Calliope: A Distributed, Scalable Multimedia Server

Andrew Heybey, Mark Sullivan, and Paul England
Bellcore

Calliope: A Distributed, Scalable Multimedia Server

Andrew Heybey Mark Sullivan Paul England

Bell Communications Research
Morristown, NJ 07960

Abstract

Calliope is a distributed multimedia server constructed from personal computers. Preliminary performance measurements indicate that Calliope can be scaled from a single PC producing about 22 MPEG-1 video streams to hundreds of PCs producing thousands of streams. The system can store both variable- and constant-rate video and audio encodings and can deliver them over any network supported by the underlying operating system. Calliope is cost-effective because it requires only commodity hardware and portable because it runs under Unix.

1 Introduction

A multimedia server records and plays data that has real-time delivery requirements. Examples of multimedia data include video and audio streams; some typical applications that might use a multimedia server are video mail, video bulletin boards, and video databases. Two crucial requirements of a multimedia server are that it should be low cost and that it should scale up as demands on the system increase. This paper describes the design and performance of Calliope, a distributed multimedia server constructed from personal computers. Calliope achieves low cost by running on commodity PCs and is easy to scale up by virtue of its distributed architecture.

A traditional network file server cannot be used as a multimedia server for several reasons. First, multimedia data has timeliness constraints. Clients have limited buffering, so data that arrives too late will result in an interruption in audio or a still frame; data that arrives too early will overflow the buffer and be discarded. Traditional file servers make no delivery guarantees. Second, a traditional file system is not optimized for a multimedia workload. Multimedia clients are willing to accept relatively long delays when a read or write begins, but very little jitter when a large file is being accessed.

Calliope uses several strategies to solve these problems. Timeliness constraints are addressed by dividing the system into real-time and non-real-time components so that data delivery deadlines can be met without using a hard real-time operating system. One or more *Multimedia Storage Units* (MSUs) handle the real-time services while a single *Coordinator* machine handles the non-real-time functions (see Figure 1). The MSUs send and receive multimedia data, manage disk storage, and process simple VCR commands from clients. The Coordinator maintains bookkeeping information, serves as an initial point of contact for users, and schedules requests at MSUs. For very small installations, the Coordinator and MSU software may run on the same machine. Larger Calliope installations still have a single coordinator, but add more MSUs as storage requirements or user bandwidth requirements increase. To achieve as much performance as possible using relatively low performance machines, the MSUs are optimized to handle multimedia files. The MSU has a large-block file system to store large, sequentially-accessed files and its process architecture is oriented toward efficiently moving large quantities of data through the system from disk to network interface (or vice versa for recording).

Calliope can record and play back in real-time both constant bit-rate streams, such as MPEG, and variable bit-rate ones, such as those used by the MBone tools [5]. Since there are many different network protocols and audio/video coding

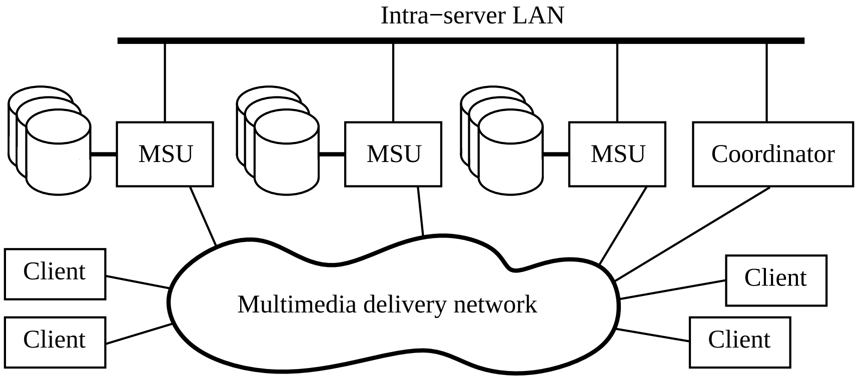


Figure 1: Calliope System Architecture.

standards, Calliope is extensible. Simple modules can be added if necessary to handle different network packet formats and to extract timing information from new encodings.

Calliope does not rely on special-purpose hardware or a hard real-time operating system. The major advantage of a software-based approach running under a standard operating system is portability. As performance of commodity computers improves, Calliope can continue to run on the most cost-effective platform.

The performance results presented in this paper show that twenty-two 1.5 Mbit/sec MPEG-1 streams can be transmitted to an FDDI network from a single Pentium PC running the Calliope MSU software. By combining MSUs, our measurements suggest that Calliope could scale to hundreds of PCs and thousands of customers. We think that such a distributed architecture is appropriate for large scale multimedia servers, but low-end machines have both performance and engineering difficulties that we never expected. In the remainder of the paper, we describe the system architecture of Calliope, measurements of server performance, and some of the PC-related problems we encountered while building Calliope.

2 System Architecture

Figure 1 shows a Calliope installation with three MSUs. The intra-server network connecting the Calliope components is a relatively low-bandwidth network such as Ethernet. The multimedia delivery network that connects Calliope to its clients is a higher-bandwidth network such as FDDI or ATM. If necessary, a Calliope installation could eliminate the intra-server network and use the multimedia delivery network to carry both intra-server and client-server traffic. This would reduce the number of clients served by each MSU, since some network interface bandwidth would have to be dedicated to MSU control messages. In Calliope, the Coordinator and MSUs communicate using TCP connections. Clients use TCP connections to communicate control information to Calliope and UDP for real-time data delivery. MSUs do not communicate with one another.

The subsections that follow describe Calliope's components and some of its applications. We start with a section on the applications to show what kind of services the system supports. Next, we describe the ways in which the Coordinator allocates resources to provide those services. Finally, we describe the storage management and real-time features of the MSU.

2.1 The Client Interface

We have experimented with several different kinds of client applications for Calliope's multimedia services. Some are fairly simple video-on-demand programs that browse Calliope's table of contents, select content to play, and issue simple VCR commands. We have also used Calliope in a simple video mail application and have used it to record MBone presentations. Finally, we have experimented with an application that keeps simple indices of recorded seminars. Users can examine the index and skip to the portion of the seminar that interests them.

Our primary clients have been Unix™ workstations and PCs running the MBone teleconferencing tools. We have also used Windows™

PCs with hardware-assisted MPEG and JPEG decoders. In one application, we have treated a PC-based MPEG client as dumb set-top box that accepts raw MPEG over UDP. So far, we have not programmed any commercial set-top boxes to act as Calliope clients.

To begin using Calliope, a client establishes a session with the Calliope coordinator. The client can then request a listing of available content, play existing content, or record new content. With appropriate permissions, the client can delete an item of content or make other administrative changes to the server configuration.

The Calliope Coordinator associates a *content type* with each item in its table of contents. Calliope uses the content type to determine the rate at which the content is to be played. Content type also tells whether the content is played at constant or variable rate. Types may be composite; for example, we have a VAT [17] audio type, an RTP (the Internet Real-time Transport Protocol [13]) video type and a *Seminar* type composed of one VAT and one RTP stream. In the current implementation, clients may not define new types without the help of a system administrator to update the Coordinator's internal databases.

Before sending or receiving multimedia content, the client must create a UDP socket and register that socket with Calliope as a *display port*. In Calliope, display ports associate a string name, a content type, and the socket's IP address and port number. The display port need not be on the same machine as the client program that creates it. The software that owns the port socket can be a software encoder/decoder that is part of the client application or a simple driver for a hardware device.

Display ports for composite types can be constructed from previously-registered display ports of the component types. In our example above, a Seminar display port is composed of display ports for RTP and VAT. A client can register many display ports of the same type and unregister them as necessary, but all ports are associated with a single client-Coordinator session. When this session is dropped, the Coordinator deallocates its local representation of the ports.

To read an item of Calliope's content, the client sends a request containing the content name and the name of a display port. Calliope checks that the port and the content have the same type and assigns the resources required to play the content. When it is ready to play the content stream, Calliope creates a control connection to the client on which the client can send VCR commands: pause, play, seek, and quit. For some content items, clients will also be able to issue fast forward and fast backward commands (this is described in more detail in Section 2.3.1). Recording on a Calliope server is similar to reading, but the client request must also contain an estimate of the recording length.

2.2 Coordinator Architecture

The Coordinator is the global resource manager for Calliope. It maintains a small administrative database and a set of scheduling queues. The database contains information about customers, content stored on Calliope, and resources owned by the system. The Coordinator uses the database to tell what MSUs are available, how many disks each one has, and how much disk space remains unused. The Coordinator also keeps track of which items of content are stored on each disk and what content types are available. As the previous section stated, each item of content has a type. The content type entry contains a *bandwidth consumption rate* which gives the expected rate at which content of this type is to be played and recorded.

The Coordinator uses its databases to authenticate clients, satisfy their requests for the table of contents, and to allocate resources as clients play or record content. When Calliope receives a read request, the Coordinator finds an MSU with a disk that both contains the requested content and has enough bandwidth available to satisfy the request. As the Coordinator assigns resources to clients, it keeps track of load by processor and disk. If a client's request cannot be satisfied, the Coordinator queues the request until an MSU with the necessary resources becomes available.

The Coordinator informs an MSU of a scheduling decision by sending a message describing the client's request. Once the request is scheduled, the client does not communicate with the Coordinator until the stream is terminated. VCR commands between the client and Calliope go directly to the MSU. As soon as it is ready to deliver the content stream, the MSU establishes a control stream (TCP connection) with the client for these commands. After a "quit" command from the client, the MSU informs the coordinator that the stream has been terminated.

If a client requests a composite content type, the Coordinator creates a *stream group* to play the content. A stream group has one member for each of the atomic subtypes of the composite type. For example, if a client played an item of the composite Seminar type described in the previous section, the RTP video and VAT audio subitems would become a two-item stream group. All streams in a stream group are controlled by the same VCR commands. Calliope assigns all streams in a group to the same MSU so that the client's commands can start and stop all streams simultaneously. Synchronizing the streams would be difficult if streams from the same group were assigned to different machines.

When a client records content on a Calliope server, the Coordinator must allocate disk storage as well as bandwidth. The client write request includes an estimate of the length of the recording (in seconds). The Coordinator uses this estimate and the content type information to determine how much disk space the recording will consume. It must schedule the request on an MSU that has both disk space and bandwidth available. If the client overestimates the length of the recording, the unused space will be returned to the system once the recording session has completed.

To support variable rate content, the content type table contains separate rates for disk space and bandwidth consumption. A constant-rate stream will consume disk bandwidth and disk space at the same rate. For data with a variable-rate encoding, the system should allocate bandwidth more conservatively than disk space. The bandwidth consumption rate should be closer to the stream's peak rate and the storage consumption rate should be closer to the average rate.

Calliope's Coordinator has some support for fault tolerance. The Coordinator detects when one of the MSUs fails by a break in the TCP connection between the coordinator and the MSU. When an MSU is down, the Coordinator marks it as unavailable in the scheduling database. When the MSU becomes available again, it contacts the Coordinator and is restored to the scheduling database. Calliope does not recover from Coordinator failures.

2.2.1 Disk and Network Scheduling

Calliope's MSUs use double buffering and careful disk and network device scheduling to ensure that data is delivered at a regular rate and peripheral devices are fully utilized. In the case of a read, double buffering means the network process is sending data to a client from one main memory buffer while the disk process is loading the other. When the network process empties its buffer, the disk process should already have filled the other buffer. The two processes swap buffers for the next cycle.

To allocate bandwidth of a single disk, we give the disk a *duty cycle* which is divided into *slots*. Each slot is long enough to read or write a single disk block for one client stream. The number of slots in a cycle is the maximum number of block transfers that can be accomplished during the time it takes for a single stream to transmit its block. In other words, the cycle must be short enough that each client's disk transfer completes before the network transfer runs out of data to transmit.

In order to replay variable-rate data packets at the correct times, the network process constructs a *delivery schedule* as the data is recorded. The schedule associates delivery times with data packets. When variable-rate data is replayed, the network process uses the stored delivery schedules to determine when packets are delivered to the network. For constant bit-rate streams, the delivery schedule is calculated rather than stored. The arrival times in delivery schedules are not absolute; they are offsets from the beginning of the recording session.

When it stores the delivery schedule and data on disk, Calliope interleaves them in a single file using a data structure similar to a primary B-tree [4]. In a primary B-tree, the file blocks contain both data and a search tree. The top parts of the search tree are stored in internal pages containing only keys and pointers to lower-level pages. The lower parts the tree are stored in leaf pages with the data. In our case, the key for the search tree is delivery time. A sequential scan of the B-tree gives the data packets in the order they must be delivered to the network.

Calliope's variant on B-tree is called Integrated B-tree (IB-tree) because it integrates the internal pages into the data pages. Calliope's large disk blocks allow it to make internal pages smaller than leaf pages without increasing the height of the tree in most cases. We use 28 KByte internal pages (with 1024 keys) and 256 KByte data pages. As the B-tree is constructed, Calliope creates the internal pages and data pages in a manner similar to normal B-tree construction. When an internal page fills up, it is copied into the current data page instead of being written separately on disk. During seeks, Calliope traverses the internal pages of the search tree in the usual way. During sequential reads, the internal pages are read in as part of the data page but ignored. They are so small and only appear in 0.1% of the data pages so they do not affect read bandwidth appreciably. On writes, the IB-tree writes both data page and internal page using a single disk transfer and seek. If the two pages were stored separately, the internal page writes would add slots to Calliope's disk duty cycle and the extra seeks would reduce disk utilization.

Finally, Calliope does not use a real-time operating system and FreeBSD timers have only 10 ms granularity, so delivery times are only approximate. While timer granularity will introduce jitter, clients will have to be able to handle the jitter introduced by the multimedia delivery network anyway. We assume that clients have enough buffer space to smooth any jitter introduced by either the approximate scheduling or the intervening network. A 200 KByte buffer will hold more than one second of 1.5 Mbit/sec video. Calliope will not add more than 150 milliseconds of jitter in the worst case (see Section 3.2.1) and any network that introduces more than 850 milliseconds of jitter is probably not usable for video delivery.

2.3 MSU Software Architecture

Each MSU is a PC with a set of disks, an interface to Calliope's intra-server network and an interface to the external high-speed network. The MSU runs a simple multi-process control program that assigns a process to each network device and disk while a central process handles RPCs from the Coordinator and from clients. The MSU processes must communicate in order to share resources and to start and stop streams. Instead of using expensive semaphore operations, the MSU processes communicate using a shared memory queue structure that relies on the atomicity of memory read and write instructions to produce atomic enqueue and dequeue operations.

When the client starts a read stream, the MSU's disk process loads data from disk into a shared memory buffer. The network process then packetizes the buffer and sends it out through the high speed interface. The network process ensures that packet delivery proceeds on schedule. The disk process makes sure that the network process always has buffered data ready to send. When data is recorded, the network process fills buffers and the disk process writes full ones to disk. The description of Calliope's performance in Section 3 contains more detail about the MSU data path.

2.3.1 Fast Forward and Backward

The MSU cannot send a data stream to clients at a higher rate than the one at which it was received. To implement fast forward and fast backward scans, we used an offline filtering program. Thus far, we have only implemented filtering for MPEG streams. The filtering program reads the recorded stream, selects every fifteenth video frame, recompresses the filtered stream, and loads it into the server. For the fast-backward version, the frames are stored in the filtered stream in reverse order. This filtering procedure is not automatic in the current implementation; an administrator has to produce the fast forward and fast backward versions of the content.

An administrative interface is used to load the fast forward and fast backward files into the server in a way that allows the server to associate the files with the fast forward and fast backward VCR commands. The MSU remembers which files contain the normal rate, fast forward, and fast backward versions of the same content. If a client issues a command to switch from normal rate to fast forward, the MSU seeks to the frame in the fast forward file corresponding to the current frame of the normal rate file. The user will experience a few seconds of delay as the MSU waits for the client's next disk slot, reads the required block of the fast forward file into memory, and starts delivering frames to the network device. Switching back to normal rate follows the same procedure.

We briefly considered implementing a dynamic fast forward and fast backward mechanism that extracted the fast rate streams directly from the normal rate stream. In such a scheme, the MSU would use the schedule information to skip over frames, delivering only selected ones to the network. This scheme was impractical for two reasons.

First, if the data stream uses inter-frame compression, some frames may not be safely skipped. In inter-frame compression, decoding a given frame requires information in other frames around it in the stream. If the stream uses intra-frame compression or no compression, the MSU could send selected frames to the user. MPEG uses a combination of inter-frame and intra-frame compression. Most frames are inter-encoded, but intra-encoding is used for every N-th frame, where N is a parameter determined at the time of encoding (typically, fifteen to thirty). The MSU could implement fast forward for MPEG streams by sending selected intra-encoded frames. However, the MPEG encoders that we have produce an opaque stream with no framing information. While recording, the MSU would have to search the stream to find the intra-coded frames. Parsing the MPEG stream is too expensive to do in real time.

The second reason we did not implement this scheme is that it would make disk scheduling hard. Skipping frames allows the MSU to send a fast forward stream using the same network resources as a normal-rate stream. However, fast forward delivery has a larger impact on disk usage than normal rate delivery. If the MSU reads from disk only the frames that it will send in the fast forward stream, it has to issue many small read requests instead of a few 256 KByte ones. This will significantly worsen disk performance. A more practical approach is to read all of the stream's frames from the disk and then skip over the unneeded frames once they are in memory. However, in this case, the MSU must read fast forward streams from disk at several times the normal stream rate. Allowing users to switch back and forth between high-rate and low-rate disk reads would complicate our scheduler.

2.3.2 MSU Extensibility

The MSU is designed so that support for new protocols can be added to the system easily. A "protocol" in this context is something no more than complex than (for example) RTP—essentially a header definition and a few control messages. The MSU currently supports RTP [13], VAT [17] audio, some home-grown protocols and any protocol and/or encoding which can be handled by transmitting fixed sized packets at a constant rate.

An MSU *protocol extension module* is comprised of two functions. The first performs any operations required by the protocol beyond the normal sending or receiving of data packets. For example, the RTP protocol uses two ports—one for control messages and one for data. The RTP module for the MSU manages the control socket. During recording, the RTP module interleaves the control messages with the rest of the data stream before the data is given to the disk process. On output, the opposite process is performed. The MSU calls the second extension function during recording to construct a delivery sechedule. This function creates a delivery time to use when the incoming packet is replayed. By default, the MSU derives the delivery time from the packet's arrival time. If there is a timestamp in the protocol's header, then a protocol extension function may derive delivery time from the timestamp. Using the sender-generated protocol timestamp instead of the packet's arrival time has the advantage that it does not include the effects of network-induced jitter.

2.3.3 MSU File System

The MSU has to manage files that are often large (a two hour MPEG-1 movie is 1.35 GByte) and are usually read and written sequentially. Instead of the BSD fast file system, the MSU uses a simple user-level file system tuned to the multimedia workload. The MSU file system does its own memory management and uses raw disk I/O to avoid user-to-kernel-space copying.

Instead of a block cache, available main memory is organized into large buffers to support read ahead and write behind. Large disk blocks and large buffers mean that large amounts of data can be transferred per disk access, lessening the performance impact of disk seeks. With 256 KByte transfers, the MSU achieves 70% of the maximum disk transfer bandwidth. Large file block size also decreases the size of the file system meta-data to the point that it can be entirely cached in main memory. An LRU block cache would impair performance because there is not enough data locality or sharing to make the cache effective. Different clients may view the same data at approximately the same time, but a 256 KByte buffer contains only about one second of 1.5 Mbit/sec MPEG-1 video. To share data in a block cache, clients would have to be synchronized to within a second of one another. Without some application-level coordination, this is unlikely to occur. Locality of reference by a single client is also unlikely. Clients tend to access multimedia files sequentially so the client would rarely rereference its own cached data.

The current implementation of the MSU does not employ disk head scheduling. The MSU services the customers for each disk in a round-robin fashion, resulting in random seeks between disk transfers. Further gains in disk performance could be realized by ordering the I/Os to minimize seek distances. We may eventually implement disk head scheduling, but we do not expect large performance improvements for two reasons. First, disk rotation and head settling times contribute a substantial fraction of the total seek time; disk head scheduling will not have any effect on these. Second, the MSU's large block size already limits the effects of seeks on disk performance. Using a simple program that simulated 24 concurrent users reading random 256 KByte disk blocks, we found that an elevator scheduling algorithm improves throughput by only about 6% for our disks.

In the current implementation, Calliope's MSU does not stripe files over its disks. When a client writes a file, all blocks go to a single disk. It would be easy to lay out a file so that consecutive blocks are on "adjacent" disks. The disk process in this case would read or write blocks from its disks in a round-robin fashion.

Disk bandwidth management is a little more complex in this case. The duty cycle must cover all N disks and contain $D * N$ slots, where D is the number of slots in a single disk's duty cycle. When a client arrives at the MSU, it is allocated a disk slot and must wait at most $D * N - 1$ slots before the MSU begins to deliver data.

The advantage to this kind of disk management is that we can still utilize the disks well even if workload is unpredictable. If an MSU has N items of content striped across N identical disks, all of the system's customers can access any of the N items. If each of the items were on separate disks, only $1/N$ of the system's customers can access any one item of content. In the non-striped case, we can make copies of popular content on several disks, but we must anticipate usage trends in order to choose the content to copy. We must also use additional disk space to get additional disk bandwidth.

One disadvantage of striping is that the client must delay every time it issues a VCR command while a disk slot becomes available. As in the non-striping case, the client waits at most as long as the duty cycle. If the MSU file system used striping, this delay is N times as long as it is in the non-striped case. We initially felt that this delay would be unacceptable to our users. In retrospect, we were probably wrong.

The second disadvantage is that management of streams with different fixed rates or a combination of variable and fixed rates is harder in a striped environment. When all streams play at the same rate, controlling the rate at which users enter the system ensures that none of the disks becomes overloaded. If different files are consumed at different rates, the block size must vary by file or clients will progress across the disk at different rates depending on the content they are playing or recording. Variable block sizes make file system implementation more challenging. Allowing users to move from disk to disk at different rates would allow disks to become oversubscribed.

3 Performance

To demonstrate the scalability and capacity of the system, we have run several kinds of performance measurements. First, we measured the throughput capacity of our hardware by running simple programs that move data through the same data path as the MSU. These measurements give an approximate upper bound on the performance of the MSU for our hardware. Second, we estimated the MSU's throughput by observing how well packet delivery deadlines are met when the system delivers varying numbers of constant-rate and variable-rate streams. Finally, we measured the load on the internal network and Coordinator due to client requests in order to determine how many MSUs can be combined into a single system.

In our experiments, the MSU software runs under the FreeBSD [7] operating system version 2.0.5 on a 66 MHz Pentium PC from Micron Computer Corporation. The Micron has two busses: a fast PCI bus and a slower EISA bus. Each MSU contains one or more Buslogic EISA bus fast-differential SCSI host adaptors each having one or more 2 GByte Seagate Barracuda disks, 32 MBytes of main memory, a single SMC ISA bus Ethernet interface to the internal network and a DEC DEFPA PCI bus FDDI interface to the high speed network.

3.1 Baseline Measurements

In order to estimate the maximum potential throughput of Calliope, we measured the performance of several simple programs exercising memory, disks, and network interface. Since these programs perform almost no computation, they are measuring the speed at which the PC hardware and operating system can move data from disk through memory and out the FDDI interface.

The baseline tests use one process per device as the MSU does. The disk process is a sim-

	FDDI only	Disks only			Disks and FDDI			
		Disk 1	Disk 2	Disk 3	FDDI	Disk 1	Disk 2	Disk 3
0 disk	8.5							
1 disk (one HBA)		3.6			5.9	3.4		
2 disk (one HBA)		2.8	2.8		4.7	2.4	2.4	
2 disk (two HBA)		2.9		2.9	2.3	2.7		2.7
3 disk (two HBA)		2.2	2.2	2.7	1.4	1.9	1.9	2.5

Table 1: Baseline Performance Measurements. The measurements show the throughputs of simple test programs writing to the FDDI interface and reading from the disks (in MBytes/sec). The table is divided across the top into three groups of experiments: FDDI only, disks only, and both FDDI and disks running simultaneously. Within each group, each row of the table shows the performance of each component that took part in a particular run. Tests labeled "one HBA" (one SCSI Host Bus Adaptor) had all disks on the same SCSI chain; those labeled "two HBA" had at least one disk on each of two SCSI chains. See the text for a detailed description of the test programs used.

ple program that performs 256 KByte reads of the raw disk device at random offsets. The network process is a slightly modified version of the ttcp [16] program, which sends data from memory to a peer process on a machine across the FDDI network. We changed ttcp so it did not touch the data before sending (since the MSU network I/O process (IOP) does not touch its data). We also changed it to send successive segments from a large buffer instead of repeatedly sending a smaller buffer. If a small buffer is used, ttcp reports artificially high performance because the buffer stays in the processor cache, speeding up the user-to-kernel-space copy. The arguments to ttcp were:

ttcp -t -u -s -l 4096 -L 1m -n 100000 ⟨dest⟩

-t	Transmit mode.
-u	Use UDP instead of TCP.
-l 4096	Send 4k packets (default is 8k).
-n 100000	Send 100000 packets.
-s	Send from memory, not stdin.
-L 1m	Step through 1MB buffer while sending (we added this option).

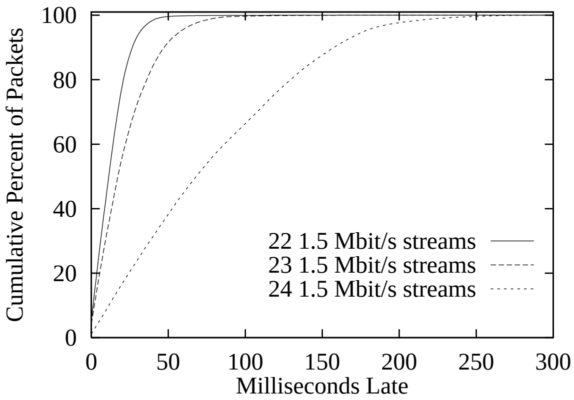
Note that, while ttcp misreports network bandwidth for transmitting UDP packets under some versions of Unix, it is accurate when run on FreeBSD. Some systems silently discard packets when the interface output queue is full, leading ttcp to report inflated performance numbers. FreeBSD returns ENOBUFS when the interface output queue is full. Ttcp then sleeps briefly and tries to send the packet again.

Table 1 displays the results of the baseline measurements. According to the measurements, the highest throughput attainable with our particular hardware and operating system combination is 4.7 MByte/sec[1]. This corresponds to the "2 disk (one HBA)" experiment in which the FDDI wrote 4.7 MByte/sec, and the 2 disks read 2.4 MByte/sec each. With only one disk running, the FDDI can go faster, but the disk does not then produce enough data to keep up with the network.

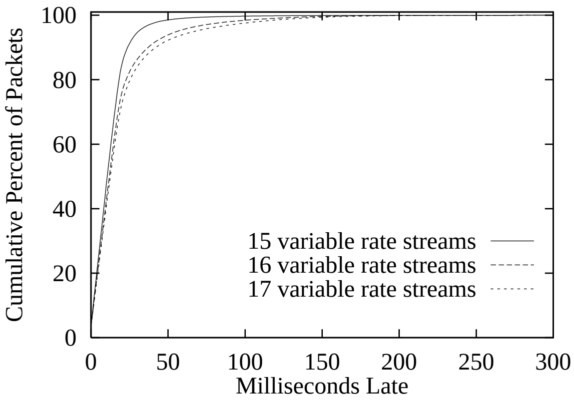
It is surprising that we could not achieve better performance with multiple SCSI host bus adaptors (HBAs) running simultaneously. As Table 1 shows, the FDDI performance is dramatically lower when running two disks on two separate HBAs versus two disks on the same HBA, while performance achieved by the disks in this case is only slightly improved. We believe that this effect is a hardware problem either on the part of our motherboard or the HBAs.

We first became aware of the problem when we noticed that the system clock slowed down if multiple HBAs were active. Upon further investigation, we discovered that the system was missing clock interrupts. It turned out that "in" and "out" instructions (which are needed both to service interrupts and to read the hardware timer used in FreeBSD to keep time) could take a very long time when two HBAs were running. Specifically, the sequence of instructions needed to read the hardware timer took approximately 4 microseconds with no disk activity; it occasionally took a millisecond with one HBA running, and often took 20 milliseconds with two HBAs running. To measure this phenomenon, we disabled interrupts and timed the code using the

[1] All of the measurements in this section are in 10^6 bytes/sec units.



Graph 1: Cumulative Packet Delivery Distribution of Constant Bit Rate Streams. The curves show the percent of packets delivered within a given number of milliseconds of their deadline. The three workloads depicted show the service provided as the MSU reaches its performance limit.



Graph 2: Variable Bit Rate Cumulative Packet Delivery Distribution. The curves show the percent of packets delivered within a given number of milliseconds of their deadline for variable rate streams. The streams have average rates ranging from 635 to 877 KBit/sec. The three workloads depicted show the service provided as the MSU reaches its performance limit.

Pentium microprocessor's internal cycle counter. We worked around the bug by changing FreeBSD to keep time using the Pentium cycle counter (so that missed clock interrupts would not affect the time of day, though they might result in timer interrupts being late).

3.2 MSU Throughput

3.2.1 Constant-Rate Streams

Graph 1 shows how closely an MSU with 22, 23 and 24 constant-rate streams keeps to the real-time packet delivery schedule. In each experiment, the MSU delivered streams from two disks on one host bus adaptor for six minutes, and generated approximately 16480 four KByte FDDI packets per stream. The graph's X-axis is the number of milliseconds a packet was sent after its deadline. The Y-axis gives the cumulative percentage of all packets delivered in the experiment that fall in each one-millisecond bin.

The measurements show that an MSU can deliver up to 22 1.5 MBit/sec streams with very good performance for each stream. In this case, only 0.4 percent of the packets are delivered more than 50 milliseconds late and no packets are more than 150 milliseconds late. As more streams are added the quality first degrades gradually and then dramatically. With 24 streams, only 38 percent of the packets are delivered within 50 milliseconds of their deadline.

This performance is approximately 90% of the baseline performance, indicating that the overhead introduced by the MSU for constant-rate streams is not excessive. The biggest performance problem of the system is the one discussed above in Section 3.1. The current baseline performance might be considerably improved with different hardware.

3.2.2 Variable Rate Streams

Graph 2 shows how closely an MSU with 15, 16, and 17 variable-rate streams keeps to the real-time packet delivery schedule. For these tests, three different files encoded by NV [6] were used (so the files were played multiple times to produce the required number of streams). The three different files used in the test had average rates of 650, 635, and 877 KBit/sec. Note that the performance for variable-rate streams (by this metric) is substantially worse than that of the constant-rate streams. There are several reasons for this performance difference.

First, the packet size of the variable-rate streams is much smaller than the four KByte packets used for the constant-rate tests. Most of the packets in the streams are about one KByte long. There is four times as much processing overhead for both the MSU code and the kernel networking code.

Second, the video is quite bursty. NV encodes a frame and then sends it out as quickly as possible, resulting in bursts of back-to-back packets. Measured using a 50 millisecond sliding window, the peak rates of the files ranged from 2.0 to 5.4 MBit/sec. It is impossible for the MSU to preserve the exact timings for many streams with bursty traffic on this time scale due the non-real-time nature of the MSU. Packets at the tails of the bursts will be delayed relative to their deadline.

In addition, the clients in the variable-rate tests viewed only three different files. All of the streams in the tests were started simultaneously. This means that every time the data file contained a burst of packets, one third of the streams in the test transmitted the burst at the same time. Thus, the MSU often had more packets to deliver simultaneously than the apparent data rates would suggest and some packets were necessarily delivered late. Furthermore, when tested while transmitting only a single file, the MSU could only produce 11 streams instead of 15. This unrealistic scenario is a limitation of our automated test setup; in practice clients do not start streams in synchrony nor would there be such a limited selection of content.

3.2.3 Bottlenecks

At present, the bottleneck in our system is that we cannot make use of more than one SCSI host bus adaptor simultaneously, limiting the data rate to 4.7 MBytes/sec. The next bottleneck is memory bandwidth. Our system can read memory at 53 MByte/sec, write it at 25 MByte/sec, and copy at 18 MByte/sec. As the MSU reads a file from disk and sends it to a client, the data traces the following path through the memory of the MSU PC:

1. Write (DMA from disk to user memory in the raw disk read).
2. Copy (user space buffer to kernel mbuf in network send).
3. Read (UDP checksum).
4. Read (DMA to FDDI interface).

Therefore, the fastest rate at which our test system could move data along this path (assuming that DMA accesses memory at the same speed as the processor) is:

$$\frac{1}{\frac{1}{25}+\frac{1}{18}+\frac{2}{53}} = 7.5\text{MByte/sec.}$$

To measure this disk-less data path, we replaced the disk I/O process in our baseline measurements with a process that simply wrote constant values into memory buffers. Using this process setup, the system moved data at about 6.3 MByte/sec (that is, it sent UDP packets at that rate while another process simultaneously wrote memory buffers at the same rate). The difference between the maximum 7.5 MByte/sec and the measured value is due to other memory accesses occurring in addition to the data movement. In particular, we believe that instruction fetches account for much of the difference. All the code necessary for the MSU (and kernel) to move the data will not fit in the first-level instruction cache, and the second-level cache will be flushed by all the data movement.

3.3 Scalability

The Coordinator and internal network are the only shared resources in the system, so their capacity will eventually limit system size. During normal operation, most of the load on the Coordinator and network consists of client requests for new streams and stream termination notifications. In a real system, the maximum request rate depends on the number of MSUs in the system and the length of the average viewing session.

To measure the effect of scheduling requests on shared resource loads, we have created a fake MSU which, when scheduled, delays for 50 ms and then reports that the user has terminated the stream. We start two of these MSUs on different machines and started two clients who together sent 10,000 requests to the coordinator at a rate of about 60 requests per second. We measured the Coordinator's CPU utilization at 14% and the network utilization at 6%, both relatively insignificant loads. Even if sessions are as short as one minute, a large scale implementation of Calliope serving 3000 simultaneous streams (150 MSUs at 20 streams each) would need to service only 50 requests per second.

4 Related Work

Commercial network based video and multimedia servers have become quite common in recent years. Several large-scale distributed video servers — including ones by DEC, Hewlett-Packard, Oracle, IBM, and Microsoft — are under production or currently available. Starlight sells a smaller-scale ethernet-based centralized video server. While some of the commercial systems have designs similar to ours, the focus of the paper is on performance issues and implementation experience which the commercial vendors have not published. Furthermore, many of the commercial systems rely on custom hardware and operating systems to achieve acceptable performance while Calliope uses commodity hardware and OS software for greater portability and lower cost.

We are familiar with three research multimedia servers comparable to ours. Freedman and DeWitt [8] describe simulations of a video-on-demand system in which clients request data as their buffers empty rather than having the server deliver the data at a pre-determined rate. Smith [14] focuses on coding and protocol issues rather than on the architecture and scalability of the server itself. Gelman et al. [9] describes a multimedia server built from custom hardware. Their system includes an end-to-end network distribution system for video-on-demand that uses buffering in switches and specialized telephone line-cards.

Other groups have considered specific aspects of server design, most notably, disk layout strategies. Lougher and Shepherd [11] describe a file system for striping multimedia data across several disks with file system meta-data on a separate disk. They concentrate on constant rate streams but can support streams of different rates by varying disk I/O sizes so that reads happen at constant intervals. Kalns and Hsu [10] describe a video server implemented using the Vesta parallel file system on an IBM SP-1 parallel computer. It uses JPEG encoding at a constant frame rate but variable frame size, so some mapping from timestamp or frame to byte offset would be required in order to seek to a given frame or time offset, though the paper does not mention how seeking is implemented. Chang and Zakhor [3] describe a disk layout scheme for storing video encoded using a scalable coding on RAID disks in such a way as to be able to serve as many clients as possible from the array. The encoding used is constant rate. Vin and Rangan [18] analyze a method of interleaving data on disk to make the most of disk bandwidth but do not address variable-rate streams.

Another class of systems that include support for video include several research and commercial DBMSs [15] [2] [1]. Some of these systems support sophisticated query-by-image-content features, but do not support real-time data delivery — the essential feature of Calliope.

5 Conclusions

In this paper, we have sketched the architecture and performance characteristics of Calliope—a distributed, scalable real-time storage system for multimedia data. Calliope consists of a collection of off-the-shelf PCs running Unix. One machine, designated the Coordinator, maintains a database of the stored content and serves as the point of contact for new clients. The rest of the machines are Multimedia Storage Units (MSUs) which record and play back real-time content for clients. Calliope is extensible in order to support many types of networks, protocols, and audio or video encodings.

We have measured the performance of Calliope running on Pentium-based PCs under the FreeBSD operating system. Our test system can support 22 1.5 MBit/sec simultaneous constant-rate streams or 15 variable-rate streams (at an average rate of 635 to 877 KBit/sec and peaks up to 5.4 MBit/sec). We have shown that inexpensive hardware and a general-purpose operating system can be used to transmit many audio and video streams with good performance.

Acknowledgements

We would like to thank Bob Gray, Peter Bates, Sid Devadhar, Ravi Jain, Ben Melamed, Andy Ogielski, Marc Pucci, Norman Ramsey, and Yatin Saraiya for comments and suggestions.

References

[1] A. Biliris. The performance of three database storage structures for managing large objects. In *Proc. ACM SIGMOD Conference*, San Diego, California, June 1992.

[2] M. J. Carey et al. Shoring up persistent applications. In *Proc. ACM SIGMOD Conference*, Minneapolis, MN, May 1994.

[3] E. Chang and A. Zakhor. Scalable video data placement on parallel disk arrays. In *Storage and Retrieval for Image and Video Databases II*, volume SPIE 2185, 1994.

[4] D. Comer. The ubiquitous B-Tree. *ACM Computing Surveys*, 11(4), 1979.

[5] H. Eriksson. MBone: The multicast backbone. *Communications of the ACM*, 37:54–60, 1994.

[6] Ron Frederick. Experiences with real-time software video compression. In *Proceedings of the Sixth International Workshop on Packet Video*, pages F1.1–1.4, Portland, Oregon, September 1994.

[7] FreeBSD operating system. Information and source code available via HTTP from www.freebsd.org.

[8] C. Freedman and D. DeWitt. The SPIFFI scalable video-on-demand system. In *Proc. ACM SIGMOD Conference*, pages 352–363, 1995.

[9] A. Gelman, H. Hobrinski, L. Smoot, S. Weinstein, M. Fortier, and D. Lemay. A store and forward architecture for video on demand service. In *Proceedings of the IEEE ICC'91*, Denver, Colorado, 1991.

[10] E. Kalns and Y. Hsu. Video on demand using the Vesta parallel file system. In *Third Annual Workshop on Input/Output in Parallel and Distributed Systems*, April 1995.

[11] P. Lougher and D. Shepherd. The design of a storage server for continuous media. *The Computer Journal*, 36(1):32–42, 1993.

[12] Coding of moving pictures and associated audio (MPEG). Int'l Organization for Standardization/Int'l Electrotechnical Institute, September 1990. ISO/IEC JTC1/SC2/WG11.

[13] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. Internet draft: draft-ietf-avt-rtp-07, Internet Engineering Task Force, March 1995.

[14] B. Smith. Implementation techniques for continuous media systems and applications. Technical Report UCB/CSD 94/845, Computer Science Division—EECS, U.C. Berkeley, December 1994.

[15] M. Stonebraker and M. Olson. Large object support in POSTGRES. In *Ninth International Conference on Data Engineering*, 1993.

[16] TTCP network performance measurement program. Available for anonymous ftp from ftp.sgi.com:sgi/src/ttcp.

[17] VAT audioconferencing program. Available for anonymous ftp from ftp.ee.lbl.gov: conferencing/vat.

[18] H. Vin and P. V. Rangan. Designing a multiuser HDTV storage server. *IEEE Journal on Selected Areas in Communications*, 11(1):153–164, January 1993.

Biographies

Andrew Heybey (ath@bellcore.com) received his BS and MS in EECS from the Massachusetts Institute of Technology. He worked for several years at MIT before joining Bellcore in 1993. His interests include networks, real-time protocols, operating systems, and computer architecture.

Mark Sullivan (sullivan@bellcore.com) received his BS in Math Sciences from Stanford University and MS/PhD in Computer Science from the University of California at Berkeley. Since 1992, he has worked as a parallel computing researcher at Bellcore. His interests include database management systems, networks, fault tolerance, and poetry.

Paul England (england@bellcore.com) holds a BS in physics from the University of Birmingham, and a PhD in physics from Imperial College London. He has been a Research Scientist at Bellcore since 1986 and has worked extensively on the design and characterization of novel semiconducting, superconducting and optoelectronic devices. His current interests include applications and architectures for networked multimedia.