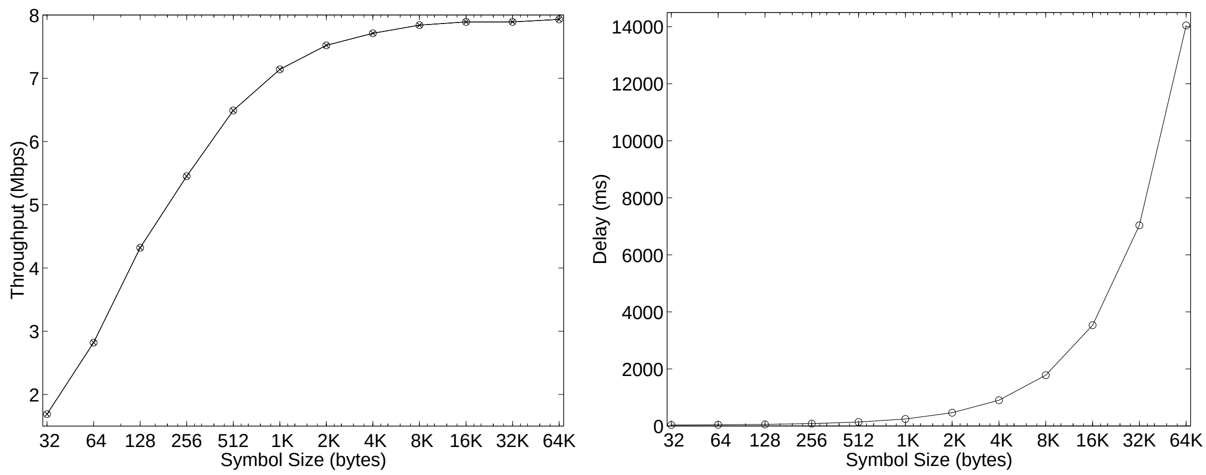
This section present our example scheme for the integrated error management mechanism using the Cauchy based Reed-Solomon Erasure code in a multimedia storage server offering services like NoD or VoD. We have tested different code configurations and used a minimum coding limit of about 6 Mbps to find the configuration useful. In this scenario, we want to be able to recover from a single disk error[6] in the disk array.

The Cauchy-based Reed-Solomon Erasure code experiments show that this code can reach decoding times appropriate for a multimedia environment like ours. We use a (256,224) over $GF(2^8)$ Cauchy-based Reed-Solomon Erasure code. Thus, any 32 out of 256 data blocks can be corrupted and the code will still be able to recover the damaged data. In the storage system, this means that data is striped over eight different disks where one is a parity disk. The communication system transmits each block as a single packet and can get about 12.5 % of the packets garbled by noise or lost in the network. Thus, the algorithm should be able to recover from most of the losses according to the measurements reported in Section 3.1.1.

There are also various factors to consider when deciding the network packet size, because the codeword symbol size has impact on the decoding performance. Since each symbol in the correcting code is sent as a separate packet, the packet size affects the coding performance. Decoding performance, throughput and start-up delay, depends on the codeword configuration, the symbol size, and the available CPU resources. This also applies to encoding operations, but decoding speed is most interesting. To find suitable symbol size values, we extended the tests performed in Section 4.1.4.2 on the Cauchy-based

[6] It should be adequate to be able to recover from a single disk crash using today's top-end disks, because, assuming disk errors occur independently, the probability of two disks failing at the same time is very low, i.e., 10^{12} using Seagate Barracuda and Cheetah disks [202] or Western Digital Enterprise disks [207].

Reed Solomon Erasure code using symbol sizes from 32 B to 64 KB on a 167 MHz Sun UltraSparc 1 using the reported codeword configuration and a worst case loss scenario, i.e., dropping 12.5 % of the packets or one disk out of eight correspondingly. Figure 4.4A shows that the throughput increases with larger symbol sizes. If we use symbol sizes equal or larger than 1 KB, the client system will be able to decode streams with data rates of 6 Mbps and more on this system (the performance levels out after 1 KB symbols). However, increasing throughput by using larger symbol sizes also increases the worst case start up delay (see Figure 4.4B), because there is more data to decode per codeword. Therefore, to reduce the experienced start-up delay, we do not use symbol sizes larger than 8 KB. This results in a worst-case decoding delay of 2 seconds or less on the 167 MHz Sun UltraSparc 1.



(A) Throughput. (B) Delay.

Figure 4.4: Cauchy-based Reed Solomon erasure decoding.

Furthermore, we do not want the codeword to span different files, and each codeword has a fixed length, i.e., half a codeword will on average not be used and will result in some fragmentation on the disk. A large symbol size gives a greater fragmentation, i.e., a codeword with 256 symbols and a 1 KB symbol size gives an average (max) fragmentation of 128 KB (255 KB). However, compared to a 5 minutes, 6 Mbps video clip of 225 MB, this still might be applicable. Additionally, each network has an MTU, and each packet must fit in the MTU. If a packet has a larger size than the MTU, the packet is fragmented into two or more fragments introducing extra overhead in the network. Moreover, many small packets might result in an overflow in the scheduling queue in the routers (congestion) if the maximum queue length is reached, and packets might be lost.

To avoid further fragmentation, we should use a disk array stripe which suits the correction code codeword. Thus, a stripe that contains a single codeword or a solution where a codeword is distributed on several whole stripes might be convenient. Using our (256,224) over $GF(2^8)$ Cauchy-based Reed-Solomon Erasure coding scheme, striping the codeword symbols across 8 disks results in 32 codeword symbols stored on each disk. This gives us four different schemes for our prototype shown in Figure 4.5 using packet sizes (or codeword symbols) of 1 KB, 2 KB, 4 KB, and 8 KB. These packet sizes imply codewords of 256 KB, 512 KB, 1024 KB, and 2048 KB (containing 224 KB, 448 KB, 896 KB, and 1792 KB original data), respectively. Using a 2 KB packet as an example and 64 KB disk blocks (striping units), the 224 information packets are stored on the seven data disks, i.e., 32 symbols within each disk block. These data are also used to generate 32 parity symbols which are stored in one disk block on the parity disk. Thus, this configuration corresponds to RAID-level 4 write operation.

The performance gain of our integrated error management mechanism is achieved during the data retrieval and transmission to the remote clients. During retrieval simulate a RAID level 0 read operation

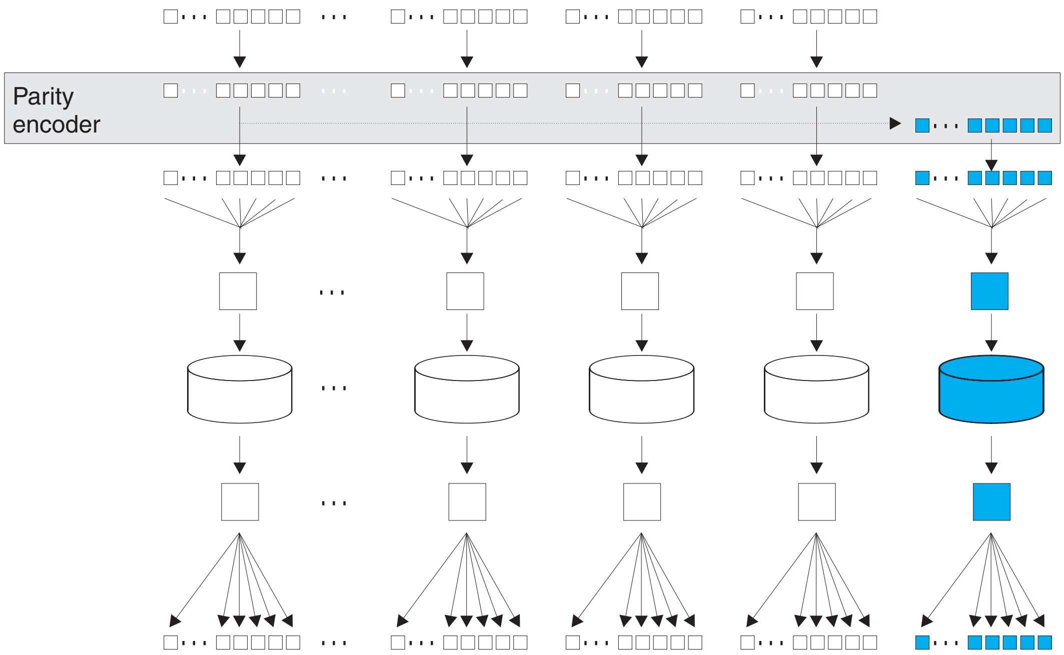


Figure 4.5: Integrated error management prototype design.

reading both information and parity data, and each striping unit consists of multiple packets (symbols) each transmitted separately as shown in Figure 4.5. Thus, the storage system parity data is reused for FEC recovery data removing the server side FEC encoding operation.

4.1.6 Scheme Shortcomings

As mentioned before, we cannot recover from all losses in the network, i.e, we can adapt to less amount of loss, but the scheme can not adapt to higher loss without reconfiguring the scheme. Furthermore, our assumption regarding losses occurring in small bursts only is often not valid. Thus, despite our recovery scheme, some receivers may experience losses. For example, if more than 32 packets per transmitted codeword are damaged or corrupted in our scheme, the decoding function will not be able to recover the missing data. Nevertheless, we still want the best possible quality of the data playout. One possible solution is to additionally use some kind of loss concealment at the client side as for example described in [127], i.e, this approach is data-type specific and is not appropriate at the system level. If a packet is lost and cannot be recovered, the loss can be hidden from the receiver by some sort of reproduction of the lost data. This can for example be done by simply copying the previous data element or by interpolation using the surrounding data to produce a replacement for the lost packet(s). Another example is to use a hybrid recovery scheme. We can for example combine FEC and ARQ like in type I or type II hybrid ARQ [32, 67]. However, in this thesis we assume that our error model is valid and regard greater loss scenarios as future work.

4.2 Zero-Copy-One-Copy Memory Architecture

Memory management has been a critical issue in high performance systems for a long time. Several zero-copy architectures removing physical data copying, i.e., reducing resource consumption of individual clients, have been designed to optimize resource usage and performance using shared memory, page remapping, or a combination of both [129]. However, concurrent clients requesting the same data require their own set of resources. Traditional broadcast or multicast is an easy way of dealing with per client resource allocation. To minimize a possible start up delay, a couple of broadcast partitioning schemes are proposed [129]. 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, in a *zero-copy-one-copy memory architecture*[7], to eliminate all physical in-memory copy operations, to share one single copy of a data element in memory, and to have a short start-up delay. This subsection describes this memory architecture.

4.2.1 In-Kernel Disk-to-Network Data Path

In this subsection, we look at the zero-copy data path reducing resource requirements on a per-data-element basis. We present our basic idea, use one appropriate existing mechanism as a basis for further design, and describe the new kernel interface through the new stream system calls.

4.2.1.1 Basic Idea

The basic idea of our zero-copy data path is shown in Figure 4.6. The application process is removed from the data path, and in-kernel copy operations between different subsystems are eliminated by sharing a memory region for data. The file system sets the `b_data` pointer in the `buf` structure to the shared memory region. This structure is sent to the disk driver, and the data is written into the `buf.b_data` memory area. Furthermore, the communication system also sets the `m_data` pointer in the `mbuf` structure to the shared memory region, and when the `mbuf` is transferred to the network driver, the driver copies the data from the `mbuf.m_data` address to the transmit ring of the network card.

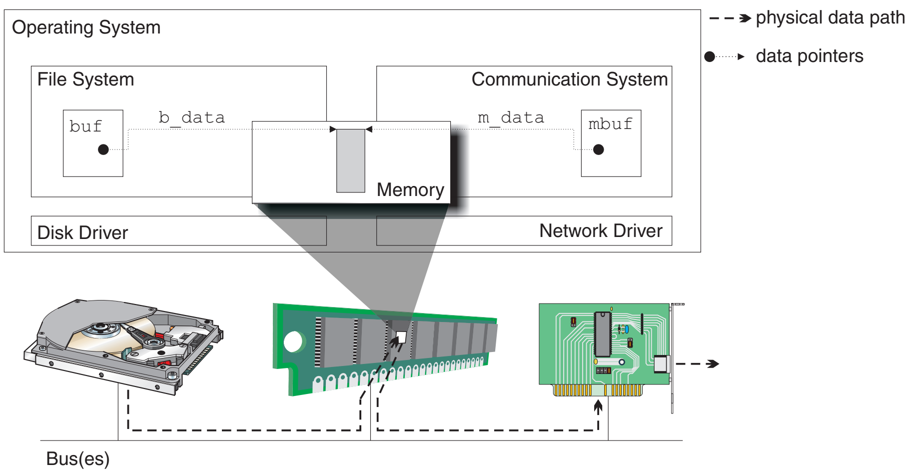


Figure 4.6: Basic idea.

4.2.1.2 Choosing Mechanism

To implement this design, we have analyzed several proposed zero-copy data path designs (see Section 3.2.3). In INSTANCE, we have a specialized system and do not need to support all general operating system operations. Therefore, we have used a mechanism that removes the application process from the data path and transfers data between kernel subsystems without copying data (see for example [28, 58, 186, 101]). As a starting point, we have chosen to use the *Multimedia M-buf* (MMBUF) mechanism [27,

[7] The name *zero-copy-one-copy* refers to a combination of a zero-copy implementation of an in-kernel data path with a broadcasting scheme that serves multiple clients with one in-memory copy.

28, 36], because of its clean design and reported performance. Below, *MMBUF* denotes this mechanism in general whereas *mmbuf* refers to the data structure containing the MMBUF variables similar to mbuf.

MMBUF is developed to reduce data copying in an MoD server in the MARS project [28] at Washington University, St. Louis, as an enhancement to the 4.4 BSD UNIX. A new kernel buffer management system provides a zero-copy data path for networked multimedia applications by unifying the buffering structure in file I/O and network I/O. This buffer system looks like a collection of clustered mbufs that can be dynamically allocated and chained. Both the mbuf header and the buffer cache header is included in the mmbuf header used for network and file I/O respectively, so by manipulating the header, the mmbuf can be transformed either into a traditional buffer that a file system and a disk driver can handle or an mbuf which the network protocols and network drivers can understand. Furthermore, for multimedia applications a new interface is provided to retrieve and send data which coexist with the old buffer cache based data path for applications that use the `read()` and `write()` interface. A `stream_open()` system call corresponding to the traditional `open()` opens a file and initializes the use of the file descriptors. The system call `stream_read()` bypasses the old buffer cache and reads data from a file into an mmbuf chain. Both synchronous (blocking) and asynchronous (non-blocking) semantics are supported. Moreover, the data is sent to the network using a new `stream_send()` call which converts the mmbuf chain into a cluster mbuf chain without any copies. At setup time, each stream is allocated a ring of buffers, each of which is an mmbuf chain. The size of each buffer element, i.e., the mmbuf chain, depends on the size of the multimedia frame it stores, and each buffer element can be in one of the four states: empty, reading, full, or sending. Furthermore, to coordinate the data read and send activities, two pointers (read and send) to the ring buffer are maintained. Then, for each periodic invocation of the stream process, these pointers are used to handle data transfers. If the read pointer is pointing to a buffer element in the empty state, data is read into this chain of mmbufs and the pointer is advanced to the next succeeding chain on which the next read is performed. If the send pointer is holding a full buffer element, the data stored in this buffer element is transmitted. Finally, both read and send requests for multiple streams can be bunched together in a single call minimizing system call overhead.

4.2.1.3 Changes Made to the Native MMBUF Mechanism

To support our requirements and system components, including a later version of NetBSD, and to further increase the performance of the original MMBUF mechanism, we have made the following modifications:

- The allocation and deallocation of mmbufs have been changed to use the pool mechanism in NetBSD [197], because this reduces the time used to allocate and free a buffer item. On a PentiumIII 933 MHz machine, `malloc()` and `free()` (which is used in the original design) used 5.80 μs and 6.48 μs, respectively. Using the pool mechanism, these times are reduced to 0.15 μs for both allocating and deallocating the mmbuf. To allocate and free mmbuf memory clusters, we have added some functionality from the NetBSD pool mechanism like locks to the original mmbuf cluster pool to assure that one cluster is not handed over to more than one stream.

- We reimplemented the stream application programmer interface (API) system calls. The interface is almost identical, but the underlying functions are somewhat changed. We have for example removed the timing mechanisms used by the real-time upcall mechanism in MARS [28] which guarantee periodic data transfer[8]. We have also implemented some additional system calls to perform more extensive measurements.

- The native MMBUF system transmitted data on an ATM network. We had to modify the network send routine to allow UDP/IP processing (because the NLF prototype uses UDP, see Section 4.3).

[8] In this thesis, we have only looked at achieving maximum performance. Guaranteed services and QoS provision are a subject of future work.

In addition to the possibility of sending the whole mmbuf chain as a single packet, we enabled sending each mmbuf as an own network packet. This is because the disk performance is better reading large blocks, and we read more data than a single packet can hold. The integrated error management scheme also uses a smaller packet size compared to the amount of data read each read operation.

- When sending data from a chain in the MMBUF buffer manager, the original design issued a wait until all the data in the chain was sent before a new read on that chain could be executed. This might introduce some delays (the process will sleep) if the queue in the network driver is long. The traditional mechanisms just deliver the data to the queue and return. In our initial performance experiments, we achieved a higher throughput using traditional system calls. However, a new memory block is allocated for the chain regardless of whether the data is transmitted or not (as in traditional mbuf manner), so to avoid this delay, we removed this strict waiting on the alternating buffers and let the call return after handing the data over to the driver, i.e., the possible blocking delay is removed.

The MMBUF mechanism integrates the buffering architecture of the file system and the communication system. As shown in Figure 4.7, the `mmbuf` data structure contains both the file system `buf` structure and the communication system `mbuf` structure. The data pointers in these structures then point to a shared data area (an *mmbuf external memory cluster*), i.e., all the `buf.b_data` pointers point to the start of the data area, and each `mbuf.m_data` pointer is advanced by a configurable size, i.e., the used packet size which is 1 KB, 2 KB, 4 KB, and 8 KB (see Section 4.1.5.2).

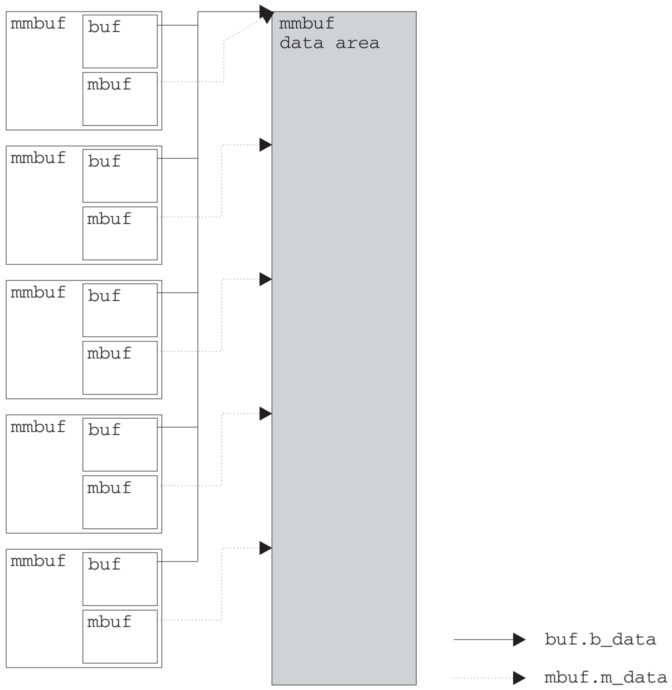


Figure 4.7: Using the `mmbuf` as both `buf` and `mbuf`.

Flags are used to manage synchronization between the subsystems preventing both subsystems from accessing the data area at the same time. This means that the MMBUF mechanism transfers data between different protection domains using a transfer semantic or transfer model somewhere between the *move* and the *share* model described in Section 3.2.2.

4.2.1.4 Stream System Calls

To make use of the zero-copy data path between disk and network, we have implemented the adapted MMBUF system calls and some additional system calls to perform more extensive measurements. To replace the traditional system calls used in a server-like application shown in Figure 4.8A, we have created `stream_open()`, `stream_read()`, `stream_send()`, and `stream_close()` presented in Figure 4.8B:

`stream_open(int num, void *sarray):` To initialize a server retrieving data from disk and transmitting it to a remote client over a network connection, we use the `stream_open()` system call. With a stream we implicitly mean a data transfer from the storage system to the network, and this call merges these operations into one operation, i.e., this system call opens the requested file and creates a connected network socket to the client's IP address and port number. Thus, it is a replacement for `open()`, `socket()`, and `connect()`.

`stream_read(int num, void *sarray):` This system call avoids physical in-memory copy operations from the file system in kernel space to the application in user space and is a substitute for the traditional `read()` system call. `stream_read()` issues a read operation from disk where data is put in the mmbuf data area instead of using the traditional file system buffers. This system call will not copy data up to the application in user space, but only return when the disk request is sent to the disk driver (asynchronous mode). Strict waiting for the disk read to complete is supported (synchronous mode), but it requires a kernel recompilation.

`stream_send(int num, void *sarray):` This system call replaces the traditional `send()` on a connected socket to avoid physical in-memory copy operations from the application in user space to the communication system in kernel space. It will check if the read operation issued by `stream_read()` is finished. If the data is retrieved from disk, `stream_send()` will access the same data area and transmit it through the communication system. Otherwise, this system call goes to sleep and will be woken by the file system when the disk operation finishes. This system call will return after putting the packets into the driver queue (asynchronous mode). Strict waiting for the packets to be sent is removed (see above), but is supported (synchronous mode). However, it requires a kernel recompilation.

`stream_close(int num, void *sarray):` This system call tears down the stream by closing the file descriptor of the transmitted file and the network socket, i.e., it is a replacement for `close()` which traditionally must be called twice.

The input parameters for all these system calls are `num` describing the number of streams and `*sarray` pointing to a `streamState` array. The first parameter is used to make operations on several streams

```
filefd = open(...);
socketfd = socket(...);
connect(socketfd, ...);

while (more to send) {
   read(filefd, buf, length);
   send(socketfd, buf, length);
}

close(filefd);
close(socketfd);
```

(A) Pseudo code of a server using the traditional system calls.

```
stream_open(num, sarray);


while (more to send) {
   stream_read(num, sarray);
   stream_send(num, sarray);
}

stream_close(num, sarray);
```

(B) Pseudo code of a server using the stream system call.

Figure 4.8: Replacement of traditional system calls with our new stream API.

```
struct streamState {
    int fd;                  /* Data file descriptor */
    int sockfd;              /* Socket descriptor */
    int nochains;            /* Number of alternating chains */
    int read_len;            /* Amount of data to read/send into/from in one operation*/
    char *file_path;         /* Filename of file to open */
    char *nlffile_path;      /* Filename of NLF meta-data file to open (if NLF is used) */
    int flags;               /* Set required operations (NLF/prefetching) and return errors */
    struct sockaddr_in addr; /* Used to open and connect the socket during stream_open */
    char *IPaddress;         /* Destination IP address */
    u_int16_t portno;        /* Destination port number */
};
```

Figure 4.9: The `struct streamState` used in the stream system calls.

within one system call (reducing system call overhead), and the value indicates how many streams to serve for this system call. The `streamState` structure, shown in Figure 4.9, describes the input parameters and the state of each stream, e.g., file name, file descriptor, socket descriptor, how much data to read and send, some flags, destination IP address and port number, etc. The `*sarray` pointer points to the start of an array holding one `streamState` structure for each stream.

This new stream API reduces the overhead of the traditional data and control path shown in Figure 4.10. As depicted in Figure 4.11A, the data remains in the kernel whereas the control is handed over to the application process. Furthermore, the time to make a system call is substantial compared to just making a call to a function in the same memory space (Section B.3 presents the system call overhead). Furthermore, an unfortunate property of traditional disk operations is that they are indeterministic. This means that the application is not guaranteed a delivery deadline. Thus, data which should be sent periodically might be delayed due to high disk workload. To deal with this problem, we use *prefetching* of data, i.e., try to retrieve the data in advance before it is requested. In addition to the straight forward replacements of the traditional system calls above, we have implemented two other system calls (with identical input parameters) to further reduce the overhead in our MoD server as presented in Figure 4.11B:

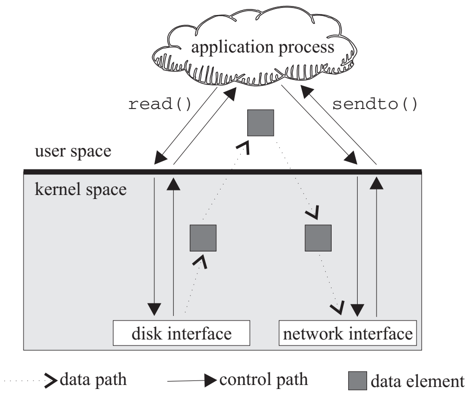
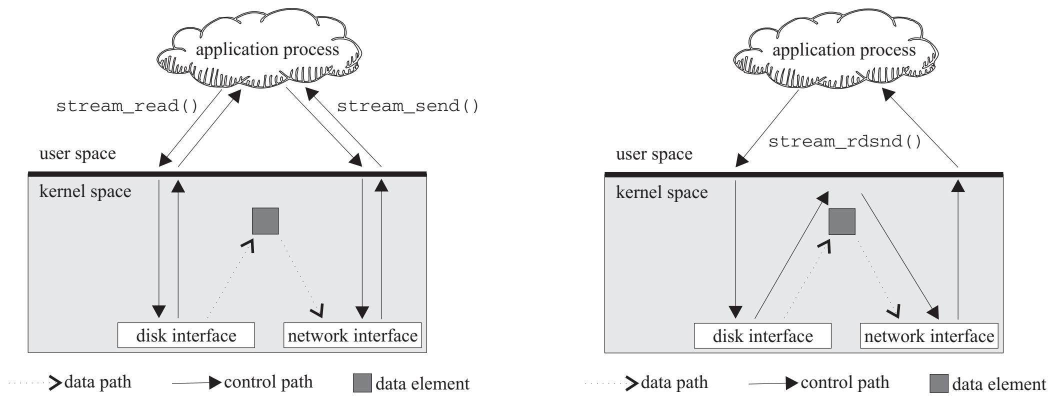


Figure 4.10: Traditional `read()`/`send()` data and control path.

`stream_rdsnd(int num, void *sarray):` This system call merges `stream_read()` and `stream_send()`, i.e., instead of returning the control back to user space, this system call immediately processes the data arriving from disk through the communication system to the network. Thus, the kernel is accessed only one time.

`stream_sndrd(int num, void *sarray):` This system call extends `stream_rdsnd()` with prefetching using two (or more) alternating buffers for read and send operations respectively. One



(A) The `stream_read()`/`stream_send()` system calls. (B) The `stream_rdsnd()` system call.

Figure 4.11: The data and control path using our new system calls.

buffer is used for disk operations while the other is used for sending data through the communication system. The read operation is then trying to retrieve the data to be transmitted to the client one period in advance.

Finally, we have implemented a system call for repositioning the file offset (moving the read/write file pointer), i.e., for use in jumps, restarting, etc.:

`stream_seek(void *state, off_t offset, int op):` This is a stream version of the traditional `lseek()` system call. You may set the file offset according to where you are now and according to the start of the file. Additionally, `stream_seek()` may restart the file, i.e., setting offset to 0 and clearing the end-of-file flag used in the stream. The seek operation will immediately affect streams without prefetching. In the case of prefetching, the current implementation, will reposition the read offset, but the send operation will first send the prefetched data remaining in memory. Thus, the send operation does not see the seek operation.

The input parameters for this system call are different compared to the other stream system calls, because the seek operation is not equal for each stream. `*state` is a `streamState`-pointer to the selected stream. `offset` describes the offset, and `op` describes how to seek: (1) `OFF_SEEK_SET` sets offset to `offset`, (2) `OFF_SEEK_CUR` sets offset to `offset` plus current location, and (3) `OFF_SEEK_REW` rewinds the file to the start.

4.2.2 Broadcasting Scheme

The zero-copy data path eliminates the in-memory copy operations in our storage node, but each client that is served by the storage node still requires its own set of resources in the storage node. For each client request, the data for the respective client has to be stored in the virtual address space of the serving process. This means that physical memory is allocated and mapped into the virtual address space of the process according to available resources and a global or local allocation scheme. This approach is also called user-centered allocation, because each client has its own share of the resources. However, traditional memory allocation on a per-client basis suffers from a linear increase of required memory with the number of clients.

In order to better utilize the available memory, several systems use so-called data-centered allocation where memory is allocated to data objects rather than to a single process. We design our storage node to use a periodic service like pyramid broadcasting [164] or its derivatives (see Section 3.2.4). In this approach, data is split in partitions of growing size, because the consumption rate of one partition is assumed to be lower than the downloading rate of the subsequent partition. Each partition is broadcasted in short intervals on separate channels. A client does not send a request to the storage node, but instead it tunes into the channel transmitting the required data. The data is cached on the receiver side, and during the playout of a partition, the next partition is downloaded. Performance evaluations show that data-centered allocation schemes scale much better with the numbers of users compared to user-centered allocation [164]. The total buffer space required is reduced, and the average response time is minimized by using a small partition size at the beginning of a movie.

4.2.2.1 Choosing Protocol

In Section 3.2.4, we found periodic service protocols like pyramid-, skyscraper-, or harmonic broadcasting suitable for our purpose, and which protocol to use is dependent on several factors. Pyramid broadcasting requires that each channel transmits at least at full consumption rate, and the buffering requirement at the client is substantial, i.e., 50 % of the video. Skyscraper broadcasting requires less client buffering at the cost of more channels broadcasting at the playout rate making the server workload higher. Thus, the harmonic broadcasting schemes seem to be suitable for our purposes, because the transmission rate of channel S_i decreases as i increases. However, pure harmonic broadcasting may fail to deliver data in time for playout, but this problem is solved by the successor schemes [116, 117]. In [118], a harmonic broadcasting scheme with no startup delay is proposed, but this scheme requires that a client buffers the start of all videos in advance, at a high memory cost, before selecting which video to download and view. This video-prefix buffering requires a lot of buffer space for videos which will never be watched, and we find this scheme unrealistic. We therefore assume a *cautious harmonic broadcasting* protocol [116], because of its simpleness. Quasi- and polyharmonic broadcasting further reduce the required server bandwidth slightly, but they also increase complexity. In the next section, the cautious harmonic broadcasting protocol is presented.

4.2.2.2 Cautious Harmonic Broadcasting

Like harmonic broadcasting, cautious harmonic broadcasting [116] breaks a video into n equal-length segments of duration $d = D/n$ where D is the total duration of the video. The size S of the video is given by $S = D \times b$ where b is the playout rate of the video, and each segment S_i has size S/n. The original harmonic scheme repeatedly broadcasts segment S_i with bandwidth b/i in separate streams, i.e., one stream per segment. This gives a total bandwidth requirement for this video of

$$B_{hb} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} = \sum_{i=1}^{n} \frac{b}{i} = b \sum_{i=1}^{n} \frac{1}{i} = b \times H(n)$$

where $H(n)$ is the harmonic number of n. The harmonic series diverges [56], but from Table 4.1 we see that the total server bandwidth requirement per video increases slowly with the number of segments, and with a reasonable number of segments, the total server bandwidth requirement will be below six times the consumption rate. However, harmonic broadcasting may fail to deliver data in time, so the cautious harmonic protocol changes this transmission scheme slightly within one video to guarantee in-time data delivery (see proof in [116]). The first stream transmits segment S_1 at bandwidth b, but the second stream is different, i.e., the second stream transmits segments S_2 and S_3 alternately at full bandwidth. Then stream i, $i = [3, n]$, transmits segments S_{i+1} with bandwidth $b_i = b/i$. Thus, to guarantee in-time data delivery within the same startup latency limits, the total bandwidth requirement is increased to

$$B_{chb} = 1+1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\cdots+\frac{1}{n-1} = 2b+\sum_{i=3}^{n-1} \frac{b}{i} = \frac{b}{2}+(bH(n-1)) = b\left(\frac{1}{2} + H(n-1)\right) \quad (4.1)$$

This means that $b \times (1/2 - 1/n)$, $n > 3$, more units of bandwidth is required compared to the original harmonic broadcasting.

n	$H(n)$	n	$H(n)$	n	$H(n)$
1	1.000	10	2.929	100	5.187
2	1.500	20	3.598	200	5.878
3	1.833	30	3.995	300	6.283
4	2.083	40	4.279	400	6.570
5	2.283	50	4.499	500	6.793

Table 4.1: Harmonic numbers.

The rest of the scheme is identical to the harmonic broadcasting scheme. The client must wait until the first stream transmitting segment S_1 restarts at the beginning before starting to receive (and view). At the same time, the reception of the all other streams dedicated for this video transmitting the remaining segments is also started. Thus, data will arrive out-of-order, and about 40 % client side buffering is needed [82].

4.2.2.3 Client Overhead

The harmonic broadcasting scheme can save a lot of server resources, but implies an increased client overhead. Looking at the example described in Section 2.2 where 1000 concurrent clients retrieve a 3 minute 3.5 Mbps video clip in a true NoD scenario, i.e., the server has one stream for all clients, the performance gain is large applying the cautious harmonic broadcasting scheme. In the stream-per-client approach, the server will have to transmit 1000 concurrent 3.5 Mbps streams. Using cautious harmonic broadcasting and assuming a maximum startup latency of 5 seconds (excluding transmission delay), the video is partitioned into 36 segments. The total server bandwidth requirement for this news clip is then

$$B = b\left(\frac{1}{2} + H(n-1)\right) = 3.5 \text{ Mbps} \times \left(\frac{1}{2} + H(35)\right) = 3.5 \text{ Mbps} \times (0.5 + 4.147) = 16.26 \text{ Mbps}$$

using equation 4.1. However, the client will experience a higher workload and resource requirement, because it has to receive all the streams concurrently (only during the download/playout of the first segment) and buffer data for later use. On average, the client will receive 16.26 Mbps in this example, and this will decrease as the playout of segments finishes (number of streams goes down). Receiving this decreasing data rate from the network should be no problem using existing network cards. The disk I/O (assuming we have to buffer arriving data on disk) can be slightly higher compared to network I/O as we both write the incoming data on disk and read the segment currently presented to the client, i.e, approximately 16.26 Mbps + 3.5 Mbps = 19.76 Mbps in the example. However, the first segment is viewed as it arrives at the client, and there might be no need for buffering the data in this segment. Nevertheless, using a Seagate Cheetah X15 (ST318451LW) [203], which achieves a minimum data rate of 299 Mbps, this data rate is no problem.

4.2.3 Integrating the In-Kernel Data Path and the Broadcasting Scheme

The integration of the zero-copy data path and the delay-minimized broadcasting scheme is depicted in Figure 4.12. The data file is split in partitions, and each of these partitions are transmitted continuously in rounds using the zero-copy data path (see Figure 4.6) where data is sent to the network using UDP and IP multicast. Thus, our memory architecture removes all in-memory copy operations making a zero-copy data path from disk to network, i.e., the resource requirement per-data element is minimized. Additionally, this mechanism reduces the per-client memory usage by applying a delay-minimized broadcasting scheme running on top of the in-kernel data path.

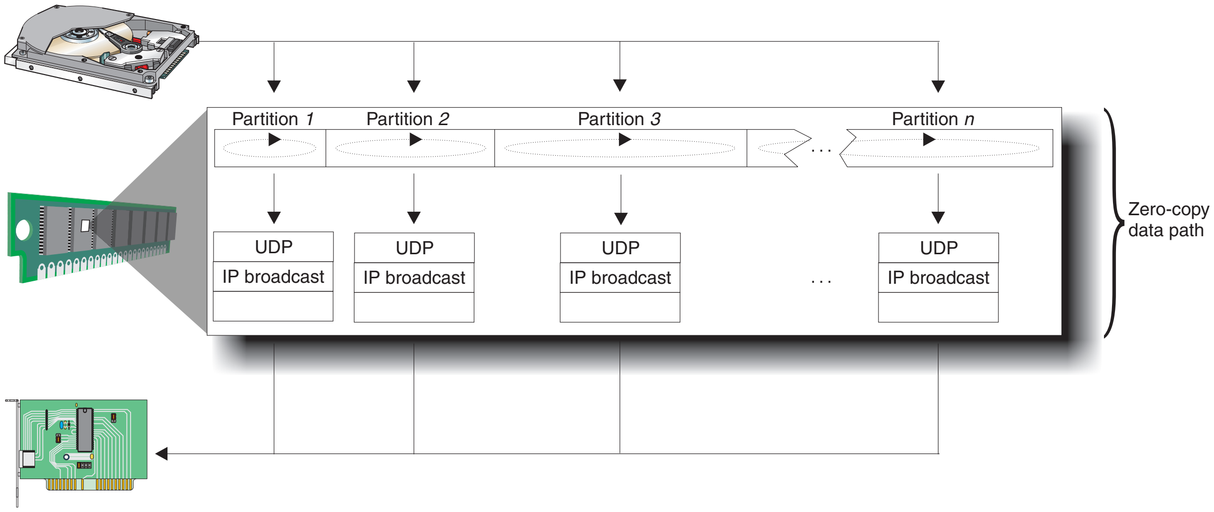


Figure 4.12: The Zero-Copy-One-Copy Memory Architecture.

4.2.4 Zero-Copy-One-Copy Prototype

The zero-copy-one-copy memory architecture design presented in the previous section is only partly implemented in our prototype. The zero-copy data path reducing per-data-element resource requirement is fully implemented and tested. However, the delay-minimized broadcasting scheme reducing the per-client resource demand is at the current stage not implemented. There are several reasons for this. The broadcasting scheme found suitable for our purpose (cautious harmonic broadcasting) is not implemented, but only proved mathematically. Likewise, most of the other broadcast protocols presented in literature are also to the author's knowledge not fully implemented, but they are rather simulated in a simulation environment and proved mathematically[9]. Another reason is that there is no easy way of determining the exact performance gain using a broadcasting scheme. The broadcasting scheme enables several clients to share a single data element in memory, i.e., we are able to support a lot of users, but still only a limited number of concurrent streams. Thus, the server workload will be approximately constant using the broadcasting scheme regardless of the number of clients. An implementation would only be used for proof of concept which already is done in theory and by simulations [62, 78, 164]. Furthermore, implementation of such a scheme requires a testbed able to handle broadcast transmissions. At present, we have no such infrastructure meaning that broadcast experiments could not be performed.

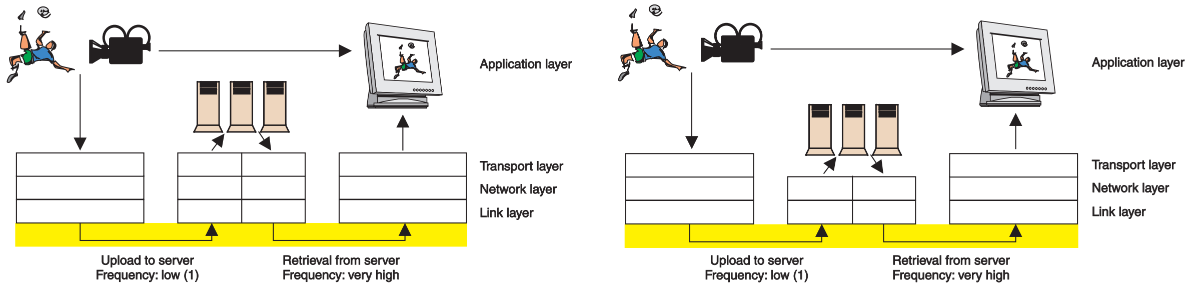
4.3 Network Level Framing

Each time a client retrieves data from a server, the data 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 [168] show that the sender latency is dominated by the transport level checksum operation, i.e., most of the time is consumed due to reformatting data into network packets and calculating the checksum. This operation is repeated for each client and seems to be 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.

To reduce this unnecessary workload in the communication system protocols, i.e., performing the

[9]However, at SIGMETRICS'01, lessons learned from an implementation of periodic broadcast will be presented [22].

same operations on the same data for each packet transmitted (Figure 4.13A), we regard the server as an *intermediate node* in the network where only the lower layers of the protocol stack are processed (Figure 4.13B). 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.



(A) Traditional server storage.

(B) NLF.

Figure 4.13: Traditional server storage versus NLF.

This section focuses on the design and implementation of the network level framing (NLF) concept [71], which enables us to reduce the server workload by reducing the number of operations performed by the communication system at transmission time. We use the Internet protocol suite [153], and from Section 4.1.1, we see that ARQ-based schemes, like TCP, are not suitable in a broadcast multimedia scenario with real-time requirements. UDP is appropriate, for example, according to [146][10], and we use this protocol in our MoD server. Below, be look at the basic idea, the design and implementation, and the performance of NLF.

4.3.1 Basic Idea

As shown in Figure 4.13B, 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, i.e., UDP 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 [152], we use UDP as the transport level protocol to transmit data from the server to the clients.

4.3.2 When to Store Packets

If data is arriving from a remote site as depicted in Figure 4.13B 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 are transmitted correctly to

[10]TCP may be the appropriate protocol for control messages between server and client [146].

the server. This is shown as phase 1 in Figure 4.14. Finally, the integrated error management scheme described in Section 4.1 may use a different block size than the arriving packet size (each packet in a TCP stream might even vary as TCP may gather several transmitted blocks into one packet) and also requires that we transmit the redundant parity data calculated by the storage system. Consequently, we cannot store incoming packets directly.

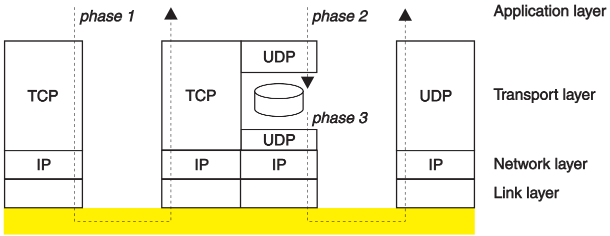


Figure 4.14: NLF in detail.

To be able to prefabricate UDP packets of correct size and with a correct, partly completed header, we store packets processed through the transport level protocol (phase 2 in Figure 4.14), i.e., after the UDP packet has been generated and the checksum has been calculated, but before handing it over to the IP protocol.

4.3.3 Splitting the UDP Protocol

To preprocess data through the UDP layer and store packets on disk, we split the traditional BSD `udp_output()` procedure [169], according to phases 2 and 3 in Figure 4.14. This gives two distinct functions as depicted in Figure 4.15: (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.

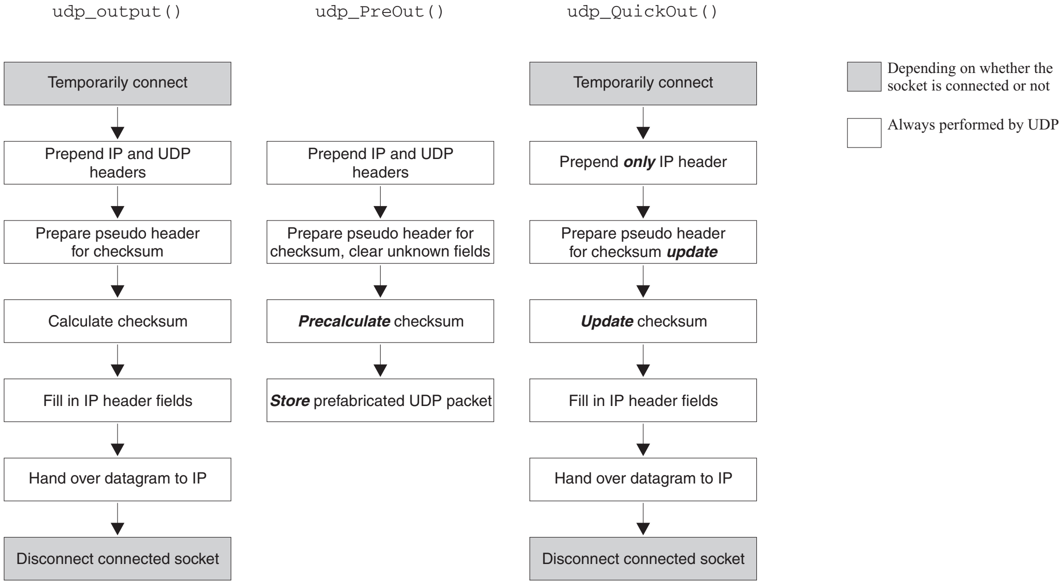
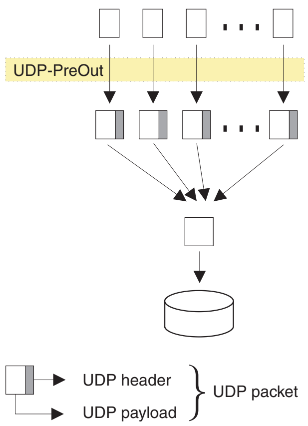


Figure 4.15: `udp_output()`, `udp_PreOut()`, and `udp_QuickOut()`.

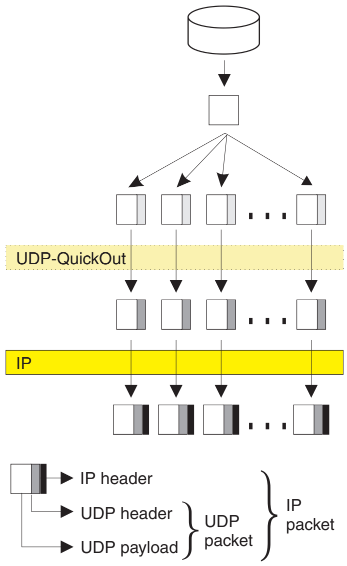
By looking at Figure 4.15, one might wrongly assume that the same (or even more) work is performed in our `udp_PreOut()` and `udp_QuickOut()` functions compared to the traditional `udp_output()`. It is important to note, however, that in our MoD scenario where the multimedia data is retrieved multiple 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, is considerably simplified (see Section 4.3.5.2).

4.3.4 Prefabrication and Data Transmission

The process of prefabricating UDP packets and storing them on disk is depicted in Figure 4.16A. The data 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 packets are concatenated to form a disk block, and the newly formed blocks are written to disk.



(A) Prefabricating transport level packets and storing them on disk.



(B) Retrieving transport level packets from disk and sending them to the client.

Figure 4.16: The prefabrication and data transmission processes.

The data retrieval from disk and the process of sending the packets to the network is sketched in Figure 4.16B. The whole UDP packet is retrieved from disk 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 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. Furthermore, since the server application will only 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 our zero-copy, in-kernel data path. This will additionally reduce system call overhead and data movement operations.

4.3.5 Checksum Operations

Originally, the UDP checksum is calculated over the three areas displayed in Figure 4.17: 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 4.18, 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 [169], 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.

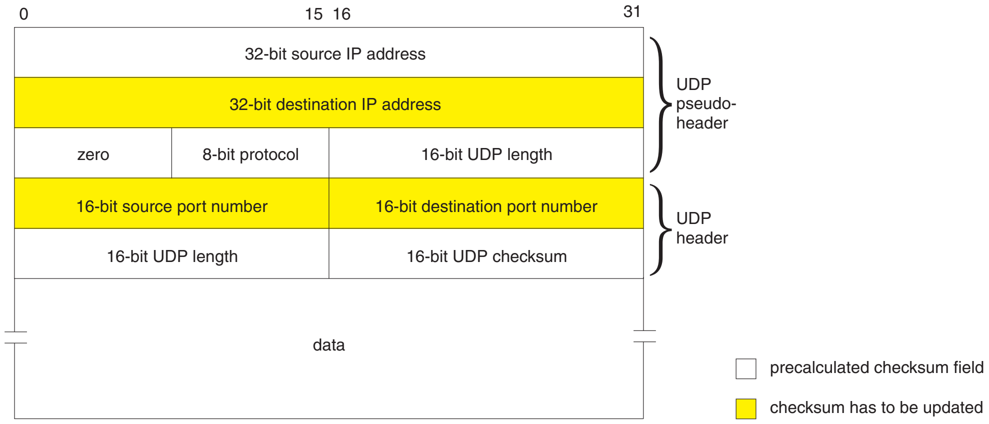


Figure 4.17: Used fields for UDP checksum computation.

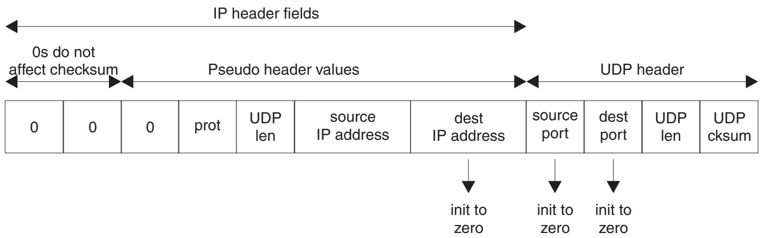


Figure 4.18: `udpiphdr` structure with the fields used for checksum.

The checksum procedure in our NLF mechanism is 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 the checksum of the new fields (shaded in Figure 4.17) in the pseudo and UDP headers. The two checksum operations are described next.

4.3.5.1 Precalculating Checksum

The `udp_PreOut()` function generates UDP packets with (precalculated) checksums (see Figure 4.16A), 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 4.18) before the checksum is calculated using the traditional `in_cksum()` function, in order to assure that these fields will have no affect on the checksum.

4.3.5.2 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 4.19. 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()` continues the protocol processing in the same way 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 4.19: Checksum calculation procedure used in `udp_QuickOut()`.

4.3.6 Network Level Framing Prototype

To be able to test and evaluate the NLF mechanism, we have designed and implemented a prototype of the NLF mechanism in NetBSD. This design differs slightly from the basic idea. We store the communication system meta-data (packet headers) in an own meta-data file. 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 right 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. We split the headers and the media data into two files ("split-file" storage) because our multimedia server also uses integrated storage and communication system error management (see Section 4.1). The recovery scheme uses fixed size blocks (either one packet contains data from several disk blocks or one disk block contains a power-of-two number of packets), each to be transmitted as a packet. Storing packets, including headers, contiguously on disk will greatly increase the complexity of the integrated error management scheme storing both packet headers and media data (codeword symbols) within one disk block. Storing the meta-data in a separate file also 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.

Additionally, we do not store incoming packets in the current prototype. We prefabricate the transport level packets using 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 their own file. Furthermore, the data touching checksum operation is the main CPU cycle consumer, so our prototype precalcuates the checksum over the packet payload only (application level data) and stores this on disk, but the packet header is generated on-the-fly during transmission time. This simplifies the prefabrication function and reduces the storage requirement of the meta-data file by 80 %. Nevertheless, the performed prefabrication should still be sufficient to prove the advantages of the NLF concept.

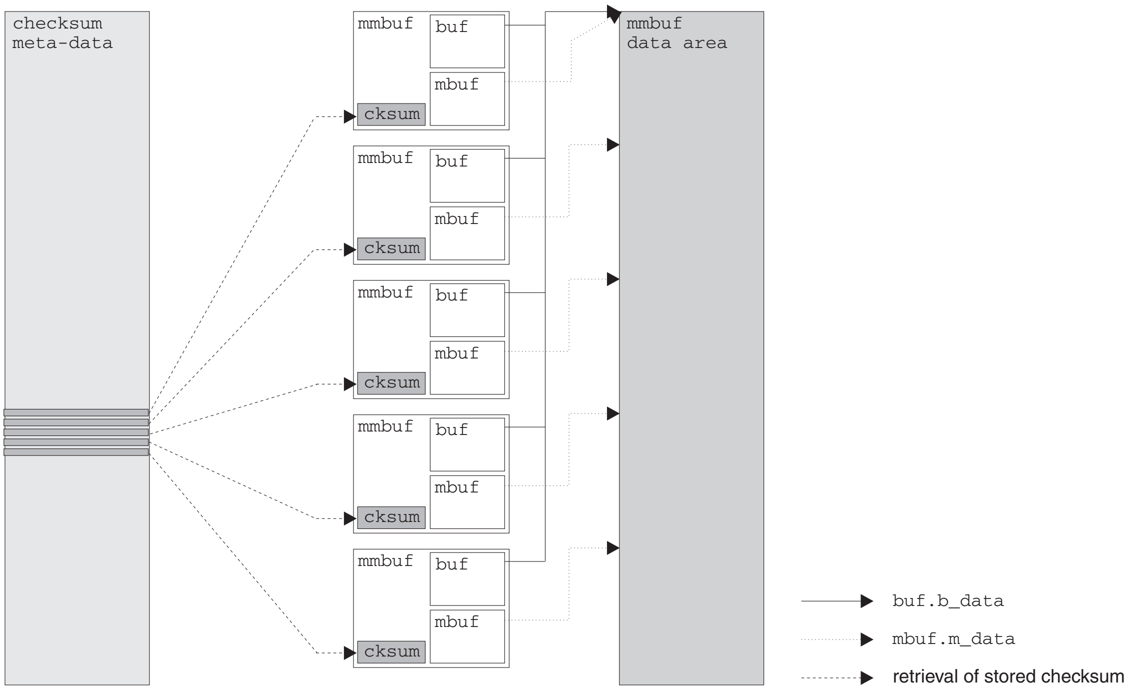


Figure 4.20: The NLF prototype implementation using the zero-copy data path.

The idea of the prototype implementation is shown in Figure 4.20. The checksum meta-data is retrieved from disk and stored in memory during stream setup. As the size of an mbuf has increased to 256 B in the current version of NetBSD, and the data area per mmbuf is 512 B, we have some room for additional variables. We have therefore added a checksum variable (integer) which holds the stored checksum value. During stream read and each buffer initialization the checksum value is retrieved for each packet according to the data read from disk. To find the correct offset in the meta-data file, we use the formula

$$checksum\_offset = \frac{data\_read\_offset}{\texttt{NBPMMBUF}} \times \texttt{sizeof(int)}$$

where `NBPMMBUF` is the size of each packet (number of bytes in each mmbuf), and data_read_offset is the current file offset in the streamed file. For each MMBUF in the current chain, the checksum_offset is advanced by the size of the checksum. When the mmbuf is sent to the transport protocol the checksum value in the mmbuf is used as an argument for the NLF checksum procedure, i.e., the checksum routine performed at transmission time only calculates the checksum of the packet header and adds it to the stored checksum to obtain the final checksum.

4.4 Putting It All Together

Above, we have described our three mechanisms which individually will improve performance in an MoD server. They are orthogonal, meaning that they can be used alone or combined in a system. In this section, we look at how to integrate them in one MoD storage server as shown in Figure 4.21.

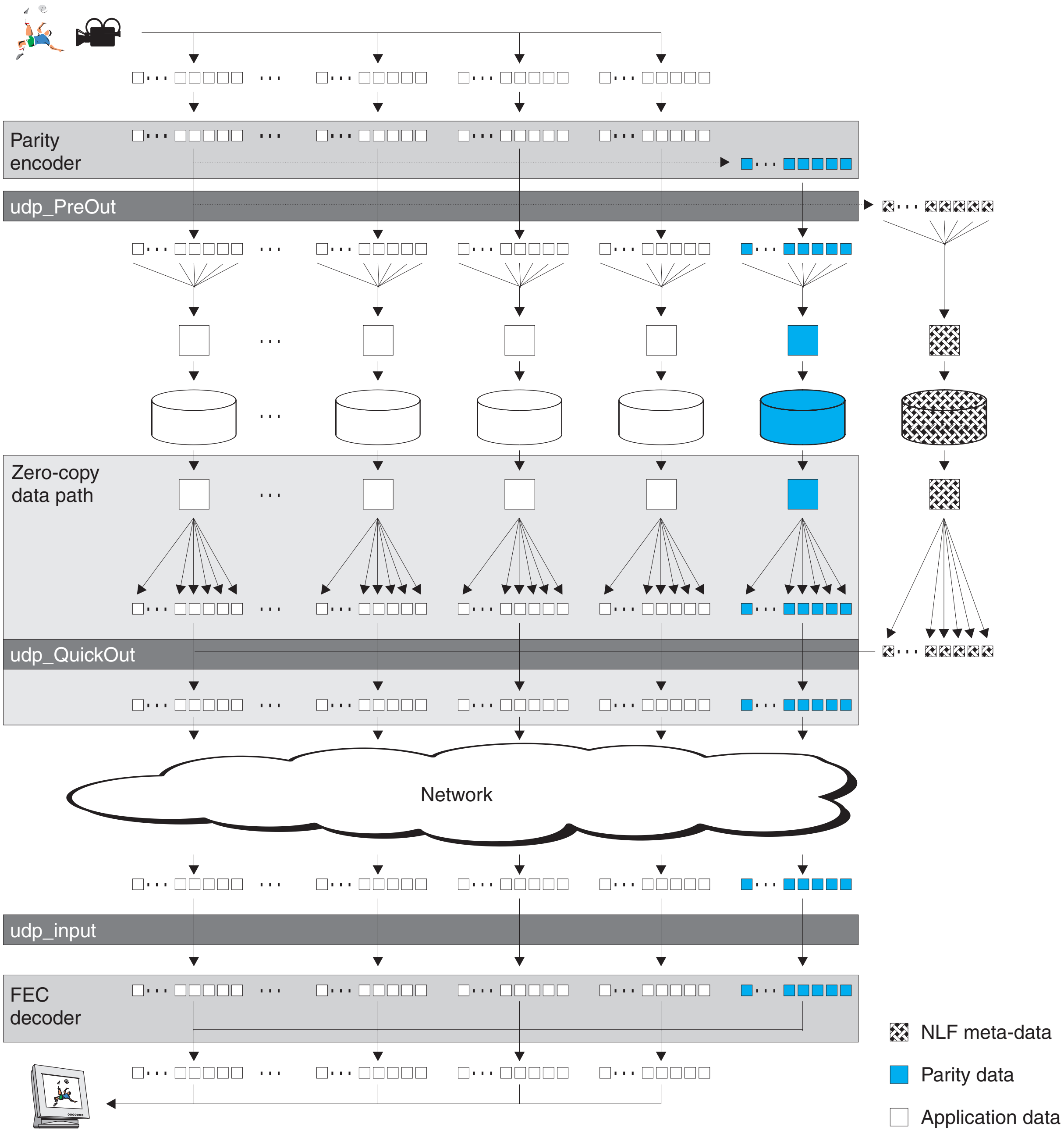


Figure 4.21: Integrating all the mechanisms in the INSTANCE MoD server.

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 code blocks, each to be transmitted as a UDP packet. 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. Additionally, we also transmit the calculated parity information to the remote clients which also must be stored in a network level format. The performance analysis in Section 4.1.5.2 shows that the correcting scheme performs best using symbol (packet) sizes 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 [135], disk requests of 64 KB or larger are suggested, and these numbers are backed up by our own disk efficiency analysis (see Section B.1 for details). Therefore, we perform (multiple) disk requests using the maximum disk block size in NetBSD of 64 KB and afterwards split the read data block into several packets depending on the selected packet size.

The integrated error recovery and NLF write operation is shown in Figure 4.21. The data is handed over to the storage system where write operations are done à la RAID level 4 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 (only the checksum) as described in Section 4.3.4. The NLF meta-data is then stored in a separate meta-data file. To reuse the stored parity data for FEC in the communication system, read operations are performed in a RAID level 0 fashion. Additionally, NLF meta-data is retrieved from disk, containing the precalculated checksum for either application or parity data, and is processed through the `udp_QuickOut()` where only a quick assembly of the packet is performed.

The memory architecture has no direct implications on the other mechanisms and can be used in all cases. However, as the integrated error management scheme uses fixed size packets, we have to set this size in the `NBPMMBUF` variable describing the amount of data in the mmbuf. The NLF mechanism is implemented using the zero-copy data path as described in Section 4.3.6, and can therefore not be used without the MMBUF mechanism in the current prototype.

4.5 Discussion and Conclusions

The delay introduced by the decoding function in the integrated error management scheme may be increased depending on the codec used. For example, in an MPEG stream, frames are reordered and the frame size may vary. The decoder has always one future I-frame and one P-frame in the buffer, because the B-frames refer to future frames. Thus, before decoding the MPEG stream, the entire group-of-pictures must be available. The delay introduced by the error management scheme, will therefore increase if the amount of data required buffered by the MPEG decoder exceed the amount of data within one codeword, i.e., one might have to receive (and possibly reconstruct) several codewords before the MPEG decoder can start.

In [90, 91], a similar approach to NLF is presented where video is preformatted and stored in the form of network packets. 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. Furthermore, checksum caching in memory in a multimedia storage server, as described in [112, 134], 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 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.

The choice of a periodic broadcast protocol, i.e., cautious harmonic broadcasting, is based on minimizing the server workload as much as possible at a fair complexity of the partitioning and transmission scheme. We have assumed that network and client bandwidth can retrieve all the broadcasted streams belonging to one file concurrently, and that there is enough buffer space (disk or memory) to buffer about 40 % of the data at the client. However, this assumption may not hold in some cases. In the given NoD example playing out a high data rate video stream (3.5 Mbps), the client has to support an incoming data rate during the playout of the first segment of 16.2 Mbps (this bandwidth requirement drops as more and more segments finish). Thus, if the client hardware does not support this bandwidth, it cannot be serviced by our NoD server using the cautious harmonic broadcasting protocol scheme. However, the problem of insufficient client bandwidth can be solved in several ways:

- The bandwidth requirement is dependent on the number of segments in the harmonic broadcasting protocol where each augment is transmitted in a separate stream. Thus, by reducing the number of segments (and thereby the number of streams), the client bandwidth requirement is relaxed at the cost of increased startup latency due to longer segments.

- The latency issue from the solution above can be solved at the cost of increased server bandwidth requirement. The same number of segments are used reducing client bandwidth requirement, but each segment is transmitted in several streams (each transmitting the whole segment) with different starting times. For example, the latency may be halved transmitting each segment in two streams separated in time by the half of the length of the segment, but this doubles the server bandwidth requirement.

- Different broadcasting protocols can be used, for example where only one segment is retrieved at a time lowering the client bandwidth requirement, i.e., using pyramid broadcasting with an $\alpha = 1.5$ the client bandwidth requirement will be 5.25 Mbps. However, this client bandwidth requirement is decreased at the cost of a higher server bandwidth requirement, because all streams must be transmitted at the data rate of the consumption rate or higher.

- Another solution would be to support quality adaption where the stream is divided into a base quality layer and several quality enhancement layers as proposed in [88]. In case of insufficient client bandwidth, the quality enhancement layers of the stream are discarded reducing the playout rate and thereby the bandwidth requirement. Reducing the playout rate to 1 Mbps by dropping quality enhancement layers reduces the bandwidth requirement to 4.64 Mbps using the described cautious harmonic broadcasting protocol scheme which is a data rate that can be supported to Norwegian homes today by broadband companies [179].

The various solutions make a trade-off between server and client requirements, and different scenarios (applications, environments, users) may have different preferences, i.e., an analysis should be made in each case. Nevertheless, users having a high data rate connection and sufficient client-side resources should be able to retrieve the data in full quality. Thus, we believe that quality adaption in combination with a broadcasting scheme, that is configured to "satisfy" both server and client requirements, is a good solution.

Another assumption is that we assume that the different multimedia files will be accessed by a large number of concurrent users. The gain in offering such a periodic data delivery depends on the number of viewers. If the total resource requirement for the periodic task exceeds the total requirement of each client in a unicast scenario, it is more efficient to allocate one stream for each client. Thus, a broadcast scheme is best suited for popular videos, e.g., the top ten. For less popular videos, some kind of contingency channels [146] could be allocated where resources are set aside for single unicast streams, though still using the zero-copy data path.

Chapter 5

Performance Evaluation

In the previous chapter, we described the design and implementation of our operating system enhancements to improve the I/O performance of multimedia storage servers. In this chapter, we describe the performance measurements carried out on our mechanisms using a simple benchmark application server transmitting data from our server machine to another machine over a point-to-point network. Since this thesis focus on I/O performance and nor process management, we implemented a simple process-per-stream server, but a future server will probably benefit form a thread-based architecture. Furthermore, each measurement is carried out several times to get a credible result, and we present statistics like maximum, minimum, average, standard deviation, and confidence intervals. In the tables, the values are labeled max, min, avg, stdev, and ci, respectively. Each measurement is also performed using different packet sizes (see Section 4.1.5.2), and when listing the different values for each size, we have included the packet size in parenthesis. In order to see the relationship between the expected improvements from removing bottlenecks and the actual measured performance gain, we estimate the expected performance increase based on our system analysis (limitations are summarized in Section 3.4 and some measurements are presented in Appendix B) and compare the measured and estimated performance to determine if other factors have influenced the results.

In the first section of this chapter (Section 5.1), we describe the experimental environment. Sections 5.2, 5.3, and 5.4 describe the specific experiments and present the experimental results from our three mechanisms. Section 5.5 presents an evaluation of the total server improvement, and we discuss and analyze the results further in Section 5.6.

5.1 Experimental Environment and Performance Monitoring Tool

There are many different factors that may influence the measured results when performing experimental tests on a system. Different hardware architectures have different instruction sets, and each subsystem often has several parameters that could be varied for each test, e.g.:

- hardware parameters like the size of memory, machines and architectures, hardware components like disks, buses, network cards, etc., and
- software parameters like the size of the kernel address space, size of different kernel sub-maps, file system (disk) block size, mbuf external cluster size, transport level packet size, network MTU size, etc.

Furthermore, the application could make read and send requests of different sizes, and each test should be run several times to get a credible result. However, if we were to consider all these parameters and use different configurations for each one of them, the number of tests would be very large. To limit the scope of this thesis, we have limited the number of variable parameters and their values. We have used only one set of hardware (described below), and we have kept the size of the kernel memory maps, the file system block size, and the mbuf (and mmbuf) external cluster size constant using default NetBSD values. What we have varied in our tests are the background load, the number of concurrent streams, and the transport level packet size. Furthermore, we tested the different configurations of our system, i.e., three stream versions of the server (using `stream_read()`/`stream_send()`, `stream_rdsnd()`, and `stream_sndrd()`) with and without NLF. The results are then compared with the results of a traditional server. The tests are performed using packet sizes of 1 KB, 2 KB, 4 KB, and 8 KB, because these packet sizes perform best using the integrated error management scheme (see measurements in Section 4.1.5.2). Furthermore, if not described explicitly in the text, each test is performed 100 times to get a trustworthy evaluation of the system, i.e., we try to eliminate infrequent events like garbage collection, log files being reset, etc. during our experiments. Some of the tests could have been performed on other packet sizes, i.e., the tests on the data-path and NLF. However, the tests carried out achieve the performance improvement we required and projected, so further tests using other packet sizes or other system configurations are regarded as future work.

We have performed the tests using a *Dell Precision WorkStation 620* with a PentiumIII 933 MHz processor and 256 MB Rambus direct random access memory (RDRAM) running NetBSD 1.5ALPHA2. This machine has only one 9.1 GB, 10,000 rounds-per-minute SCSI hard disk. Therefore, we simulated a RAID system by storing both application level data and parity data on the same disk. We connected this machine to another PC using an isolated, point-to-point gigabit ethernet network (see Section A.1 for more details).

To measure time in kernel and user space, we have implemented a software probe using the Intel `RDTSC` instruction. This instruction reads the processor cycle count giving a nano-second granularity and is used in a similar way as described in [187]. To avoid issues affecting the measured cycle count like out-of-order execution, we used the `CPUID` instruction in the software probe forcing every preceding instruction in the code to complete before allowing the program to continue. The overhead of executing this software probe comprises *206 cycles* running inside the kernel and *273 cycles* in user space. These values equal to both the lower and upper limits of the 99% confidence interval and should therefore be reliable. On our 933 MHz test machine, the overhead corresponds to about 0.22 μs and 0.29 μs, respectively. In the performance results presented in the following subsections, this overhead is already subtracted (for further information about the software probe, see Section A.2).

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()` [198] which returns information about the resources utilized by the current process.

5.2 Integrated Error Management

In Section 4.1.4.2 and 4.1.5.2, we performed some performance tests to find a suitable code configuration including codeword and symbol sizes, because this has an impact on the efficiency of the system. To further analyze the performance of our prototype and to see if our error management scheme is applicable in a real system, we have designed a simple benchmark application in which we transfer a 225 MB file between the server process and the client process. We performed a worst-case decoding performance test introducing a maximum number of errors within the code's correcting limit on several different client machines using symbol (packet) sizes of 1KB, 2 KB, 4 KB, and 8 KB on the (256,224) over $GF(2^8)$ Cauchy-based Reed-Solomon Erasure code.

5.2.1 Client Side

By reconstructing the lost data at the client side instead of waiting for a retransmission, the clients may experience a better data presentation at the cost of increased memory and CPU usage compared to a scenario where we have long retransmission latencies. Our measurements show that our code is able to decode the data in time for a high data rate presentation. The average decoding performance on our different machines is displayed in Table 5.1. Most of our experiments show that a standard MPEG-2 DVD data stream of 3.5 Mbps with maximum data loss can be recovered in time. Furthermore, there are no large performance differences when varying the symbol size between 1 KB and 8 KB, and all these configurations are adequate regarding throughput. However, depending on the requested data rate our tests indicate some minimal hardware requirement. For example, the Tech Pentium (166 MHz) machine is able to support a data rate of only about 3 Mbps.

	Throughput (Mbps)				Delay (s)			
Machine	**1 KB**	**2 KB**	**4 KB**	**8 KB**	**1 KB**	**2 KB**	**4 KB**	**8 KB**
Sun UltraSparc 1, 167 MHz, Solaris 2.6	6.02	6.51	6.36	6.20	0.30	0.56	1.16	2.45
Sun Ultra 4, 300 MHz, Solaris 2.6	9.99	10.71	10.55	10.49	0.18	0.33	0.69	1.47
Dell PowerEdge 6300 500 MHz, Red Hat Linux 6.1	8.16	7.43	6.52	6.47	0.14	0.49	1.11	2.25
Dell Inspiron 7000, 400 MHz, Red Hat Linux 6.1	10.03	9.27	9.48	9.59	0.18	0.40	0.82	1.54
Cinet PPI-600, 350 MHz, Red Hat Linux 6.1	8.96	9.22	9.05	8.89	0.20	0.39	0.85	1.63
Tech Pentium, 166 MHz, Red Hat Linux 6.1	3.18	3.32	3.34	3.25	0.58	1.11	2.16	4.50
Tech AMD K7, 700MHz, Red Hat Linux 6.1	17.10	17.40	16.67	16.53	0.11	0.21	0.44	0.95
Cinet PPI-600, 350 MHz, NetBSD 1.4.1	10.35	10.71	10.37	10.12	0.18	0.34	0.71	1.45
Dell Inspiron 7000, 400 MHz, NetBSD 1.4.2	10.62	10.05	10.52	10.66	0.17	0.36	0.70	1.38
Cinet PPI-600, 350 MHz, OpenBSD 2.6	14.87	16.28	14.85	14.22	0.12	0.22	0.49	1.04
Dell Precision 620, 933 MHz, NetBSD 1.5ALPHA2	23.03	20.67	20.80	20.75	0.08	0.18	0.35	0.71

Table 5.1: Average decoding throughput and start-up delay varying the symbol (packet) size.

Table 5.1 also shows the experienced start-up decoding delays in our experiment. Due to the decoding cost, the client might experience an increased start-up delay ranging from about 0.1 - 4.5 seconds (if a maximum number of errors occurs) depending on the processor speed and the block size used. The delay increases with the size of the symbol, and since there are no large differences in throughput of the evaluated schemes, a symbol size of 1 KB or 2 KB is appropriate.

Finally, as there usually is some variance in the accessibility of the processor (unless some kind of reservation-based scheduling is provided), and thus in the decoding throughput, some client-side buffering should be provided. Nevertheless, despite all the overhead introduced on the client side, the recovery from lost packets can be made in time to support average MPEG-2 DVD video playout, with exception of the Intel Pentium (166 MHz) machine, assuming client hardware similar to the machines we used in our experiments.

5.2.2 Server Side

The server side performance gain, when integrating the error management mechanisms and reading the parity data from disk, is substantial. Storing and retrieving parity data from disk requires no extra storage space compared to traditional RAID systems, because one disk is already allocated for parity data. Furthermore, it requires no extra time for data retrieval, because the recovery disk is read in parallel with the original application data. We have no overhead (neither acknowledgment handling nor buffer space) managing retransmissions, and the usage of the parity data from the RAID system as FEC recovery data offloads the storage node from the encoding operation resulting in significant performance improvements. As shown in Table 5.2, a PC with a PentiumIII 933 MHz CPU is capable of encoding data at a maximum throughput of 24.34 Mbps (1 KB packets), 22.07 Mbps (2 KB packets), 22.67 Mbps (4 KB packets), 22.94 Mbps (8 KB packets) using the Cauchy-based Reed Solomon Erasure code. The table also shows the effective CPU time for the parity encoding of the 225 MB file. Without this encoding operation, the same system can achieve a throughput of nearly 1 Gbps (see Section 5.3.2.2). The only overhead on the storage node side in our approach is 12.5 % of increased buffer space and bandwidth requirement to hold the redundant data in memory and transmit it over the buses and the network. Thus, using our integrated error management mechanism, the storage node workload is greatly reduced compared to traditional FEC schemes, and the clients experience a smoother data playout compared to ARQ based schemes, because no latency is introduced due to retransmissions. Finally, note that the decoding performance presented in Table 5.1 also applies to the storage system recovery rate. If we assume that we use 9 GB hard disks in our disk array, the recovery of a whole disk takes about 420 seconds (assuming a throughput of 22 Mbps on our PentiumIII 933 MHz PC and that the system is dedicated to rebuild the failed disk).

	Size	max	min	avg	stdev	99% ci	95% ci
Measured throughput (Mbps)	1 KB	24.69	22.29	24.34	1.70	[22.34 - 24.67]	[22.41 - 24.61]
	2 KB	22.27	21.30	22.07	0.75	[22.32 - 24.50]	[22.38 - 24.50]
	4 KB	22.84	22.33	22.67	0.35	[22.31 - 24.45]	[22.34 - 24.45]
	8 KB	23.12	22.81	22.94	0.13	[22.31 - 24.38]	[22.32 - 24.38]
Used CPU time (s)	1 KB	76.71	64.94	72.15	2.43	[68.21 - 76.67]	[68.34 - 76.45]
	2 KB	81.42	78.15	80.72	0.75	[78.36 - 81.33]	[78.50 - 81.30]
	4 KB	79.21	78.80	79.02	0.09	[78.80 - 79.18]	[78.87 - 79.18]
	8 KB	78.57	77.98	78.32	0.12	[78.02 - 78.57]	[78.06 - 78.54]

Table 5.2: Cauchy FEC encoding.

5.3 Zero-Copy-One-Copy Memory Architecture

In this section, we describe our performance evaluation of the traditional disk-to-network data and control path using system calls like `read()` and `send()` (illustrated in Figure 4.10) versus our zero-copy data and control path using the system calls described in Section 4.2.1.4. First, we try to analyze the expected gain using our basic tests described in Appendix B measuring copy performance and system call overhead. Then, we analyze the performance gain transmitting only one stream, and afterwards we look at the multi-stream scenario. Finally, we look at the expected performance gain using the broadcasting protocol on top of the zero-copy data path.

5.3.1 Expected Gain

By using our zero-copy data path, the time to perform the copy operations should be saved. For example, reading 64 KB data from the storage system into the specified buffer in user space requires on average 137.1 μs to perform the `copyout()` function (see Table B.2 in Section B.2). If using 4 KB packets, the `copyin()` function uses on average 1.65 μs to copy data into the kernel. Thus, transmitting a 1 GB file in 64 KB blocks at a time, requires 16384 read operations of 64 KB and 262144 send operations of 4 KB. In all, this should be 16384×137.1 μs + 262144×1.65 μs ≈ 2.7 s used for copy operations alone. This time should be saved using our stream API instead of the native NetBSD read and send operations.

If we look at the system call overhead when transmitting 1 GB in 64 KB blocks (the kernel splits the 64 KB block into smaller packets), the `stream_read()` and `stream_send()` system calls require 16384 read and 16384 send operations, respectively. Using the `stream_rdsnd()` system call, which merges the read and send operations into one operation, should reduce this cost by 50 %. Our measurements in Section B.3 (see Table B.3) show that on average 0.40 μs is used each time we perform an access to the kernel. This means that about 16384 $\times$ 0.40 μs = 6.55 ms should be saved in kernel access time. Thus, by reducing the number of system calls, we might expect a small decrease in time to transmit the data, but it will not be very significant.

In a multi-stream scenario, all concurrent streams share the set of resources, and the average performance will decrease per stream. However, the used CPU time should be approximately constant with the exception of adding time to perform context switches.

5.3.2 Single Stream Scenario

We performed three types of experiments on a single stream. First, we used the `stream_read()` and `stream_send()` system call pair (see Figure 4.11A), which eliminate the data copy operation, but still hand over the control to the application before data is read and sent to the network. Second, we used the `stream_rdsnd()` system call (see Figure 4.11B) to eliminate data copy operations and to refrain from handing over the control to the application before data is read and sent to the network. Finally, we also prefetch data from disk using the `stream_sndrd()` system call which has an identical data and control path as `stream_rdsnd()` (see Figure 4.11B).

To measure the performance of the different servers (using the different system calls), we have measured the time it takes to transfer a 1 GB file from the storage system to the network interface. Each test is repeated 100 times to get a reliable result, and the data transfer time is measured using the software probes described in Section 5.1. Furthermore, to see the system behavior under different kinds of machine workloads, we have run the tests with two background workloads: (1) no workload, i.e., no other processes running on the machine except the operating system itself, and (2) high CPU load, i.e., 10 concurrent processes just using the CPU performing time consuming float and memory copy operations. In addition to the CPU bound background workload, I/O bound workloads occupying the storage system and the network could be used. However, in order to test the system with as authentic a workload as possible, we tested the system in a multi-stream scenario, because the load imposed on an MoD server will be the load from several concurrent streams.

5.3.2.1 Transmission Time Reading Data From a Disk Storage System

The performance results, for reading and transmitting a 1 GB file to the remote client, are shown in Table 5.3. As we can see, the disk storage system is a bottleneck, and the performance gain of using a zero-copy data path is absorbed in the large data retrieval times from disk, i.e., the average throughput of all measurements, regardless of using copy operations or not, is about 215 Mbps. To see the real performance benefit of the MMBUF zero-copy mechanism, we performed the same measurements on a 1 GB file in a memory file system (in some systems called a ramdisk). This means that the time to retrieve data is reduced from a disk access to a memory copy operation.

5.3.2.2 Transmission Time Reading Data From a Memory File System

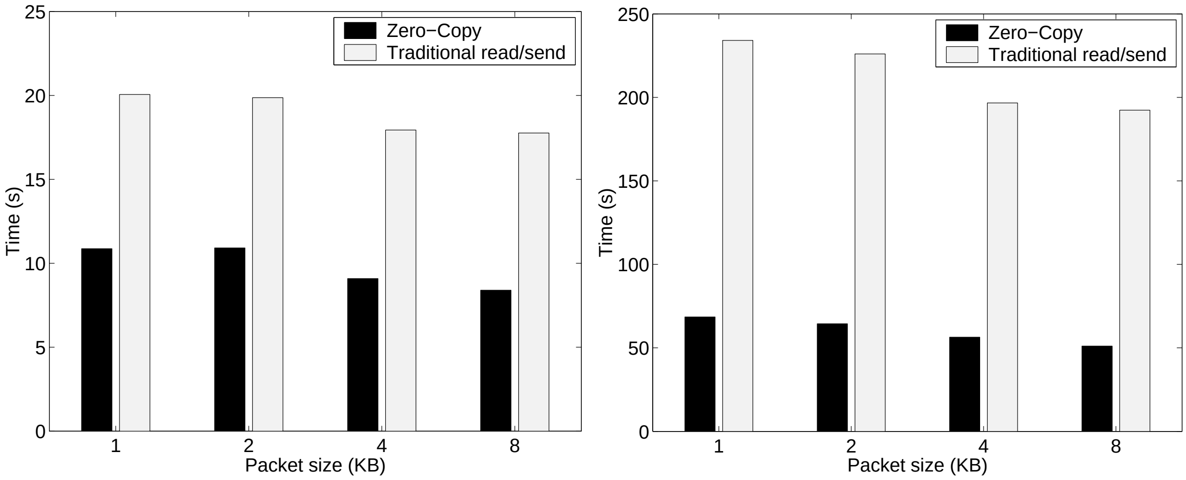
The average results, reading 1 GB data from a memory file system[1] and sending it to the remote client, are plotted in Figure 5.1 where the zero-copy data path is represented by the `stream_read()` and `stream_send()` system calls. Table 5.4 presents several statistics calculated from all the measurements. If there is no load on the system (Figure 5.1A), the time to transmit is reduced to 54.19 % (1 KB packets), 54.96 % (2 KB packets), 50.67 % (4 KB packets), and 47.30 % (8 KB packets) of the traditional `read()/send()` time. Furthermore, if we have a highly loaded machine (Figure 5.1B),

[1]Holding a 1 GB file in a memory file system requires a lot more memory than we have in our machine (256 MB). Since the size of the memory file system is limited, we performed these experiments using a $\sim$30 MB memory file system, and we simulated a large file read by repeatedly (38 times) reading a 28662512 B file using the `stream_seek()` system call to reset the file offset.

Operation	Size	max	Min	Avg	stdev	99% ci	95% ci
`read()/`	1 KB	41.81	40.62	40.70	0.19	[40.62 - 41.81]	[40.62 - 41.22]
`send()`	2 KB	41.47	40.62	41.41	5.40	[40.62 - 41.47]	[40.62 - 41.18]
	4 KB	41.49	40.62	40.69	0.15	[40.62 - 41.49]	[40.62 - 41.20]
	8 KB	41.74	40.62	40.69	0.19	[40.62 - 41.74]	[40.62 - 41.26]
`stream_read()/`	1 KB	40.08	39.27	39.31	0.10	[39.27 - 40.08]	[39.28 - 39.57]
`stream_send()`	2 KB	40.72	40.61	40.63	0.02	[40.61 - 40.72]	[40.61 - 40.70]
	4 KB	40.98	40.60	40.62	0.05	[40.60 - 40.98]	[40.60 - 40.78]
	8 KB	41.35	40.60	40.62	0.08	[40.60 - 41.35]	[40.60 - 40.68]
`stream_rdsnd()`	1 KB	39.58	39.27	39.30	0.06	[39.27 - 39.58]	[39.28 - 39.57]
	2 KB	40.82	40.61	40.63	0.03	[40.61 - 40.82]	[40.61 - 40.70]
	4 KB	40.79	40.60	40.62	0.03	[40.60 - 40.79]	[40.60 - 40.70]
	8 KB	40.81	40.60	40.62	0.03	[40.60 - 40.81]	[40.60 - 40.71]
`stream_sndrd()`	1 KB	39.60	39.28	39.29	0.05	[39.28 - 39.60]	[39.28 - 39.38]
	2 KB	40.85	40.58	40.61	0.05	[40.58 - 40.85]	[40.59 - 40.83]
	4 KB	41.47	40.59	40.61	0.10	[40.59 - 41.47]	[40.59 - 40.83]
	8 KB	41.46	40.58	40.62	0.10	[40.58 - 41.46]	[40.59 - 40.84]

Table 5.3: Time (in seconds) to transfer 1 GB from a disk (no load).

the respective times are reduced to 29.27 % (1 KB packets), 28.51 % (2 KB packets), 28.67 % (4 KB packets), and 26.57 % (8 KB packets) of the native NetBSD read and send operations. This shows that a zero-copy data path also scales much better than the traditional mechanisms, because less resources, like the CPU, are used. The zero-copy mechanism degrades by factors 6.30, 5.90, 6.20, and 6.08, i.e., the time to transmit 1 GB data is increased by these factors, whereas the performance of the traditional server is reduced by factors 11.67, 11.37, 10.96, and 10.83, respectively to increasing packet size.



(A) No background load.

(B) High CPU background load.

Figure 5.1: Average time to transmit 1 GB from a memory file system.

If we compare the different stream versions, we can observe that by using the `stream_rdsnd()` system call instead of the `stream_read()/stream_send()` pair, the performance is increased marginally. This means that compared to other time consuming operations, the overhead of making a system call (see Section B.3) can almost be ignored. Moreover, the results from the prefetching scenario, i.e., us-

Operation	Load	Size	max	Min	Avg	stdev	99% ci	95% ci
`read()/`	No	1 KB	20.16	20.00	20.06	0.04	[20.00 - 20.16]	[20.00 - 20.15]
`send()`		2 KB	19.95	19.81	19.87	0.03	[19.81 - 19.95]	[19.82 - 19.92]
		4 KB	18.02	17.85	17.94	0.03	[17.85 - 18.02]	[17.87 - 18.01]
		8 KB	19.01	17.51	17.76	0.27	[17.51 - 19.01]	[17.52 - 18.51]
	High CPU	1 KB	273.48	208.96	234.10	13.25	[208.96 - 273.48]	[211.24 - 260.64]
		2 KB	236.56	208.73	225.99	5.11	[208.73 - 236.56]	[215.60 - 235.46]
		4 KB	209.35	183.82	196.67	5.53	[183.82 - 209.35]	[186.73 - 208.46]
		8 KB	207.12	169.62	192.33	7.02	[169.62 - 207.12]	[177.54 - 205.53]
`stream_read()/`	No	1 KB	10.89	10.07	10.87	0.08	[10.07 - 10.89]	[10.87 - 10.89]
`stream_send()`		2 KB	10.93	10.91	10.92	0.01	[10.91 - 10.93]	[10.91 - 10.93]
		4 KB	9.22	9.02	9.09	0.03	[9.02 - 9.22]	[9.04 - 9.19]
		8 KB	8.49	8.36	8.40	0.02	[8.36 - 8.49]	[8.37 - 8.44]
	High CPU	1 KB	76.20	58.86	68.51	3.71	[58.86 - 76.20]	[60.85 - 75.96]
		2 KB	75.90	56.02	64.43	4.20	[56.02 - 75.90]	[57.87 - 74.59]
		4 KB	65.98	48.34	56.38	3.97	[48.34 - 65.98]	[49.07 - 64.91]
		8 KB	68.32	43.54	51.11	4.11	[43.54 - 68.32]	[44.46 - 60.66]
`stream_rdsnd()`	No	1 KB	10.89	10.06	10.87	0.08	[10.06 - 10.89]	[10.86 - 10.88]
		2 KB	10.92	10.90	10.92	0.00	[10.90 - 10.92]	[10.91 - 10.92]
		4 KB	9.28	8.85	9.01	0.15	[8.85 - 9.28]	[8.87 - 9.26]
		8 KB	8.42	8.22	8.28	0.06	[8.22 - 8.42]	[8.24 - 8.42]
	High CPU	1 KB	61.01	47.67	54.47	3.38	[47.67 - 61.01]	[48.30 - 60.61]
		2 KB	60.51	45.25	53.93	3.12	[45.25 - 60.51]	[48.03 - 59.27]
		4 KB	53.67	40.66	46.74	2.68	[40.66 - 53.67]	[41.48 - 52.14]
		8 KB	51.97	34.17	42.69	3.04	[34.17 - 51.97]	[36.73 - 48.77]
`stream_sndrd()`	No	1 KB	13.06	12.51	13.02	0.05	[12.51 - 13.06]	[12.98 - 13.05]
		2 KB	12.63	12.48	12.52	0.03	[12.48 - 12.63]	[12.49 - 12.62]
		4 KB	11.14	11.02	11.08	0.05	[11.02 - 11.14]	[11.02 - 11.14]
		8 KB	10.91	10.80	10.85	0.04	[10.80 - 10.91]	[10.80 - 10.90]
	High CPU	1 KB	124.34	98.79	111.01	5.09	[98.79 - 124.34]	[99.83 - 120.64]
		2 KB	134.57	103.99	118.85	5.74	[103.99 - 134.57]	[107.81 - 131.82]
		4 KB	119.82	92.87	104.92	5.56	[92.87 - 119.82]	[94.76 - 116.65]
		8 KB	106.77	78.70	92.91	6.66	[78.70 - 106.77]	[81.71 - 105.79]

Table 5.4: Time (in seconds) to transfer 1 GB from a memory file system.

ing the `stream_sndrd()` system call, show a larger time to process data through the system compared to `stream_rdsnd()`. Prefetching should be performed by using free available time in the storage system to retrieve data to be used next. However, as data is read from a memory file system, the CPU is used to read the data, and the CPU is never idle in our tests. Thus, there is no spare time to retrieve data in the prefetching scenario, which means that there is no time to be saved. We use in the remaining tests only the `stream_read()`/`stream_send()` pair and the the `stream_rdsnd()` system call. Nevertheless, prefetching, as performed in `stream_sndrd()`, will be very useful in a system having a disk storage system with a retrieval rate that is equal to or better than the rest of the storage server components.

Finally, since the disk represents the major bottleneck in the system, and this is solved by using a memory file system (at the cost of memory and CPU resources in the test bed), we will in the remaining tests read data from memory assuming a faster persistent storage system.

Size	max	min	avg	stdev	99% ci	95% ci
1 KB	173.09	0.69	2.28	15.55	[1.71 - 6.96]	[1.77 - 2.69]
2 KB	107.36	1.44	4.63	3.84	[3.65 - 16.66]	[3.77 - 7.34]
4 KB	212.17	3.21	7.73	31.41	[5.81 - 61.20]	[6.03 - 10.86]
8 KB	139.58	2.17	14.72	6.30	[11.74 - 50.16]	[12.38 - 17.28]

Table 5.5: Time to execute `ether_output()` per transport level packet (μs).

	Total			`ip_output()`			`ether_output()`		
Size	max	min	avg	max	min	avg	max	min	avg
4 KB	33535	30891	31886.90	31532	29073	30005.30	2003	1805	1881.60
8 KB	24068	22939	23320.90	22969	21887	22233.40	1133	1052	1087.50

Table 5.6: Amount of loss in number of packets.

5.3.2.3 Server Side Congestion

As some of these values indicate a throughput of approximately 1 Gbps when reading data from the memory file system, we checked to see if the network card could be a bottleneck by counting transmitted packets. Our tests show that all packets are correctly transmitted through the UDP/IP layers, but we experience some loss when enqueuing the packet on the interface output queue and when retrieving the packet from the queue for transmission. The packets are dropped at two places. If the transport level packet is fragmented in the IP protocol, `ip_output()`, the system checks whether there is room in the queue for all the fragments of the current packet. If not, all the fragments are dropped. If there seem to be enough free buffers in the queue, all the fragments are sent to the upper layer of the ethernet layer. A new check on the queue length is performed, in `ether_output()`, where fragments are again dropped if the queue is overloaded. This means that the times presented above to send data have a small source of error. The time to process the packets through the `ether_output()` function is not included for those packets dropped from the IP layer. If the packet size is 1 KB, the `ether_output()` function is always executed, because we do not have any fragmentation. If the packet size is above the interface MTU (2KB, 4 KB, or 8 KB), fragmentation occurs and packets might be dropped in the IP layer. In our measurements, we experienced some loss using 4 KB and 8 KB packets[2]. To see the margin of error in our tests, we measured the time to execute the `ether_output()` function (and the remaining IP instructions) for all fragments depending on the transport level packet size and the amount of loss. Table 5.5 shows execution times, and Table 5.6 shows the number of packets lost during transmission of 1 GB of data[3]. If the packets are dropped in `ip_output()`, the time to execute the `ether_output()` function should be added in the measurements above. For example, using 4 KB packets, we lose on average 30005 packets whose execution time through `ether_output()` corresponds to about 224 ms. For a 8 KB packet size, the corresponding time to execute `ether_output()` for the lost packets is about 322 ms. This means that our measurements are about 2 - 3 % too low using 4 KB or 8 KB packets in the zero-copy data path. Furthermore, when the network card driver dequeues packets and tries to pack the data into the transmit ring on the network card, the packets are occasionally put back on the queue, because the transmit ring is full. This means that the network card is not able to send data fast enough when receiving bursts of packets, and the 1 Gbps network card is therefore a bottleneck. If we have blocking system calls waiting

[2] That we do not experience any loss using 1 KB and 2 KB packets, is probably because there are more instructions to execute with respect to the amount of data compared to packet sizes of 4 KB and 8 KB.

[3] We measured the amount of loss by inserting monitors (counters) into `ip_output()` and `ether_output()`. Thus, all traffic through IP is monitored, i.e., if another process has transmitted a packet, this packet is also counted giving a small margin of error.

for all the packets to be sent, this delay will be added to the total time used to transmit the file. In our tests, we do not wait until all packets are sent before proceeding with the other operations, because then all the read operations would have to wait before all packets are sent in a chain giving even more delays. Since we do not wait until all packets are sent, there might be some packets left in the driver's transmit ring when we execute the last probe (in user space) measuring the time. Thus, a small source of error is introduced into the measurements by this re-enqueuing, but as the total time to transmit the whole file is quite large, this source of error should be negligible. This server side congestion means that we have met the Gbps capacity of the network card and the network card is therefore a bottleneck in our testbed.

5.3.2.4 Estimated Versus Measured Performance Gain

If we now compare the performance gain with the expected gain described in Section 5.3.1, we see that our prediction holds quite well. Compared to the used CPU time presented in Table 5.7 and Figure 5.2, we see that using the stream API (`stream_read()`/`stream_send()`) on average reduces the kernel time by 3.98 seconds using 4 KB packets. The copy operations in this scenario consume about 2.7 seconds. Moreover, time to execute `ether_output()` for dropped packet in the ethernet queue is about 0.2 seconds. The stream API also saves time to process the data through the file system removing most of these instructions, e.g., time to do a buffer cache lookup, buffer allocation, and calling functions like `dofileread()`, `vn_read()`, `ffs_read()`, `bread()`, `bio_doread()`. These functions further call other functions. Using the MMBUF mechanism, this chain of functions is strongly simplified and adapted to our stream. All functions is merged into one function in our stream buffer manager. Likewise, the time to process data through the socket layer is simplified and merged into the send function in the stream buffer manager. Finally, using native NetBSD functionality, block breakdown into smaller packets is done in user space requiring more system calls, i.e., one for each packet to send. Thus, we save more time than just removing the overhead of copy operations, but as we can see the time saved by not performing any memory copy operations still constitutes most of the increase in performance. Furthermore, the expected gain in reducing the number of system calls also seems to be correct. Using 4 KB packets the time is reduced by approximately 8.9 ms (the values in Table 5.7, indicating 10 ms, is rounded off to have two decimals) whereas we expected to reduce the time by about 6.5 ms. This gap

Operation	Size	max	min	avg	stdev	99% ci	95% ci
`read()/`	1 KB	11.29	10.88	11.06	0.08	[10.89 - 11.27]	[10.93 - 11.19]
`send()`	2 KB	11.32	10.87	11.07	0.09	[10.87 - 11.24]	[10.89 - 11.21]
	4 KB	10.19	9.79	9.98	0.08	[9.80 - 10.15]	[9.86 - 10.12]
	8 KB	9.38	9.02	9.17	0.07	[9.02 - 9.33]	[9.05 - 9.29]
`stream_read()/`	1 KB	7.37	7.11	7.27	0.05	[7.18 - 7.37]	[7.19 - 7.37]
`stream_send()`	2 KB	7.10	6.86	6.98	0.05	[6.87 - 7.08]	[6.88 - 7.07]
	4 KB	6.09	5.88	6.00	0.05	[5.88 - 6.08]	[5.90 - 6.07]
	8 KB	5.96	5.75	5.86	0.05	[5.75 - 5.95]	[5.77 - 5.94]
`stream_rdsnd()`	1 KB	7.39	7.16	7.27	0.05	[7.17 - 7.37]	[7.18 - 7.36]
	2 KB	7.08	6.84	6.97	0.05	[6.85 - 7.08]	[6.86 - 7.05]
	4 KB	6.10	5.86	5.99	0.04	[5.88 - 6.06]	[5.91 - 6.05]
	8 KB	5.94	5.72	5.84	0.05	[5.73 - 5.93]	[5.76 - 5.93]
`stream_sndrd()`	1 KB	7.85	7.48	7.62	0.06	[7.49 - 7.79]	[7.51 - 7.71]
	2 KB	7.27	7.05	7.16	0.05	[7.07 - 7.26]	[7.07 - 7.25]
	4 KB	6.27	6.05	6.14	0.05	[6.07 - 6.26]	[6.07 - 6.23]
	8 KB	6.15	5.89	6.00	0.05	[5.91 - 6.10]	[5.91 - 6.08]

Table 5.7: CPU time (in seconds) used by the kernel transferring 1 GB.

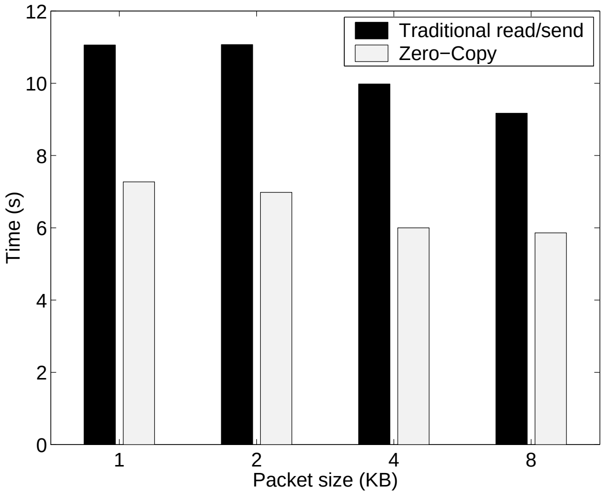


Figure 5.2: Histogram over used CPU time in the kernel.

in time is probably due to a decrease in the number of instructions merging the read and send operation into one system call.

5.3.3 Multi-Stream Scenario

The previous section shows that we can send data in Gbps speed (limited by the Gbps network card) using our zero-copy data path in our testbed. However, how fast a system can shovel a single stream out is not of primary interest in our MoD scenario or most other applications. Most multimedia streaming applications do not need support for one stream transmitting at Gbps speed. For example, the maximum bandwidth of MPEG-2 DVD is about 10.08 Mbps including audio, video, and sub-pictures [184]. Thus, a more appropriate metric of server capacity is how many concurrent clients can the system support at a given bandwidth requirement. As stated in the introduction of this thesis, our goal is to the reduce resource requirements in a traditional server to support more concurrent clients. Therefore, we have looked at performance when transmitting several concurrent streams (each to be broadcast to several concurrent clients) using a packet size of 1 KB and 2 KB[4]. Each stream transmitted 1 GB of data from a memory file system through the communication system to the network. However, as memory is a scarce resource, several streams had to read the same file. This means that streams using traditional system calls could have a caching effect on the buffer cache which will not be present in our storage node broadcasting data. To make equal conditions for all tests, we removed this caching effect by reducing the total buffer cache size from 13168 KB to 200 KB. Finally, during these tests the memory file system consumed approximately 35 % of the CPU resources for all tests.

5.3.3.1 Transmission Times

Our results show that our zero-copy mechanism performs better than the native NetBSD mechanisms. Figure 5.3A/C presents the difference in time to transmit the data per stream depending on packet size, and this time increases faster using traditional system calls compared to our stream mechanism. In the *Zero-Copy 1* measurement, we have used the `stream_read()/stream_send()` system call pair. The test *Zero-Copy 2* represents a scenario where the system call overhead is reduced by merging the read and send operations into one single system call, i.e., `stream_rdsnd()`. Figure 5.3B/D shows the

[4]Since we experienced loss using packet sizes 4 KB and 8 KB, we only performed this experiment using 1 KB and 2 KB packets.

per client throughput, and the *Zero-Copy 2* test indicates a throughput improvement per stream of at least a factor of two, e.g., using 1 KB packets and 150 streams, our stream mechanism achieves a throughput of 5.10 Mbps per stream while in contrast the native NetBSD transmits at 2.03 Mbps per stream.

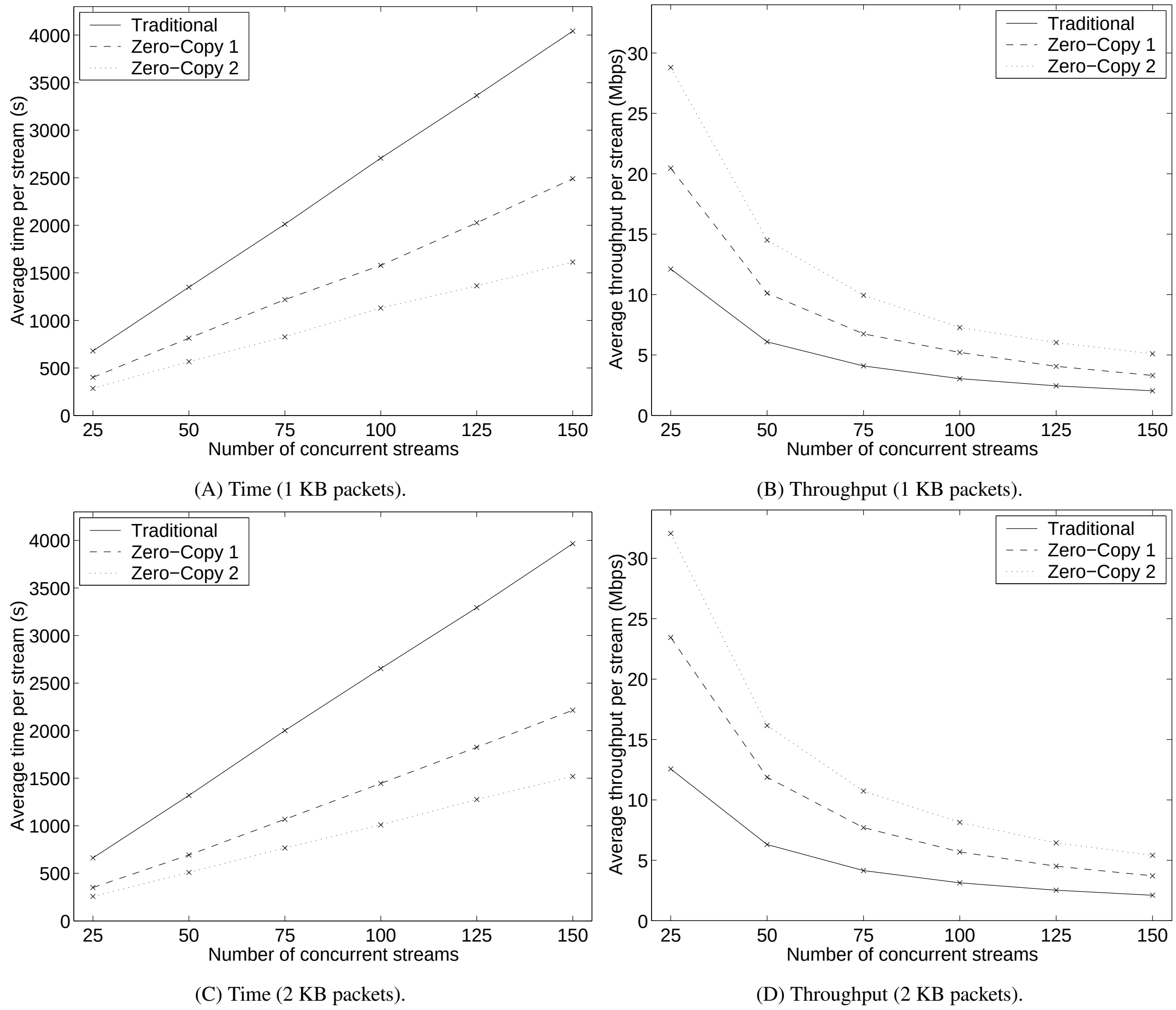


Figure 5.3: Average per stream performance transmitting 1 GB from a memory file system.

5.3.3.2 Context Switches

If there are several concurrent processes on a machine, each process is usually allowed to run for a certain amount of time or until it voluntarily suspends itself, e.g., waiting for I/O to complete. This also consumes resources and reduces the overall system performance. The measured number of context switches per stream we experienced in our multi-stream experiments is nearly constant, because we always have to wait on a read before we can transmit the requested data block, i.e., approximately one context switch per read call ($\sim$16,700). However, in the case where idle time on the storage devices can be used for prefetching, each process can run on the processor for a longer time without being preempted due to waiting on I/O to complete. The total server performance in such a scenario will decrease slightly with the number of simultaneous streams.

However, the measurements in the previous section also show an improvement by reducing the number of system calls. This improvement is larger than expected compared to the cost of accessing the kernel (see Section B.3). However, by looking at the context switches, the `stream_read()`/-`stream_send()` pair has 99.9 % involuntary context switches, i.e., because a higher priority process becomes runnable or the current process exceeded its time slice. The `stream_rdsnd()` system call has about 99.5 % voluntary context switches, i.e., usually waiting for a resource to become available. This may have a large impact on performance. Involuntary context switches are triggered by an interrupt or exception where the interrupt also requires communication with the controller, e.g., to signal an "end-of-interrupt". Processes performing voluntary context switches go to sleep of their own free will, i.e., running in the kernel in this case. The cost related to transferring control to another process is the same, but the difference is how the context switch is activated (see Figure 5.4). To give an indication of some (not all the communication with the controller is included) of the interrupt overhead, we ran a simple micro-benchmark[5] measuring the number of cycles used in our testbed. The results is shown in Table 5.8. Thus, an involuntary context switch is more expensive to perform, and this may be a possible explanation to the unexpected high difference in performance.

involuntary context switch:	*interrupt dispatch → context store → <scheduler> → context restore → interrupt return*
voluntary context switch:	*context store → <scheduler> → context restore*

Figure 5.4: Simplified description of voluntary and involuntary context switches.

Operation	Number of cycles
Interrupt dispatch	145
Interrupt return	141
Sending "end-of-interrupt" to controller	625
Context store	72
Context restore	72

Table 5.8: Context switch and interrupt overhead.

5.3.3.3 Server Side Congestion

Also in the multi-stream scenario, we experience some server-side congestion. However, as more data is transmitted in total, one might expect more data to be lost. This is not the case, because as the system uses more time to perform context switches, the network card is able to use this time to empty (or at least reduce the number of pending packets in) the queue. Nevertheless, as Table 5.9 shows, we do experience some loss meaning that we do have a small margin of error. The packets dropped in the `ether_output()` function are negligible with respect to processing time (but of course the client will experience a lost packet due to the network card bottleneck). This means that the measurements using 1 KB packets is as close as possible to an error free transmission result. When using 2 KB packets, each packet is fragmented, and the `ip_output()` function discards some data before sending it down the protocol stack. This means that one protocol layer is not processed for these packets resulting in a slightly lower time measurement result (see Table 5.5). Nevertheless, compared to the single-user experiments, the amount of loss per stream is very small. For example, transmitting 50 concurrent streams, 0.09 packets are lost per stream which corresponds to about 0.41 μs on average. Compared to the average time to transmit the 1 GB file in this test of 509.05 seconds, the margin of error is about 8 $\times 10^{-8}$ %, i.e., the packet drop rate is insignificant with respect to the time to process the data through the operating system.

[5]This benchmark is implemented by Åge Kvalnes, University of Tromsø. The cycle count is measured using the `RDTSC` instruction, but the code is not serialized using the `CPUID` instruction (see Section A.2.1).

			`ip_output()`				`ether_output()`			
	Size	**Streams**	**max**	**min**	**avg**	**stream avg**	**max**	**min**	**avg**	**stream avg**
`stream_read()/`	1 KB	all	0	0	0	0	0	0	0	0
`stream_send()`	2 KB	25, 50	0	0	0	0	0	0	0	0
		75	163	0	18.11	0.24	6	0	0.67	0.01
		100, 125	0	0	0	0	0		0	0
		150	1293	0	161.62	1.08	31	0	3.88	0.03
`stream_rdsnd()`	1 KB	25	0	0	0	0	5552	1059	2418.60	96.74
		50	0	0	0	0	8584	2679	5058.80	101.18
		75	0	0	0	0	9399	0	6054.50	80.73
		100	0	0	0	0	283	0	55.20	0.55
		125	0	0	0	0	1854	0	261.30	2.09
		150	0	0	0	0	445	0	51.00	0.34
	2 KB	25	0	0	0	0	0	0	0	0
		50	272	79	128.60	2.57	7	1	4.40	0.09
		75	1684	912	1269.70	16.93	40	21	28.00	0.37
		100	2622	1898	2201.30	22.01	71	41	51.60	0.52
		125	5563	3005	3911.40	31.29	107	55	82.20	0.66
		150	6359	5241	5883.00	39.22	147	112	134.60	0.90

Table 5.9: Number of packets lost for all concurrent streams.

5.3.3.4 Estimated Versus Measured Performance Gain

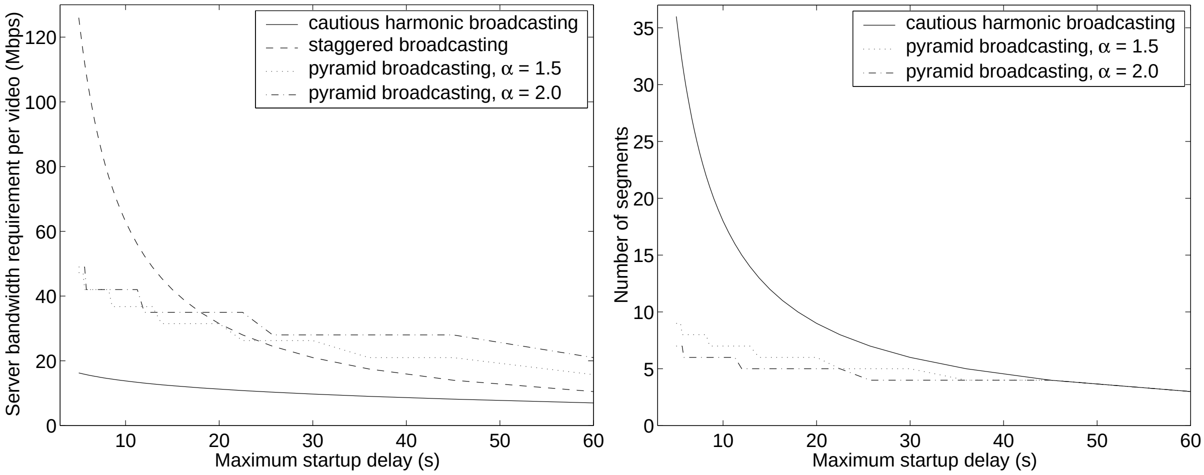
The performance gain in our multi-stream scenario is as expected. The gain per stream is even larger compared to the single stream scenario, because the stream mechanism requires less resources and therefore scales better. The amount of used CPU time is also as estimated. The measured times are slightly larger due to more context switches.

However, the large difference between using the `stream_read()`/`stream_send()` system call pair and the `stream_rdsnd()` system call is not as first expected. A further analysis shows that the difference probably is due to having involuntary versus voluntary context switches, respectively. An involuntary context switch is triggered by an interrupt or an exception which additionally introduces communication with the interrupt controller.

5.3.4 Broadcasting Protocol Gain

A broadcast protocol for periodic services improves the efficiency of our MoD server. Such schemes reduce the server bandwidth requirement at the cost of increased client workload and resource requirement. In Section 4.2.2.3, we showed one example looking at the scenario described in Section 2.2. The traditional stream-per-client approach requires the server to transmit 1000 concurrent 3.5 Mbps streams. Using cautious harmonic broadcasting and assuming a maximum startup latency of 5 seconds, the video is partitioned into 36 segments (transmitted on 35 channels), and the total server bandwidth requirement is 16.26 Mbps regardless of the number of concurrent clients using equation 4.1. Increasing the size of each segment to 10 seconds, further reduces the required bandwidth to 13.79 Mbps. As the stream-per-client approach overhead increases with the number of clients, the broadcasting scheme overhead increases with the number of segments. In Figure 5.5A, we show the relationship between startup latency and server bandwidth requirement in our NoD example. As we can see, the server bandwidth requirement is greatly reduced, regardless of maximum startup delay. The figure also shows staggered broadcasting and pyramid broadcasting. In the pyramid broadcasting scheme, we have varied the α

parameter which is the segment increase value. The number of channels needed are found using a first segment of 5 seconds and increasing each following segment with a factor of α, i.e., we increased and counted segments until we reached a total size of 3 minutes. Each channel transmits at a bandwidth $\alpha \times b$ where b is the playout rate. The total server bandwidth requirement per video is lowest using the cautious harmonic broadcasting, i.e., 16.26 Mbps compared to pyramid broadcasting 47.25 Mbps (α = 1.5) and 49 Mbps (α = 2) and staggered broadcasting 126 Mbps. All schemes have a significant improvement over the stream-per-client scenario with 1000 concurrent clients (3.42 Gbps). The pyramid scheme has a stair-shaped graph, because the number of segments is not increased for each startup latency. The number of needed segments is depicted in Figure 5.5B[6], and we see that we need fewer segments using pyramid broadcasting, but at the cost of increased bandwidth. Furthermore, not every segment require an own channel, because several segments can be multiplexed into one channel using the low bandwidth transmission in cautious harmonic broadcasting.



(A) Maximum startup latency versus server bandwidth requirement.

(B) Maximum startup delay versus number of segments.

Figure 5.5: Broadcast protocol comparison using 3.5 Mbps, 3 minutes videos.

5.4 Network Level Framing

In this section, we describe our performance evaluation of the NLF mechanism. First, we analyze the expected gain using statistics and measurements from others and by performing a simple measurement transmitting data with and without checksum operations. Then, we look at the performance gain measured in the kernel using our NLF implementation in NetBSD.

5.4.1 Expected 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 of the integrated error management scheme shows, 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.

[6]Staggered broadcast needs only one segment, but the broadcast is restarted after a certain interval requiring several channels, i.e., video_length/maximum_delay which will be one more channel than cautious harmonic broadcasting.

Data touching operations like checksum calculation are addressed as one of the major time consumers in the end-systems [38, 85]. For example, in a performance measurement described in [84], the processing overhead of data touching operations of the UDP/IP protocol stack is 60 % of the total software processing time. In [113], 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 packets on disk including output from the data touching, CPU intensive checksum calculation. The gain of our checksum pre-computation depends on the packet size. In our scenario using the described packet sizes 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 % (1 KB packets), 0.39 % (2 KB packets), 0.19 % (4 KB packets), and 0.10 % (8 KB packets) of UDP data. In our prototype implementation, we generate the header during transmission time, i.e., the on-line checksum procedure 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, and the time spent on checksum operations should therefore be reduced to nearly the same (we must add some time for initializing the checksum function, etc.). 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 [84], we might expect a processing speed-up of a factor of 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 approximate 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 5.10. The throughput improvement of our approach should be nearly the same[7] 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	$\sim$420	$\sim$385	$\sim$9.1
8 KB	$\sim$450	$\sim$405	$\sim$11.1

Table 5.10: Maximum measured UDP throughput.

5.4.2 Kernel Measurements

To see the real performance of NLF, we have measured the time spent in the traditional UDP protocol and the respective time spent in our protocol using the `udp_QuickOut()` function. The experiment was carried out in our test environment by transmitting a large data file of 225 MB[8] using the `stream_rdsnd()` system call providing a zero-copy data path as described in Section 4.2.1.4. The time was measured using the software probe described in Section A.2, and we sent 1 KB, 2 KB, 4 KB, and 8 KB packets, i.e., this corresponds to sending 219483, 109742, 54871, and 27436 packets respectively. When looking at the results below, keep in mind that the protocol processing is "protected" by `splsoftnet()` which blocks soft network interrupts (protocol stacks), i.e., a packet is allowed to be processed through the communication protocols without being interrupted by other processes sending packets. Furthermore, no other user level processes are allowed to start running before the current process returns from the kernel or voluntarily performs a context switch, e.g., waiting for a disk operation

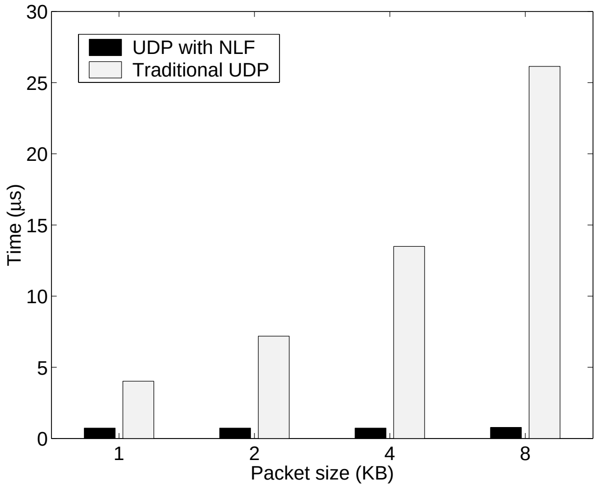
[7]It will not be exactly the same, because we still have to update the checksum. The checksum is not completely removed, but only reduced to a minimum.

[8]We used a 225 MB file in the measurements using in-kernel probes, because the memory area holding the probe data was limited, and we performed one measurement per packet (each requiring two probes).

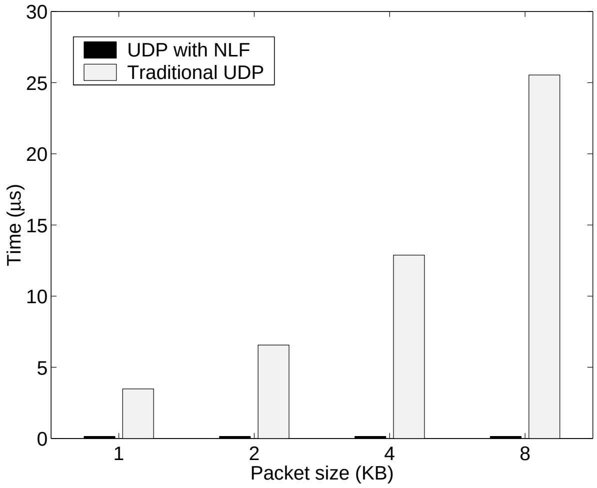
to complete. Also note that our implementation is in the C-programming language with the portable version of the checksum routine as a base, whereas the traditional UDP protocol uses the performance optimized checksum routine in assembly language modified for each CPU according to [20].

5.4.2.1 Protocol Processing Time

Table 5.11 and Figure 5.6A present the measured times spent in the UDP protocol. As we can see, our version of the UDP protocol is faster than the traditional protocol. Furthermore, as we only compute the checksum over the packet headers, the overhead of processing the packet through the UDP protocol is approximately constant using NLF (0.14 μs for the checksum operation and 0.77 μs for the entire UDP protocol). Using the traditional protocol, we see that this overhead is dependent on the packet size.



(A) Whole UDP protocol (including checksum).



(B) Checksum operations.

Figure 5.6: Average time per packet spent in the UDP protocol and on checksum operations.

Table 5.11 and Figure 5.6B present the measured times calculating the checksums. 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 95.98 % (1 KB packets), 97.87 % (2 KB packets), 98.91 % (4 KB packets), and 99.45 % (8 KB packets) when using NLF. 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 5.7. 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 or traditional UDP.

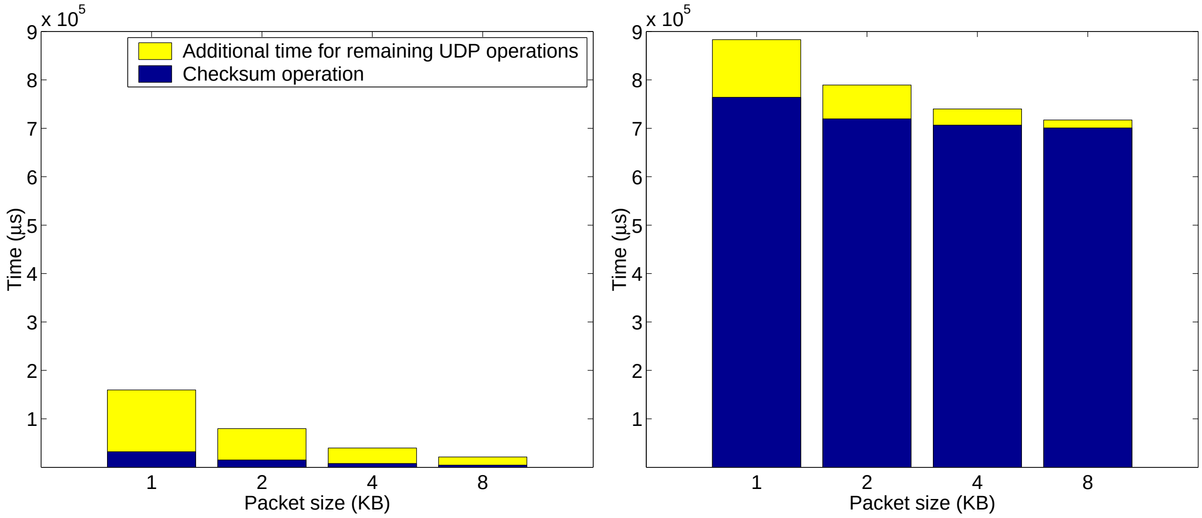
5.4.2.2 Transmission Time

Even though we experience some server-side congestion (see Section 5.3.2), we have also measured the time to transmit 1 GB from a memory file system to the network using the zero-copy data path and NLF. We have only tested this in a single stream scenario, because we expected even more loss

Operation	Size	max	min	avg	stdev	Total†	99% ci	95% ci
UDP with NLF	1 KB	34.82	0.57	0.73	0.21	159534.04	[0.65 - 1.65]	[0.66 - 0.93]
	2 KB	28.54	0.49	0.73	0.20	79842.12	[0.64 - 1.66]	[0.65 - 1.29]
	4 KB	15.67	0.58	0.73	0.16	39795.15	[0.64 - 1.27]	[0.65 - 1.23]
	8 KB	9.39	0.47	0.77	0.22	21089.07	[0.64 - 1.40]	[0.65 - 1.28]
Traditional UDP	1 KB	44.97	3.54	4.02	0.72	883065.69	[3.71 - 5.90]	[3.75 - 4.48]
	2 KB	63.49	3.64	7.19	0.92	789312.83	[6.70 - 8.71]	[6.76 - 7.99]
	4 KB	52.20	12.63	13.49	1.29	740045.14	[12.81 - 14.50]	[12.91 - 14.05]
	8 KB	63.94	9.95	26.14	1.80	717267.33	[25.20 - 40.74]	[25.36 - 26.82]
`in_QuickCksum()`	1 KB	25.84	0.11	0.14	0.17	32054.33	[0.12 - 0.33]	[0.12 - 0.28]
	2 KB	16.83	0.11	0.14	0.12	14937.14	[0.12 - 0.29]	[0.12 - 0.24]
	4 KB	9.65	0.11	0.14	0.15	7825.87	[0.12 - 0.30]	[0.12 - 0.28]
	8 KB	10.79	0.11	0.14	0.32	4292.73	[0.12 - 0.30]	[0.12 - 0.24]
`in_cksum()`	1 KB	62.84	2.97	3.48	0.95	764056.33	[3.11 - 4.43]	[3.14 - 4.04]
	2 KB	50.03	3.21	6.56	1.33	719533.76	[6.07 - 7.67]	[6.12 - 7.00]
	4 KB	59.99	11.97	12.88	1.54	706522.90	[12.22 - 20.81]	[12.32 - 13.41]
	8 KB	75.25	9.30	25.54	2.14	700819.37	[24.45 - 45.24]	[24.54 - 26.30]

†This column is the total amount of time used for the whole stream.

Table 5.11: Time per packet spent in the UDP protocol and on checksum operations (μs).



(A) UDP with NLF.

(B) Traditional UDP.

Figure 5.7: Accumulated UDP protocol execution time versus total time spent on checksum operation.

due to an overloaded network queue in the multi-stream scenario. Table 5.12 shows the results using the `stream_rdsnd()` system call with NLF in a no load scenario. Compared to the corresponding results without NLF in Table 5.4, we see a small improvement, i.e., 2.2 % (1 KB packets), 11.6 % (2 KB packets), 24.0 % (4 KB packets), 22.0 % (8 KB packets). However, comparing the amount of server-side loss in Table 5.6 and 5.13, we also see that the amount of loss has increased. Using 1 KB packets, all loss is in the `ether_output()` function, which means that this function is executed in this case. However, using the other packet sizes, we experience loss in `ip_output()` and approximately 391 ms (2 KB packets), 679 ms (4 KB packets), and 695 ms (8 KB packets) should be added to our measured times.

Adding the execution time for the ethernet function, then gives a smaller improvement of 2.2 % (1 KB packets), 8.1 % (2 KB packets), 16.5 % (4 KB packets), 13.6 % (8 KB packets), i.e., with and without NLF.

Operation	Size	max	min	avg	stdev	99% ci	95% ci
`stream_rdsnd()`	1 KB	10.65	10.61	10.63	0.01	[10.61 - 10.65]	[10.62 - 10.64]
	2 KB	9.83	9.60	9.65	0.05	[9.60 - 9.83]	[9.62 - 9.82]
	4 KB	6.88	6.82	6.85	0.01	[6.82 - 6.88]	[6.82 - 6.87]
	8 KB	6.48	6.44	6.46	0.01	[6.44 - 6.48]	[6.44 - 6.47]

Table 5.12: Time to transfer 1 GB from a memory file system using NLF (no load).

	Total			`ip_output()`			`ether_output()`		
Size	max	min	avg	max	min	avg	max	min	avg
1 KB	14515	12665	13514.70	-	-	-	14515	12665	13514.70
2 KB	88310	85080	86938.00	85812	82659	84492.70	2526	2351	2445.30
4 KB	93608	86446	90463.80	90968	83809	87811.30	2691	2597	2652.50
8 KB	48848	47430	48436.00	47576	46266	47184.80	1300	1164	1251.20

Table 5.13: Amount of loss in number of packets using NLF.

5.4.2.3 Estimated Versus Measured Performance Gain

To see the total speed up in the amount of time used in the operating system kernel, we also measured the used CPU time in the kernel per process. In this experiment, we transmitted the 225 MB file using both the traditional UDP protocol and NLF. As Table 5.14 and Figure 5.8 show, the total CPU time used by the kernel is greatly reduced using NLF compared to using the traditional UDP protocol. The total time is reduced in average with 51.32 % (1 KB packets), 54.07 % (2 KB packets), 61.16 % (4 KB packets) and, 61.66% (8 KB packets). This means that our prediction of having a processing speed-up of a factor of two in Section 5.4.1 holds. Furthermore, the time to execute the checksum procedure is reduced by about 95.9 - 99.5 % depending on packet size when using NLF. This is approximately as expected in Section 5.4.1.

Operation	Size	max	min	avg	stdev	99% ci	95% ci
UDP with NLF	1 KB	773013	726169	748494.85	9906.31	[726169 - 773013]	[729842 - 768811]
	2 KB	705007	643219	673464.21	9842.35	[643219 - 705007]	[655569 - 694999]
	4 KB	524133	475994	495996.71	8141.70	[475994 - 524133]	[481033 - 512695]
	8 KB	505235	453829	480951.96	10384.19	[453829 - 505235]	[462563 - 501402]
Traditional UDP	1 KB	1567585	1501467	1537586.03	13364.58	[1501467 - 1567585]	[1509630 - 1563475]
	2 KB	1509054	1415464	1466213.96	16497.26	[1415464 - 1509054]	[1435745 - 1500248]
	4 KB	1316653	1243363	1276913.82	12076.36	[1243363 - 1316653]	[1254025 - 1299215]
	8 KB	1278349	1212957	1254541.61	13022.20	[1212957 - 1278349]	[1225299 - 1274588]

Table 5.14: The total time spent in the kernel using the traditional UDP protocol and using NLF (μs).

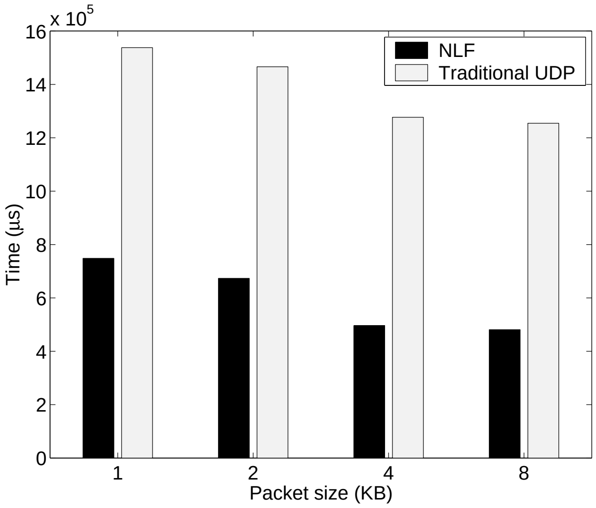


Figure 5.8: Time spent in the kernel using traditional UDP and using NLF (μs).

5.5 Total Server Performance

The INSTANCE enhancements are orthogonal, and as shown in the previous sections, each mechanism increases performance in a particular part of the system. The integrated error management removes the 24 Mbps (1 KB packets) encoding bottleneck on our test machine at the cost of 12.5 % extra storage and bandwidth using the current scheme configuration. Thus, 12.5 % of the data transmitted is parity data for error recovery which means that the effective bandwidth of the server is reduced. Nevertheless, the encoding bottleneck and the retransmission overhead are removed which means that an effective throughput of about 875 Mbps could be achieved as the zero-copy measurements show a total throughput close to 1 Gbps.

Figure 5.9 shows the gain in used CPU time in the kernel running a server without FEC. We see that using the zero-copy data path together with NLF reduces the used CPU time by 66.18 % (1 KB packets), 70.37 % (2 KB packets), 75.95 % (4 KB packets), and 75.25 % (8 KB packets) compared to the traditional data path. If the server should also perform FEC encoding, an additional overhead of 346.92 seconds (1 KB packets), 388.54 seconds (2 KB packets), 380.50 seconds (4 KB packets), and 375.45 seconds (8 KB packets) should be added to the approximately 10 seconds used for the server using the traditional data path (same test as in Section 5.2.2, but on a 1 GB file). Additionally, the total time used by the system includes the time spent executing instructions in the server application, but as these times are below 0.5 seconds regardless of which server we use, we have not added this overhead into the figure.

The cautious harmonic broadcasting scheme increases the client reception rate (but not the consumption rate), and this influences the integrated error management scheme recovering from lost and damaged packets at the client side. We can decode the data as it is played out or when the data arrives. Decoding at presentation time reduces the CPU requirement at the cost of also buffering the parity data until it is presented to the user. Decoding at arrival increases the decoding performance requirement, because all streams are decoded concurrently. However, some of the machines used in our test are capable of decoding multiple streams from the broadcast protocol. For instance, our NoD example requires 16.26 Mbps, and some of the tests indicate that even a 350 MHz machine can decode these streams. Nevertheless, PentiumIV processors are already relatively cheap today, and in a couple of years, almost everyone will have a machine capable of the decoding load imposed by retrieving several concurrent streams. Thus, the FEC decoding operation imposed on the client by the integrated error management scheme will not be a bottleneck using the broadcast protocol.

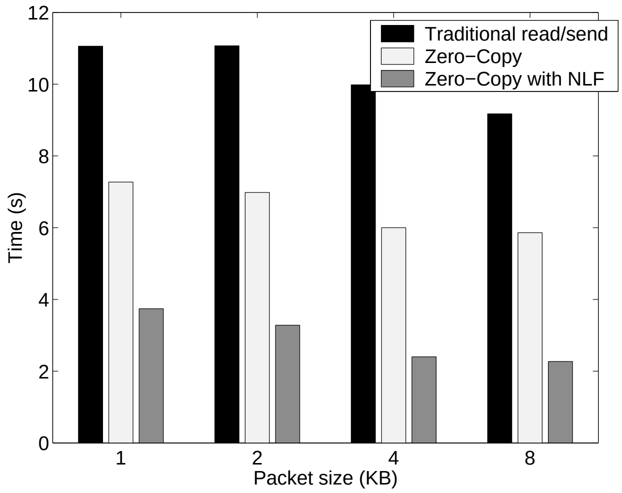


Figure 5.9: Histogram over used CPU time in the kernel.

5.6 Discussion and Conclusions

Our performance experiments show that a single disk storage system is a severe bottleneck in a Gbps environment (see Section 5.3.2.1). However, even though we need a persistent storage system and not only a memory file system which is erased during a crash or reboot, our performance tests show the benefits of the zero-copy data path assuming an adequate storage system. The storage system bottleneck can be solved using parallel off-the-shelf disks. The Seagate Cheetah X15 (ST318451LW) [203] achieves a minimum data rate of 299 Mbps. Thus, connecting several such disks to the new SCSI fiber channel interface, which offers extremely fast bus data rates of up to 3.2 Gbps for a dual loop, may solve the Gbps storage system bottleneck. Additionally, our tests indicate the performance of servers that use memory-based files after system start-up or the performance of servers using future mechanical-free storage devices. Companies like Opticom ASA [200] are building such storage devices. If this technology, based on memory films, can be made persistent, it will remove the need for mechanical devices, such as disks and tapes. At the current stage of the development of this technology, Opticom projects that their memory films are (will be) about one million times faster than any mechanical system.

Figure 5.7 shows that, by using NLF, 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 [38, 142], the overhead of filling in header fields can be reduced. In this context, the headers will always be identical, with the exception of the checksum field, 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 described in Figure 4.19 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. However, such an implementation of NLF and a further analysis of this topic are considered as future work.

Conventional wisdom says that bigger packets are better due to less costs transmitting the data [153]. The packet sizes are so far determined only on the recovery code's coding performance, and based on the experiments evaluating these schemes, a packet size of 1 KB or 2 KB seems to be appropriate with respect to start-up latency and decoding throughput. The zero-copy and NLF experiments show better performance using larger packets, but the amount of server-side congestion also increases (due to low processing overhead and network card bottleneck). However, the packet size also affects performance in other components like the network itself and the routers. Fragmentation in the IP layer is not always appropriate which means that each packet should be below the path's MTU. On the other side, the Internet is heterogeneous and different links might have different MTUs. Furthermore, a basic problem of routers is that they are store-and-forward devices. They normally receive the entire input packet, validate the IP header including the IP checksum, make their routing decision, and start sending the output packet. Assuming a 1 Gbps network line, the latency time in each node is given by $\frac{p_1+p_2}{2^{30}} + p_3$ where p_1 is the amount of information data (in bits) in each packet, p_2 is the header size (in bits), and p_3 is processing and queuing overhead. This means that a smaller packet size is better. In summary, using large packets, the reduced costs associated with the network are less data to send (less packet headers), fewer routing decisions, and reduced protocol processing and device interrupt handling overhead. Smaller packets give less packet fragmentation and reduce latency in the intermediate nodes. Thus, the optimal packet size is determined by several factors which vary for each link in a heterogeneous environment, and more research is therefore required in this area [153].

In the multi-stream tests above, several streams read data from the same file stored in a memory file system. We minimized the buffer cache in the file system to minimize any caching effects. In the broadcasting scheme, each stream will read data from different files, and caching effects will not be present. However, minimizing the buffer cache this way does have side effects. When using a small buffer cache, the amount of memory (and the number of buffers) in the cache is decreased. There might be a problem when a process requests a new buffer for an I/O operation, and the process might have to wait longer for an available buffer compared to the case of using a normal buffer cache. On the other hand, a larger buffer will probably have available buffers, but also gain performance due to caching effects. To see the difference, we made a simple monitor (taking small snapshots only) to see the number of times a new buffer had to be allocated, how many times a requested buffer was not available (which means that the process had to retry until a buffer was available), and the size of each request (`buf->b_bufsize`). Our results show that a small cache benefited minimally from caching, i.e., a new buffer has to be allocated, whereas the large cache has large caching gains. On average, in the small cache scenario only, every other process had to retry to allocate one buffer during the whole transmission, i.e., (almost) no latency is added due to multiple calls to `getblk()`. We do not know, however, how long time the process slept before an available buffer was allocated. The size of each data request to the storage system also varied according to the amount of available memory. The average size was about 20 KB and 50 KB using small and large caches, respectively. Thus, a small cache will also have more requests to the storage system still only retrieving the same amount of data. Thus, both small and large buffer caches introduce a small margin of error in our multi-stream tests, and the memory file system requires CPU resources, memory space and bandwidth, and causes data in the cache to be flushed more often. Future tests should therefore be performed using a disk-based storage system capable of delivering data at the requested rate (or faster) where each stream can read data from separate files, i.e., no caching effects and no additional latencies waiting for available buffers.

As described in Section 5.3.2, we have a source of error in the measurements, because we experienced some congestion in the ethernet queue. This means that neither the `ether_output()` function nor the low-level network card driver code are executed for all the packets processed through UDP/IP. However, as the loss-experiments show, the margin of error is about 2 - 3 %, and if we add the execution time of the `ether_output()` function for the lost packets, we still have a huge improvement compared to the traditional data path. Increasing the queue length (`if_snd.ifq_maxlen`), which is by default 50 (`IFQ_MAXLEN`) or, as in our case, increased by the network card driver to 511 (`TI_TX_RING_CNT` - 1), will help if the network card is sometimes idle. This means that data is sent in large bursts and there is a long time period between each burst making the driver queue empty before each burst, but this is not the case in our scenario. Figure 5.10 depicts a plot of the queue level in one of the transmissions where we monitored the queue length each time a packet was to be enqueued, and as we can see the driver has no idle time. The network card is simply not able to transmit the packets fast enough, i.e., the data is processed too fast through the operating system, and we therefore have a congestion within the end-system itself. Nevertheless, most of the data processing operations are executed for all the packets, i.e., all packets are processed through the UDP/IP protocols. This means that even though we have some server side loss, the results give a good indication of the improvements using our mechanisms. There is, however, a need for some rate control mechanism which is not provided by UDP or a mechanism which waits if the queue is overloaded so that we do not lose packets in the server end-system.

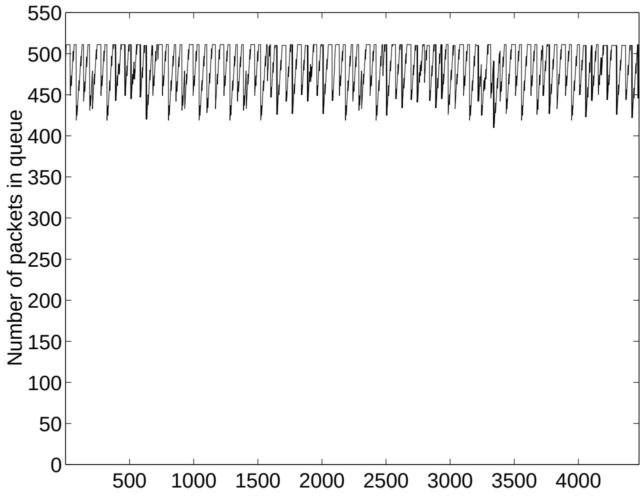


Figure 5.10: Plot of the ethernet queue level (amount of packets in the queue).

If we look at the used CPU time and assume that the disks and the network card are not a bottleneck, the used CPU times to process data through the system indicate a throughput of 1.49 Gbps and 3.83 Gbps using the zero-copy data path without and with NLF, respectively. Thus, if the operating system processing was the only bottleneck, we should be able to achieve data transmissions at these speeds. Note also that, in these tests, data is read from a memory file system making a memory copy operation, i.e., replacing data in the cache and consuming CPU cycles and memory bandwidth. This means that data could be sent even faster if the only processing done by the CPU was the disk driver execution (assuming a sufficient disk-based storage system bandwidth). This means that using the mechanisms described in this thesis, the operating system is no longer a bottleneck, because data can be processed through the system faster than our hardware components can manage.

In summary, our mechanisms perform as expected. Storing parity data on disk removes the FEC encoding bottleneck. The zero-copy data path and NLF reduce the time to process data from disk to network interface, and the broadcasting protocol enables data sharing between concurrent clients by broadcasting one set of data to all clients viewing the same file.

Chapter 6

Conclusions

In the research performed in the context of the INSTANCE project, we addressed the challenging problem of optimizing performance of an MoD server. In this chapter, we summarize the contributions of this thesis and explore directions for future research.

6.1 Summary

The emergence of high data rate applications like NoD and VoD presenting audio and video data to remote viewers has made the traditional I/O data path a bottleneck in a server supporting a lot of concurrent users. The operating system mechanisms are traditionally designed in a layered structure and optimized for management of small files. In the case of multimedia applications, these operations are not efficient, and the operating system itself imposes a lot of overhead transferring data from the storage system to the communication system.

In this thesis, we have identified three different bottlenecks in traditional systems in the context of high data rate multimedia applications: (1) error management, (2) memory management, and (3) communication protocol processing. For example, we have redundant functionality, in-memory copy operations even though data is not manipulated, and repeated identical operations on the same date elements. Based on system analysis and literature work, we have designed and implemented three solutions each improving the performance. Our performance tests show that we overcome the limitations imposed by the identified bottlenecks.

6.2 Contributions and Critical Review of Claims

The overall goal of the INSTANCE project and this thesis is to improve the I/O performance of MoD servers by avoiding the major bottlenecks in the common case operation of storage nodes, i.e., retrieving data from disk and sending it to remote clients. We achieve this goal by applying three orthogonal techniques:

- *integrated error management*, which reuses storage system parity data and thereby removes the FEC encoding operation at the server side,
- *zero-copy-one-copy memory architecture*, which eliminates all in-memory copy operations by integrating the buffering schemes and sharing one single data element between all concurrent users using a delay-minimized broadcasting scheme, and
- *NLF*, which reduces the transport level protocol processing by precalculating the packet payload checksum.

The performance of these mechanisms is summarized by the following critical review of the claims stated in Section 1.4:

Claim 1: *Error management is a potential, but removable, bottleneck.*
Integrated error management frees the MoD server from all resource intensive error management tasks by using precomputed parity data in a FEC scenario. In general, using n % parity data, we can recover from a total data corruption rate of n % in either the storage or the communication system. The described and tested scheme configuration allows recovery from one disk failure and 12.5 % packet loss or corruption in the network. Furthermore, the decoding time at the client is relative to the number of network errors and introduces in the worst case only a delay of 100-700 ms depending on the packet size on a client with a 933 MHz CPU. The decoding throughput measurements show that the multimedia data stream can be decoded in time for a hiccup free presentation on the client side on several different machines and architectures.

Claim 2: *Memory copy operations are a potential, but removable, bottleneck.*
The zero-copy-one-copy memory architecture integrates the file system and communication system buffering and creates an in-kernel data path. The removal of copy operations reduces the amount of needed memory and CPU resources, and minimizes the time to transmit data. We have shown that the zero-copy data path increases the number of concurrent clients that can be supported by at least 100 %.

Claim 3: *Concurrent clients represent a potential, but removable, bottleneck.*
Data is periodically broadcasted using the zero-copy-one-copy memory architecture. Thus, an unlimited number of users can receive the broadcasted packets. Furthermore, using the cautious harmonic broadcasting protocol, the server bandwidth requirement per stream is reduced, and the number of streams is maximized using the zero-copy data path. Thus, by broadcasting data, the number of users is no longer a bottleneck.

Claim 4: *Transport level checksum operations are a potential, but removable, bottleneck.*
The NLF mechanism significantly reduces the only data touching operation at transmission time in our MoD server. The resource requirement per stream is reduced, and combined with our in-kernel data path, the used CPU time by the kernel is reduced by 66.18 - 75.95 % depending on the packet size.

Claim 5: *By removing the mentioned bottlenecks, the operating system is no longer a critical component.*
The bottlenecks addressed in this thesis strongly limit system throughput in the context of high data rate applications. Our results show that using our mechanisms in a single stream scenario and thereby eliminating the noted bottlenecks, data can be processed faster through our system compared to what standard off-the-shelf hardware can handle. Assuming that hardware is not the bottleneck, our tests measuring the consumed CPU cycles indicate a throughput of 3.83 Gbps using the copy-free data path and NLF. This data rate is above the hardware limitations of network cards today, i.e., the operating system is no longer a critical component. Another possible bottleneck is the peripheral component interconnect (PCI) bus. A 64-bit, 66 MHz PCI bus can at maximum transfer 4.2 Gbps where a transfer consists of one address phase and any number of data phases [204], i.e., the effective data rate is decreased due to bus idle time between transfers and addressing and signaling packets.

In a multi-stream scenario, the overall system performance drops slightly. The overhead of context switches is substantial, but if we use a faster CPU (which is available), data could still be processed through the operating system faster than (our) hardware is currently capable of handling. This overhead will also decrease if we use a thread-based server whereas we now have one process for each stream.

> Today, the network card bottleneck can be removed similar to the storage system bottleneck. There exist network cards supporting virtual networks which provide load balancing by spreading the transmitted data over parallel network cards to increase outbound bandwidth, e.g., the 3Com network card used in our tests [1, 176]. There is, however, lack of support in many operating systems (at least NetBSD[1]). Furthermore, parallel Gbps network cards will reach the single PCI bus bandwidth limit, but several PCI buses can be used connected with a PCI bridge.
>
> Thus, the overall conclusion is that there still are parts of the operating systems that might be critical in high data rate systems. Our tests show that hardware limits the performance in our experiments, but hardware components can be organized in parallel to increase performance. To see whether the operating system still is a bottleneck requires further tests on parallel high-performance hardware. However, removing the bottlenecks addressed in this thesis is a marginal step towards the goal of invalidating the phrase *operating systems are not getting faster as fast as hardware* [109] – at least in the context of our special read-only MoD scenario.

Thus, in the context of high data rate multimedia systems, our bottleneck claims hold. The bottlenecks can be removed, and this makes the hardware in our testbed a bottleneck. The fifth claim about the operating system as critical component in a multimedia environment will in a lot of scenarios be true. However, to prove this claim generally, there has to be done more research in this area.

For the example application described in Section 2.2, transmitting a 3.5 Mbps, 3 minutes video to 1000 concurrent users, the resource requirement is drastically reduced when using the mechanisms proposed in this thesis. The amount of memory is reduced from 192 KB to 64 KB (67 % reduction) per read and transmit operation, and the in-memory copy operations over the user-kernel boundary are eliminated. The number of system calls is reduced by 50 % using the `stream_rdsnd()` system call, the buffer cache operations are omitted, and the communication protocol processing is reduced greatly. The needed number of streams per video clip is reduced using a broadcasting scheme which implies a further reduction in total memory usage and bandwidth requirement. For example allowing a maximum delay of five seconds, the number of streams is reduced from 1000 to 35, and the total server bandwidth requirement is reduced from 3.42 Gbps to 16.26 Mbps using the cautious harmonic broadcasting scheme. It is also noteworthy that, the server requirement will increase further with more clients using a per-client connection, whereas the workload will be approximately constant regardless of the number of clients using a broadcast protocol.

In summary, we have shown that the INSTANCE approach and its potential to improve the I/O performance of a single storage node by a factor of two is to the best of our knowledge unique. There are several important results in the area of zero-copy implementation and in the area of pyramid broadcasting schemes; and some works report the usage of prefabricated packets. However, none of them has reported a combination of two or more of these three techniques and their corresponding integrated performance improvement.

6.3 Critical Assessments

The mechanisms proposed in this thesis mainly address application scenarios where data is streamed to the client without any data manipulations at the server side. Furthermore, the server design is optimized for streaming without any user interaction. Which mechanisms to use depends on the client behavior (interactions and file access patterns) and the supported functionality (encryption, changing video codec, etc.), i.e., different applications may need different mechanisms.

Since we started this work in 1998, there has been considerable improvements in hardware developments. For example, new network cards today can perform on-board checksum calculations offloading

[1] Such support is currently being implemented, see http://mail-index.netbsd.org/tech-kern/2001/07/13/0017.html and http://mail-index.netbsd.org/tech-kern/2001/07/13/0020.html

this intensive task from the CPU. However, even though the hardware has been available for a while, the operating systems have only recently added support for this[2]. This means that the CPU intensive checksum operation can now be performed by the network card itself which means that the current NLF mechanism might be unnecessary unless it is extended as described in Section 6.4.3. Nevertheless, the NLF functionality will still be valuable in systems not supporting on-board checksumming.

Our current prototype and test environment is limited by two elements: the speed of the single disk and the number of receivers. However, these limitations have been accounted for in our test cases and do not effect the usefulness of the results presented here. First, the performance of a chain of eight high-speed disks or a RAID system using disks like the Seagate Cheetah X15 (ST318451LW) [203], each capable of achieving a minimum data rate of 299 Mbps, connected to a 64 bit 66 MHz PCI bus are able to deliver a total data rate above the 1 Gbps limitation of the network card. Second, the usage of IP multicast to multicast data to multiple receivers instead of IP unicast to send data to a single receiver has no influence on the server load and performance of the send operation.

6.4 Open Issues and Future Work

An MoD system enhanced with the mechanisms described in this thesis has better resource utilization and performance compared to a system using traditional operating system mechanisms. However, there are still a lot of open questions and unsolved tasks, and the MoD server design presented here can be extended in several ways. In the subsections below, we give some examples.

6.4.1 Implementation Issues

The current data prefetching approach only prefetches one data element, i.e., data for the next operation in a one-block-read-ahead manner, whereas our disk-based storage system performance evaluation in Section B.1 indicates that the larger the read operation the better. Reading and transmitting more data at a time is not a solution, because our tests show that we might lose packets in the packet queue before data is transmitted to the network card if bursts of data larger than 64 KB are sent to the communication system. Therefore, a possible solution could be to replace the traditional FFS file system which has block allocation and data placement policies that are suboptimal for multimedia data storage with special multimedia file systems like *Symphony* [143, 144] or *Minorca* [166]. Additionally, one could have a prefetching pool preloading multiple 64 KB data elements at a time reducing the disk access frequency. The data pool should have a lower limit on the number of elements in the pool. When the number of elements remaining in the pool drop below the lower limit, a group of new elements is fetched from the storage system.

The current NLF prototype precomputes the checksum over the application level data, i.e., not over the packet header itself. Figure 5.7 shows that it is the packet header generation and the checksumming that are the most time consuming operations in our modified UDP protocol and not the data payload checksum calculation. Thus, by additionally using the idea of pregenerating header templates [38, 142] and precomputing the checksum over the template, the overhead of filling in header fields and the header checksum operation can be reduced - both for UDP and IP. Furthermore, we now hold the whole NLF meta-data file containing the precomputed checksums in memory, because we do not want the low-bandwidth disk accesses of the meta-data file to interfere with the high data rate accesses of the application information data. However, this is at the cost of much memory space. One way of improving this is to implement a "sliding window" over the meta-data file, i.e., holding only the most relevant parts of the file in memory.

[2]NetBSD added support for in-bound and out-bound IPv4, TCP, and UDP checksumming to NetBSD-current in June 2001, see http://mail-index.netbsd.org/tech-net/2001/06/02/0002.html.

Our zero-copy data path is only accessible through our stream API in our prototype. However, if traditional applications should be able to use this mechanism without reimplementing the parts making I/O requests, we should extend the interface to include standard API. This could simply be done by including an extra flag when opening a file indicating that the file should be streamed from disk to network without any server-side modifications, i.e., using the zero-copy data path. If a stream flag is set, this indicates that the `f_ops` file operation pointer in the `struct file` kernel descriptor table, i.e., one entry for each open kernel vnode and socket, should point to our stream operations vector (`streamops`) instead of the traditional file vnode operations vector (`vnops`) using the traditional data path. In the case of merging the stream system calls, we could, for example, make the read perform both disk and network I/O, whereas the send call will be empty.

Our tests show that packets are occasionally dropped in the network driver queue due to overload. Depending on the latency requirement, this can be handled differently. In an application which does not have room for delays, one might drop the packet like the current prototype does. However, if a small buffering is applied or if the timing requirements is flexible, we should add a control mechanism. If the packet is dropped, `ether_output()` or `ip_output()` calls `m_freem()` which frees the mbuf chain containing data for this packet. A control mechanism can for example check the mbuf type (`MT_MMBUF` in case of a stream) before calling `m_freem()` to drop the packet. The error is then returned to our buffer manager, and the packet is just retransmitted to the communication system.

6.4.2 Short-Term Experimental Issues

In the short term, further tests could be performed on the existing prototype. As our infrastructure does not have support for broadcast or multicast, we could not perform tests on the broadcasting scheme in our memory architecture. It might be hard to come up with specific figures or numbers of performance gain regarding broadcast, because the resource requirement is constant regardless of whether we have one or several concurrent clients, but we move focus from clients to streams, where each stream can support several (unlimited) clients. However, it should be proved that the scheme works in practice, and the experienced workload and delay on the various clients should be measured.

Even though we purchased a top-end workstation for performance measurements, we meet some hardware limitations and bottlenecks. This forced us to use a memory file system and prevented us from evaluating the prefetching mechanism we proposed, because there is no free time in the storage system to prefetch data. The current mechanism should probably perform better in a system where the storage system is not a bottleneck like the tests in the MARS project [28] indicate. Furthermore, some of our tests show that the 1 Gbps network card is a bottleneck, i.e, either the queue in the communication system or the queue on the network card (the transmit ring) is overloaded. Furthermore, the amount of memory was a limiting factor in our experiments, because we used a memory file system to overcome the storage system bottleneck, and each stream holds the NLF meta-data file in memory. Extended performance tests could be performed on an even better machine using a disk array and several parallel network cards.

6.4.3 Long-Term Research Issues

Our testbed did not embody a disk array, and our tests only simulated a RAID system storing both parity data and application data on one disk. In a future version of our system, the recovery code should also replace the recovery scheme in the disk array driver. However, as the mean time between failure in a single disk also increases, one might consider applying the integrated error management idea on only the communication system where parity data is generated only with respect to the network error model and then stored on disk to save the FEC encoding operation. Then other kinds of codes could be more suitable, and other codes should be tested for better performance, e.g., Turbo [46, 193] or Tornado [29, 96, 97] codes, and one might also look at adaptable schemes [86, 139], for example, giving priorities to packets or correcting only the most important data like the I-frames in MPEG or sending the FEC parity data in a separate stream.

The packet sizes are so far determined only on the recovery code's coding performance. However, the packet size also affects performance in other components like the network itself. For example, if we use a larger packet size than the MTU of the network, IP will split the transport level packet into smaller fragments each transmitted as one packet. If only one of the fragments is damaged or lost, the whole packet will be damaged because the recovery scheme is applied at the transport level. A further analysis on an optimal packet size should be performed, especially if we also look into new codes.

Earlier in this thesis, we claimed that because all concurrent users share one single data element and because the data rate in an MoD system is high, there is almost no gain in applying caching of data in the buffer cache. However, in the delay-minimized broadcasting scheme, the first partition(s) will be quite small, e.g., about 4.4 MB using the average DVD data rate of 3.5 Mbps and a startup delay of 10 seconds. As all partitions are broadcast in rounds, the data in the first partitions will be reused quite frequently, e.g., every tenth second, so in this case there can be a lot of disk accesses saved by pinning the data for the first partition(s) in memory (of course depending on the amount of memory and partition size).

In INSTANCE, we have looked at optimizing resource usage and performance in an MoD scenario. However, a hiccup free presentation of video and audio data requires a smooth data delivery, e.g., data must be delivered within a certain delay bound. To guarantee services and a perfect data playout, each stream must have guaranteed access to the required resources, e.g., by admission control on the number of concurrent streams (each stream is started by the information provider) and/or by scheduling algorithms supporting real-time tasks. This means that QoS support is important and that resources in the network must be guaranteed and reserved using protocols like the real-time protocol (RTP) and resource reservation protocol (RSVP). One interesting approach in this area is the real-time Linux operating system [201]. A real-time executive runs in parallel with the traditional Linux operating system. Real-time processes are executed by the real-time executive, and if no real-time processes are running, the Linux kernel will run as one of the real-time executive tasks. Thus, the system provides hard real-time capabilities [172] and may be a sound foundation for providing real-time guarantees. Another interesting approach is the DROPS operating system [74, 183] which is built to remedy the lack of resource reservation support in traditional operating systems. Resource managers are allowed to reserve CPU, memory, and driver level resources. Since DROPS also provides some performance enhancements such as a timing-aware streaming interface with a zero-copy application-to-network data path [18, 94], this system may provide a nice starting point for resource reservations and reservation enforcement.

When designing our current server, we assumed that the users started a video playout and watched it to the end without any interactions like pause, rewind, forward, or fast playout. Thus, the support for interaction is very limited, especially with the cautious harmonic broadcasting scheme. The client should be able to pause and immediately restart if the next broadcast partition continued to be received by the client and rewind if previous data is cached on the client side and not discarded after playout. However, arbitrary jumps forward and fast forward playout are not supported using the chosen broadcast protocol. To support a true-MoD application where each client can perform arbitrary VCR-operations, such support should be investigated where a possible approach could be to have set-aside resources (contingency channels) for clients performing interactions [146].

So far we have concentrated on origin servers, i.e., the main data server in the distributed network of serving machines, but our long-term goal is to provide large scale, like national-wide, continent-wide, and even world-wide, support for MoD services. Proxies will then be used to cache data closer to the client. However, in the current prototype, we have only optimized data retrieval. In the case of proxies, the data upload operation is also important, and future research could focus on an efficient upload protocol.

Going world-wide makes the network topology heterogeneous where a lot of different networks are used. The various subnetworks can for example differ in loss rates and bandwidth capabilities. We also want to support different kind of receivers or data sinks. Thus, transmitting all parity data and a full quality data stream to all clients regardless of underlying hardware capacity and client QoS requirement, may be wasteful. Therefore, we would also need some kind of adaptation in the data stream using quality layers, e.g., both in frame rate, color depth, resolution, and recovery capability. For example, if a client has an error free connection, no parity data should be transmitted, or if the network cannot manage the bandwidth of a full quality playout, the quality of the transmitted data should be reduced by dropping the upper quality layers of the stream [88]. This adaptation could be achieved by simply adding a priority level into the packets. All packets are then transmitted from the servers, but if a network cannot manage the transmitted data rate, the router discards a certain number of packets according to the priorities. Likewise, if the data sink cannot manage or the user does not want full quality playout, the data packets are dropped according to the priority levels.

6.5 Final Remarks

By performing experimental research on our prototype design in NetBSD, we demonstrated that the I/O performance in a multimedia server can be greatly improved by using simple techniques removing redundant functionality, in-memory copy operations, and pre-executing operations that are performed several times and are common for all data transfers. Our proposed mechanisms are not hardware dependent and should therefore also benefit from future technology developments.

Our critical review of the claims showed that the listed bottlenecks in the operating system can be eliminated, and compared to the performance of standard off-the-shelf hardware, we could send data faster through the operating system than the hardware devices could handle.

It is important that special, existing mechanisms for all subsystems are integrated in a single system to guarantee services and to optimize performance. We believe that our approach is a step towards such an integrated system using existing and new mechanisms. However, there are many aspects not addressed in this thesis, and there are many unsolved questions. Nevertheless, we think that our research results are a step in the right direction and should be a sound foundation for further research activities.

Appendix A

Detailed Testbed Description

In this appendix, we give further details about our test machine and describe the time measuring tool including complicating issues, implementation, data extraction from the kernel, and probe overhead.

A.1 Detailed Server Machine System Description

We have performed our experiments on a *Dell Precision WorkStation 620* (see Table A.1) which we have connected to another PC using an isolated, point-to-point Gbps network [176]. This machine (WorkStation 620) has an Intel 840i chipset [192], with two PCI buses, running an Intel PentiumIII 933 MHz processor. We had 256 MB RDRAM [195] and had one 9.1 GB, 10.000 RPM SCSI hard disk. The integrated dual-channel Ultra160 SCSI controller is able to provide 160 MBps throughput on the primary channel, i.e., the storage system performance is limited by the single hard disk. The disk controller was

Dell Precision WorkStation 620	
Processor	
Type	Dual Intel Pentium III Xeon 933 MHz with a front-side (memory) bus external speed of 133 MHz and internal math coprocessor
1st level cache	32 KB (16 KB data cache; 16 KB instruction cache)
2nd level cache	256 KB
System information	
Chipset	Intel 840i slot2 chipset [192]
PCI bus speeds	32-bit, 33.3 MHz *and* 64-bit bus running at 33.3 MHz and 66.6 MHz
Memory	
Type	PC800 error checking and correction RDRAM in the rambus in-line memory module slots
Size	256 MB
Storage system	
Disk	9.1 GB, 10.000 rounds-per-minute SCSI hard disk
PCI SCSI controller	Integrated dual-channel Adaptec 7899 Ultra 160/M LVD (160 MB/s)
Communication system	
Gbps network card	3Com EtherLink server network interface card (3C985B-SX) [176]
network card (for external connections)	Integrated 10/100 3Com Ethernet controller (3C920 based and 3C905-TX compatible)

Table A.1: Server machine (DELL Precision WorkStation 620) system description [181].

connected to the 32-bit, 33.3 MHz PCI bus whereas the network card was connected to the 64-bit, 66.6 MHz PCI bus.

A.2 Time Measurement Tool

For measuring time, both in kernel and user space, we have implemented a software probe. This probe reads the processor cycle count in the time-stamp counter register, and returns the current cycle count number since last register reset or machine reboot. The probe is implemented using the Intel `RDTSC` instruction, described in Table A.2, and is used in a similar way as described in [187]. Below, we discuss some issues affecting the cycle count, and we present our software probe and quantify the probe overhead.

Instruction	Description
`RDTSC`	The `RDTSC` instruction [188, 189, 190], opcode 0F 31, was introduced with the Pentium Processors. This instruction allows the programmer to access the time-stamp counter which keeps an accurate count of every cycle that occurs on the CPU. It is a 64-bit model specific counter that is incremented on every clock cycle. `RDTSC` loads the 32 high-order bits of the cycle count register into the EDX register and the 32 low-order bits into the EAX register.
`CPUID`	The `CPUID` instruction [188, 189, 190], opcode 0F A2, was introduced with the Pentium Processors. This instruction serves primarily to identify processors providing processor identification information in the EAX, EBX, ECX, and EDX registers. Additionally, and more important in our context, the `CPUID` has another function (or side-effect). `CPUID` works as a serializing instruction, which means that it flushes the pipeline and waits for all pending operations to finish before proceeding. This is useful when using the `RDTSC` instruction for testing purposes on a PentiumII or Pentium Pro processor (or higher).

Table A.2: Used instructions (supported on Intel-based machines) in our time measurement tool.

A.2.1 Issues Affecting the Cycle Count

There are several issues affecting the measured cycle count for a set of instructions. In the following subsections, we briefly discuss some of them, i.e., out-of-order execution, influence of caching, and counter overflow.

A.2.1.1 Out-of-Order Execution

The first issue affecting the cycle count is that processors starting with Pentium Pro, Pentium II, and above support out-of-order execution where instructions are not necessarily performed in the order that they appear in the source code. This is an important issue when counting the number of cycles spent on executing (parts of) a program, i.e., the `RDTSC` instruction is not a serializing instruction, and it does not necessarily wait until all previous instructions have been executed before reading the counter. Likewise, subsequent instructions may begin execution before the counter is read. This means that on these processors, we might get a misleading cycle count. In order to keep the `RDTSC` instruction from being performed out-of-order, a serializing function is required. We therefore use the `CPUID` instruction (see Table A.2) in our software probe forcing every preceding instruction in the code to complete before allowing the program to continue.

When using the `CPUID` instruction, the programmer must take into account the cycles it takes for the instruction to complete. However, the time to execute `CPUID` varies according to the instructions in the CPU pipeline when the instruction is called. In our performance measurements, we will use the probe several places giving different kinds of pending instructions. Therefore, with regard to execution time, every operation before execution of the `CPUID` can cause a change in execution time, and it will never stabilize unless the code stream is exactly the same between two successive executions of the `CPUID` instruction. In many of the experiments we perform on our system, two successive runs will probably not be the same, because we measure large operations retrieving data from disk and sending it to the network. These operations generate their own kernel threads, i.e., the instructions to complete before executing our probe will vary according to the scheduling of the threads and where in the source code the probe is placed.

Our choice of using `CPUID`, despite it's varying execution time (probe overhead), is based on the following observations:

- In our system, we perform large operations including disk, memory, and network operations. This means that the varying number of cycles spent executing `CPUID` will be minimal compared to the large number of cycles used to perform the code instructions. Furthermore, as described below in Section A.2.2, the cycle variation is relatively small compared to the average `CPUID` execution time.
- The pending instructions in the CPU pipeline will (usually, unless we have a context switch) be part of the instruction set we wish to test in the measurement.
- When not using `CPUID`, we risk that we measure wrong instructions and that we miss some important pieces of code in our measurements.

In our performance tests, we would like to get as accurate measurements as possible, so we have chosen to use `CPUID` to get the source code serialized, i.e., all preceding operations will be finish before reading the cycle count the first time and all our instructions will be executed before reading it the second time. This will be at the cost of an inaccurate probe overhead, but the variance is too small to affect the results in any great extent (see Section A.2.4).

A.2.1.2 Caching

When measuring the same piece of code several time, different results may occur. This may be due to the influence of caching in the system. In our system, we retrieve data from disk and transmit it to the network. Thus, using the traditional data path through the file system, this is an important issue, because the file system keeps data in the buffer cache[1]. However, we have performed tests using very large files, so all the used data should be paged out of the cache before it is reused. Nevertheless, before each test, we have "flushed" the cache by reading another large file. Furthermore, using our new streaming system and the MMBUF mechanism, we do not cache data in memory, because we think the benefits of caching in our system might be minimized by high bandwidth streams and by the fact that concurrent clients should be served by a multicast transmission and not retrieve data individually from disk. In a similar way, the caching of code instructions affect the measured cycle count. The first time the code instructions (or data) are brought into the cache (or even into memory from disk), a large number of cycles are used.

In order to use `RDTSC` effectively and later analyze the measured cycle count in a correct manner, these caching effects must be taken into account. As mentioned above, the caching of data will probably not be an issue in our measurements, but to be certain, we "flush" the buffer cache between consecutive runs. The caching of code instructions is probably more important. However, as the data rates are high, each piece of code will be executed a large number of times in a loop. Thus, the cycle count average will take into account these varying measurements. Finally, each test is also performed several times to get an accurate result.

A.2.1.3 Counter Overflow

As mentioned above, the `RDTSC` counter is divided into a lower and upper 32-bit counter. On fast processors, the 32 low-order bits of the cycle counter may overflow, i.e., on our 933 MHz machine it

[1]The size of the buffer cache is an important issue. By default NetBSD used 10% of the first 2 MB of memory and 5% of the remaining memory for bufpages, and the buffers are replaced in an least recently used fashion.

will exceed the 32-bit value every 2^{32} cycles / (933×10^6 cycles/second) = 4.6 second. Because our measurement intervals are typically greater 4.6 seconds, we must use the whole 64-bit value, and since our system supports 64-bit values, we read the whole cycle count into one `unsigned long long` variable[2].

A.2.2 Software Probe

This section describes the software probe we implemented for reading the processor clock cycle count register. The probe data is placed in a structure as shown in Figure A.1 and includes the cycle count, the probe location, and a probe id. The probe data structure should contain a minimum of data to reduce the amount of needed memory holding the probe data in kernel space, and the probe function is kept as simple as possible to avoid a high probe execution overhead (see Section A.2.4) which will influence the performance measurement results too much.

```
struct instance_probe {
  u_int64_t cycle_count; /* The processor cycle count read by RDTSC */
  int location;          /* The location in the code when we read the cycle count register */
  int id;                /* The id of the probe, e.g., packet number */
}
```

Figure A.1: The `instance_probe` structure.

The implementation of the probe function, called `probe_entry()`, is shown in Figure A.2. First, we lock the probe mechanism by changing the system interrupt priority level. We store the location and id of the probe, and increase the index of array storing the probe data. Then, we execute the `CPUID` instruction for serializing, i.e., to avoid out-of-order execution effects where previous instructions have not been executed or where subsequent instructions may begin execution before the counter is read. When all pending instructions have finished, we execute the `RDTSC` instruction, and the whole 64-bit value is read into a `u_int64_t` (`unsigned long long`) variable as described in Section A.2.1.3. Finally, the system interrupt priority level is reset.

A.2.3 Probe Data Extraction

We modified and used the `__sysctl()` system call to extract the probe data from the kernel. The probe data array is copied out as a raw data and dumped to a file on disk. To analyze and present the data in a more readable format, we implemented a program presenting the measurements in MATLAB format. In addition to just retrieving probe data from the kernel, we added functionality in `__sysctl()` to quantify the probe execution overhead, to clear the probe ids, and to see the number of measurements currently made in the kernel. The latter is to see if the probe data array is overloaded, i.e., the size of the data from the measurements in a test exceed the size of memory allocated in the kernel, and whether the probe data must be extracted while the test is executing.

A.2.4 Probe Execution Overhead

A software probe used for measurements will also require some resources, and our probe consumes some CPU cycles serializing the operations, retrieving the clock cycle counter, and managing the probe identification information. The probe execution time is overhead and should be subtracted from the recorded number of cycles. Therefore, we have performed tests to determine the probe costs. However,

[2] Reading the whole cycle count can be done in two different ways, because `RDTSC` place the 64-bit cycle counter into two different registers. This means that we can either read each of these registers into different `unsigned long` variables, i.e., supporting only 32-bit values, or read the whole 64-bit value into a `unsigned long long` variable, i.e., supporting 64-bit values.

```
struct instance_probe iprobe[440000];        /* Array of probes holding measurement data */

void
probe_entry(location, id)
     int location;
     int id;
{
  int index, pl;

  pl = splimp();                             /* Change the system interrupt priority level */

  index = probe_index++;                     /* Index into the probe array to be used */
  probe_index = probe_index%INSTANCE_PSIZE;  /* Update the "ring buffer" index for the next
                                              * measurement,i.e., do not exceed the
                                              * INSTANCE_PSIZE size */

  iprobe[index].location = location;         /* Store the location of the probe */
  iprobe[index].id       = id;               /* Store the id of the probe */
  __asm __volatile(".byte 0x0f,0xa2");                                 /* CPUID */
  __asm __volatile(".byte 0x0f,0x31" : "=A" (iprobe[index].cycle_count));     /* RDTSC */

  splx(pl);                                  /* Reset the system interrupt priority level */
  return;
}
```

Figure A.2: The `probe_entry()` function.

as described in Section A.2.1.1, the `CPUID` execution time varies according to the preceding operations, and it will not stabilize before we have made a few consecutive calls. According to [187], the best way to measure the overhead is "to call the instruction three times, measure the elapsed time on the third call, then subtract this measurement from all future measurements". Thus, to follow Intel's instructions, we have measured the probe execution time as shown in Figure A.3.

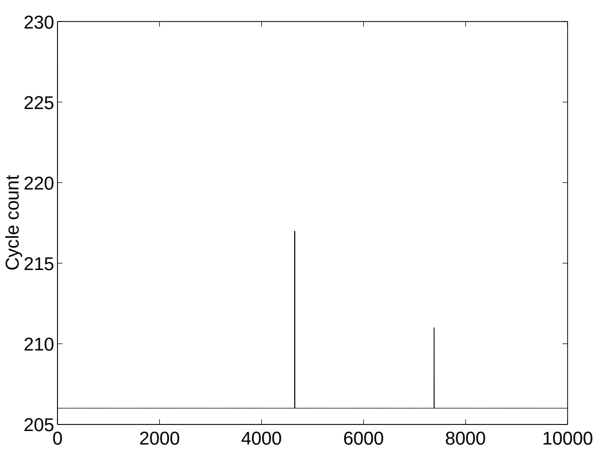
```
for(i = 0 ; i < 10000 ; i++) {
  probe_entry(0, 1);    /* Execute probe 1 */
  probe_entry(0, 2);    /* Execute probe 2 */
  probe_entry(0, 3);    /* Execute probe 3 */
  probe_entry(0, 4);    /* Execute probe 4 */

  /* Calculate overhead by subtracting cycle count in probe 4 from cycle count in probe 3 */
  overhead[i] = iprobe[probe4].cycle_count - iprobe[probe3].cycle_count;
}
```

Figure A.3: Pseudo code for testing the execution overhead of the probe.

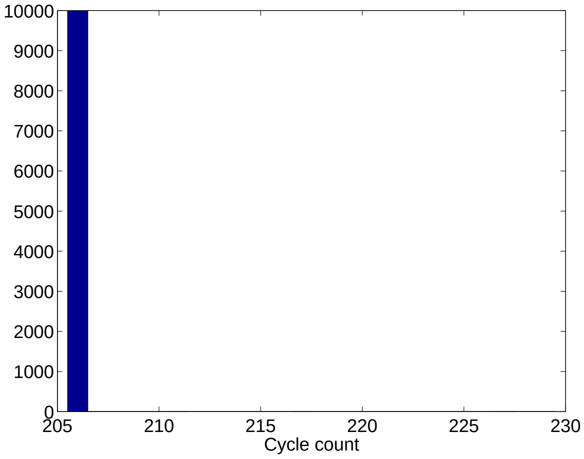
Furthermore, as we use probes in both kernel and user space, we have implemented a software probe for use in kernel space and a software probe for use in user space, therefore we do not have any problems accessing memory in either address space. However, the execution time may then vary slightly, and we have tested the two versions separately. For each overhead measurement, we have collected 10000 values.

Our tests indicate an overhead of *206 cycles*, *227 cycles*, and *206 cycles* which is average, maximum and minimum overhead, respectively, when running the probe in the kernel (see Figure A.4A-B and Table A.3). The results from the probe overhead experiments in user space are shown in Figure A.4C-D and Table A.3. The test indicates an overhead of *273 cycles*, *382 cycles*, and *265 cycles* which is average, maximum and minimum overhead, respectively.

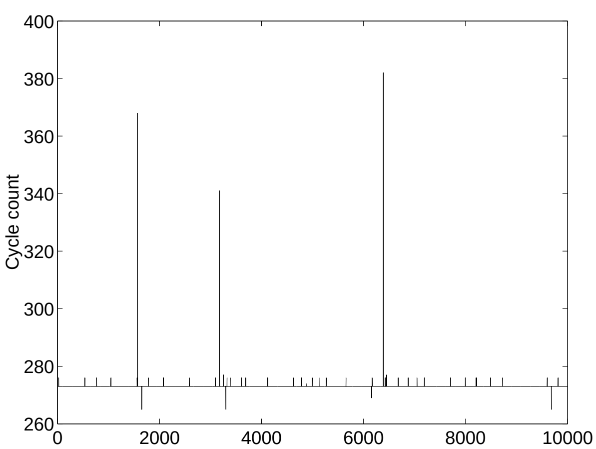
In the performance measurements described in this thesis, we have used a probe overhead of *206 cycles* when the probe is executed in the kernel and of *273 cycles* when the probe is executed in user space, because this is the average value in both our probe execution overhead measurements. This corresponds to about 0.22 μs and 0.29 μs on our 933 MHz test machine, respectively. Furthermore, the



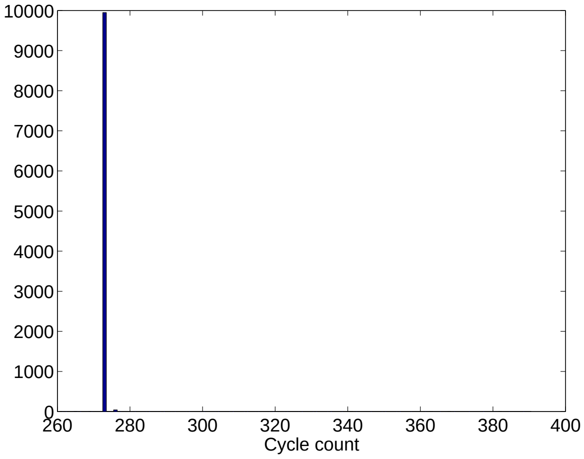
(A) Plot of a probe overhead test (kernel space).



(B) Histogram over the probe overhead test (kernel space).



(C) Plot of a probe overhead test (user space).



(D) Histogram over the probe overhead test (user space).

Figure A.4: Results from one of the probe overhead experiments (with 10,000 iterations).

	max	**min**	**avg**	**stdev**	**99% ci**	**95% ci**
Kernel space	227	206	206.00	0.24	[206 - 206]	—
User space	382	265	273.05	1.85	[273 - 273]	—

Table A.3: Results from one of the probe overhead experiments.

average value is equal to both the lower and upper limits of the 99% confidence interval. In most of the performance experiments in this thesis, however, the overhead variance is negligible, because most of the tests will consume a lot more CPU cycles than the probe itself. Only in the basic performance tests described in Appendix B, the variance in the probe overhead might have some small influence on the results, because we measure operations that do not last very long, i.e., making the probe overhead a considerable part of the total measured time (up to about 50%) of the test. Nevertheless, each performance test is run several times (10000) in all experiments, and the average value should therefore in any test give a good overall result.

Appendix B

Basic Performance Tests

This chapter describes the results from some basic performance experiments we performed to see the overhead in traditional operating system for several operations. Several of these tests have been performed earlier and presented in various papers by others, but we wanted to see if the traditional bottlenecks still represent a degrading factor on the overall system performance. The tests have been performed using the software probe described in Section A.2.

B.1 Disk Efficiency Versus Amount of Data Read per I/O Operation

The performance of the storage (disk) system depends on several parameters, and performance of a read (and write) operation is highly dependent on the disk characteristics. The disk throughput is dependent on the disk arm movement speed (seek time), the spindle speed (rotational latency and data transfer speed), the size of the transfer request, the scheduling of the requests, etc. Furthermore, the request model is also important, and in our high-data rate MoD system a lot of data is sent to remote clients. In this section, we describe our evaluation of disk performance versus the I/O operation size.

To predict the disk behavior, we used a very simple analytic model calculating the disk throughput, disk efficiency, and number of possible concurrent users in a disk array when varying the block size. We assumed a Seagate Cheetah 36LP (ST336704LC) disk drive [202]. The disk transfer rate, disk transfer time, and disk throughput are given by equation B.1, B.2, and B.3, respectively:

$$
\begin{aligned}
transfer\_rate &= \frac{data\_per\_track}{rotation\_speed} && \text{(B.1)}\\
transfer\_time &= seek\_time + rotational\_latency + \frac{data\_size}{transfer\_rate} && \text{(B.2)}\\
achieved\_disk\_throughput &= transfers\_per\_second \times data\_size \\
&= \frac{1s}{transfer\_time} \times data\_size && \text{(B.3)}
\end{aligned}
$$

The transfer rate can typically vary according to where on the disk drive data is stored, because the outermost tracks often contain more data than innermost tracks. However, we assume that we do not have a zoned disk, we have an equal amount of data in all tracks, and all requests are of equal size. Nevertheless, in our calculations, we have used the average formated transfer rate (35.5 MBps) given by Seagate [202]. The data transfer time is also dependent on seek time and rotational speed, and in our calculations, we used the average seek time (5.20 ms) and rotational latency (2.99 ms) of the Cheetah disk, but in practice these values will vary depending on where the disk arm and disk head are located.

The achieved disk efficiency in percent compared to the maximum disk transfer rate is given by equation B.4:

$$disk\_efficiency = \frac{achieved\_disk\_throughput}{maximum\_disk\_throughput} \times 100 \tag{B.4}$$

Furthermore, the number of possible concurrent users depends on the data rate each client is asking for and how much data is read per I/O request. In our scenario, we assume that all video clips are stored at the same playout rate (6 Mbps), and one stripe of data (eight disks) is read per request. Thus, the playout duration per I/O request and the I/O request frequency per client is given by equation B.5:

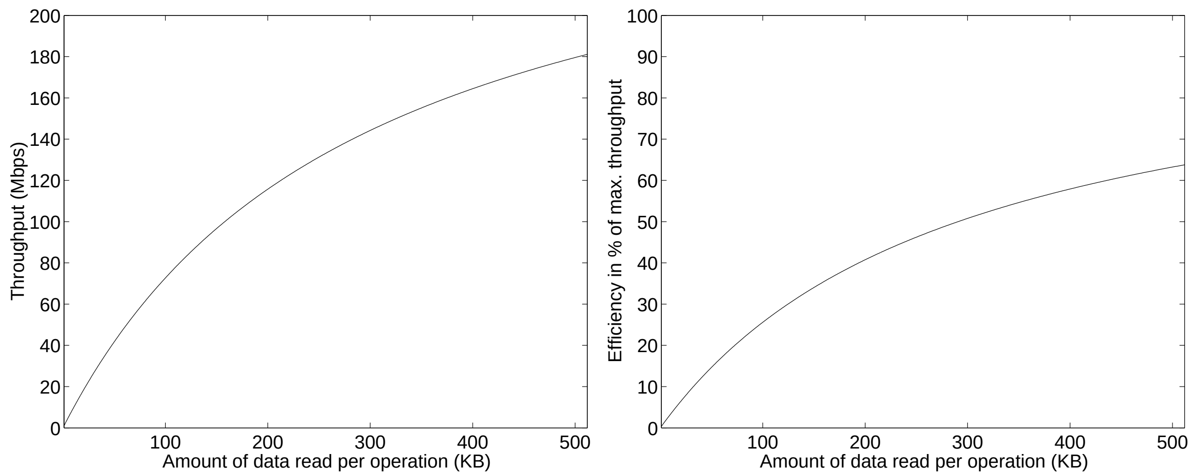
$$request\_frequency = \frac{data\_size \times number\_of\_disks\_containing\_information\_data}{playout\_rate} \tag{B.5}$$

The average transfer rate from the disk array is given by a slightly modified version of equation B.1 multiplying the transfer rate with the number of disks (the maximum data rate is however limited by the controller and bus bandwidth). The achieved seek time and rotational latency will usually increase as the number of parallel disks increase, but we still assume that the average numbers per disk can be used. The number of possible concurrent users is then given by equation B.6:

$$number\_of\_concurrent\_users = request\_frequency \times transfers\_per\_second \tag{B.6}$$

where transfers per second is calculated using the transfer time in equation B.2.

By using this simple analytic model and the average disk characteristics of the Seagate Cheetah disk, we present the relationship between disk efficiency and number of consecutive data reads per I/O operation in Figure B.1. Our results show that a few large requests usually perform better than many small requests, because more requests result in more disk seeking and rotation latencies. In addition, if a disk read spans several block, the request is traditionally broken into pieces such that each piece fits within the disk block, and each piece is then scheduled to fetch the block to be read [157]. However, improvements are available in some more modern systems where I/O requests for several contiguous blocks on disk are clustered together making a single large I/O operation on the entire range of sequential blocks [100]. Each disk request also introduces interrupt overhead to the system in addition to decreasing the disk throughput by increasing the number of seeks and rotation delays. Nevertheless, disk seek and rotation latencies are the main bottlenecks in disk I/O operations. Therefore, the larger the amount of data read per operation, the higher the throughput. In other words, increasing disk block size increases performance. Consequently, to reduce the overhead and increase the disk throughput, we should use a large disk block or make large reads if contiguous blocks are read in one operation.



(A) Disk throughput versus amount of data read per operation

(B) Disk efficiency versus amount of data read per operation

Figure B.1: Disk block size versus disk efficiency.

In our analytic disk efficiency calculation, we have tested disk blocks larger than 64 KB. However, in many operating systems, e.g., NetBSD, the maximum disk block size is 64 KB (by default 4 KB).

Theoretically, this value can be changed, but one reason for this limit is that (most) disk controllers use 16-bit registers, and one block is therefore not able to hold more data than 2^{16} bytes (64 KB). Consequently, we use in our prototype as large a disk block size as possible, i.e., 64 KB. This choice is also in accordance with other research in this area where read requests of up to 64 KB (and even above) are proposed for high data rate multimedia systems [135, 146].

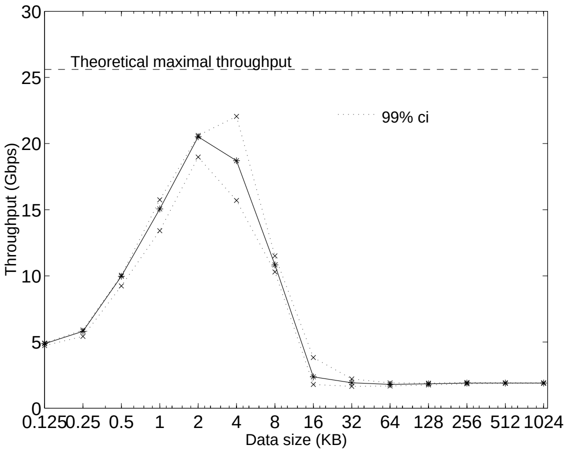
B.2 Data Copy Performance

In Section 3.2.1, we described the traditional I/O-pipeline in UNIX-based operating systems. Common user-level applications access the file and network I/O system services for example through the `read()`/`recv()` and `write()`/`send()` system calls for data transfers, respectively. A major bottleneck in high throughput systems is that these system calls copy data from the kernel memory region to the application buffer in user space, and vice versa. This is expensive for various reasons [42]:

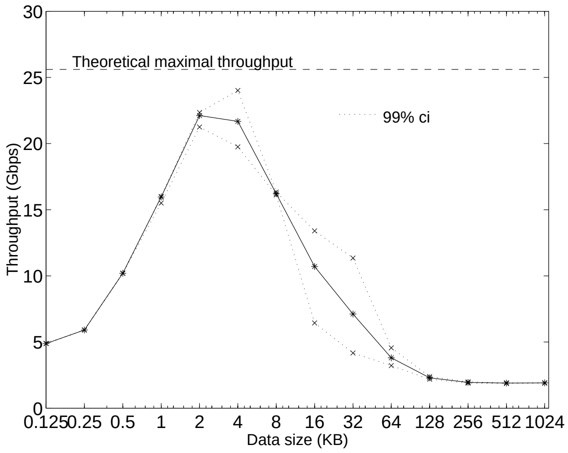
- The bandwidth of the main memory is limited, and every copy operation is effected by this.
- A lot of CPU cycles are consumed for every copy operation. Often, the CPU must move the data word-by-word from the source buffer to the destination, i.e, all the data may flow through the CPU. This means that the CPU will unavailable during the copy operation.
- Data copy operations affect the cache. Since the CPU accesses main memory through the cache, useful information resident in the cache before the copy operation is flushed out with the data being copied.

In NetBSD, the transfer of data between user and kernel space is done by the `copyout()` and `copyin()` functions in the operating system kernel. To copy data within the kernel, one might use the `memcpy()` function. To see the available copy throughput using these functions on our test machine, we implemented a simple system call which only copies data from user space to the kernel, copies the data inside the kernel from one buffer to another, and finally, copies the new buffer out to user space. A test program made 10000 calls, and the results are presented in Figure B.2, Table B.1, and Table B.2. In the experiments, we tested different data sizes, and since these operations consume CPU cycles, we made tests with and without interrupts, i.e., when interrupts are turned off the code is executed as an atomic instruction which is not suspended due to another task. The test was performed with no other load than the system itself listening on the network and performing basic tasks within the system.

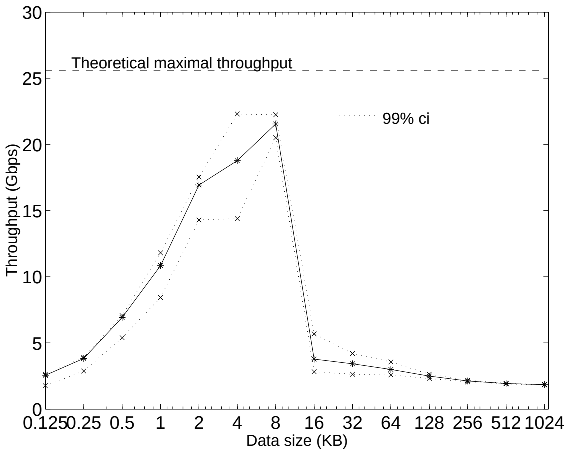
Theoretically, using PC800 RDRAM and the dual memory channels on the 840 chipset, each of which can support 12.8 Gbps of memory bandwidth, we can pump data to and from memory at 25.6 Gbps [195]. The measured average copy throughput is shown in Figure B.2 where we also have plotted the 99% confidence interval. The throughput varies according to the size of the data element being transferred between the buffers with best performance when the size is between 1 KB and 8 KB. Within this interval, we experienced a throughput well above 20 Gbps, and for some data sizes, we experience a measured throughput close to the theoretical maximum. For data sizes below 1 KB and above 8 KB, the performance is reduced drastically towards a throughput of below 2 Gbps. Table B.1 additionally present various statistics on the throughput measurements. If we compare the results from the tests with and without interrupts, we see a small degradation when allowing other tasks to be executed on the CPU. Thus, even on this minimal load, other tasks that interrupt the copy operation decreasing the perceptual performance. A heavily loaded server will, because the copy operations are CPU dependent, have large, indeterministic delays when data must be copied within the system before data is transmitted onto the network. Moreover, if we look at the traditional server approach where both `copyout()` and `copyin()` are performed, the effective throughput is approximately halved. Even if data only were to be copied from the file system to the communication system for example using the `memcpy()` function, the copy overhead would be substantial. The times to copy a data element using the tested functions is



(A) `copyin()`.



(B) `copyout()`.



(C) `memcpy()`.

Figure B.2: Average throughput for the copy operations depending on data size in Gbps.

presented in Table B.2. If data copying is performed, the times for the `copyout()` and `copyin()` functions must be added to the total execution time for a specific request in the traditional case. If data is only copied within the kernel, only the time for the `memcpy()` function should be added.

The exact reason for the large peek between 1 KB and 8 KB compared to the other measured data sizes is yet somewhat unclear. The performance of the copy operations increases up to data sizes of 4 KB and 8 KB, and then the performance drops. The results are not unique to our experiments, but are also confirmed by two other independent measurements described in [81] (NetBSD 1.4.1) and [147, 148] (OpenBSD). There might be a lot of reasons for these results, e.g., paging, context switches, translation-lookaside buffer and cache flushes, page limits and buffer alignment, etc. The increase is similar to the tests performed by Intel in [191], but they do not have any tests for larger sizes than 4 KB. The question is therefore why the performance drops using larger sizes than 8 KB? In our tests, the buffers used for `copyout()` and `copyin()` are in separate address spaces (user and kernel space), and the buffers used for `memcpy()` are both in kernel space, i.e., our tests include buffers separated in memory and buffers close in memory. Furthermore, we tested the buffers in kernel space using page alignment and without page alignment. In all experiments, the results are similar – copy operations perform best using sizes around 4 KB and 8 KB. However, one highly probable explanation is the cache. Our test machine have a 16 KB level one data cache which means that we will have caching effects using sizes above this limit. In [45], copy performance tests are performed using transfer sizes from 4 KB to 4 MB. The results show that the copy performance decreases as the transfer size increases due to caching effects. Thus, the cache size is a plausible explanation. However, to be certain, more tests are required on other machines having smaller and larger cache sizes.

In summary, our results demonstrate that data copy operations still represent a potential bottleneck of the disk-to-network data path, and as the CPU is involved in the copy operations, the perceptual copy performance will be even further decreased on heavily loaded machines. If possible, such operations therefore should be omitted making a zero-copy data path a good solution for high throughput systems.

Table B.1: Statistics on copy operation performance in Gbps.

Operation	Interrupts	Data size	max	min	avg	stdev	99% ci	95% ci
`copyin`	**off**	**0.125 KB**	4.92	2.51	4.87	0.09	[4.73 - 4.92]	[4.78 - 4.92]
		0.25 KB	5.91	3.22	5.81	0.13	[5.43 - 5.91]	[5.46 - 5.91]
		0.5 KB	10.05	1.13	9.97	0.23	[9.24 - 10.05]	[9.44 - 10.05]
		1 KB	15.92	3.72	15.06	0.64	[13.41 - 15.75]	[13.66 - 15.75]
		2 KB	21.64	6.06	20.51	0.79	[18.98 - 21.60]	[19.21 - 21.41]
		4 KB	22.54	1.94	18.70	1.83	[15.70 - 22.05]	[16.04 - 21.70]
		8 KB	12.97	0.72	10.83	0.30	[10.30 - 11.51]	[10.42 - 11.31]
		16 KB	7.23	0.02	2.37	0.36	[1.79 - 3.83]	[1.89 - 3.15]
		32 KB	6.75	0.03	1.92	0.14	[1.65 - 2.21]	[1.70 - 2.17]
		64 KB	6.36	0.07	1.80	0.08	[1.68 - 1.91]	[1.73 - 1.89]
		128 KB	3.90	0.13	1.84	0.06	[1.76 - 1.89]	[1.79 - 1.88]
		256 KB	2.55	0.25	1.90	0.06	[1.84 - 1.94]	[1.86 - 1.94]
		512 KB	2.25	0.44	1.90	0.06	[1.88 - 1.92]	[1.89 - 1.92]
		1024 KB	2.05	0.71	1.90	0.08	[1.88 - 1.92]	[1.90 - 1.91]
	on	**0.125 KB**	4.71	0.19	4.68	0.08	[4.61 - 4.71]	-
		0.25 KB	5.80	0.17	5.73	0.13	[5.41 - 5.80]	-
		0.5 KB	10.03	0.57	9.65	0.44	[8.49 - 10.03]	[8.66 - 10.03]
		1 KB	15.34	5.18	14.34	0.75	[12.53 - 15.34]	[12.90 - 15.28]
		2 KB	20.00	1.83	16.06	2.26	[14.07 - 19.77]	[14.27 - 19.56]
		4 KB	22.51	3.46	18.73	1.88	[15.91 -22.26]	[15.96 - 21.87]
		8 KB	12.65	3.24	10.97	0.30	[10.44 - 11.63]	[10.58 - 11.43]
		16 KB	7.26	0.02	2.00	0.27	[1.58 - 2.81]	[1.66 - 2.62]
		32 KB	6.38	0.04	1.71	0.11	[1.53 - 2.01]	[1.56 - 1.93]
		64 KB	5.83	0.07	1.68	0.07	[1.56 - 1.81]	[1.58 - 1.79]
		128 KB	3.10	0.13	1.74	0.05	[1.65 - 1.81]	[1.68 - 1.80]
		256 KB	1.98	0.25	1.76	0.06	[1.71 - 1.81]	[1.73 - 1.80]
		512 KB	1.80	0.69	1.73	0.07	[1.71 - 1.75]	[1.73 - 1.74]
		1024 KB	1.87	0.70	1.80	0.07	[1.78 - 1.81]	[1.79 - 1.81]
`copyout`	**off**	**0.125 KB**	4.89	0.19	4.89	0.05	[4.89 - 4.89]	-
		0.25 KB	5.91	0.13	5.91	0.08	[5.91 - 5.91]	-
		0.5 KB	10.20	0.77	10.20	0.11	[10.20 - 10.20]	-
		1 KB	16.00	0.49	15.96	0.22	[15.51 - 16.00]	[15.78 - 16.00]
		2 KB	22.35	9.00	22.12	0.32	[21.25 - 22.35]	[21.51 - 22.35]
		4 KB	24.38	0.30	21.68	1.30	[19.75 - 24.01]	[19.75 - 23.79]
		8 KB	16.44	3.46	16.21	0.21	[16.12 - 16.34]	[16.16 - 16.33]
		16 KB	13.47	2.66	10.71	1.82	[6.44 - 13.40]	[7.20 - 13.29]
		32 KB	12.20	2.86	7.12	1.36	[4.17 - 11.35]	[4.77 - 10.32]
		64 KB	5.49	2.50	3.81	0.29	[3.22 - 4.56]	[3.31 - 4.47]
		128 KB	2.54	0.13	2.30	0.06	[2.19 - 2.39]	[2.22 - 2.37]

continues on next page

continued from previous page

Statistics on copy operation performance in Gbps.								
Operation	**Interrupts**	**Data size**	**max**	**min**	**avg**	**stdev**	**99% ci**	**95% ci**
		256 KB	2.01	0.25	1.95	0.06	[1.89 - 1.99]	[1.91 - 1.99]
		512 KB	1.93	0.44	1.90	0.07	[1.87 - 1.92]	[1.89 - 1.91]
		1024 KB	1.99	0.71	1.91	0.08	[1.89 - 1.92]	[1.90 - 1.92]
	on	**0.125 KB**	4.86	4.43	4.81	0.01	[4.81 - 4.81]	-
		0.25 KB	5.85	1.68	5.85	0.06	[5.70 - 5.85]	[5.85 - 5.85]
		0.5 KB	10.11	4.04	10.11	0.06	[10.11 - 10.11]	-
		1 KB	15.89	1.21	15.89	0.22	[15.89 - 15.89]	-
		2 KB	22.07	7.47	21.97	0.15	[21.94 - 21.97]	[21.97 - 21.97]
		4 KB	24.40	1.67	21.72	1.37	[19.58 - 24.15]	[19.62 - 23.97]
		8 KB	16.44	4.15	16.20	0.19	[16.12 - 16.35]	[16.13 - 16.34]
		16 KB	13.32	2.93	9.71	2.26	[4.40 - 13.15]	[5.11 - 12.93]
		32 KB	12.23	3.15	7.10	1.41	[4.73 - 11.50]	[5.27 - 10.95]
		64 KB	4.00	2.06	3.58	0.25	[3.00 - 4.22]	[3.10 - 4.07]
		128 KB	2.27	0.13	2.09	0.07	[1.96 - 2.22]	[1.99 - 2.20]
		256 KB	1.88	0.25	1.82	0.06	[1.75 - 1.87]	[1.77 - 1.86]
		512 KB	1.81	0.69	1.74	0.07	[1.71 - 1.75]	[1.73 - 1.75]
		1024 KB	1.86	0.70	1.77	0.07	[1.02 - 1.78]	[1.77 - 1.78]
`memcpy`	**off**	**0.125 KB**	2.64	0.06	2.55	0.10	[1.76 - 2.63]	[2.52 - 2.63]
		0.25 KB	3.92	2.08	3.82	0.15	[2.87 - 3.90]	[3.43 - 3.90]
		0.5 KB	7.09	3.89	6.92	0.19	[5.39 - 7.06]	[6.63 - 7.06]
		1 KB	11.88	1.13	10.84	0.92	[8.42 - 11.80]	[9.44 - 11.75]
		2 KB	17.80	0.89	16.93	0.51	[14.29 - 17.53]	[16.25 - 17.38]
		4 KB	22.67	2.64	18.77	2.14	[14.39 - 22.30]	[15.56 - 22.07]
		8 KB	22.53	2.54	21.54	0.42	[20.50 - 22.24]	[21.33 - 21.90]
		16 KB	7.11	1.09	3.77	0.52	[2.82 - 5.68]	[2.96 - 4.93]
		32 KB	7.07	0.04	3.42	0.33	[2.63 - 4.19]	[2.78 - 4.00]
		64 KB	6.82	0.07	2.99	0.21	[2.58 - 3.56]	[2.64 - 3.48]
		128 KB	3.22	0.13	2.49	0.08	[2.31 - 2.63]	[2.36 - 2.61]
		256 KB	2.32	0.25	2.12	0.06	[2.05 - 2.17]	[2.07 - 2.16]
		512 KB	1.95	0.44	1.93	0.07	[1.90 - 1.94]	[1.92 - 1.94]
		1024 KB	1.91	0.70	1.84	0.07	[1.83 - 1.86]	[1.84 - 1.85]
	on	**0.125 KB**	3.36	0.06	3.33	0.07	[3.21 - 3.36]	-
		0.25 KB	4.67	0.31	4.58	0.10	[4.44 - 4.67]	[4.45 - 4.67]
		0.5 KB	8.30	0.26	8.04	0.30	[7.29 - 8.30]	[7.40 - 8.30]
		1 KB	13.56	1.36	12.58	0.93	[11.02 - 13.56]	[11.21 - 13.56]
		2 KB	18.35	1.93	16.62	1.46	[14.08 - 18.14]	[14.35 - 18.02]
		4 KB	24.09	0.38	20.04	2.48	[16.06 - 23.87]	[16.17 - 23.63]
		8 KB	23.08	7.23	22.03	0.25	[21.67 - 23.03]	[21.88 - 22.41]
		16 KB	6.34	1.64	2.85	0.33	[2.29 - 4.09]	[2.36 - 3.59]
		32 KB	6.61	1.75	3.05	0.27	[2.55 - 3.79]	[2.65 - 3.66]
		64 KB	3.90	0.07	2.77	0.14	[2.46 - 3.10]	[2.52 - 3.04]
		128 KB	2.65	0.13	2.29	0.09	[2.12 - 2.46]	[2.16 - 2.43]
		256 KB	2.11	0.25	1.91	0.06	[1.85 - 1.96]	[1.86 - 1.95]
		512 KB	1.74	0.69	1.73	0.07	[1.71 - 1.74]	[1.72 - 1.74]
		1024 KB	1.83	0.70	1.75	0.07	[1.73 - 1.76]	[1.75 - 1.76]

Table B.2: Statistics on copy operation performance in μs.

Operation	Interrupts	Data size	max	min	avg	stdev	99% ci	95% ci
`copyin`	**off**	**0.125 KB**	0.38	0.19	0.20	0.00	[0.19 - 0.20]	-
		0.25 KB	0.59	0.32	0.33	0.00	[0.32 - 0.35]	-
		0.5 KB	3.39	0.38	0.38	0.00	[0.38 - 0.41]	[0.38 - 0.40]
		1 KB	2.05	0.48	0.51	0.00	[0.48 - 0.57]	[0.48 - 0.56]
		2 KB	2.52	0.71	0.75	0.00	[0.71 - 0.80]	[0.71 - 0.79]
		4 KB	15.74	1.35	1.65	0.00	[1.38 - 1.94]	[1.41 - 1.90]
		8 KB	84.21	4.70	5.65	0.01	[5.30 - 5.93]	[5.40 - 5.86]
		16 KB	6937.41	16.87	53.18	0.60	[31.85 - 68.09]	[38.71 - 64.52]
		32 KB	7021.08	36.17	128.48	0.60	[110.51 - 148.22]	[112.65 - 143.95]
		64 KB	7162.29	76.75	274.33	1.33	[256.13 - 290.55]	[258.77 - 281.97]
		128 KB	7422.10	250.28	535.90	1.57	[515.93 - 556.35]	[518.92 - 546.67]
		256 KB	7927.02	765.16	1036.34	1.88	[1004.46 - 1060.42]	[1008.94 - 1049.74]
		512 KB	8940.14	1735.35	2063.40	2.51	[2035.19 - 2082.09]	[2039.59 - 2067.32]
		1024 KB	10996.58	3804.18	4128.02	3.93	[4076.48 - 4155.87]	[4082.05 - 4114.62]
	on	**0.125 KB**	5.06	0.20	0.20	0.00	[0.20 - 0.21]	-
		0.25 KB	11.35	0.33	0.33	0.00	[0.33 - 0.35]	-
		0.5 KB	6.69	0.38	0.40	0.00	[0.38 - 0.45]	[0.38 - 0.44]
		1 KB	1.47	0.50	0.53	0.00	[0.50 - 0.61]	[0.50 - 0.59]
		2 KB	8.33	0.76	0.97	0.00	[0.77 - 1.08]	[0.78 - 1.07]
		4 KB	8.83	1.36	1.65	0.00	[1.37 - 1.92]	[1.40 - 1.91]
		8 KB	18.86	4.83	5.57	0.00	[5.25 - 5.85]	[5.34 - 5.77]
		16 KB	6891.89	16.82	62.67	0.59	[43.52 - 77.39]	[46.58 - 73.78]
		32 KB	6975.85	38.29	145.26	1.02	[121.59 - 160.12]	[126.44 - 156.84]
		64 KB	7129.82	83.82	293.34	1.02	[270.31 - 313.65]	[273.61 - 308.50]
		128 KB	7403.95	314.93	566.47	1.44	[540.73 - 590.98]	[543.92 - 582.48]
		256 KB	7939.01	987.21	1113.92	1.95	[1079.33 - 1141.82]	[1084.77 - 1131.12]
		512 KB	11355.92	4334.77	4527.70	3.94	[4470.48 - 4560.29]	[4479.75 - 4516.30]
		1024 KB	11092.79	4171.73	4367.65	3.81	[4315.40 - 4398.56]	[4322.26 - 4356.48]
`copyout`	**off**	**0.125 KB**	5.01	0.20	0.20	0.00	[0.20 - 0.20]	-
		0.25 KB	15.22	0.32	0.32	0.00	[0.32 - 0.32]	-
		0.5 KB	4.98	0.37	0.37	0.00	[0.37 - 0.37]	-
		1 KB	15.69	0.48	0.48	0.00	[0.48 - 0.49]	[0.48 - 0.48]
		2 KB	1.69	0.68	0.69	0.00	[0.68 - 0.72]	[0.68 - 0.71]
		4 KB	100.12	1.25	1.42	0.01	[1.27 - 1.55]	[1.28 - 1.55]
		8 KB	17.63	3.71	3.77	0.00	[3.73 - 3.79]	[3.74 - 3.78]
		16 KB	45.89	9.07	11.78	0.02	[9.11 - 18.95]	[9.19 - 16.97]
		32 KB	85.47	20.00	35.60	0.06	[21.52 - 58.52]	[23.65 - 51.20]
		64 KB	195.29	88.94	128.75	0.08	[107.13 - 151.42]	[109.36 - 147.35]
		128 KB	7319.08	385.02	426.93	1.03	[409.09 - 446.44]	[411.56 - 439.35]
		256 KB	7897.94	974.00	1007.38	1.78	[980.16 - 1034.98]	[983.73 - 1022.65]
		512 KB	8938.55	2029.04	2068.42	2.84	[2039.25 - 2086.83]	[2042.11 - 2068.03]
		1024 KB	10985.64	3925.68	4118.12	3.85	[4070.50 - 4128.20]	[4074.08 - 4105.93]
	on	**0.125 KB**	0.22	0.20	0.20	0.00	[0.20 - 0.20]	-
		0.25 KB	1.14	0.33	0.33	0.00	[0.33 - 0.33]	-
		0.5 KB	0.94	0.38	0.38	0.00	[0.37 - 0.38]	-
		1 KB	6.30	0.48	0.48	0.00	[0.48 - 0.48]	-

continues on next page

continued from previous page

Statistics on copy operation performance in μs.								
Operation	**Interrupts**	**Data size**	**max**	**min**	**avg**	**stdev**	**99% ci**	**95% ci**
		2 KB	2.04	0.69	0.70	0.00	[0.70 - 0.70]	-
		4 KB	18.27	1.25	1.41	0.00	[1.26 - 1.56]	[1.27 - 1.56]
		8 KB	14.70	3.71	3.77	0.00	[3.73 - 3.79]	[3.74 - 3.78]
		16 KB	41.69	9.16	13.46	0.03	[9.29 - 27.77]	[9.44 - 23.90]
		32 KB	77.62	19.96	35.57	0.05	[21.23 - 51.62]	[22.30 - 46.35]
		64 KB	237.46	110.98	137.10	0.09	[115.80 - 162.69]	[119.87 - 157.32]
		128 KB	7302.80	431.03	469.94	1.18	[439.42 - 499.34]	[444.00 - 491.41]
		256 KB	7919.95	1039.36	1083.54	2.13	[1044.85 - 1114.07]	[1049.03 - 1102.37]
		512 KB	11340.47	4326.32	4518.10	3.95	[4467.28 - 4559.20]	[4471.18 - 4505.47]
		1024 KB	11182.03	4201.05	4440.96	4.03	[4387.42 - 7670.08]	[4391.98 - 4426.24]
`memcpy`	**off**	**0.125 KB**	16.35	0.36	0.38	0.00	[0.36 - 0.54]	[0.36 - 0.38]
		0.25 KB	0.92	0.49	0.50	0.00	[0.49 - 0.66]	[0.49 - 0.56]
		0.5 KB	0.98	0.54	0.55	0.00	[0.54 - 0.71]	[0.54 - 0.58]
		1 KB	6.77	0.64	0.71	0.00	[0.65 - 0.91]	[0.65 - 0.81]
		2 KB	17.12	0.86	0.91	0.00	[0.87 - 1.07]	[0.88 - 0.94]
		4 KB	11.56	1.35	1.65	0.00	[1.37 - 2.12]	[1.38 - 1.96]
		8 KB	24.07	2.71	2.84	0.00	[2.74 - 2.98]	[2.79 - 2.86]
		16 KB	111.83	17.17	32.97	0.04	[21.49 - 43.31]	[24.78 - 41.30]
		32 KB	6953.79	34.51	72.63	0.60	[58.23 - 92.94]	[61.01 - 87.93]
		64 KB	7034.54	71.58	164.78	0.60	[137.00 - 189.16]	[140.39 - 184.63]
		128 KB	7282.43	302.82	394.62	1.03	[371.91 - 423.00]	[374.70 - 414.33]
		256 KB	7818.86	840.45	927.46	1.78	[900.71 - 953.83]	[903.99 - 943.26]
		512 KB	8907.75	2001.88	2036.15	2.58	[2010.02 - 2052.09]	[2012.78 - 2039.17]
		1024 KB	11126.93	4088.66	4257.92	3.75	[4210.94 - 4269.60]	[4214.65 - 4248.50]
	on	**0.125 KB**	15.09	0.28	0.29	0.00	[0.28 - 0.30]	-
		0.25 KB	6.16	0.41	0.42	0.00	[0.41 - 0.43]	-
		0.5 KB	14.90	0.46	0.48	0.00	[0.46 - 0.52]	[0.46 - 0.52]
		1 KB	5.59	0.56	0.61	0.00	[0.56 - 0.69]	[0.56 - 0.68]
		2 KB	7.93	0.83	0.93	0.00	[0.84 - 1.08]	[0.85 - 1.06]
		4 KB	80.51	1.27	1.56	0.01	[1.28 - 1.90]	[1.29 - 1.89]
		8 KB	8.45	2.64	2.77	0.00	[2.65 - 2.82]	[2.72 - 2.79]
		16 KB	74.57	19.26	43.38	0.04	[29.87 - 53.38]	[33.96 - 51.72]
		32 KB	139.69	36.93	80.75	0.06	[64.41 - 95.65]	[66.70 - 92.05]
		64 KB	700.09	125.14	178.28	0.84	[157.31 - 198.43]	[160.76 - 193.62]
		128 KB	7272.97	368.49	430.62	1.45	[396.90 - 460.14]	[401.29 - 452.81]
		256 KB	7868.76	925.00	1027.73	1.86	[994.99 - 1056.97]	[999.33 - 1047.82]
		512 KB	11381.01	4480.33	4543.22	4.07	[4490.15 - 4574.46]	[4494.16 - 4529.57]
		1024 KB	11208.39	4277.28	4484.77	3.77	[4435.63 - 4505.83]	[4439.87 - 4475.31]

B.3 System Call Overhead

Accessing the operating system kernel from user space is time consuming, because a lot of different actions are performed in order to switch between user and kernel mode. The system call interface is called in the library. This interface sets up the arguments appropriately, e.g., selecting which system call number to use and the parameters, and generates a trap instruction that causes the process to switch from user into kernel mode. Once in kernel mode, control passes to an entry point called `syscall()`, which takes the system call number passed in as part of the trap instruction, indexes into a vector of pointers to system functions, and invokes the appropriate function. When the system call is finished, the entire process unwinds and control is returned to the user process running in user mode.

In [73], the overhead of the system call `getpid` is measured on different Linux-based operating systems using the processor cycle count register (the Intel `RDTSC` instruction). The total costs of accessing the kernel were measured to be 223 and 524 cycles for Linux and L^4Linux, respectively. To measure the overhead of accessing the NetBSD kernel by making a system call in a similar way, we implemented an empty system call, i.e., just entering the kernel and then returning. A test program made 10000 calls to the kernel and the results measured by our software probe (see Section A.2) are displayed in Figure B.3 and Table B.3. On average, about *0.4 μs* (*369 cycles*) are used to access the kernel per system call. This means that the overhead accessing the kernel in NetBSD is somewhere between traditional Linux and L^4Linux.

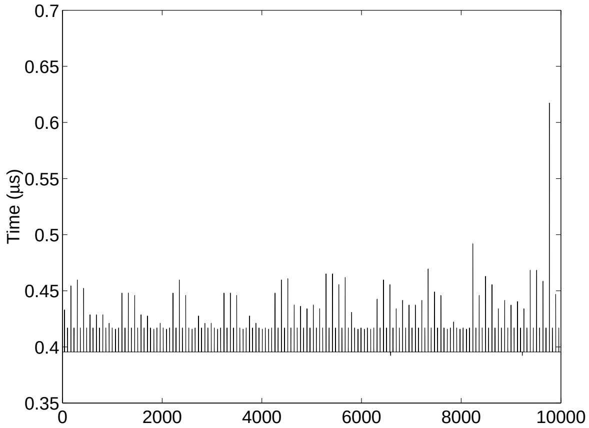


Figure B.3: Plot of system call overhead in μs.

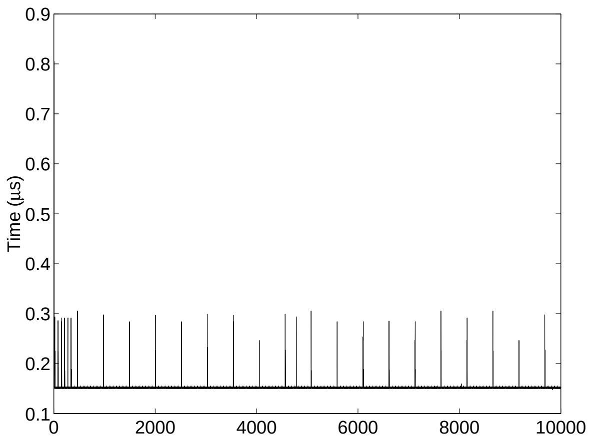
max	min	avg	stdev	99% ci	95% ci
0.62	0.40	0.40	0.00	[0.40 - 0.43]	[0.40 - 0.40]

Table B.3: System call overhead in μs.

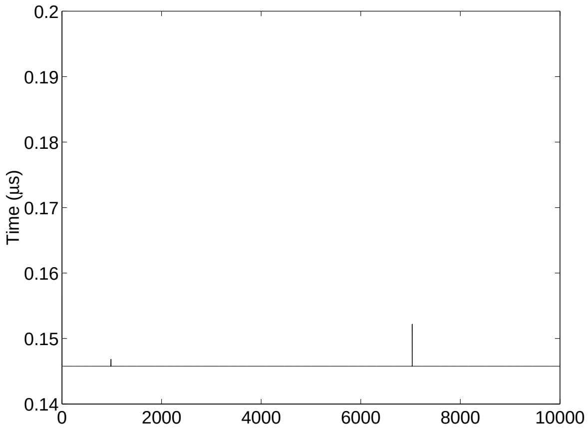
B.4 Pool and Memory Allocation/Deallocation Overhead

When reading data from disk, we need some memory to hold the data in the operating system kernel. To see whether we should use a pool mechanism or traditional memory allocation/free operations, we have measured the time to get and free a pool item, i.e., an mmbuf memory cluster from the `mmclpool`, and the time to allocate and free memory using `malloc()` and `free()`. As we have implemented two versions of the `mmclpool`, i.e., using the NetBSD pool mechanism and our own mmbuf memory cluster pool, we have tested both of these implementations. The NetBSD pool mechanism uses `poolget()` and `poolput()` to allocate and free items respectively, whereas our pool uses `clusterpool_get()` and `clusterpool_put()`. As we read data in 64 KB blocks from disk (see Section B.1), we have only looked at pool items of size 64 KB.

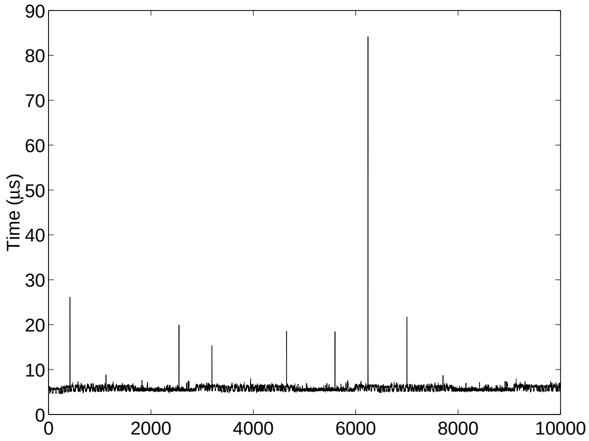
The results are shown in Figure B.4 (note that they have different y-axis) and Table B.4. The tests indicate that it is faster to use the pool mechanisms rather than allocating new memory each time a



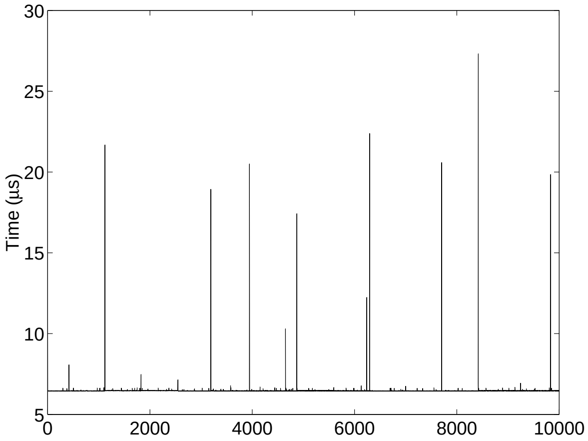
(A) `poolget()`.



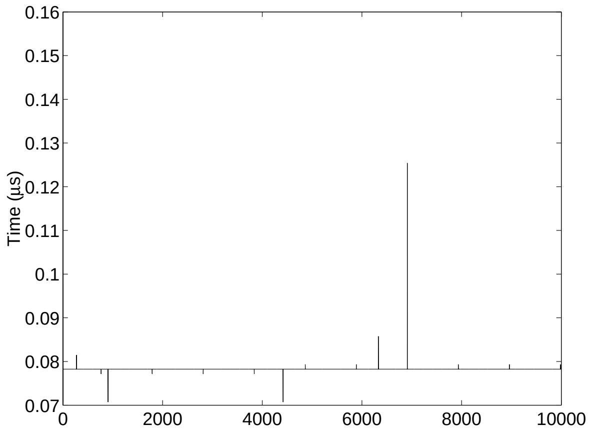
(B) `poolput()`.



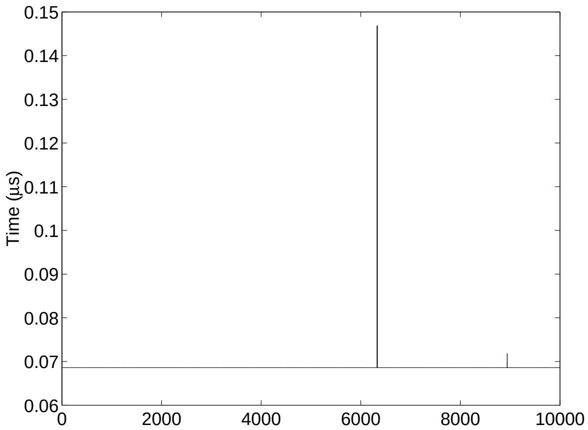
(C) `malloc()`.



(D) `free()`.



(E) `clusterpool_get()`.



(F) `clusterpool_put()`.

Figure B.4: Plot of time to get and free an mmbuf memory cluster in μs.

memory region is needed. The times to allocate a memory area using the NetBSD pool mechanism, our own pool, and the traditional allocation mechanism are in average 0.15 μs, 0.08 μs, and 5.80 μs, respectively. The respective operations to free the memory area consume the same proportional amount of time on the CPU. Thus, both pool mechanisms are faster than traditional allocation/free operations. Furthermore, our pool mechanism is slightly faster than the NetBSD pool mechanism. This is due to our specialized cluster pool implementation. The NetBSD pool mechanism supports more functionality and

	max	min	avg	stdev	99% ci	95% ci
poolget()	0.81	0.15	0.15	0.00	[0.15 - 0.23]	[0.15 - 0.15]
poolput()	0.19	0.15	0.15	0.00	[0.15 - 0.15]	—
malloc()	84.14	4.62	5.80	0.01	[4.80 - 7.09]	[5.09 - 6.66]
free()	27.32	6.42	6.48	0.00	[6.43 - 6.63]	[6.44 - 6.49]
clusterpool_get()	0.17	0.07	0.08	0.00	[0.08 - 0.08]	—
clusterpool_put()	0.15	0.07	0.07	0.00	[0.07 - 0.07]	—

Table B.4: Time to get and free a pool item and to allocate and free memory in μs.

different pools with different item sizes, item limits, allocation/free routines, etc. Our mechanism has only one pool and only the functionality needed for our purpose, i.e., our mechanism is smaller where the number of instructions used to retrieve and return a pool item therefore is reduced.

The above result assumes that the pool has available pool items. If the pool does not have any items available, more memory is mapped to the pool in sizes of two pool items (128 KB)[1] in our `mmclpool`. As depicted in Figure B.5, the time to map memory to the pool is dependent on the amount of already allocated memory where the time increases with the amount of used memory. This applies to both our implementations using either the NetBSD pool mechanism or our own mmbuf cluster pool, because both implementations use the same mechanism for allocating more memory. However, this memory may be allocated during boot time (pool initialization) or when the element is needed, and it remains in the pool as long there is need for it. Thus, this operation is performed once, i.e., there is no allocation/free for each time a pool item is needed, and the overhead is therefore negligible. The time to do a `malloc()` seems to be the same regardless of the amount of memory already allocated.

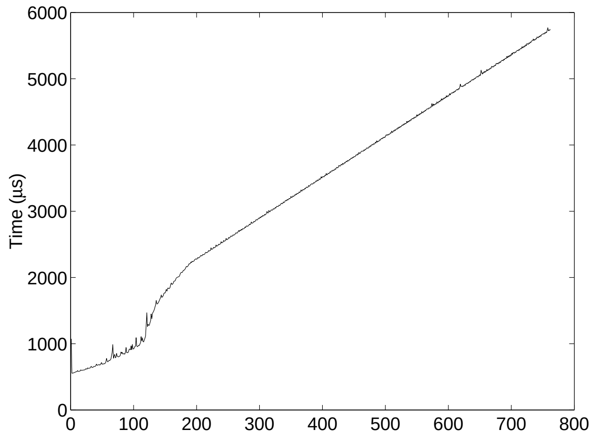


Figure B.5: Plot of time to get a pool item in μs (no items available in pool).

[1] In this test, we have allocated more memory in sizes of two pool items. However, this is a configurable parameter and better performance might be achieved using a larger size.

Appendix C

Abbreviations

Prefixes

n	nano (10^{-9})
μ	micro (10^{-6})
m	milli (10^{-3})
K	Kilo (2^{10})
M	Mega (2^{20})
G	Giga (2^{30})

Acronyms and Abbreviations

ACK	Acknowledgment
API	Application Programming Interface
ATM	Asynchronous Transfer Mode
ARQ	Automatic Repeat Request
avg	Average (in tables only)
bps	Bits per Second
Bps	Bytes per Second
BSD	Berkeley Software Distribution
ci	Confidence Interval (in tables only)
CoW	Copy-on-Write
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
DVD	Digital Versatile Disk
fbufs	Fast Buffers
fps	Frames Per Second
GB, Gb	Giga byte, Giga bit
GF	Galois Field
HDTV	High Definition Television
IETF	Internet Engineering Task Force
I/O	Input/Output
ILP	Integrated Layer Processing
INSTANCE	Intermediate Storage Node Concept
IP	Internet Protocol
IPC	Interprocess Communication
KB, Kb	Kilo byte, Kilo bit

L/MRP	Least/Most Relevant for Presentation
LoD	Learning-on-Demand
max	Maximum (in tables only)
MB, Mb	Mega byte, Mega bit
mbuf	Memory Buffer
min	Minimum (in tables only)
MMBUF	Multimedia M-buf mechanism
mmbuf	The MMBUF data structure
MoD	Media-on-Demand
MTU	Maximum Transfer Unit
MPEG	Moving Picture Expert Group
NLF	Network Level Framing
NoD	News-on-Demand
OS	Operating System (in tables only)
QoS	Quality-of-Service
PC	Personal Computer
PCI	Peripheral Component Interconnect
PIO	Programmed I/O
RAID	Redundant Array of Inexpensive (Independent) Disks
RDRAM	Rambus Dynamic Random Access Memory
RSVP	Resource Reservation Protocol
RTO	Retransmission Timeout
RTP	Real-Time Protocol
RTT	Round-Trip Time
SCSI	Small Computer Scalable Interface
stdev	Standard Deviation (in tables only)
TCP	Transport Control Protocol
UDP	User Datagram Protocol
VCR	Video Cassette Recorder
VoD	Video-on-Demand
WWW	World Wide Web
XOR	Exclusive Or

Bibliography

Literature References

[1] 3com: *"3Com EtherLink Server - Gigabit EtherLink Server NIC - User Guide"*, Model no. 3C985B-SX, Part no. 09-1151-002, January 1999

[2] Abbott, M. B., Peterson, L. L.: *"Increasing Network Throughput by Integrating Protocol Layers"*, IEEE/ACM Transactions on Networking, Vol. 1, No. 5, October 1993, pp. 600 - 610

[3] Aggarwal, C. C., Wolf, J. L., Yu, P. S.: *"Design and Analysis of Permutation-Based Pyramid Broadcasting"*, ACM/Springer Multimedia Systems, Vol. 7, No. 6, November 1999, pp. 439 - 448

[4] Allman, M., Glover, D., Sanchez, L.: *"Enhancing TCP Over Satellite Channels using Standard Mechanisms"*, RFC 2488, January 1999

[5] Allman, M.: *"A Web Server's View of the Transport Layer"*, ACM Computer Communication Review, Vol. 30, No. 5, October 2000

[6] Almulhem, A., El-Guibaly, F., Gulliver, T. A.: *"Adaptive Error Correction for ATM Communications using Reed-Solomon Codes"*, Proceedings of the 1996 IEEE SoutheastCon, Tampa, FL, USA, April 1996, pp. 227-230

[7] Anderson, E. W.: *"Container Shipping: A Uniform Interface for Fast, Efficient, High-Bandwidth I/O"*, PhD Thesis, Computer Science and Engineering Department, University of California, San Diego, CA, USA, 1995

[8] Anderson, D. C., Chase, J. S., Gadde, S., Gallatin, A. J., Yocum, K. G., Feeley, M. J.: *"Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet"*, Proceedings of the 1998 USENIX Annual Technical Conference, New Orleans, LA, USA, June 1998

[9] 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

[10] Biersack, E. W.: *"Performance Evaluation of Forward Error Correction in an ATM Environment"*, IEEE Journal on Selected Areas in Communications, Vol. 11, No. 4, May 1993, pp. 631-640

[11] Biersack, E. W., Bernhardt, C.: *"A Fault Tolerant Video Server Using Combined RAID 5 and Mirroring"*, Proceedings of the 1997 Multimedia Computing and Networking (MMCN'97), San Jose, CA, February 1997

[12] Blaum, M., Brady, J., Bruck, J., Menon, J.: *"EVENODD: An optimal Scheme for Tolerating Double Disk Failures in RAID Architectures"*, Proceedings of the 21st Annual International Symposium om Computer Architecture (ISCA'94), Chicago, IL, USA, April 1994, pp. 245 - 254

[13] Blömer, J., Kalfane M., Karp, R., Karpinski, M., Luby , M., Zuckerman, D.: *"An XOR-Based Erasure-Resilient Coding Scheme"*, Technical Report TR-95-048, International Computer Science Institute (ICSI), The University of California at Berkeley, CA, USA, August 1995

[14] Bobrow, D. G., Burchfiel, J. D., Murphy, D. L., Tomlinson, R. S., Beranek, B.: *"Tenex, A Paged Time Sharing System for the PDP-10"*, Communications of the ACM, Vol. 15, No. 3, March 1972, pp. 135 - 143

[15] Bolot, J.-C., Fosse-Parisis, S., Towsley, D.: *"Adaptive FEC-based Error Control for Internet Telephony"*, Proceedings of the 18th IEEE Conference on Computer Communications (INFOCOM'99), New York, NY, USA, March 1999

[16] Bolot, J.-C., Turletti, T.: *"Adaptive Error Control for Packet Video in the Internet"*, Proceedings of the 1996 IEEE International Conference on Image Processing (ICIP '96), Lausanne, Switzerland, September 1996

[17] Bolot, J.-C., Vega-Garcia, A.: *"The Case for FEC-based Error Control for Packet Audio in the Internet"*, To appear in ACM/Springer Multimedia Systems, 1999

[18] Borriss, M., Härtig, H.: *"Design and Implementation of a Real-Time ATM-Based Protocol Server"*, Proceedings of 19th IEEE Real-Time Systems Symposium (RTSS'98), Madrid, Spain, December 1998

[19] Boyce, J. M., Gaglianello, R. D.: *"Packet Loss Effects on MPEG Video Sent Over the Public Internet"*, Proceedings of the 6th ACM International Multimedia Conference (ACM MM'98), Bristol, UK, September 1998, pp. 181-190

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

[21] Braden, R. (ed.): *"Requirements for Internet Hosts – Communication Layers"*, RFC 1122, October 1989

[22] Bradshaw, M. K., Wang, B., Sen, S., Gao, L., Kurose, J., Shenoy, P., Towsley, D.: *"Periodic Broadcast and Patching Services - Implementation, Measurement, and Analysis in an Internet Streaming Video Testbed"* (poster), Proceedings of the Joint International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS 2001 / Performance 2001), Cambridge, MA, USA, June 2001

[23] Brustoloni, J. C.: *"Interoperation of Copy Avoidance in Network and File I/O"*, Proceedings of the 18th IEEE Conference on Computer Communications (INFOCOM'99), New York, NY, USA, March 1999

[24] Brustoloni, J. C., Steenkiste, P.: *"Effects of Buffering Semantics on I/O Performance"*, Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96), Seattle, WA, USA, October 1996, pp. 227 - 291

[25] Brustoloni, J. C., Steenkiste, P.: *"Evaluation of Data Passing and Scheduling Avoidance"*, Proceedings of the 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'97), St. Louis, MO, USA, May 1997, pp. 101 - 111

[26] Brustoloni, J. C., Steenkiste, P.: *"User-Level Protocol Servers with Kernel-Level Performance"*, Proceedings of the 17th IEEE on Computer Communications (INFOCOM'98), San Francisco, CA, USA, March/April 1998

[27] Buddhikot, M. M., Chen, X. J., Wu, D., Parulkar, G. M.: *"Enhancements to 4.4BSD UNIX for Efficient Networked Multimedia in Project MARS"*, Proceedings of the 5th IEEE International Conference on Multimedia Computing and Systems (ICMCS'98), Austin, TX, USA, June/July 1998

[28] 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

[29] Byers, J. W., Luby, M., Mitzenmacher, M., Rege, A.: *"A Digital Fountain Approach to Reliable Distribution of Bulk Data"*, Proceedings of the ACM conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'98), Vancouver, Canada, August/September 1998, pp. 56-67

[30] Cai, L. N., Chiu, D., McCutcheon, M., Ito, M. R., Neufeld, G. W.: *"Transport of MPEG-2 Video in a Routed IP Network - Transport Stream Errors and Their Effects on Video Quality"*, Proceedings of the 6th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS'99), Toulouse, France, October 1999, pp. 59-73

[31] Cao, P.: *"Application-Controlled File Caching and Prefetching"*, PhD Thesis, Department of Computer Science, Princeton University, Princeton, NJ, USA, January 1996

[32] Carle, G., Biersack, E. W.: *"Survey of Error Recovery Techniques for IP-based Audio-Visual Multicast Applications"*, IEEE Network, Vol. 11, No. 6, November/December 1997

[33] Carter, S. W., Long, D. D. E.: *"Improving Video-on-Demand Server Efficiency through Stream Tapping"*, Proceedings of the 6th International Conference on Computer Communications and Networks (ICCCN '97), Las Vegas, NV, USA, September 1997

[34] Chang, F., Gibson, G. A.: *"Automatic I/O Hint Generation through Speculative Execution"*, Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99), New Orleans, LA, USA, February 1999, pp. 1 - 14

[35] Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., Patterson, D. A.: *"RAID: High-Performance, Reliable Secondary Storage"*, ACM Computing Surveys, Vol. 26., No. 2, June 1994, pp. 145 - 185

[36] Chen, X. J., Buddhikot, M. M., Wu, D., Parulkar, G. M.: *"Enhancements to 4.4BSD UNIX for Efficient Networked Multimedia in Project MARS"*, Technical Report WUCS-97-38, Department of Computer Science, Washington University, St. Louis, MO, USA, 1997

[37] Cheriton, D. R.: *"The V Distributed System"*, Communications of the ACM, Vol. 31, No. 3, March 1988, pp. 314 - 333

[38] 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

[39] 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

[40] Chu, H.-K. J.: *"Zero-Copy TCP in Solaris"*, Proceedings of the 1996 USENIX Annual Technical Conference, San Diego, CA, USA, January 1996, pp. 253 - 264

[41] Cranor, C. D., Parulkar, G. M.: *"Universal Continuous Media I/O: Design and Implementation"*, Technical Report WUCS-94-34, Department of Computer Science, Washington University, St. Louis, MO, USA, 1994

[42] Cranor, C. D.: *"The Design and Implementation of the UVM Virtual Memory System"*, PhD Thesis, Sever Institute of Technology, Department of Computer Science, Washington University, St. Louis, MO, USA, August 1998

[43] Cranor, C. D., Parulkar, G. M.: *"Design of Universal Continuous Media I/O"*, Proceedings of the 5th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'95), Durham, NH, USA, April 1995, pp. 83-86

[44] Cranor, C. D., Parulkar, G. M.: *"The UVM Virtual Memory System"*, Proceedings of the 1999 USENIX Annual Technical Conference, Monterey, CA USA, June 1999

[45] Cranor, C. D., Parulkar, G. M.: *"Zero-Copy Data Movement Mechanisms for UVM"*, Submitted to the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99), New Orleans, LA, USA, February 1999 (not approved)

[46] Dahl Andersen, J.: *"A Turbo Tutorial"*, Report TELE-15 ISSN 1396-1535, Institute of Telecommunication, Technical University of Denmark, October 1999

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

[48] Dan, A., Shahabuddin, P., Sitaram, D., Towsley, D.: *"Channel Allocation under Batching and VCR Control in Movie-On-Demand Servers"*, Journal of Parallel and Distributed Computing, Vol. 30, No. 2, November 1995, pp. 168-179. Also as IBM Research Report, RC 19588, September 1994

[49] Dan, A., Sitaram, D.: *"Multimedia Caching Strategies for Heterogeneous Application and Server Environments"*, Multimedia Tools and Applications, Vol. 4, No. 3, May 1997, pp. 279 - 312

[50] Dresler, S., Hofmann, M.: *"Adaptive Error Correction to Support Heterogeneous Multicast Groups"*, Proceedings of 6th Open Workshop on High Speed Networks, Stuttgart, Germany, October 1997, pp. 169-174

[51] Druschel, P.: *"Operating System Support for High-Speed Communication"*, Communication of the ACM, Vol. 39, No. 9, September 1996, pp. 41-51

[52] Druschel, P., Abbot, M. B., Pagels, M. A., Peterson, L. L.: *"Network Subsystem Design"*, IEEE Network, Vol. 7, No. 4, July 1993, pp. 8 - 17

[53] Druschel, P., Peterson, L. L.: *"Fbufs: A High-Bandwidth Cross-Domain Transfer Facility"*, Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP'93), Asheville, NC, USA, December 1993, pp. 189 - 202

[54] Druschel, P., Peterson, L. L., Davie, B. S.: *"Experiences with a High-Speed Network Adapter: A Software Perspective"*, Proceedings of the ACM Conference on Communications Architectures, Protocols and Applications (SIGCOMM'94), London, UK, September 1994, pp. 2-13

[55] Eager, D. L., Vernon, M. K.: *Dynamic Skyscraper Broadcasts for Video-on-Demand"*, Proceedings of the 4th International Workshop on Advances in Multimedia Information Systems (MIS '98), Istanbul, Turkey, September 1998, pp. 18-32

[56] Edwards, C. H., Penney, D. E.: *"Calculus and Analytic Geometry"*, 3rd edition, Prentice-Hall, 1990

[57] Fall, K. R.: *"A Peer-to-Peer I/O System in Support of I/O Intensive Workloads"*, PhD Thesis, Computer Science and Engineering Department, University of California, San Diego, CA, USA, 1994

[58] Fall, K., Pasquale, J.: *"Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability"*, Proceedings of the 1993 USENIX Winter Technical Conference, San Diego, CA, USA, January 1993, pp. 327 - 333

[59] Fall, K., Pasquale, J.: *"Improving Continuous-Media Playback Performance with In-Kernel Data Paths"*, Proceedings of the 1st IEEE International Conference on Multimedia Computing and Systems (ICMCS'94), Boston, MA, USA, May 1994, pp. 100-109

[60] Feldmeier, D. C.: *"A Framework of Architectural Concepts for High-Speed Communication Systems"*, IEEE Journal on Selected Areas in Communications, Vol. 11, No. 4, May 1994, pp. 480 - 488

[61] Fitzgerald, R., Rashid, R.F: *"The Integration of Virtual Memory Management and Interprocess Communication in Accent"*, ACM Transactions on Computer Systems, Vol. 4, No. 2, May 1986, pp. 147 - 177

[62] Gao, L., Kurose, J., Towsley, D.: *"Efficient Schemes for Broadcasting Popular Videos"*, Proceedings of the 8th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'98), Cambridge, UK, 1998

[63] Garofalakis, M. N., Özden, B., Silberschatz, A.: *"On Periodic Resource Scheduling for Continuous-Media Databases"*, The VLDB Journal, Vol. 7, No. 4, 1998, pp. 206-225

[64] Garcia-Martinez, A., Fernadez-Conde, J., Vina, A.: *"Efficient Memory Management in VoD Servers"*, to appear in: Computer Communications, 2000

[65] Grimaldi, R. P.: *"Discrete and Combinatorial Mathematics - An Applied Introduction"*, Second edition, Addison-Wesley, 1989

[66] Griwodz, C.: *"Wide-Area True Video-on-Demand by a Decentralized Cache-Based Distribution Infrastructure"*, PhD Thesis, TU Darmstadt, Germany, April 2000

[67] Haccoun, D., Pierre, S.: *"Automatic Repeat Request"*, in: Gibson, J. D. (ed.): The Communications Handbook, CRC press, 1997

[68] Halsall, F: *"Data Communications, Computer Networks and Open Systems"*, Fourth edition, Addison-Wesley, 1995

[69] Halvorsen, P., Goebel, V., Plagemann, T.: *"Q-L/MRP: A Buffer Management Mechanism for QoS Support in a Multimedia DBMS"*, Proceedings of 1998 IEEE International Workshop on Multimedia Database Management Systems (IW-MMDBMS'98), Dayton, OH, USA, August 1998, pp. 162 - 171

[70] 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
http://www.acm.org/sigops/sosp99/posters.html

[71] 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

[72] Halvorsen, P., Plagemann, T., Goebel, V.: *"Integrated Error Management for Media-on-Demand Services"*, Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001), Anchorage, AK, USA, April 2001, pp. 621-630

[73] Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., Wolter, J.: *"The Performance of μ-Kernel-based Systems"*, Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP'97), Saint-Malo, France, October 1997

[74] Härtig, H., Baumgartl, R., Borriss, M., Hamann, C.-J., Hohmuth, M., Mehnert, F., Reuther, L., Schönberg, S., Wolter, J.: *"DROPS - OS Support for Distributed Multimedia Applications"*, Proceedings of the 8th ACM SIGOPS European Workshop, Sintra, Portugal, September 1998

[75] Holland, M., Gibson, G. A.: *"Parity Declustering for Continuous Operation in Redundant Disk Arrays"*, Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), Boston, MA, USA, October 1992, pp. 23-35

[76] Holland, M., Gibson, G. A., Siewiorek, D. P.: *"Fast, On-Line Failure Recovery in Redundant Disk Arrays"*, Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing (FTCS'93), Toulouse, France, June 1993, pp. 421-433

[77] Hellerstein, L., Gibson, G. A., Karp, R. M., Katz, H. M., Patterson, D. A.: *"Coding Techniques for Handling Failures in Large Disk Arrays"*, Algorithmica, Vol. 12, No. 2/3, August/September 1994, pp. 182 - 208

[78] Hua, K. A., Sheu, S.: *"Skyscraper Broadcasting: A New Broadcasting Scheme for Metropolitan Video-on-Demand System"*, Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'97), Cannes, France, September 1997, pp. 89-100

[79] Hua, K. A., Cai, Y., Sheu, S.: *"Patching: A multicast Technique for True Video-on-Demand Services"*, Proceedings of the 6th ACM International Multimedia Conference (ACM MM'98), Bristol, UK, September 1998, pp. 191-200

[80] ISO/IEC: *"Working Draft for Open Distributed Processing - Reference Model - Quality of Service"*, ISO/IEC JTC 1/SC 21 NQoS1, July 1997

[81] Jorde, E.: *"Buffer Management in INSTANCE"* (in Norwegian), Master Thesis, UniK - Center for Technology at Kjeller/Department of Informatics, University of Oslo, Norway, 2000

[82] Juhn, L.-S., Tsend, L.-M.: *"Harmonic Broadcasting for Video-on-Demand Service"*, IEEE Transactions on Broadcasting, Vol. 43, No. 3, September 1997, pp. 268 - 271

[83] Kamath, M., Ramamritham, K., Towsley, D.: *"Continuous Media Sharing in Multimedia Database Systems"*, Proceedings of the 4th International Conference on Database Systems for Advanced Applications (DASFAA'95), Singapore, April 1995, pp. 79 - 86

[84] 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

[85] 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

[86] Keller, R., Effelsberg, W., Lamparter, B.: *"XMovie: Architecture and Implementation of a Distributed Movie System"*, ACM Transactions on Information Systems, Vol. 13, No. 4, October 1995, pp., 471-499

[87] Kitamura, H., Taniguchi, K., Sakamoto, H., Nishida T.: *"A New OS Architecture for High Performance Communication Over ATM Networks: Zero-Copy Architecture"*, Proceedings of the 5th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'95), Durham, NH, USA, April 1995, pp. 87-90

[88] Krasic, C., Walpole, J.: *"QoS Scalability for Streamed Media Delivery"*, Technical Report CSE-99-011, Department of Computer Science and Engineering, Oregon Graduate Institute, Portland, OR, USA, September, 1999

[89] Krishnan, R., Venkatesh, D., Little, T. D. C.: *"A Failure and Overload Tolerance Mechanism for Continuous Media Servers"*, Proceedings of the 5th ACM International Multimedia Conference (ACM MM'97), Seattle, WA, USA, November 1997, pp. 131-142

[90] Kumar, M., Kouloheris, J. L., McHugh, M. J., Kasera, S.: : *"A High Performance Video Server for Broadband Network Environment"*, Proceedings of the 1996 Multimedia Computing and Networking (MMCN'96), San Jose, CA, USA, January 1996, pp. 410-421

[91] 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

[92] Leffler, S. J., McKusick, M. K., Karels, M. J., Quarterman, J. S.: *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley Publishing Company, 1989

[93] Lei, H., Duchamp, D.: *"An Analytical Approach to File Prefetching"*, Proceedings of the 1997 USENIX Annual Technical Conference, Anaheim, CA, USA, January 1997

[94] Löser, J., Härtig, H., Reuther, L.: *"A Streaming Interface for Real-Time Interprocess Communication"*, Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Bavaria, Germany, May 2001

[95] Lu, G.: *"Communications and Computing for Distributed Multimedia Systems"*, Artech House Publishers, 1996

[96] Luby, M., Mitzenmacher, M., Shokrollahi, M. A., Spielman, D. A., Stemann, V.: *"Practical Loss-Resilient Codes"*, Proceedings of the 29th annual ACM Symposium on Theory of Computing (STOC'97), El Paso, TX, USA, May 1997, pp. 150-159.

[97] Luby, M., Mitzenmacher, M., Shokrollahi, M. A.: *"Analysis of Random Processes via And-Or Tree Evaluation"*, Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, USA, January 1998, pp. 364-373

[98] Lund. K., Halvorsen, P., Goebel, V., Plagemann, T.: *"Requirements Analysis and Design for a Flexible Learning-on-Demand System"*, Proceedings of the Fourth World Conference on Integrated Design & Process Technology 1999-2000 (IDPT 1999 - 2000), Dallas, Texas, June 2000

[99] McAuley, A. J.: *"Reliable Broadband Communication Using a Burst Erasure Correcting Code"*, Proceedings of the ACM Symposium on Communications, Architectures and Protocols (SIGCOMM'90), Philadelphia, PA, USA, September 1990, pp. 297-306

[100] 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

[101] Miller, F. W., Keleher, P., Tripathi, S. K.: *"General Data Streaming"*, Proceedings of the 19th IEEE Real-Time System Symposium (RTSS'98), Madrid, Spain, December 1998

[102] Miller, F. W., Tripathi, S. K.: *"An Integrated Input/Output System for Kernel Data Streaming"*, Proceedings of the SPIE/ACM Multimedia Computing and Networking (MMCN '98), San Jose, CA, USA, January 1998, pp. 57 - 68

[103] Mogi, K., Kitsuregawa, M.: *"Hot Mirroring: A Study to Hide Parity Upgrade Penalty and Degradations During Rebuilds for RAID5"*, Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 1996 pp. 183-194

[104] Moser, F., Kraiß, A., Klas, W.: *"L/MRP: A Buffer Management Strategy for Interactive Continuous Data Flows in a Multimedia DBMS"*, Proceedings of the 21th IEEE International Conference on Very Large Databases (VLDB'95), Zurich, Switzerland, September 1995, pp. 275 - 286

[105] Nahrstedt, K., Steinmetz, R.: *"Resource Management in Networked Multimedia Systems"*, IEEE Computer, Vol. 28, No. 5, May 1995, pp. 52 - 63

[106] Nakajima, T., Tezuka, H.: *"Virtual Memory Management for Interactive Continuous Media Applications"*, Proceedings of the 4th IEEE International Conference on Multimedia Computing and Systems (ICMCS'97), Ottawa, Canada, June 1997

[107] Ng, R., T., Yang, J.: *"Maximizing Buffer and Disk Utilization for News-On-Demand"*, Proceedings of the 20th IEEE International Conference on Very Large Databases (VLDB'94), Santiago, Chile, September 1994, pp. 451 - 462

[108] Nonnenmacher, J., Biersack, E. W.: *"Reliable Multicast: Where to use FEC"*, Proceedings of IFIP 5th International Workshop on Protocols for High Speed Networks (PfHSN'96), Sophia Antipolis, France, October 1996, pp. 134-148

[109] 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

[110] Pai, V. S.: *"IO-Lite: A Copy-free UNIX I/O System"*, Master of Science Thesis, Rice University, Houston, TX, USA, January 1997

[111] Pai, V. S., Druschel, P., Zwaenepoel, W.: *"IO-Lite: A unified I/O buffering and caching system"*, Technical Report TR97-294, Rice University, Houston, TX, USA, 1997

[112] 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

[113] 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

[114] Papadopoulos, C., Parulkar, G. M: *"Retransmission-Based Error Control for Continuous Media Applications"*, Proceedings of the 6th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'96), Zushi, Japan

[115] Papadopoulos, C., Parulkar, G. M, Varghese, G.: *"An Error Control Scheme for Large-Scale Multicast Applications"*, Proceedings of the 18th IEEE Conference on Computer Communications (INFOCOM'98), San Francisco, CA, USA, March/April 1998

[116] Pâris, J.-F., Carter, S. W., Long, D. D. E.: *"Efficient Broadcasting Protocols for Video on Demand"*, Proceedings of the 6th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '98), Montreal, Canada, July 1998, pp. 127-132

[117] Pâris, J.-F., Carter, S. W., Long, D. D. E.: *"A Low Bandwidth Broadcasting Protocol for Video on Demand"*, Proceedings of the 7th International Conference on Computer Communications and Networks (ICCCN'98), Lafayette, LA, USA, October 1998, pp. 690 - 697

[118] Pâris, J.-F., Long, D. D. E., Mantey, P. E.: *"Zero-Delay Broadcasting Protocols for Video-on-Demand"*, Proceedings of the 1999 ACM Multimedia Conference (ACM MM'99), Orlando, FL, USA, November 1999, pp. 189 - 197

[119] 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

[120] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., Zelenka, J.: *"Informed Prefetching and Caching"*, Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP'95), Cooper Mountain, CO, USA, December 1995, pp. 79 - 95

[121] Pasquale, J., Anderson, E., Muller, P. K.: *"Container Shipping - Operating System Support for I/O-Intensive Applications"*, IEEE Computer, Vol. 27, No. 3, March 1994, pp. 84 - 93

[122] Pasquale, J., Anderson, E. W., Fall, K., Kay, J. S.: *"High-performance I/O and Networking Software in Sequoia 2000"*, Digital Technical Journal, Vol. 7, No. 3, December 1995, pp. 84 - 96

[123] Paxson, V.: *"End-to-End Routing Behavior in the Internet"*, Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'96), Palo Alto, CA, USA, August 1996, pp. 25-38

[124] Paxson, V.: *"End-to-End Internet Packet Dynamics"*, Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'97), Cannes, France, September 1997, 139-152

[125] Paxson, V., Allman, M.: *"Computing TCP's Retransmission Timer"*, RFC 2988, November 2000

[126] Perkins, C., Kouvelas, I., Hodson, O., Hardman, V., Handley, M., Bolot, J.-C., Vega-Garcia, A., Fosse-Parisis, S.: *"RTP Payload for Redundant Audio Data"*, RFC 2198, September 1997

[127] Perkins, C., Hodson, O., Hardman, V.: *"A Survey of Packet Loss Recovery Techniques for Streaming Audio"*, IEEE Network, Vol. 12, No. 5, September/October 1998, pp. 40 - 48

[128] 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

[129] 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

[130] Plank, J. S.: *"A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems"*, Software - Practice and Experience, Vol. 27, No. 9, September 1997, pp. 995-1012

[131] Postel, J.: *"User Datagram Protocol"*, RFC 768, August 1980

[132] Postel, J. (ed.): *"Internet Protocol"*, RFC 791, September 1981

[133] Postel, J. (ed.): *"Transmission Control Protocol"*, RFC 793, September 1981

[134] 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

[135] 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

[136] Rizzo, L.: *"Effective Erasure Codes for Reliable Computer Communication Protocols"*, ACM Computer Communication Review, Vol. 27, No. 2, April 1997, pp. 24-36

[137] Rorabaugh, C. B.: *"Error Coding Cookbook - Practical C/C++ Routines and Recipes for Error detection and Correction"*, McGraw-Hill, 1996

[138] Rashid, R., Robertson, G.: *"Accent: A Communication-Oriented Network Operating System Kernel"*, Proceedings of the 8th ACM Symposium on Operating System Principles (SOSP'81), New York, NY, USA, 1981, pp. 64 - 75

[139] Rosenberg, J., Schulzrinne, H.: *"An RTP Payload Format for Generic Forward Error Correction"*, RFC 2733, December 1999

[140] Rotem, D., Zhao, J. L.: *"Buffer Management for Video Database Systems"*, Proceedings of the 11th International Conference on Data Engineering (ICDE'95), Tapei, Taiwan, March 1995, pp. 439-448

[141] Roth, A., Moshovos, A., Sohi, G. S.: *"Dependence Based Prefetching for Linked Data Structures"*, Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII), San Jose, CA, USA, October 1998, pp. 115 - 126

[142] 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

[143] Shenoy, P. J., Goyal, P., Rao, S. S., Vin, H. M: *"Symphony: An Integrated Multimedia File System"*, Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking (MMCN'98), San Jose, CA, USA, January 1998, pp. 124 - 138

[144] Shenoy, P. J.: *"Symphony: An Integrated Multimedia File System*, PhD Thesis, Distributed Multimedia Computing Laboratory, Department of Computer Science, University of Texas at Austin, Austin, TX, USA, August 1998

[145] Silberschatz, A., Korth, H. F., Sudarshan, S.: *"Database System Concepts"*, McGraw-Hill, 1997

[146] Sitaram, D., Dan, A.: *"Multimedia Servers - Applications, Environments, and Design"*, Morgan Kaufman Publishers, 2000

[147] Skevik, K.-A.: *"Memory Management for High-Speed Protocols"*, Master Thesis, UniK - Center for Technology at Kjeller/Department of Informatics, University of Oslo, Norway, 2000

[148] Skevik, K.-A., Plagemann, T., Goebel, V., Halvorsen, P.: *"Evaluation of a Zero-Copy Protocol Implementation"*, to appear at the Euromicro Workshop on Multimedia and Telecommunications, Warsaw, Poland, September 2001

[149] Smith, J. M., Traw, C. B. S.: *"Giving Applications Access to Gb/s Networking"*, IEEE Network, Vol. 7, No. 4, July 1993, pp. 44 - 52

[150] Srinivasan, V., Ghanwani, A., Gelenbe, E.: *"Block Loss Reduction in ATM Networks"*, Elsevier Computer Communications, Vol. 19, No. 13, November 1996, pp. 1077-1091

[151] Steinmetz, R., Nahrstedt, K.: *"Multimedia: Computing, Communications & Applications"*, Prentice Hall, 1995

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

[153] Stevens, W. R.: *"TCP/IP Illustrated, Volume 1 - The Protocols"*, 16th printing, Addison Wesley Longman, February 2000

[154] Stodolsky, D., Gibson, G., Holland, M.: *"Parity Logging Overcoming the Small Write Problem in Redundant Disk Arrays"*, Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93), San Diego, CA, USA, May 1993, pp. 64-75

[155] Stone, J., Greenwald, M., Partridge, C., Hughes, J.: *"Performance of checksums and CRC's over real data"*, IEEE/ACM Transactions on Networking, Vol. 6, No. 5, October 1998, pp. 529-543

[156] 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

[157] Tanenbaum, A. S.: *"Operating Systems - Design and Implementation"*, Prentice Hall, 1987

[158] Tanenbaum, A. S.: *"Modern Operating Systems"*, Prentice Hall, 1992

[159] Tanenbaum, A. S.: *"Computer Networks"*, Third edition, Prentice Hall, 1996

[160] Tezuka, H., Nakajima, T.: *"Simple Continuous Media Storage Server on Real-Time Mach"*, Proceedings of the 1996 USENIX Annual Technical Conference, San Diego, CA, USA, January 1996

[161] Thadani, M. N., Khalidi, Y. A.: *"An Efficient Zero-Copy I/O Framework for UNIX"*, Technical Report SMLI TR-95-39, Sun Microsystems Laboratories Inc., May 1995

[162] Thekkath, C. A., Nguyen, T. D., Moy, E., Lazowska, E. D: *"Implementing Network Protocols at User Level"*, IEEE/ACM Transactions on Networking, Vol. 1, No. 5, October 1993, pp. 554 - 565

[163] Tzou, S.-Y., Anderson, D. P.: *"The Performance of Message-passing using Restricted Virtual Memory Remapping"*, Software - Practice and Experience, Vol. 21, No. 3, March 1991, pp. 251-267

[164] Viswanathan, S., Imielinski, T.: *"Metropolitan area Video-on-Demand Service Using Pyramid Broadcasting"*, Multimedia Systems, Vol 4., No. 4, 1996, pp. 197-208

[165] Vogel, A., Kerhervé, B., Gecsei, J., von Bochmann, G.: *"Distributed Multimedia and QoS: A Survey"*, IEEE Multimedia, Vol. 2, No. 2, summer 1995, pp. 10 - 19

[166] Wang, C., Goebel, V., Plagemann, T.: *"Techniques to Increase Disk Access Locality in the Minorca Multimedia File System"* (Short Paper), Proceedings of the 7th International Multimedia Conference (ACM MM'99), Orlando, FL, USA, October 1999

[167] Wang, J. L., Silvester, J. R.: *"Delay Minimization of the Adaptive Go-Back-N ARQ Protocols for Point-to-Multipoint Communications"*, Proceedings of the 8th IEEE Conference on Computer Communications (INFOCOM'89), Ottawa, Canada, April 1989, pp. 584-593

[168] 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,

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

[170] Yajnik, M., Kurose, J., Towsley.: *"Packet Loss Correlation in the MBone Multicast Network"*, Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM'96), London, UK, November 1996, pp. 94-99

[171] Yau, D. K. Y., Lam, S. S.: *"Operating System Techniques for Distributed Multimedia"*, Technical Report TR-95-36 (revised), Department of Computer Sciences, University of Texas at Austin, Austin, TX, USA, January 1996

[172] Victor Yodaiken, V.: *"The RTLinux Manifesto"*, Proceedings of the 5th Linux Expo, Raleigh, NC, USA, March 1999

[173] Zhang, A., Gollapudi, S.: *"QoS Management in Educational Digital Library Environments"*, Technical Report CS-TR-95-53, State University of New York at Buffalo, New York, NY, USA, 1995

[174] Özden, B., Rastogi, R., Silberschatz, A.: *"Buffer Replacement Algorithms for Multimedia Storage Systems"*, Proceedings of the 3rd IEEE International Conference on Multimedia Computing and Systems (ICMCS'96), Hiroshima, Japan, 1996

World Wide Web (HTTP) References

[175] 3com: *"Ethernet Packet Loss Reference"*, http://www.support.3com.com/infodeli/tools/netmgt/temwin/-temnt97/091293ts/ploss3.htm, September 1998

[176] 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

[177] Advanced Hardware Architectures, http://www.aha.com/, May 2001

[178] BBC News online, http://news.bbc.co.uk/, June 2001

[179] Bredbåndsfabrikken (in Norwegian), http://www.bredband.no/, June 2001

[180] CNN.com - Live Video/Audio, http://europe.cnn.com/video/popup/top.video.exclude.html, June 2001

[181] Dell Computer Corporation: *"Technical Specifications: Dell Precision WorkStation 620 Systems User's Guide"*, http://support.euro.dell.com/docs/systems/ws620/en/ug/specs.htm, 2000

[182] Doehner, B.: *"Overview of Forward Error Correction"*, http://bugs.wpi.edu:8080/EE535/hwk97/-hwk4cd97/bad/bad.html, May 1997

[183] DROPS - The Dresden Real-Time Operating System Project, http://os.inf.tu-dresden.de/drops/, June 2001

[184] DVDDemystified.com: *"DVD Frequently Asked Questions"*, http://dvddemystified.com/dvdfaq.html, March 2000

[185] Ginier-Gillet, P., Di Minico, C. T.: *"1000BASE-T: Gigabit Ethernet Over Category 5 Copper Cabling"*, 3com, http://www.3com.com/technology/tech_net/white_papers/503047.html, December 1999

[186] Hewlett-Packard Company: *"Linux Programmer's Manual - sendfile"*, http://devresource.hp.com/-STKL/man/RH6.1/sendfile_2.html, May 2001

[187] Intel Corporation: *"Using the RDTSC Instruction for Performance Monitoring"*, ftp://download.intel.com/-software/idap/media/pdf/rdtscpm1.pdf, 1998

[188] Intel Corporation: *"Intel Architecture Software Developer's Manual - Volume 1: Basic Architecture"*, Order Number 243190, ftp://download.intel.com/design/pentiumii/manuals/243190.htm, 1999

[189] Intel Corporation: *"Intel Architecture Software Developer's Manual - Volume 2: Instruction Set Reference"*, Order Number 243191, ftp://download.intel.com/design/pentiumii/manuals/243191.htm, 1999

[190] Intel Corporation: *"Intel Architecture Software Developer's Manual - Volume 3: System Programming"*, Order Number 243192, ftp://download.intel.com/design/pentiumii/manuals/243192.htm, 1999

[191] Intel Corporation: *"Block Copy Using PentiumIII Streaming SIMD Extensions"*, Revision 1.9, ftp://download.intel.com/design/servers/softdev/copy.pdf, 1999

[192] Intel Corporation: *"Intel 840 Chipset - Product Overview"*, http://developer.intel.com/design/chipsets/840/, 2000

[193] NASA Jet Propulsion Laboratory: *"JPL turbo codes page"*, Communications Systems and Research Section (JPL), Information Processing Group, NASA, http://www331.jpl.nasa.gov/public/, May 2001

[194] MPEG.org: *"DVD Technical Notes - Bitstream breakdown"*, http://mpeg.org/MPEG/DVD, March 2000

[195] Mulder, F.: *"Rambus Direct RDRAM"*, http://www.pcaccelerate.com/Memory/RDRAM-1/rdram-1.html, January 2000

[196] NetBSD Foundation, http://www.netbsd.org/, May 2000

[197] NetBSD Programmer's Manual, *pool* manual page, http://www.tac.eu.org/cgi-bin/man-cgi?pool++NetBSD-1.5, March 2000

[198] NetBSD Programmer's Manual, *getrusage* manual page, http://www.tac.eu.org/cgi-bin/man-cgi?-getrusage++NetBSD-1.5, June 2001

[199] NRK.no (in Norwegian), http://www.nrk.no/magasin/nyheter/sendinger/, June 2001

[200] Opticom ASA, http://www.opticomasa.com, March 2001

[201] RTLinux - The Realtime Linux, http://www.rtlinux.org/, June, 2001

[202] Seagate, Disk Products, http://www.seagate.com/cda/products/discsales/guide/, February 2000

[203] Seagate, Disk Products by Product Number, http://www.seagate.com/cda/products/discsales/index, May 2001

[204] TechFest: *"PCI Local Bus Technical Summary"*, http://www.techfest.com/hardware/bus/pci.htm, June 2001

[205] TV2 Interactiv (in Norwegian), http://frihet.tv2.no/, June 2001

[206] Turbo Codes Company, http://www.turboconcept.com/index.php3, May 2001

[207] Western-Digital, Products, http://www.western-digital.com/products/, October 1999