JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2000; **00**:1–7 *Prepared using* smrauth.cls *[Version: 2003/05/07 v1.1]*

Practice

Development and Evolution of a Heterogeneous Continuous Media Server: a case study[‡]

D. J. Makaroff[1*†] and N. C. Hutchinson[2]

[1]*Department of Computer Science, University of Saskatchewan, Saskatoon, SK, CANADA*
[2]*Department of Computer Science, University of British Columbia, Vancouver, BC, CANADA*

SUMMARY

Media server software is significantly complicated to develop and maintain, due to the nature of the many interface aspects which must be considered. This paper provides a case study of the design, implementation, and evolution of a continuous media file server. We place emphasis on the evolution of the software and our approach to maintainability. The user interface is a major consideration, even though the server software would appear isolated from that factor. Since continuous media servers must send the raw data to a client application over a network, the protocol considerations, hardware interface and data storage/retrieval methods are of the paramount importance. In addition, the application programmer's interface (API) to the server facilities has an impact on both the internal design and the performance of such a server. We discuss our experiences and insight into the development of such software products within a small research-based university environment. We experienced two main types of evolutionary change: requirements changes from the limited user community and performance enhancements/corrections. While the former were anticipated via a generic interface and modular design structure, the latter were surprising and substantially more difficult to solve.

KEY WORDS: multimedia, file servers, software evolution, software development, interface design

*Correspondence to: Department of Computer Science, University of Saskatchewan, 57 Campus Drive, Saskatoon, SK, CANADA
†E-mail: makaroff@cs.usask.ca
Contract/grant sponsor: Canadian Telecommunications Institute for Telecommunications Research; contract/grant number:

Received 31 January 2002
Revised 19 January 2004
Accepted 12 December 2003

INTRODUCTION

All software undergoes changes from the design to the deployment phase. Software that is used for significant lengths of time must evolve as requirements change and improvements are desired. Lehman et al. summarized general observations about the nature of these changes and codified them as a set of laws[1]. One law, in particular, states that the complexity of a system tends to increase without specific measures to maintain or decrease it. Our deployment of a continuous media file server in the context of a multi-year university research project confirmed that some of these laws apply to research-based mid-size software systems as well as the large systems studied by Lehman.

One of the areas of recent prolific research and development with regard to computer systems has been the introduction of multiple forms of media to the computing experience. In particular, authoring tools, server and network distribution systems and end-client viewing systems have been studied and marketed extensively. Varying degrees of success have been achieved, due to the difficulty in leveraging emerging technological advances and the need to have both high-performance and real-time capabilities. In some cases, high-performance has been achieved by a simplistic software structure and overly-redundant hardware configurations.

At the outset of this research, many potential technologies existed for supplying the requirements, but none had been proven to be applicable in our domain. We anticipated that it might be necessary to change hardware architectures, network protocols and media encoding formats. Thus, the initial design focused on a small number of key algorithms that abstracted from the hardware specifications and provided a modular design to isolate dependencies on outside factors. Evolution of the software was expected to occur in the implementation phase. A user community did not exist and the requirements evolved during our experimentation with both the available hardware and the fundamental algorithms. As the project matured through the integration and testing phases, there were further significant alterations, some of which were inspired by research collaborators who used our system as a component in a much larger system.

The context was different than the commercial software environment since the deadlines imposed were for conferences and/or demos, rather than due to market pressures. Our objectives were to fulfill our research goals, and so, customer involvement was minimal, in so far as we were our own customers.

Since continuous media encoding formats had a history of multiple standards which were themselves evolving, we knew that a server that restricted support to only one media format was doomed to failure. As well, many existing systems were dependent on performance characteristics of specific hardware, that are subject to quick changes. We needed a system that was format independent and designed at a level protected from specific hardware implementations. Long-term use of the system would likely have required wholesale changes to major components. Even with our initial recognition of these evolution points, unexpected evolution occurred.

The approach taken in the Continuous Media File Server (CMFS) was to build a system that provided high-performance and real-time guarantees of delivery of heterogeneous media documents on commodity hardware. This hardware could be combined in a heterogeneous fashion to enable linear scalability of the system from single workstation-class machines to networks of such workstations and multiprocessor machines, without restrictions on configurations. The server was developed in the context of a research project with the Canadian Institute for Telecommunications Research (CITR), whose research task was to explore enabling technologies for broad-band services [2]. The delivery of continuous media was one aspect of this project; others included quality of service monitoring and negotiation, media synchronization, multimedia databases as well as scalable video encoding methods. The server and the media synchronization were the most crucial aspects for the functionality of the project, being the two end points of system interactions.

Access to the server was through an Application Programmer's Interface (API) that enabled flexible and convenient modes of user interaction. This provided the initial incentive and framework for requirements of the system. Modular design was a huge focus of the project, since a major goal was to enable comparisons of different algorithms for the major components. These included: a)admission control for storage and/or retrieval of media objects, b) internal meta-data storage for the media objects and c) transmission protocols for sending and receiving of the raw media data.

The development team consisted of one PhD. student, two Master's students, both later hired as research associates in conjunction with the project, one visiting Master's student and two faculty members. Other members of our department and research collaborators in our major project provided insight and suggestions on occasion.

This paper presents a case study of software evolution in the context of the development of the UBC Continuous Media File Server. We first discuss the issues related to the hardware/software environments and the opportunities/limitations they offered the development team. Then we elaborate on the implementation and evolution associated with the various interfaces the system has to the real world and to other software components. The evolution of the server with respect to the details of the services and functionality desired in the system is next investigated, followed by the experience and evaluation with respect to performance issues. We conclude with some lessons learned about prototype development and maintenance of a distributed server in the context of a university research project intended for external collaboration. The focus of the paper is the intentional design for evolution and an analysis of the process of subsequent changes that occurred. Many of these changes were anticipated in the structural design, which made them straightforward to implement. Unfortunately, maintenance changes due to performance problems were neither as easy to anticipate, nor as straightforward to solve.

ENVIRONMENT AND ENVIRONMENTAL CHALLENGES

The four most important goals of the server were real-time streaming, variable-bit-rate support, heterogeneity and scalability. Other projects and products existed at the time which could serve constant-bit-rate video files and had a fixed capacity [3, 4, 5]. This did not seem interesting from a research point of view. The first two goals influenced the algorithms and the storage design significantly, while the last two goals influenced the architecture and the storage design.

The CMFS was intended to be a product/infrastructure that could support various client applications and multiple data formats. We knew from the start that this would require some change to accommodate these new issues, but we also designed our architecture to require as little change as possible for new data formats. As well, the server was to be deployed on heterogeneous hardware; thus dependencies on specific network protocols and device characteristics were avoided. The result was a software architecture that was highly modular, but were still able to meet its real-time performance objectives.

In particular, we selected the following characteristics to include:

- isolate hardware dependencies to a middleware layer
- provide a self-contained client API and library which requested services based on a message passing protocol
- design a storage system that had no dependencies on encoding formats
- component design that permitted the execution of software components on separate, heterogeneous hardware with a seamless interface and scalable performance.

These features were key factors in the design process that followed. We believed that adherence to these principles would ensure that the major goals and that of a flexible user interface could be maintained. Changes to the API could first be tested on a client platform and then integrated into the server.

The initial intended application was a News-On-Demand prototype. This was somewhat different from the general video-on-demand environment that was the main focus of the research community in the mid-1990's. In particular, many news stories are shorter than 5 minutes and are required immediately, whereas a feature-length movie or television show tends to be an order of magnitude longer in length, implying the possibility of different user preferences in terms of latency, jitter, etc.

Support Layer Software

Modular design was a key component of the system, but the real-time performance of object-oriented environments was a frightening prospect. We decided that very lightweight objects would be necessary and chose to pursue object-based design which was supported by a threaded programming environment.

Thus, we required a real-time operating system and threads programming environment that would be capable of resource reservation and delivering data from the server to the client with real-time guarantees. Several appropriate environments existed, although they did not have the same level of accessibility to the research community. User-level and kernel-level threads packages were exploding in popularity at the time and the POSIX Threads API was becoming standardized. Most of these packages were in very early commercial stages, or were university research projects. The very first version of Solaris had recently been released which contained SUN's version of POSIX threads. Though they claimed that real-time capabilities were available, they actively discouraged the use of real-time facilities in application level code. Further research [6] has shown that the schedulers in this system could lead to total system failure in an overload situation. AIX had real-time process scheduling, but it could only be used by the superuser.

No kernel-level threads abstraction had real-time scheduling capabilities (with the aforementioned exception of Solaris). AIX did not provide scheduling control at the thread level. We also thought that the learning curve involved in commercial Real-time Operating Systems (i.e. QNX and others), would be significant, but perhaps unavoidable. Selecting a commercial system also restricted our ability to deliver the software to the research community as an open-source project.

Performance was also a concern. One of the particular reasons we chose not to use native threads in the AIX operating system was the results of some early experiments. These indicated that the native AIX process synchronization primitives were an order of magnitude slower than the equivalent operations on the same hardware with our own user-level threads package.

Previous research in operating systems in our department had produced several useful software artifacts. One of these was a simple and efficient threads package (UBC pthreads [7]) that had low overhead and implemented process synchronization, as well as intra-address space inter-process communication and used a round-robin CPU scheduler. There was familiarity with the API and a substantial code base of sample applications. Also at our disposal was a real-time scheduling system provided by another Ph.D. thesis [8], that investigated performance of real-time scheduling on multi-processor and uniprocessor systems. We decided to merge these facilities into a package called Real-Time Threads (RTT)[9] over which we had complete control and access to the source code and original developers. The threads in this package are user-level threads within a single process which can communicate with threads in the same process or in other processes. The modular design of pthreads itself enabled the new scheduling algorithm to simply replace the existing algorithm. We added 32 priority levels, as opposed to three, and incorporated deadlines and starting times. Processes were scheduled to run to completion according to priority and the Earliest Deadline First algorithm (EDF) selected between equal priority. The starting time was used to place a process in a non-runnable state, by setting its starting time to some time in the future. Completion of this integration was remarkably simple, as the design used by the previous authors adhered to the same layering concept. This task was completed in a few days, though further enhancements continued during the remainder of development. The most significant was the addition of Asynchronous I/O into the support layer. Here we had a clear example of software reuse with ease. The functionality was slightly altered and a greatly enhanced software package was the result. It continues to be used in courses at UBC and in some research environments [10].

We envisioned a system that would be able to simply add a disk device and increase both the capacity and the deliverable bandwidth. The modular design allowed us to have a thread per disk, so the management of I/O on each disk could be completely separate. The only problem with this approach is that disk I/O reads are normally blocking operations. Adding another thread which makes a blocking request would not increase the bandwidth.

We needed an asynchronous I/O interface to permit multiple outstanding disk requests to be generated from a single process. Only IBM AIX and Solaris provided this feature in their operating system interface at that time. A user-level process may initiate several disk requests simultaneously and asynchronously wait for their completion. This is necessary even for a single disk system as the application cannot know the optimal order of disk requests. This should be left to the device driver and the disk controller.

Other operating systems have recently provided AIO facilities. Windows NT has had this for a while, as does FreeBSD and Linux, though these facilities are not well-tested, nor widely deployed. Recent experiments done in our lab confirm that there is a substantial difference in AIO read performance between the implementations of FreeBSD, Linux, and Solaris on the same hardware. This leads us to believe that there are some fundamental flaws in the implementations of this facility on one or more of the operating systems. Although AIO in Linux conforms to the POSIX interface specification, it is not a complete implementation from a single vendor/implementor [11].

One particular area of expected change was with respect to the network protocols used. There was no clear choice for a real-time data transport protocol. The initial version of the real-time raw data transport protocol was XTP (eXpress Transport Protocol) [12]. At the beginning of the project, it was a promising real-time protocol with facilities for rate based data transfer and the guaranteeing of bandwidth on a per-connection basis. It was intended to be implemented over ATM networks where the signaling protocol for ATM could provide a constant-bit-rate or variable-bit-rate connection. This was used for sending the video streams from the server to the client. The protocol for storing the data in the server and communicating the parameters of UI requests needed to be reliable, so we chose to use TCP/IP. Message passing details were isolated to RT-Threads as communication was done via a Send-Receive-Reply protocol.

The applications required knowledge of the various XTP parameters involved in connection set-up and data transfer. The API simply had a data structure in the call to *CmfsOpen* that contained a protocol independent set of transmission parameters (heavily influenced by what XTP required) and implemented the specific connection negotiations in a call back mechanism wherein the client requested a connection from the server. The streamDesc parameter could also contain protocol-specific information. Thus, any change to the protocol had a very localized effect.

Two factors soon became obvious: the research and industrial community was not quickly embracing XTP, and XTP provided functionality that was not strictly necessary nor appropriate for the CMFS. Thus, a search for a replacement protocol began. RTP was in the beginnings of formation and seemed like a reasonable choice. Since it required some level of understanding that we felt required substantial research, we postponed including this in the next version of the server, opting instead for our own media transport protocol (MT) [13].

MT was based on quite a simple idea, but proved very useful. It consisted of adding sequence number information to UDP packets at the sender, so that the receiver could tell which packets had been lost. Thus, data received at the application may not be complete, but all correctly received information would be in the correct place. MT borrowed most of the necessary functionality from XTP and UDP, and was extremely lightweight. Placing different protocols inside the CMFS was straightforward. Their scope was restricted to the connection establishment call, the transmission of raw media data (stream manager), and the call to *CmfsRead* at the client. In the stream manager, a connection status bit was used to determine which protocol had been used in the connection setup and the appropriate "send" call was used.

After MT was fully functional, the exploration of RTP continued. This was done in parallel with performance tests on the server using MT. The real-time data stream was encapsulated inside an RTP stream, using RtpWrite in the Stream Manager. The client application could take advantage of the fact that RTP was used, which made timing simpler. An important lesson of the evolution was that isolating the network functionality is relatively easy to do if there are no required changes in the semantics.

The server goal of heterogeneity was to be shown by developing clients for many data formats and many operating systems/hardware platforms. Additionally, the server could run unmodified in many H/W and 0/S platforms. During implementation, we ensured that the server functionality we desired was available on nearly every operating system variant we had at our disposal. The main method we used was to perform extensive tests on the RTT package and eliminate our use of native operating system calls within the server itself. For the first testing of the entire system, three main UNIX variants were employed: AIX 3.2.5 (for the server node and IBM display client), SUN OS 4.1 (for the Parallax display client), and SUN Solaris 2.1 (for the server administrator). In this phase, clients were also implemented which displayed MPEG-1 and Quicktime in the UNIX environment.

The heterogeneity and scalability goals required network communication between the different major software components, since they were implemented as separate processes which could be resident on different computers. The original pthreads package provided inter-thread communication primitives that were able to provide both synchronization and data communication within a process. We modified the semantics of the Send/Receive/Reply procedures to copy data from a buffer in the user space of the sending process to a buffer in the user space of the receiving process. If the destination process was on another machine, this required some way of identifying the IP address and process ID, in addition to the thread identifier on the remote machine. We implemented a name-server capability that would return a thread identifier on a remote machine, so that the interface to the remote thread remained the same as to a thread in the same address space, thus simplifying the design and lessening the burden on the application programmer to know about remote versus local threads. All that was needed is to establish the IP address/port number pair of the machine which served as the name server at the initialization of an application. This was implemented as a simple addition to the thread structure control block, and involved including the IP address as a hidden part of the thread identifier (tid), so that the message passing procedures knew where to send the network packets.

As mentioned earlier, we believed the server gained a substantial amount of performance by allowing the disk controller to do what it does best: schedule disk reads. This was incorporated into an RTT facility: RttAIORequest() and RttAIOWait(), with the variant-specific code hidden at the lowest level possible. The reads were issued in parallel, and then returned in whatever order the system could perform them. A check on the status bit of each request was subsequently performed in the RttAIOWait call(). This was the only case where the direct needs of the server influenced the implementation of the threads package. Other functionality was seen as generally applicable to distributed systems.

This integration was also easy to achieve and was completed before the majority of the server design had been worked out. A high-level description of the server application-level requirements enabled the generic support layer to be established and tested in detail.

One of the developers of the system was responsible for the development of a display client for Windows, based on the ReelMagic (tm) decoder card. This work was done in conjunction with porting the server to Windows. Nearly all of the porting was confined to the details of the RTT package, as we had abstracted the operating system dependent features out of the majority of the code base. Several issues came up which only manifested themselves in the Windows environment. This led to additional incremental modifications to the threads package which altered the performance behaviour of the server. No substantial change was necessary to server code. In this part of the work, the server was also ported to Windows NT, while the client was done in Windows 95. Further experimentation with various applications similar to the server were incorporated in this student's Master's thesis on RTT [14].

The most significant observation was that occasionally a control message consisting of a single packet experienced an unusually long latency. If there were no transmission problems, the average round trip time was less than 10 ms, but on occasion it was over 3 seconds. We soon discovered that this was a problem with the TCP timeout behaviour if the SYN packet was lost. In the UNIX environment, this was never observed, but in Windows, though it was rare, it was enough of a concern to be considered. To combat this problem, we implemented a "reliable" UDP protocol for these short messages that had its own acknowledgment and timeout mechanisms, which were much more efficient than TCP for these short messages. In order to implement this mechanism, an additional compilation flag was introduced to have the control messages not use the regular RttSend mechanism, but an RttUdpSend call. From this we clearly learned that not all TCP implementations are created equal. Certain features of the protocol stack, or default parameter configurations may require changes to the way in which the support level software is coded. This is still part of the application layer, but fortunately, we were able to completely isolate this to the networking module of the threads packages. If we were to generalize, it would be to not trust the performance of networking code in general, on many operating systems. It is often also the case that substantial portions of the operating system have not truly been tested for performance on particular types of applications and special "workarounds" are necessary.

The issue of meta-data management was the primary responsibility of collaborators at another university. We required that the system be capable of standing alone or interacting with a sophisticated multimedia database system. We chose to use a simple transaction-based flat-file based database system called tdbm [15], which was also locally developed. It was UNIX-based and open-source and therefore capable of being ported to the same architectures for which we developed the CMFS. The database functionality of the CMFS were abstracted into two calls: *CmfsPutAttr* and *CmfsGetAttr*. This implementation made it possible for an evolutionary change to use another database system, potentially one which supported ODBC for accessing an external database that may have a more complicated data model.

Hardware Issues

The generic nature of the filesystem design was intended to insulate the hardware access layer from the application. Any optimized data layout mechanisms would have to be performed on the granularity of entire disk volumes and not on the track/sector/block level.

High-quality continuous media data needs to be delivered in real-time to the human user. The large size of video objects (30 frames per second, at 640 x 480 pixels) precluded the use of a download and play protocol which would have the buffering and latency problems previously mentioned.

In order to achieve real-time display of full-motion, full-screen compressed video, hardware decoding was essential. Thus, we spent early design time becoming familiar with the hardware decoders available and the software development environments and APIs that accompanied them. Two such decoders for Motion JPEG [16] were chosen: a S-video board by Parallax for the SUN environment and an Indeo board from IBM for the AIX environment. We had a contact at the video card manufacturer who aided us in the configuration and use of drivers. We found that the expertise of the driver designer was of immeasurable benefit, since the implementation of the first version of the Parallax display client utilized insider knowledge. The work done for this display client made the work for the IBM display client much simpler.

For demonstration purposes, we required a client which ran on IBM hardware. The organization of the software modules was the same as on Sun hardware; the only difference was the call to decode the M-JPEG frame and display it to the X-window. In addition, the IBM decoder card was capable of displaying directly to an NTSC device, so we utilized this function in the client to display to a television monitor. Almost no other changes were required, and the code was ported with ease. Over the passing year, the original hardware decoder based players have become defunct, either due to hardware failure and/or operating system obsolescence. No one had the energy/budget to keep these machine functional. As a result, there currently is no working display client, and this is the subject of sporadic effort.

INTERFACE ISSUES

User Interface

We had four major user interface requirements: virtual VCR functionality, negligible playback latency, the ability to synchronize multiple streams, and guaranteed real-time transmission of the media data from the server. These were complicated by the fact that the data had a variable bit-rate profile. Architectural design decisions for the first three issues are given in this section, while implications for the delivery guarantees are discussed in [17].

The mode of interacting with a Continuous Media Server was to be consistent with VCR usage. Thus, we supported user functionality that would permit the use of the buttons on a typical VCR: *play*, *stop*, *fast-forward*, *rewind* and *record*. The *play* function requires real-time delivery of the video/audio to the client workstation for decoding and presentation. *Fast-forward* and *rewind* are similar, but could be performed with additional bandwidth or some clever selection of content to transmit to achieve the same effect. *Stop* is trivial from the user's point of view, but does introduce some complexity at the server. We chose not to implement the *record* function explicitly (mostly due to the inability to easily record in real-time), but the API does have an interface for storing data to the server.

A human user's interaction influenced both the API and the information kept at the server, in terms of object attributes and state information for open connections. For example, the desire for alternate playback rates and the ability to have a flexible choice in starting positions led to the concept of "sequences". As well, if fast-motion was requested, this could be accomplished by retrieving a subset of sequences. Both of these mechanisms are used in current DVD players. It would be possible to achieve double playback speed by retrieving alternating sequences using roughly the same bandwidth as normal playback. This had a large effect on the server design at the file-system level, as well as in the protocol used to send data from the client to the server. While this cannot be considered evolutionary, it did constrain the design at other levels.

In other video server projects, a distinction was made between on-demand and near on-demand. In the latter, the high level of bandwidth required and variation in bandwidth requirements were addressed by client and/or server buffering that resulted in playback latencies for the user. One study [18] suggested that bandwidth could be significantly reduced by introducing buffering at the client and a delay of about 30 seconds. In our context, this level of playback latency was not acceptable.

Programmatic Interface

The server API can be divided into 3 groups: storage of media data, retrieval of media data, and management of media data.

- The data storage calls are: *CmfsCreate*, *CmfsWrite*, and *CmfsClose*. *CmfsCreate* takes parameters regarding the size and format of the object and returns a unique object identifier (UOI). This UOI is used in any retrieval operations to locate an instance of a media object. The object is written reliably to the server with calls to *CmfsWrite*. If all writes complete successfully, then *CmfsClose* is called to commit the transaction and write presentation attributes to the server. The notion of a sequence is user-defined in that it can be any multiple of display units[†] considered as a unit. Within an object, sequences could vary in length, based on the concept of a "scene" in the presentation, or could be a fixed length of time.
- The retrieval calls are: *CmfsOpen*, *CmfsClose*, *CmfsPrepare*, *CmfsRead*, and *CmfsStop*. *CmfsOpen* and *CmfsClose* are fairly similar to the corresponding calls in a normal file system. To open a file, the UOI is passed as a parameter. If the object is located, a real-time data connection is established between the server node and the client machine. A connection (*cid*) is returned to the client application. *CmfsStop* is called when the user terminates the delivery, and *CmfsClose* closes the connection.
- For management, the calls are: *CmfsPutAttr* and *CmfsGetAttr*. These calls allow a user to store meta-information, which includes attributes related to resource requirements of a stream.

The most significant interface call is *CmfsPrepare* which calculates the bandwidth requirements of the stream over time, determines disk and network admissibility and informs the respective disk manager of the schedule of disk reads for the new stream. This call also enables the flexible delivery of the real-time data. Two parameters alter the delivery from the default pattern: *speed* and *skip*. The *speed* parameter reflects the presentation rate in relationship to the recorded rate while the *skip* parameter can be used to achieve fast motion without an increase in bandwidth; a skip value of 1 indicates that one client defined sequence (as described previously in the CMFS storage API) is to be skipped for every one retrieved and transmitted.

A fourth group of functions was added later in the project. These provided unit conversions and utility wrappers [19]. We attempted to keep the number of API calls to a minimum and only

[†]typically sound samples or video frames

added the wrappers when it became obvious that the developers of client applications required additional facilities. This happened frequently during the course of the project. Although the basic API remained unchanged, this fourth group of functions underwent the most change. They were the easiest to change, because they did not require additional information to be stored in the server, but merely provided convenience for client applications. Changes to the first three groups of API calls were not anticipated, because we thought we had covered every reasonable possibility.

The structure and the flavour of the interface remained constant due to the substantial initial time invested in design and early experimentation. While we did not have an integrated system yet, we believed that we had encompassed the major server goals.

One concern that the client application did have was the specification and negotiation of real-time delivery parameters: the bandwidth maximum, burst packet size and frequency, as well as client buffer space needed for continuous playback using a double-buffering scheme. It took a couple of design iterations to determine how to specify this, but we finally decided that the API would have a single parameter, which was a structure named *StreamDescriptor*, that could have both generic and format/protocol specific parameters, depending on possible later evolution. This stream description was used in a call back facility.

In the course of developing our generic view of the API, we began consultation with the research groups at the other major universities involved in the project at the annual research conference for the funding agency. One of the things that was confirmed at this meeting is that it is a mistake to move too early into the implementation stage when the interface has not been agreed upon. Since the funding was not in place and the design team had not been finalized prior to this meeting, our server facilities were specified and designed, but there was no "official" implementation. Our initial work was done in isolation from the other groups and as mentioned previously, much time and effort was expended in placing ourselves in the position of both a client application developer and a human user. We were convinced that we had a cohesive and coherent set of fundamental abstractions that would provide all the necessary capabilities. We had also determined a software infrastructure that enabled the performance goals to be met.

Unfortunately, one of our collaborators had begun the implementation phase before any detailed consultation with our group or any other group. During these and subsequent discussions, it was clear that we had superior and more general notions of what such a media server could/should do in conjunction with our research objectives, and this conflicted with code that had already been written by the other university. Their implementation had begun in C++ and made assumptions about the facilities they would be able to receive from the server. In particular, they assumed a file-access metaphor, and did not consider the details of the transmission protocols and timing implications of our server design. This could have been irrelevant, if there was a suitable middleware platform available. The specs for early versions of CORBA were available, and there were a few available implementations. We had chosen not to use a middleware platform, but rather to use C as our implementation language alone for two main reasons: all the object-oriented functionality we required could be achieved by the use of RTT, and we had grave doubts about the real-time performance of middleware products.

Research objectives/budgets did not immediately allow for such a radical shift in perspective for either group, so the initial burden fell to the university in charge of integration to solve this problem. They came up with a “glue” mechanism that translated client calls in C++ to the appropriate API calls to the CMFS, a veneer which functioned as an extremely light-weight middleware. Unfortunately, though several man-months were spent on this “glue”, no amount of retrofitting could make the pieces of software inter-operable. Eventually, the basic outline of the client application was redone from scratch.

Since the server had a number of potential client applications, it was necessary to be in frequent dialog with the developers of those applications. This led to evolution of the API within the context of the general infrastructure as previously described.

MAJOR COMPONENTS AND THEIR IMPLEMENTATION

Server Architecture

The server has features which support the real-time guarantees of data delivery and the reliable storage facilities of the server. We provide an overview of the architecture and the design of the individual components so that we can further explain the evolutionary changes which took place over the next number of years.

The design of the file server is based on an *administrator node* and a set of *server nodes*, each with a processor and disk storage on multiple local I/O buses. Each node is connected to a high-performance network for delivering continuous media data to the client systems (see Figure 1).

Disk drives retrieve media data at a rate which fully utilizes the network interface of the server node. Multiple server nodes can be configured together to increase the capacity of the service and nodes can be added dynamically. A similar architecture is used in other scalable, high-performance video servers [20, 21]. Subsequent development of these types of servers has continued this trend. In particular, the Yima server [22], uses lessons that were learned in previous work at USC [23] and has a similar architecture to the CMFS, though some of the details of the scheduling algorithms differ. The Yima server has taken similar ideas of server design and carried the evolution further than the CMFS at the present time, with substantial effort in hardware investment and protocol enhancements. This system has become more complex than the CMFS in its current form. One major advantage in the evolution of the Yima system is that they have eliminated the single point of failure, which is the administrator in our current system design, and instituted an entire layer of fault-tolerance.

The design of the CMFS was split into three parts: Client Interface, Administrator, and Server Node. The software organization is described in Figure 2. The CMFS internal design is based on two system design principles: the first is the administrator model of message passing between client and server applications and the second is the periodic time unit (called a “slot”) to coordinate resource allocation and delivery mechanisms. The “slot” terminology (alternatively called “service rounds”) is standard in video server implementations.

Four main types of threads are created at the Server Node: one Node Manager, one Network Manager, a Disk Manager for every logical disk device connected to the node and a Stream Manager for every open client connection. The responsibilities of these software modules are described more fully in [24]. A client application is shown in the figure as having multiple

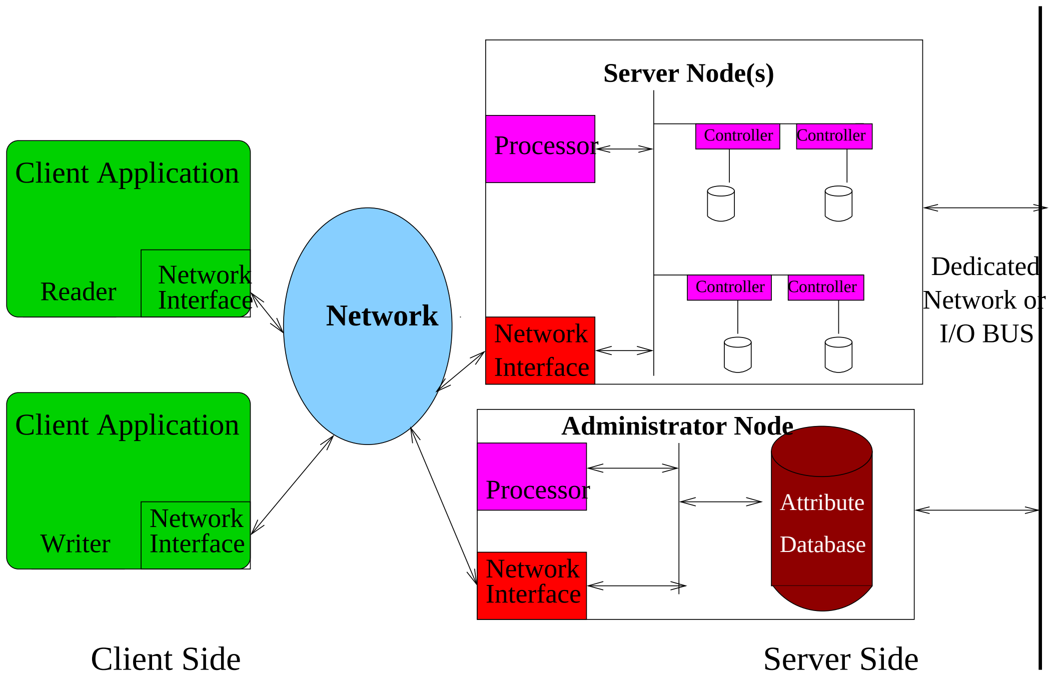

Figure 1. Organization of System

threads, but this is not strictly required. Client applications that choose not to use the threaded model must link with a non-threaded CmfsClient library.

The administrator receives messages from client applications and server nodes. The server nodes register with the administrator and clients calls are routed back to active server nodes. Separate database worker threads are created to access the database and return a result to the main administrator thread to prevent the possible blocking associated with concurrent access to the database.

The server node maintains state about active connections and the data which is being transferred. The four main threads at the server node share data structures to ensure that resources are allocated in a manner which guarantees delivery of the data from the server node.

The major control thread at the node is the "node manager", which sits in an event loop waiting for client messages. If the request requires a computationally intensive operation, the request is placed on a queue for a periodic thread (e.g. "the disk manager"). The disk manager thread and the stream managers, as well as the network managers all work on the periodic basis of the slot. The disk manager has two main tasks: performing admission control, and scheduling reads/queuing disk blocks. It requests read operations in units of what it could guarantee in a

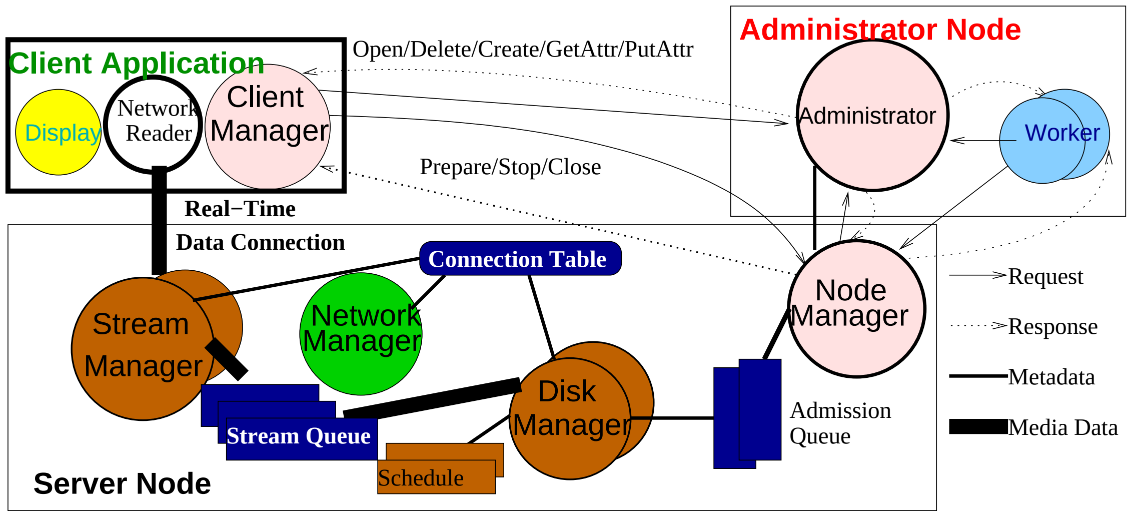

Figure 2. CMFS Software Organization

slot. The network manager allocates bandwidth to each of the connections based on a slot value. It first allocated the required bandwidth and then allocated excess bandwidth according to future requirements and the amount of data read ahead for each stream up until the maximum of the network interface. The stream managers filled the individual network output pipes at the beginning of every slot, leaving the implementation of the network protocol to perform the actual sending of the data.

Armed with this general infrastructure, a Master's student involved in the project took a weekend to develop a crude implementation of the server (version 0). Not all of the desired API was included, but we did have a client-server application that retrieved video across the network and had the functionality of playing the video object from the beginning to the end. The client did no decoding; the video object stored at the server was composed of decoded video frames and stored in the normal Unix file system as a series of files, each of which contained one frame.

The basic connectivity was achieved for the five (5) calls absolutely necessary for client display: *CmfsOpen*, *CmfsPrepare*, *CmfsRead*, *CmfsStop*, and *CmfsClose*. The interface was thereby proven in principle, though there was no admission control, no real-time data transfer and only a general purpose file system. To transmit the data, a file open call was performed at the server, and an RttSend() transferred the frame as one message (using TCP) to the client, which subsequently displayed the frame in an X-window. Even the most basic design/organization of the server was not implemented in this version. Only the basic message reception loop was designed without threads to handle any of the necessary parallelism. Fortunately, this allowed us to examine the functionality of the RTT package at the same time. We had a baseline system from which development of the various client and server components could take place. One student worked on the client development, one on the server, and the other on the support software. The next weeks saw multiple discussions on the design requirements as the three components interacted with each other.

The several hundred lines of code in the first beginnings (version 0) provided the structure for the server node, as the desired implementation details were added incrementally. These were: file/disk layout, concurrent thread execution and distribution of server functions between server components. Within four months of one developer's effort (the Ph.D. student), the admission control and data retrieval algorithms were incorporated along with the allocation of network bandwidth for the transmission of data. We called this version 1. This version was used at the initial demonstrations. We provided several demos of the server to local students and fellow faculty. The CMFS quickly became the focal point of our research group's annual Open House, where its performance was far superior to Real Networks software at the time (early 1995).

Only a few of the original design goals remained unfulfilled. Since our demonstrations and hardware environment consisted of only one or two client machines, the demonstrations did not stress the scalability of the system. Network admission control and buffer management were the next facilities to be implemented. By the end of the summer, 6 months later, we had achieved version 2 of the software. This version was stable from the design and interface point of view, but was constantly being modified as inefficiencies in the support software (RTT), internal data structures and memory usage issues were uncovered.

An issue with the disk manager was handling the reclaiming of buffers and interaction with the network manager and stream managers. The admission mechanism simulates the reclaiming of data that had been read ahead to see if reordering the reads can permit a new admission, but does not actually reclaim this data until absolutely needed. Thus, in the worst case, a block required one slot in the future could be reclaimed in favour of a block needed in the current slot. The network manager may have chosen this block for early transmission. This resulted in concurrency problems, in that the reclaiming algorithm had access to the address of a buffer that was being sent by the stream manager. A lack of careful pointer management caused two problems. Occasionally data was sent from a buffer that had been reclaimed, or a buffer was reclaimed that was still in use. This required substantial debugging and extra checks, because this issue was not foreseen. Many person hours of work was required to solve the buffer concurrency problem, and the code became convoluted and difficult to maintain, so we have avoided returning to this problem. We believe that a complete rewrite of the buffer management software is in order.

The network manager also had some interesting evolutionary issues during initial development. The main concern of the network manager is to assign transmission bandwidth to each stream manager. This involves admission control and bandwidth allocation. There were several choices for network admission control, as described in [25]. The bandwidth allocation was done by assigning credit for the number of bytes required to be sent for each connection during the current slot. If that were the only task of the network manager, it would be a straightforward process. The situation turned out to be not that simple, because we realized that there were optimizations that could be performed. If the client had sufficient buffer space and there was bandwidth on the connection and at the server, the network manager would increase the permissible bandwidth for sending ahead. This required an examination of future slots, and required that the data corresponding to this bandwidth credit had actually been read off the disk by the time bandwidth was updated. The resulting change in server code was significant. Though the concept of reading ahead had been incorporated in the initial disk delivery/admission scheme, it was a new feature in the network subsystem.

In particular, this exposed a race condition between the network manager and each stream manager and thus required synchronization control. Handling the network bandwidth assignment and the timing of stream control operations resulted in some desperate coding and virtually unmaintainable details. For the most part, bandwidth is assigned properly, except rarely at the end of a stream, where the network manager seems to get confused as to which streams are active. A careful re-implementation of this module, which is completely self-contained (all code is in the file *network.c*) is certainly within the realm of a class project.

The problem occurred rarely, and so a priority was not placed on achieving robustness; rather the effort was placed on performance evaluation for the common case. We perform short-lived performance tests that verify the real-time objectives, but occasionally the delivery of a stream could not be properly terminated. This version (version 3) of the software has stayed relatively in tact, since that period of time when performance experiments began to be performed. The details of changes required for performance evaluation are given in Section .

Further evolutionary growth was expected in our design. It has not yet been fully realized. In particular, the server utilizes buffer space for reading ahead to accommodate fluctuations in the required delivery bandwidth, but does not explicitly take caching into consideration, nor the option of multicasting the data across the network interface. Such enhancements could be made to the basic model, further increasing the number of simultaneous users that can be serviced from a server of the same scale.

Multicast could be added by creating a multicast channel in the *CmfsOpen* request. It would require a substantial change to the data structures at the server node if Application Level Multicast was implemented, as group membership is handled by the application. If IP Multicast were used, there would be very few modifications as all that is needed by the server is an IP address for the destination of the data.

Client and Integration issues

The first display client used a hardware Motion JPEG encoder/decoder card from Parallax on the SUN OS system. The work was carried out by a research associate and did not use any of the code of the server. It made use of the same interface, but server calls were mapped to the local file system, or were stubbed out entirely. In the course of this work, several interface parameters were identified that were missing in the original design. Fortunately, no new calls were required, as most of the parameters were client specific, and could be achieved by a call back mechanism, whereby the client could obtain the local information needed. This allowed true parallelism in the development process. No implementation bugs in either the client or the server impeded the progress of the other component.

The first client application of substance was the Parallax display client. We considered client synchronization from the point of view of audio, video, and text streams. The most sensitive aspect of the presentation to the human user is audio, so this was made the master stream and it controlled the timing of the other media. At the end of every second of display, the streams were re-synchronized. If transmission problems or corrupted data caused some video frames to be late during this synchronization period, frames were discarded and the first video frame of the subsequent sequence was decoded and displayed. Synchronization required timing information with respect to the data it received. In particular, frame sizes were used to determine the start of the first frame of the next synchronization period. This was stored in the server as an attribute at object creation time. This required no changes to the server code, as an attribute storage mechanism was already provided. There was a performance issue as several more kilobytes of storage in the database were required for each object.

By the time integration of components began, the code had been sufficiently tested so that fixes were less frequently needed, and changes were the result of design changes. While we had intended to use some type of source code control, it was not until this point in the project that more than one person could conceivably be involved in modifying the same group of source code files. Thus, we instituted the use of CVS, a revision control system on UNIX. There were at most 2 people working on the server at any one time, but even this provided the need to coordinate the changes using some technology support.

After the integration for one client type and the server was completed, several other software decoding clients were developed for proof-of-concept of the heterogeneity support of the server. Certain similarities in the design were recognized, and a template for client applications was developed. The Parallax client was rewritten using the generic Client Library support. The CMFS Client library was documented and left as legacy code for developing further client applications. This dream has gone unrealized, because there is substantial subtlety in using this model. This was made as part of a summer research project which was unsuccessful in completing the task of writing a client which would serve as a proxy to a software decoded application, such as MPlayer.

One particular client had requirements that exposed some major inefficiencies in the admission control algorithm and its use of internal data structures. The admission control algorithm [17] initially assumed that the sequences of data stored in the server were supersets of the data for a single slot. This made the data structure reasonably efficient. It was rare that data for a slot was noncontiguous on the disk, but this would require a dynamically allocated structure to be used in the algorithm. The potential commercial collaborator desired very fine-grained sequences: one display unit. If the entire stream was presented to the user, this did not cause any trouble, because each slot was likely composed of contiguous bytes on the disk, and a single array position could contain the necessary disk location information. Unfortunately, if a perfectly smooth fast forward delivery was to be obtained (speed 100, skip 1), then there would need to be 15 data structure elements (corresponding to the 15 display units for 500 ms of a 30 fps object) per slot. The original implementation allocated each extra structure as required, and this fine grained memory allocation greatly increased the execution time of the algorithm. Some precomputation of the number of additional structures that would be needed and the one-time allocation (per prepare) would be much more efficient. This was an unanticipated change that would have required simple design in an isolated area of the code. For various reasons, the collaboration was short lived and this change has still not been implemented. It is a very peculiar requirement, not even implemented in today's DVD players.

The technology developed for the CMFS was shown in an integrated fashion at several conferences. The developers from the various subunits arrived several days before the conference to ensure compatibility of the components. The code for the server was the most stable of all the components, primarily because the clear interface and division of responsibilities made it self-contained. A minor portion of the server was located in the interface library which was linked in to the client application, but this was extensively tested at our development location and did not require any maintenance during the demonstrations.

During the first demonstration of the News-On-Demand system, we restricted ourselves to IBM hardware as supplied at the outset of the project, as this was believed to require the least integration effort. Since this was a university project, the hardware budget restricted us to one computer as the server and one as the client. The hardware configuration for the demo did not stress any of the capabilities of the server, as it was designed to scale well with many heterogeneous clients, but it did provide a public venue for our proof of concept. Subsequent demos utilized both SUN and IBM hardware to further demonstrate the heterogeneity of the system, and to show that the server could easily support multiple encoding formats at different resolutions and frame rates. The SUN client displayed at 30 fps with 640x480 pixel resolution, while the IBM client displayed at 15 fps and 320x240 pixel resolution.

The media synchronization group requested changes that would be compatible with their vision of the project. Their model of synchronization was more extensive than ours. This had multiple video and audio objects following a multimedia document presentation scenario which was considerably more complex and allowed a designer of a document to select a future time to transmit a particular media object.

They wanted a “prepare in the future”. This led us to add another parameter to the call to *CmfsPrepare*, which was “starting time”. In this way, a document scenario could be scheduled, with bandwidth being reserved for several objects in sequence (or in parallel) for which all of the admission control would be done at the beginning of the document scenario. This violated one of our design principles that the execution time of *CmfsPrepare* would have an upper bound. We could only provide an upper bound for a single stream. This change required some effort, but it was minimal. The disk admission control algorithm added values to an array corresponding to each slot in the future until the end of the playback time. A prepare in the future has no implications for the design of the server. The values to add were merely offset from the next slot in the future to the actual time of the delivery of the object. This shows that the most flexible and powerful API call was also the most controversial. *CmfsPrepare* underwent the most significant changes of any API call, but these changes always had a small effect on the remainder of the server code.

Our effort in modularizing the interface, the retrieval, and the server node communication enabled us to keep the complexity of the software at the same level throughout the development process. Individual components may have had an increase in algorithm complexity, but the loose coupling of design features was evidence that we had a process in place to prevent uncontrolled complexity increase [1].

PERFORMANCE AND SCALABILITY ISSUES

Network Bandwidth Reservation

Since we had designed the server to be implemented in a network environment where the bandwidth could be reserved, the initial network connection establishment had hooks in it for retrieving a bandwidth guarantee from the network. This could have been via RSVP [26], or ATM virtual circuit establishment. We did not commit to any particular network enforcement as RSVP had just been introduced and early ATM devices did not implement bandwidth policing. There was earlier work [27] indicating that this process was relatively inexpensive for ATM networks and the scalability problems of RSVP for end-to-end connections with many hops in the context of the global Internet had not been fully realized. Our concern was not with the inner workings of the network, as each server node would be reserving no more than a few dozen connections at any one time, each of these reasonably long-lived. Any solution to the scalability of resource reservation in a packet-switched environment could have been utilized in the server. In an intra-net with network connection reservations that can be enforced, or with appropriate resource reservation in the Internet (e.g. DiffServ [28], Intserv [29]), guarantees can be extended past this boundary. The Internet, in general, never did implement bandwidth reservation mechanisms. Each of these mechanisms attempts to treat some levels of traffic with higher priority than others, while still maintaining guarantees with respect to delay and jitter. In a "best-effort" network like the Internet, it may be the case that the bandwidth which was reserved at the server (without any ability to enforce that reservation in the network) cannot reasonably be maintained. The users of the system will become quickly dissatisfied with the presentation quality if frames are late, and video display no longer remains smooth.

What changes were required to the server design due to the inability to apply such reservations in a real environment? To be certain, the quality must be adjusted. Either a lower frame rate, or lower spatial resolution must be sent, or both. Without extensive monitoring of the network or a clear profile of the user's preferences, the server has no idea what the appropriate action to take. Any action could either be ineffective at increasing the user's perceived quality, or even counter productive. Advances in computational power have made it reasonable to have the server do more processing and provide an adaptive stream given some policy direction from the client.

We believed that the right place for the quality adjustment to take place was at the client. To do this in the initial design required the stream to be stopped, and re-prepared (via *CmfsStop* and *CmfsPrepare*). There was the remote possibility that the new prepare request could fail, even though it requested fewer resources than currently held (in the case of downward adjustment) if a request from an unrelated user was processed during the time between receiving the stop confirmation at the client and the next prepare request. To avoid this condition, it was necessary to implement a new call for the client API called *CmfsReprepare*. This performed the stop and prepare in one atomic action. If the new prepare part of the operation failed, then the same quality of delivery continued.

The simplest manner in which to do this would be for the client stub to issue the stop and prepare as before, but this does not solve the race condition. It does ensure, however, that the interface can be tested. The implementation of *CmfsReprepare* at the server required adjusting the proposed delivery schedule at the new rate for this affected stream and re-running the admission control algorithm.

This assumed that the client has an appropriate way of determining what a new quality request would look like, in terms of object id, speed, and skip parameters. *CmfsReprepare* would use the same connection and refer to the same high-level UOI, but whether the same copy of the object would be retrieved from the same server is not quite so clear. The details of this part of the implementation and performance evaluation remain as future work. The simplest change would be to use the same object and only the speed and skip parameters would be affected.

Upon operating in degraded mode for some time, it is possible that the bandwidth that was unavailable earlier may now be available. The quality could be increased in a similar manner to how TCP increases its sending rate, until the original quality of the object is restored. We also left this as a client obligation, though it could be implemented automatically instead of being triggered by an explicit user action.

Scalability Issues

A server system with only one node and one network interface has severe capacity limitations. At the time of initial design, this was approximately OC3 (155 Mbps) for most workstation level computers. In the intervening years, the arrival of high-speed Ethernet for very low cost and the imminent deployment of Gigabit Ethernet have increased the level of service that can be provided with one "server node" on a desk-top class machine, but this does not solve the scalability issues. It would still be the case that skew in the request pattern for various streams would lead to an unbalanced load on any disk or a server within a cluster of machines. One could imagine dozens of requests for a particularly popular object, slightly offset in time, while other disks remained relatively idle.

One basic addition to the model of stream delivery that was considered midway in the design process was replication and migration. This was the topic of another Master's Thesis [30] and added a third member to the development team. Replication solves another problem in the server design: variable latency due to geographic placement of the servers. Streams can be replicated on a number of levels: between disks within a server node, between server nodes on an individual server, or between servers. The first two levels of replication increase the number of simultaneous users of an individual object at a local site. The final level of replication increases the availability and may reduce the overall cost of streaming remote objects by migrating them closer to the location where they are frequently accessed.

To support replication, there must be a way of distinguishing between the various copies of the objects as well as a method of locating the most appropriate copy for the session. This provided the impetus for the next step in evolution of the design: location server functionality. The Location Server existed as a distinct process and received registration requests whenever a copy of an object was stored in the confederation of servers that were cooperating to serve a user population. It also maintained a mapping between a high-level UOI (identified by content (e.g., Low-Resolution Star Wars MPEG-2) and the (potentially many) low-level UOIs (copies) associated with this object. The server code at the administrator was necessary to do the mapping and inform the server node which Low-Level UOI was to be delivered.

The Location Server was an optional component and a client could either request a low-level UOI or a high-level UOI. The location server could choose a copy and notify the administrator for the server node that held the copy, if the system was configured appropriately.

When multiple copies of objects are maintained, it makes sense to locate them physically near the users. This concept has been put to great use by content distribution networks, such as Akamai [32]. We then investigated the potential of permitting migration of objects. This involved another change in design, but one which was anticipated early. Migration threads were placed in the server nodes. These would be activated by a call to *CmfsMigrate*, which would originate either on the Location Server computer or from a client application request. During migration, one server node performs data storage and one performs data retrieval. The transfer of the data only utilized bandwidth not already reserved for delivery of streams, to keep previous guarantees. Additionally, the transfer needed to be reliable, so this transfer could not be done in real-time. On a busy server, migration would take place very slowly, as the guaranteed performance requirements would not allow much reserve bandwidth, but this would be temporary as the goal would be to move the object to a server that is more lightly loaded.

This version of the CMFS (version 4) has been implemented and integrated into the system. Both version 3 and version 4 were used for performance tests for the theses of the respective students. Since the hardware configurations has never been sufficient to do large scale tests with server nodes, version 4 is not used in the current performance tests. It has been several years since version 4 was tested, since the developer left the team and has taken a position in industry. There may be operating system and hardware architecture developments that make this code inoperative. The thesis is the only document describing the work, apart from the code itself, and this has not been investigated.

One final issue that caused change was the desire to have existing client applications run unmodified but to use our server. This required some software on the client machine to act as a proxy for connection establishment and control requests, but have the data delivered to the IP Address and port number of the player. The call that we added was *CmfsProxyOpen*. It existed only at the client and was translated by the library to ensure communication happened between the appropriate processes at either end.

A considerable amount of effort had gone into the functionality of the server, but it would be of little use if the real-time characteristics could not be met and the efficient use of server resources was not realized. Thus, extensive performance experiments were designed that isolated the behaviour of each component.

The first set of tests considered whether or not the admission control algorithm was successful in admitting variable bit-rate streams that could be supported by the server. The algorithm does not admit any invalid scenarios[‡] [17], but we wished a quantitative measure of the conservativeness of our algorithm. We did this by running the server without without any admission control to observe scenarios that could be supported and therefore, accepted by an optimal algorithm with perfect knowledge of future server capacity. Thus, we needed an easy

[‡]A "scenario" is a set of playback requests issued over a small interval of time (e.g. 1 minute).

way to turn admission control off. The simplest mechanism to do so was to make the disk capacity estimate an extremely large number, as this ensured that we still did the processing for admission control, but always returned a positive result. The results of the initial tests are given in [31] and [24].

The hardware available by the end of the implementation was significantly more powerful than what we started out with, but within our environment we could not construct a completely coherent environment for performance testing. The main problems we encountered were a lack of CPU performance on the original AIX machine (RS/6000 360), and a lack of network performance on Intel machines. With a SCSI II - Fast/Wide connection supporting 4 disks, we calculated that approximately 100 Mbps of bandwidth could be supported, which matched the network interface bandwidth on the ATM Taxi switch which was connected to the machine. When performance/stress tests were conducted, we found this not to be the case. We undertook a set of tests to discover which resource was causing the bottleneck.

This took the form of additional compile time constants, namely NOSEND and NOREAD. When we wanted to test the disk reading performance, we defined NOSEND, which commented out the network send, while leaving all the buffer management code functional. Likewise, to determine the actual network output, we defined NOREAD, which set the status of every read operation to "COMPLETED" as soon as it was requested and then sent arbitrary bits across the network. This mechanism worked very well for isolating functionality changes with a minimum effect on the code.

Our first results on the RS/6000 Model 360 showed us that the disk performance reached a maximum of approximately 75 Mbps. We determined that this was likely the result of the inability of the I/O buffers of the operating system to cope with the incoming data from multiple disks. When we tested the network output, it too was much less than the anticipated 100 Mbps. We felt that this was approximately about as much as the CPU could pump through the network protocol stack to the ATM driver. Although this was disappointing, the functionality had been shown and we had obtained good performance results for one disk. We were able to achieve some amount of actual performance benefit from multiple disks, but not what was expected. Since the hardware was now obsolete, we felt that exploring this in more detail was not on the critical analysis path. New hardware would bring new, but different challenges.

We began similar performance experiments on the Pentium 166 machines in the lab. To our surprise, we were also constrained by this hardware configuration. The only operating system that supported the Asynchronous I/O operations was Intel-Solaris. Even though FreeBSD claimed to offer such an I/O facility, it did not work correctly. Linux did not support this API at the time, and furthermore lacked support for kernel-level threads. What made matters worse was that the various 10/100 Mbps Ethernet cards that came with the machines did not have a mature set of drivers for all operating systems. They worked well with Windows, and some worked for FreeBSD, but performed uniformly very poorly under PC-Solaris. We were unable to reach even 10 Mbps throughput from this platform in some cases, so we were unable to perform the scalability tests in this environment either.

Currently, work has begun again on performance tests with version 3 of the software. A recent issue that has caused concern is the mechanism used for converting the server between operating modes. The main modes of operation are: disk test, network test, and full operation.

As well, the device identifiers and capacity values for the disk devices and admission control parameters are hard to set. In the initial versions, these were compile time parameters, located in a configuration header file, included by all server source code. Device names and configurations vary between operating systems and actual deployment characteristics (e.g., number of disks, capacity, buffer space at the server). Getting all configuration parameters straight is very confusing as the C preprocessor is used to properly #define the right variables. This current problem is not receiving much attention as the server has been only used in disk test, and full operation mode in recent years.

CONCLUSIONS AND LESSONS LEARNED

In this paper, we have described the evolutionary development of a Continuous Media File Server: a software development process undertaken for the purposes of examining admission control, scalability and heterogeneity with regard to the storage and retrieval of continuous media. There were many stake-holders involved in the course of this project, each contributing to the design requirements, but few contributing to the actual development of the software.

Although the scope of such a project is substantial, it does not quite resemble a large on-going software project, such as an operating system [33], or commercially supported large-scale application. For the initial stages of the project, we were our own customers. This made it substantially easier to implement our changes in design and did not have the concept of "freezing" a design and then implementing it.

Interaction with the other partners in the project confirmed that we were not sufficiently objective in assessing the requirements of a client application, despite the great effort put forth in designing the API from the client point of view. There were some significant additions that were needed, but proved possible to implement, after some negotiations which ensured they did not violate the philosophy of the server design. The early effort in interface design made the implementation of these changes possible with reasonable and localized changes in the code.

The major lessons learned throughout this exercise are the following:

- We were able to achieve substantial software reuse in the support level software (in particular pthreads, and tdbm), because it was designed to fulfill a small, narrow task, and the modular components within that task specification allowed us to insert new algorithms.
- Isolating network code is one of the most important factors, and if it is done properly, changes become easy.
- Not all implementations in the operating system have similar performance characteristics. Relying on the operating system to provide performance features needed in the application is a mistake, and often these facilities must be re-implemented in the application.
- A clearly defined API interface reduces complications in the integration process, but reduces the flexibility. The most powerful API call incurred the most change, but within

a small range of semantic difference, because we had good reasons for the initial design decisions.
- Even small projects need configuration control. Two people is enough to require such facilities, and in fact, a one person project is large enough to make use of these facilities.
- Rapid prototyping helps find design flaws early.

The large legacy of previous software in related areas, such as scheduling and databases provided us with the resources to make the software self-contained. Some of the most difficult problems to solve were those to do with the beta-versions of drivers for client hardware, and OS-level facilities, such as Asynchronous I/O. It is our belief that using more third-party software would have exacerbated these types of problems.

In particular, we found that the performance of primitive operations in the system software level were critical in the evaluation of third-party packages. It was this factor, in combination with the perceived benefit of having complete control over as much of the code base as possible that led us to use RTT and tdbm as support software.

We wanted to isolate our code from changes in operating systems as we anticipated both a lengthy development time frame and an application that could be used for several years past the research project. The performance effect of the AIX semaphore operation confirmed that our performance achievements could be too strongly tied to the efficiency of the host operating system. Additionally, if there were particular features that were not implemented in a particular operating system or implemented with slight differences in semantics (e.g., semaphore operations), having our own environment that was system independent alleviated much tedious coding of primitive system operations at the application level. It was still the case, however, that certain performance issues were only discovered in the process of ensuring that the server worked on multiple platforms. Discovering a problem on one platform led us to superior solutions on all platforms.

We encountered some surprises in the course of this work. Most of these were quantitative in nature with respect to the achievements of the underlying hardware/drivers on some platforms. This made it difficult to substantiate our claims that the server achieved the goals of scalability. Nevertheless, our lab environment did provide access to many hardware and software platforms and we gained experience in running the server and clients on many of these platforms. In particular, there were substantial differences in the system call interface between Windows and the UNIX family of operating systems. Even with a small development team, we recognized the need to use revision control software early in the project. We were not strongly versed in the practice of checking files in and out, but even the periodic use made us aware of the conflicts that our changes to the code base created. We strongly recommend the use of version control software even for a small development team.

Since we kept the development team to a minimum, due to restrictions in budget and scope, there were a lot of issues that we chose not to explore. These were deemed to be someone else's problem or issue. In all of these cases, the design anticipated that a module to be plugged in that would solve that particular problem. An example of the modular design was the introduction of RTP to the server, showing the ability to add code in a localized portion of the server which added functionality. Further examples, such as call back mechanisms, allowed the structure of the server and the client interface to remain unchanged in the face of changes to the client application needs.

We found that the experience of rapid prototyping was a great strategy in enabling a quick evaluation of the interface, especially the inter-process communication and CMFS interaction protocol design. This allowed incremental development and a testing environment in which detailed functionality could be added incrementally. This was shown to be especially true with respect to the network protocol for sending media data. There were relatively few places in the code that interfaced to the data transmission protocol, and so conditional compilation achieved this goal easily.

Maintenance of this software has received a small amount of recent attention. The hardware originally intended for the UNIX clients has become obsolete and support for the decoder card has disappeared. Current work is underway to develop additional display clients. We are also currently in the process of further evolution of the server, re-implementing the buffer management and network bandwidth allocation mechanisms. This will give us further insight into the process of evolution of the server and client-server software in general.

ACKNOWLEDGEMENTS

The authors would like to thank the members of the CMFS development design and implementation team, most notably the Principal Investigator, Dr. Gerald Neufeld, along with David Finkelstein, Roland Mechler, Oliver Kraemer, and Mark McCutcheon. Other members of the project involved in integration were Nolan Andres and David Evans. We are also indebted to Kevin Schneider for valuable comments in presentation of the results.

We also acknowledge the financial assistance of the Canadian Institute for Telecommunications Research.

REFERENCES

1. M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and Laws of Software Evolution - The Nineties View. *4th International Software Metrics Symposium (METRICS '97)*, Alberquerque, New Mexico, 1997;20–32.
2. J. W. Wong, K. A. Lyons, D. Evans, R. J. Velthuys, G. v. Bochmann, E. Dubois, N. D. Georganas, G. Neufeld, M. T. Oszu, J. Brinskelle, A. Hafid, N. Hutchinson, P. Iglinski, L. Lamont, D. Makaroff, and D. Szafron. Enabling technology for distributed multimedia applications. *IBM Systems Journal*, 1997, **36**(4):489–507.
3. W. J. Bolosky, J. S. Barrera III, R. P. Draves, R. P. Fitzgerald, G. A. Gibson, M. B. Jones, S. P. Levi, N. P. Myhrvold, and R. F. Rashid. The Tiger Video Fileserver. *6th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Zushi, Japan, 1996;29–35.
4. P.V. Rangan and H.M. Vin. Designing File Systems for Digital Video and Audio. *Proceedings 13th Symposium on Operating Systems Principles (SOSP '91), Operating Systems Review*, Pacific Grove, 1991;81–94.
5. A. Heybey, M. Sullivan, and P. England. Calliope: A Distributed, Scalable Multimedia Server. *ACM USENIX Annual Technical Conference*, San Diego, CA, 1996;65–74.
6. J. Nieh and M. S. Lam. SMART UNIX SVR4 Support for Multimedia Applications. *IEEE Multimedia Computing and Systems*, Ottawa, Canada, 1997;404–414.
7. G. W. Neufeld, M. W. Goldberg, and B. Brachman. The UBC OSI Distributed Application Programming Environment. Technical Report 90-37, University of British Columbia, Vancouver, B. C.,Canada 1990.
8. Siu-Ling Ann Lo. *An Operating System Architecture for Real-Time Tasking Abstractions*. PhD thesis, Department of Computer Science, University of British Columbia, Vancouver, B. C.,Canada,1994.

9. D. Finkelstein, R. Mechler, G. Neufeld, D. Makaroff, and N. Hutchinson. Real-Time Threads Interface. Technical Report 95-07, University of British Columbia, Vancouver, B. C.,Canada, 1995.
10. J. Wen and X. Li. Realize Network subsystem QoS Guarantee. *Operating Systems Review*, 2001;**35**(3):67–71.
11. S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan. Asynchronous I/O Support in Linux 2.5. *Proceedings of the Linux Symposium.* The Linux Symposium, Ottawa, Canada, 2003.
12. XTP Forum. *Xpress Transport Protocol Specification Revision 4.0*, 1995.
13. R. Mechler. Media Transport API. Unpublished UBC Tech Report, University of British Columbia, Vancouver, B. C., Canada,1997.
14. R. Mechler. A Portable Real Time Threads Environment. Master's thesis, Department of Computer Science, University of British Columbia, Vancouver, B. C.,Canada,1997.
15. B. Brachman and G. W. Neufeld. Tdbm: A DBM Library With Atomic Transactions. *Proceedings of the USENIX Summer Technical Conference*, San Antonio, Texas, 1992; 63–80.
16. ISO/IEC Joint Technical Committee 1. MJPEG Digital compression and coding of continuous-tone still images. Draft International Standard CD 10918, International Standards Organization: Geneva, Switzerland, 1993.
17. G. Neufeld, D. Makaroff, and N. Hutchinson. Design of a Variable Bit Rate Continuous Media File Server for an ATM Network. *IST&SPIE Multimedia Computing and Networking*, San Jose, CA, 1996; 370–380.
18. J. M. McManus and K. W. Ross. Video on Demand over ATM: Constant-Rate Transmission and Transport. *IEEE InfoComm*, San Francisco, CA, 1996; 1357–1362.
19. G. Neufeld, D. Makaroff, and N. Hutchinson. Internal Design of the UBC Distributed Continuous Media File Server. Technical Report 95-06, University of British Columbia, Vancouver, B. C., Canada, 1995.
20. C. Bernhardt and E. Biersack. The Server Array: A Scalable Video Server Architecture. O. Spaniol, W. Effelsberg, A. Danthine, and D. Ferrari, editors, *High Speed Networking for Multimedia Applications*, chapter 5, Kluwer Publ., 1996; 103–125.
21. M. Kumar, J.L. Kouloheris, M.J. McHugh, and S. Kasera. A High Performance Video Server for Broadband Network Environment. *IST&SPIE Multimedia Computing and Networking*, San Jose CA, 1996;410–421.
22. C. Shahabi, R. Zimmermann, K. Fu, and S. Yao. Yima: A Second-Generation Continuous Media Server. *IEEE Computer*,2002; **35**(7):56–64
23. S. Ghandeharizadeh, R. Zimmermann, W. Shi, R. Rejaie, D. Ierardi, and Ta-Wei Li. Mitra: A Scalable Continuous Media Server. *Multimedia Tools and Applications*,1997; **5**(1):79–108.
24. D. Makaroff, G. Neufeld, and N. Hutchinson. Design and Implementation of a VBR Continuous Media File Server. *IEEE Transactions on Software Engineering*,2001; **27**(1):13–28.
25. D. Makaroff, G. Neufeld, and N. Hutchinson. Network Bandwidth Allocation and Admission Control for a Continuous Media File Server. *6th International Workshop on Inderactive Distributed Multimedia Systems and Telecommunications Services (IDMS 99)*, Toulouse, France, 1999; 337–350.
26. L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New ReSerVation Protocol. *IEEE Network*,1993; **7**(5):8–18.
27. M. Grossglauser, S. Keshav, and D. Tse. RCBR: A Simple and Efficient Service for Multiple Time-Scale Traffic. *ACM SIGCOMM '95*, Cambridge, MA, 1995;219–230.
28. S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weisse. An architecture for differentiated services. Request for comments 2475, Internet Engineering Task Force, 1998.
29. D. Clark, S. Shenker, and L. Zhang. Supporting real-time applications in an integrated services packet network: Architecture and mechanism. *ACM SIGCOMM '92*, Baltimore, MD, 1992;14–26.
30. O. Kraemer. A Load Sharing and Object Replication Architecture for a Distributed Media Fileserver. Master's thesis, Universitat Karlsruhe, 1997.
31. D. Makaroff, G. Neufeld, and N. Hutchinson. An Evaluation of VBR Disk Admission Algorithms for Continuous Media File Servers. *ACM Multimedia*, Seattle, WA, 1997;143–154.
32. Akamai Technologies. www.akamai.org;2004.
33. L. A. Belady and M. M. Lehman. A Model of Large Program Development. *IBM Systems Journal*,1976; **15**(3):225–252.

AUTHORS' BIOGRAPHIES

D. J. Makaroff completed his Ph. D. in Computer Science from the University of British Columbia in 1998, after receiving a B. Comm. and M. Sc. in Computational Science from the University of Saskatchewan in 1985 and 1988, respectively. He has been an associate professor in the Department of Computer Science at the University of Saskatchewan since 2001. He was the primary architect of the UBC Continuous Media File Server. His research interests include performance of distributed systems and network architectures, with particular emphasis on multimedia and E-commerce systems.

N. C. Hutchinson received the B.Sc. degree in Computer Science from the University of Calgary in 1982, and the M.Sc. and Ph.D. degrees in Computer Science from the University of Washington in 1985 and 1987 respectively. After graduation he was an assistant professor of Computer Science at the University of Arizona, Tucson, Arizona for 4 years before moving to the University of British Columbia in 1991.

His research interests center around programming languages and operating systems, with particular interests in object-oriented systems, distributed systems, and file systems. He was instrumental in the design of the Emerald distributed programming language, the x-kernel communication protocol environment, the UBC Continuous Media File Server, and Elephant, a new file system offering strong support for version management of files.