A Video Server Architecture Based on Server Arrays

Johannes Dengler and Werner Geyer

Development of a client-server video-on-demand-architecture based on server arrays.
Technical report October 1994.

Johannes Dengler, Werner Geyer
Institut Eurecom
2229, Route des Cretes
Sophia Antipolis
F-06560 Valbonne
{dengler,geyer}@eurecom.fr

Supervision: Christoph Bernhardt and
Prof. Ernst Biersack.

Table of contents

1 Introduction

We describe the prototypical implementation of a distributed video-on-demand (VOD) client-server architecture in a computer network. Our approach is focused on striping (distributing) each video over several servers. Our system consists of one or more video servers which each deliver an isochronous stream of video data, further of one or more clients with graphical user interface, and exactly one meta server.

The meta server is assumed to have complete knowledge of the location of video data, video servers and video clients. Hence the meta server may act as a negotiator between video clients and video servers. Both video servers and video clients must inform the meta server of their existence at start-up time. The location of the meta server is assumed to be well-known in advance.

Each video is assumed to be striped in a well-defined order over a discrete number of servers which each will contribute to the video service with their share of video information. Our approach does not restrict the number of servers on which a video may be distributed but we make the assumption that only one video server is run on each machine at a time.

A video client gives the user access to video on demand. It may request the meta server to negotiate the play of a video between all involved parties. It performs the synchronization of video data streams, the reassembly of the video and the display of the video at play time.

The system was developed on three Sparc 2 and two Sparc 10 workstations which are connected by Ethernet and an ATM network developed at Fore, Inc. Three of the workstations are equipped with Parallax video boards that allow real-time JPEG decoding and encoding. Each server is attached to a gigabyte disk and accesses it directly through a raw disk interface.

2 Installation and Package Contents

2.1 How to Get Started

The package is delivered within a packed tar file `VOD.tar.gz`. First step is to uncompress the file by typing

```
gunzip VOD.tar.gz
```

afterwards extract the included files by typing

```
tar -xfv VOD.tar
```

Now a VOD directory with the following subdirectories and files is created:

- `Client` contains source files for the VOD client and user interface
- `Makefile` global Makefile
- `MetaServer` contains source files for the VOD meta server
- `Server` contains source files for the VOD server
- `include` contains shared include files
- `lib` contains linkable library

Before starting the installation make sure that the system fulfills the following requirements, for otherwise your installation will fail:

- Parallax board + installed software,
- Fore ATM board + installed software,
- Tcl/Tk toolkit.
- We used the GNU g++ compiler.
- Eurecom's libtools.g tools library and the appropriate include files

The global Makefile is a kind of meta Makefile which calls the specific Makefiles for each module. It offers the following functionality:

- `make clean` cleans the subdirectories (removing *.o files etc.),
- `make subdirs` calls the Makefiles in the subdirectories,
- `make install` does both of the above things,
- `make wc` performs a word count on the source files in each subdirectory.

After typing `make install` the executable files are generated in each subdirectory. Before being able to use the system one has to distribute video data on the AV disks. The process of striping is described in chapter four. Once you have striped video data on your disks you have to launch first of all the metaserver by typing

```
metaserver
```

on a machine of your choice. The metaserver is connected via Ethernet and needs thus no ATM network. Afterwards one has to launch the servers on each machine with an AV disk.

```
server [meta_server_hostname]
```

If you don't specify a hostname a default is taken. To perform a playback of your videos the graphical user interface which is located in the `Client` subdirectory has to be called by typing:

```
vplayer [meta_server_hostname]
```

For the usage of the GUI refer to section "Graphical user interface" on page 30.

2.2 Makefile Switches

In all Makefile compiler paths have to be specified. The following switches can be set in the client Makefile:

- `-D__DEBUG` Enable the debugging protocol; during run-time debug information is written to a debug file in the `/tmp` directory (default setting is disabled).
- `-D__CONTROL` Enable the display of the buffer size in the user interface (default setting is disabled that means the actual position in the video is displayed).
- `-D__STATISTICS` Enable the protocol of statistical results (queue size, frame delay, parallax display time) to files in the `/tmp` directory (default setting is disabled).
- `-D__DISPLAY` Enable the display of arrived frames (default setting is enabled). If the display is disabled the client simulates the time for display. The delay can be specified by a define in client.h.

The server Makefile allows to set

- `-D__DEBUG` Refer above.
- `-D__NOAUDIO` Disable transmission of audio information (default setting is enabled). Currently the client does not support audio. With `__NOAUDIO` set the server will transmit ONLY the video information contained in each video frame whereas with the switch cleared it will leave the duty of having to parse through video frames up to the video client.
- `-D__STATISTICS` Enable logging of statistical information regarding the service time during scheduling periods, i.e. the amount of time needing to serve all videos during each alarm function call, and the time needed to serve read requests at the disk interface level.

The Makefile of the meta server knows only the `-D__DEBUG` switch.

All Makefiles can be called with a specific module as argument or nothing which means make all. Calling make with the argument depend provokes make to browse through all source files and to append the discovered dependences to the actual Makefile.

2.3 File Contents

The following describes the file contents in each subdirectory:

Subdirectory `Client/`

- `COPYRIGHT` Copyright note for the user interface
- `Makefile` Makefile for the VOD client
- `README` user info displayed in the user interface
- `bitmaps` contains icons for the user interface
- `buffer.C` source file for the frame buffer class
- `buffer.h` header file for the frame buffer class
- `client.C` main program for the VOD client, protocol machine
- `client.h` header file for the VOD client
- `scheduler.C` source file containing a video class and list for scheduling
- `scheduler.h` header file for the video class and list
- `streammanager.C` source file for several network interface classes
- `streammanager.h` header file for network classes
- `vplayer` user interface source script for the Tcl/Tk interpreter

Subdirectory `MetaServer/`

- `Makefile` Makefile for the VOD meta server
- `MetaServer.Data` contains the video data e.g. location, frame rate, size, etc.
- `jobmanager.C` defines a job class for the handling of the two phase commit
- `jobmanager.h` header file for job classes
- `metaserver.C` main program for the VOD meta server, protocol machine
- `metaserver.h` header file for the VOD meta server
- `streammanager.C` source file for several network interface classes
- `streammanager.h` header file for network classes
- `videodata.C` source file containing a class for video data storage
- `videodata.h` header file for video data storage class

Subdirectory `Server/`

- `Makefile` Makefile for the VOD server
- `cleanipc` Macro for deleting all shared memory segments
- `cleanps` Macro for deleting orphan server processes
- `diskmanager.C` source file containing disk access classes
- `diskmanager.h` header file for disk access classes
- `scheduler.C` source file containing scheduling class
- `scheduler.h` header file for scheduling class
- `server.C` main program for the VOD server
- `server.h` header file for the VOD server
- `streammanager.C` source file for several network interface classes
- `streammanager.h` header file for network classes
- `stripetool.C` source file for the frame distributing software
- `stripetool.h` header file for the stripetool
- `vstorage.h` contains file format for video data stored on the disks
- `VODServer.[host]` generated by the stripetool, contains all video storage info.

Subdirectory include/

- `architecture.h` contains general defines valid for all modules
- `basenet.h` network interface classes (delivered by `libtools.a`)
- `list.C` definition of C++ list templates
- `list.h` header file for list templates
- `pduA.h` pdu definitions for connection between client and meta server
- `pduB.h` pdu definitions for connection between meta server and server
- `pduC.h` pdu definitions for connection between server and client
- `prlx.h` parallax interface classes (delivered by libtools.a)
- `socket.h` socket network interface classes (delivered by libtools.a)
- `systeme.h` system interface classes (delivered by libtools.a)
- `toolserror.h` error handling class for the all tools (delivered by libtools.a)
- `two_p_protocol.h` pdu definitions for the two phase commit protocol
- `util.h` some useful macros and functions

Subdirectory `lib/`

- `libtools.a` collection of useful tools for network and parallax access

3 Network Protocol Reference

3.1 Meta Control Protocol

Since the meta server has complete knowledge of the location of video data and video servers it can act as both an information server for video clients and negotiator between video clients and video servers. While the exchange of video meta information concerns video clients and the meta server, the latter involves all parts of our architecture, i.e. the video client, the meta server and all video servers at which video data resides.

The meta control protocol is handled via connection-oriented TCP over Ethernet. At start-up time both video servers and video clients establish TCP connections to the meta server. If not specified in their command line video clients and video servers assume the meta server to be run on the internet address that corresponds to `DEFAULT_META_HOSTNAME` in architecture.h. We assume that the TCP ports are also well-known in advance.

```
#define  DEFAULT_META_HOSTNAME  "fuschia.cica.fr\0"
#define  META_SERVER_PORT       6353
#define  META_CLIENT_PORT       5463
```

These connections remain persistent for the video server's or video client's lifetime. The meta server therefore knows at any time which servers and clients are presently running.

The first word of each meta control pdu contains its type. In all other fields, meta control pdus may differ. A video clients issues a META_INFO_REQ by sending a `MetaInfoPDU`, filling only the field `id`:

```
struct MetaInfoPDU {
  int         id;
  int         sid;
  int         vid;
  int         sequence;
  VideoInfo   info;
};
```

It is responded by the meta server with one or more alike structs, each completely filled with meta information about one video. The field `sequence` contains `YES` for all meta information response pdus except the very last one. The field `info` contains the video information, a struct defined as this:

```
class VideoInfo {
 public:
  int          vid;
  char         name[64];
  char         subject[64];
  char         abstract[255];
  int          fps;
  int          width;
  int          hoehe;
  int          bandwidth;
  int          qfactor;
  int          nbitspixel;
  int          number_frames;
  int          greatest_frame;
  int          smallest_frame;
  int          number_server;
  HostInfo     source[MAX_SERVER];
};
```

A META_PLAY_REQ issued by a video client invokes a two phase commit. The meta play request is carried in a meta play pdu, defined as below.

```
struct MetaPlayPDU {
  int          id;
  int          sid;
  int          vid;
  int          data;
};
```

A TWO_P_PREPARE is carried in a meta prepare pdu. The field `sid` contains the unique stream identifier for this video request. `initial_frame` contains the initial frame or video segment number unique to each video server. All other fields contain meta information.[1]

```
struct MetaPreparePDU {
  int          id;
  int          sid;
  int          vid;
  int          fps;
  int          number_server;
  int          initial_frame;
  int          hoehe;
  int          width;
  int          qfactor;
};
```

The `prepare` is responded by each party with either TWO_P_ABORT or TWO_P_READY in the field `id` of a `MetaCommitReadyAbortPDU`. Each party informs the meta server about its ATM socket addresses in the fields `pinfo` and `hinfo`.

If all turns out well, the meta server issues another `MetaCommitReady-`

1. The field name `hoehe` was used deliberately for we could not compile our code with its English meaning `height`!

`AbortPDU` with TWO_P_COMMIT in the field `id`. The pdu addressed at the video servers contain the client's ATM socket addresses while the pdu addressed at the video client contains all server's ATM socket addresses,

Please refer also to section "Finite state machines" on page 9 for further information on the complete protocol machines of video client, video server and meta server.

3.2 Video Transmission Protocol

After a successful two phase commit between a video client, video servers and the meta server a video client has knowledge of all video servers that will deliver the video. The video transmission protocol deals with all client-server communication regarding the sub-video streams that will eventually make up a video stream at the video client site. Please note that this does not at any time involve the meta server.

The video transmission protocol is based upon UDP on ATM. Transmitted control pdus normally occupy only one datagram while video data (i.e. frames as of now) may occupy more than one datagram.

Since we assume video client and video server system clocks unsynchronized a mechanism for synchronization the video start time is needed. We present such a mechanism in section "General architecture" on page 22.

The video client may at any time (i.e. before the servers trigger a timeout) tell the servers to start sending video data. It does so by sending a CONTROL_ADJUST to each video server which will immediately respond by sending its initial video frame with a time stamp enclosed. The video client may then set each server's initial start time in each server's own time. It then issues a CONTROL_START to each server and encloses the calculated deadline for transmission of each server's initial frame.

In fact, the video client may at any time repeat this procedure in order to resynchronize the video servers or recover from a lost control packet.

During play each video server sends video frames contained in UDP datagrams with a constant frame rate to the video client. They are reassembled at the client site. As for now, we use no retransmission of control pdus.

The video client may issue a CONTROL_PAUSE which will cause the video server to at once stop sending frames. In order to continue playing the video the video client will have to undertake the same steps for resynchronization as for initially starting the video with CONTROL_START, only that this time it issues a CONTROL_CONTINUE to determine the difference, if any.

A CONTROL_STOP causes the video servers to immediately stop displaying and abandon the video. A CONTROL_ABORT does the same, only with a taste of an underlying error condition.

Immediately after the last frame has been sent to the video client the video server sends a CONTROL_END to signal the end of the video. The video client may then assume that no more data will arrive on the socket dedicated to this server.

All control traffic is sent in a common struct ControlPDU with its field `code` (e.g. CONTROL_START) containing the control command type and the field `id` containing `PDU_CONTROL`. The optional field `data` currently remains unused. Data traffic (e.g. frames) is sent in pdus of struct DataPDU. Its field `Header` of type `struct DataHeader` contains various information needed for reassembly of video frames and a time stamp. Its field `Data` contains up to 4,000 byte of video data.

```
struct DataPDU {
  DataHeader    Header;
  char          Data[4000];
};

struct ControlPDU {
  int           id;         // PDU identifier
  int           code;       // feedback command from client or server
  long          timestamp;  // synchronisation time, general purpose
  unsigned long speriod;    // scheduling period
  int           data;       // command related data eg. a frame number
};

struct DataHeader {
  int           id;        // PDU identifier
  int           SID;       // unique Stream identifier
  int           FrameNum;  // unique frame number
  int           FrameSize; // frame size in bytes
  int           PDUNum;    // number of PDU
  int           Offset;    // absolut distance (bytes) to begining of frame
  int           Length;    // amount of data included in Data[]
  long          stime;     // time stamp for sending time
  unsigned long speriod;   // scheduling period
};
```

3.3 Finite State Machines

From	incoming	to	outgoing	comment
IDLE	META_INFO_REQ	IDLE	META_INFO_RSP	Meta server responds to a meta info request issued by a video client by sending all available meta data to the video client.
IDLE	META_PLAY_REQ	PREPARE	TWO_P_PREPARE to video server and video clients	The meta server received a play request by a video client. It invokes a two phase commit between all parties.
PREPARE	TWO_P_ABORT from either one party	IDLE	TWO_P_ABORT to all other parties	Receiving an abort PDU from either one party the meta server issues aborts to the other parties.
PREPARE	TWO_P_READY	PREPARE		It received ready from one party but has yet other responds pending.
PREPARE	TWO_P_READY	IDLE	TWO_P_COMMIT to video client and video servers	All parties agreed to play the requested video. Meta server issues a commit.
	META_SERVER_ DISCONNECT			A video server disconnects. It will be removed from the server list.
	META_CLIENT_ DISCONNECT			A video client disconnects. It will be removed from the client list.

Table 1: Meta server finite state machine

From	incoming	to	outgoing	comment
IDLE	VIDEOLIST from vplayer	IDLE	META_INFO_REQ	Send meta info request to meta server. Such data are passed through to the GUI.
IDLE	META_INFO_RSP	IDLE	BEGIN to GUI, followed by all meta data.	Client received meta info data from meta server. Data is passed through to GUI.
IDLE	VIDEOREQUEST	IDLE	META_PLAY_REQ	An VOD request from the user interface is passed through as a play request to the meta server.
IDLE	META_PLAY_RSP	IDLE	FAILURE to GUI	Meta server decided to send abort before 2 p commit.
IDLE	TWO_P_PREPARE	NEGOTI-ATE	TWO_P_READY	Client is ready.
IDLE	TWO_P_PREPARE	NEGOTI-ATE	TWO_P_ABORT	Client could not allocate requested resources.
NEGOTI-ATE	TWO_P_COMMIT	NEGOTI-ATE	COMMIT to GUI	The phase commit was successful. The video can be played.
NEGOTI-ATE	TWO_P_ABORT	ABORT-ED	FAILURE to GUI	Two phase commit aborted by one party other than client.
NEGOTI-ATE	VIDEOPLAY from vplayer	NEGOTI-ATE	CONTROL_ADJUST to each server	Client invokes synchronization mechanism.
NEGOTI-ATE	Data	READY	CONTROL_START to each server	Start sending video frames.
NEGOTI-ATE	Data (incomplete)	NEGOTI-ATE	CONTROL_ADJUST	Incomplete data received.
NEGOTI-ATE		NEGOTI-ATE	CONTROL_ADJUST	Time-out.
NEGOTI-ATE		ABORT-ED		Time-out after several retransmissions of CONTROL_ADJUST.

Table 2: Client finite state machine

From	incoming	to	outgoing	comment
READY		PLAYING		When client has received a sufficient number of valid frames it starts displaying the video.
READY	Data	READY		Client receives a video frame.
PLAYING	Data	PLAYING		Client receives a video frame.
PLAYING	VIDEOPAUSE from user interface	PAUSE	CONTROL_PAUSE to each server	Pressed pause button in user interface is passed through.
PLAYING	VIDEOSTOP	STOPPED	CONTROL_STOP	Pressed stop button in user interface is passed through.
PLAYING	CONTROL_END	PLAYING		Video is finished. One server sends it's 'done'.
PLAYING	CONTROL_END	DONE		Last server has sent its 'done' and frame queue is empty.
PLAYING	CONTROL_STOP from one server.	STOPPED	CONTROL_STOP to all other servers.	Client receives a 'stop' from one server.
PLAYING		READY		Frame queue is empty.
PLAYING		ABORTED		Time-out.
PAUSE	VIDEOPLAY	NEGOTIATE	CONTROL_ADJUST	Client invokes synchronization mechanism
STOPPED		DELETE		Remove video out of service queue and unmap window. Clean up resources.
DONE		DELETE		Remove video out of service queue and unmap window. Clean up resources.
ABORTED		DELETE		Remove video out of service queue and unmap window. Clean up resources.
DELETE		IDLE		

Table 2: Client finite state machine

From	incoming	to	outgoing	comment
IDLE	Listinfo menu	IDLE	LISTMETAINFO to client	User selected 'Listinfo' in menu 'Video'
IDLE	BEGIN followed by meta data from client	IDLE		Open selection window and enable user to select a video.
IDLE	Selection button / double-click	PENDING	VIDEOREQUEST	User double-clicked on a video or pressed the 'Open' button. The selection window is being removed.
PENDING	FAILURE	IDLE		The client issued a 'Failure' to the user interface.
PENDING	COMMIT	READY		The 2 phase commit was successful. The user may now start the video. The play button is enabled.
READY	FAILURE	IDLE		The client issued a 'Failure' to the user interface.
READY	Play button	PLAYING	VIDEOPLAY	The user hits the play button. Pause and stop are enabled while play is disabled.
PLAY	STOP	DONE		The video has been fully played. Disable all buttons but rewind.
PLAY	Pause button	PAUSE	VIDEOPAUSE	The user hits the pause button.

Table 3: Vplayer finite state machine

From	incoming	to	outgoing	comment
PLAY	FAILURE	IDLE		The video client reports some error condition.
PAUSE	Play button	PLAY	VIDEOPLAY	The user hits the play button.
PAUSE	Stop button	DONE	VIDEOSTOP	The user hits the stop button.
PAUSE	FAILURE	IDLE		The video client reports some error condition.
DONE	Listinfo menu	DONE	LISTMETAINFO to client	User selected 'Listinfo' in menu 'Video'
DONE	BEGIN followed by meta data from client	DONE		Open selection window and enable user to select a video.
DONE	Selection button / double-click	PENDING	VIDEOREQUEST	User double-clicked on a video or pressed the 'Open' button. The selection window is being removed.

Table 3: Vplayer finite state machine

From	incoming	to	outgoing	comment
IDLE	TWO_P_PREPARE	RESERVED	OPEN + READ_BUFFER to disk manager.	Server received a prepare from the meta server. A new scheduled video is created.
IDLE	TWO_P_PREPARE	IDLE	TWO_P_ABORT	Server could not allocate resources for the video - probably UDP sockets.
RESERVED		REMOVE	TWO_P_ABORT	Server triggered time-out for disk read responses.
RESERVED	DISK_CORRUPT, SHM_CORRUPT,	IDLE	TWO_P_ABORT	Server could not allocate resources for the video - the video was not found or no shared memory could be allocated.
RESERVED	BUFFER_READY	READY	TWO_P_READY	A buffer has been read, all resources are allocated - the video can be played.
READY	TWO_P_ABORT	REMOVE		Server received abort from meta server - remove video.
READY	TWO_P_COMMIT	COMMIT		All parties are ready for this video.
READY		REMOVE		Server triggered time-out for commit.
READY	BUFFER_READY	READY		A buffer has been read by the disk manager.
READY	DISK_CORRUPT	REMOVE		Some read error occurred after the server issued its ready to the meta server.
COMMIT	CONTROL_ADJUST	ADJUST	Data + time stamp	Receiving an ADJUST control message the server responds by sending its next frame for the video to the video client.
COMMIT		REMOVE	CONTROL_ABORT to video client	The server triggered a time-out for the first adjust control message.
ADJUST	CONTROL_ADJUST	ADJUST	Data + time stamp	Receiving an ADJUST control message the server responds by sending its next frame for the video to the video client. The first adjust frame was probably lost.
ADJUST	CONTROL_START	PLAY		The server starts playing the video.
ADJUST	CONTROL_ABORT	REMOVE		The client issued an abort before playing the video.
ADJUST	BUFFER_READY	ADJUST		A buffer has been read by the disk manager.

Table 4: Server finite state machine (scheduler)

From	incoming	to	outgoing	comment
ADJUST		REMOVE	CONTROL_ABORT to video client	The server triggered a time-out for the start control message.
ADJUST	DISK_CORRUPT	REMOVE	CONTROL_ABORT to video client	Some read error occurred while playing the video. It can no longer be played be the server.
PLAY		PLAY	Data + time stamp	The server plays the frames at a constant frame rate.
PLAY	CONTROL_PAUSE	PAUSE		The client issued a pause. The server stops sending frames on the net.
PLAY	CONTROL_ABORT	REMOVE		The client issued an abort while playing the video.
PLAY	CONTROL_STOP	REMOVE		The client issued a stop. The server removes the video from the service queue and frees all resources by entering state REMOVE.
PLAY		PLAY	READ_BUFFER to disk manager	A buffer has been completely sent to the client. It should now be refilled by the disk manager.
PLAY	BUFFER_READY	PLAY		A buffer has been read by the disk manager.
PLAY	DISK_CORRUPT	REMOVE	CONTROL_ABORT to video client	Some read error occurred while playing the video. It can no longer be played be the server.
PLAY	CONTROL_ADJUST	ADJUST	Data + time stamp	Receiving an ADJUST control message the server responds by sending its next frame for the video to the video client.
PAUSE	CONTROL_ADJUST	ADJUST	Data + time stamp	Receiving an ADJUST control message the server responds by sending its next frame for the video to the video client.
PAUSE	CONTROL_ABORT	REMOVE		The client issued an abort during the pausing.
PAUSE		REMOVE	CONTROL_ABORT to video client	The server triggered a time-out for the adjust control message.
PAUSE	BUFFER_READY	PAUSE		A buffer has been read by the disk manager.
PAUSE	DISK_CORRUPT	REMOVE	CONTROL_ABORT to video client	Some read error occurred while playing the video. It can no longer be played be the server.
REMOVE		IDLE	CLOSE to disk manager	

Table 4: Server finite state machine (scheduler)

From	incoming	to	outgoing	comment
IDLE	OPEN	RESERVED	OPEN_OK	The request for a new video stream was passed through to the disk manager. It gets new shared memory and searches for the video on its disk. If all goes well, OPEN_OK is returned.
IDLE	OPEN	IDLE	OPEN_FAILED	Either no shared memory could be allocated or the video was not found on the disk.
RESERVED	READ_BUFFER	READY	BUFFER_READY	The first read request changes the videos state.

Table 5: Server finite state machine (disk manager)

From	incoming	to	outgoing	comment
RESERVED	READ_BUFFER	IDLE	DISK_CORRUPT	An error was encountered while reading the first block of the video from the disk.
RESERVED	SHM_CORRUPT	IDLE		The other process could apparently not attach to the shared memory. We drop the video and clear shared memory.
RESERVED	CLOSE	IDLE	CLOSE_OK	The video is requested to be closed at this early state. Probably never really used.
READY	READ_BUFFER	READY	BUFFER_READY	A disk request was successfully completed.
READY	READ_BUFFER	IDLE	DISK_CORRUPT	An error was encountered while reading the next block of the video from the disk.
READY	SHM_CORRUPT	IDLE		The other process has a problem with shared memory other than attaching to it. We drop the video and clear shared memory.
READY	CLOSE	IDLE	CLOSE_OK	The video is requested to be closed.
	PING		PING_BACK	The other (parent) process pings the disk manager. Disk manager responds by pinging back.
	SHUT_DOWN			The other (parent) process issues a shut down request, indicating that the disk manager should close all open videos, clear shared memory and exit. Never used because we preferred to use a signalling mechanism for terminating the processes instead.

Table 5: Server finite state machine (disk manager)

4 Implementation

4.1 Video Server

4.1.1 General Architecture

Our video server is designed to serve multiple isochronous data streams, in our case video streams, from a SCSI hard disk to multiple clients attached to an ATM network. We made use of the transparent UDP interface to ATM supplied by Fore, Inc. and a C++ class library for further transparency of different network addressing schemes developed by Laurent Gautier at Eurecom.

One video server consists of two discrete processes: the scheduler and stream manager process (parent) and a forked-out child as the disk manager process. While the disk manager process handles all disk all operations related to disk i/o its parent serves the video data to video clients and manages all communication with the meta server and video clients. The two processes are both completely event-driven and communicate by pipes, shared memory and signalling.

Pipes are uses to pass messages like requests, responses, events and errors to the other process. A read request for example is carried over a pipe and may eventually be responded by a read response.

Shared memory is used to share video data buffer space between the two processes. Shared memory space is allocated from the system dynamically as the demand for served videos grows. For each video a shared memory segment twice the size of one read block size (defined in server.h) is allocated and logically divided in half. Each half makes one shared memory buffer that may be handed over to the other process for writing or reading. Thus, our video server needs buffer of size two for each video. We put the allocation and deletion of shared memory space into the disk manager process because it may occupy a process for a substantial amount of time. Hence the allocation of shared memory space is confirmed between the two processes in the process of a two phase commit with the meta server.

We also make explicit use of signalling for gracefully terminating parent and child processes at one time when either one encounters and error condition or a signal that will lead to program termination. Such a termination

mechanism is needing to avoid orphan shared memory segments. When either one process finds itself in such a condition it send a SIGTERM to the other process. Before terminating the disk manager will clear all shared memory space while its parent will detach from them.

The video servers `main()` function establishes the pipes and forks out the disk manager. The parent and child processes are described in detail in the sections below.

4.1.2 Scheduler and Stream Manager Process

The parent's `main()` function resembles an event dispatch loop with incoming signals triggering events. Before entering the dispatch loop the objects scheduler (class Scheduler) and stream_manager (class StreamManager) are created.

4.1.2.1 Scheduler Class Description

The class Scheduler is vital to the video server. It handles all control traffic with the meta server, all control traffic with video clients, exchanges messages and handles event and errors from the disk manager and schedules video streams. It makes use of the class StreamManager which is described in the following section. In fact, Scheduler is instantiated with an object of class StreamManager as parameter. The other two parameters are file descriptors for the pipes to and from the other process.

When instantiated, class scheduler enables alarm signalling with the alarm function `Alarm()`. Later the interval timer gets enabled in Scheduler's method `SwitchOn()`. The timer is to 1 / `PERIODS_PER_SECOND` second, currently 1 / 25 second. When a SIGALRM signal is triggered the alarm function calls Scheduler's method `Schedule()`.

Scheduler contains a sorted list of video streams currently negotiated or in service. The video streams are of class ScheduledVideo (described below) and are sorted ascending to their unique stream identifier issued by the meta server. Scheduler has two methods for adding videos to the list and for removing them: `AddVideo()` and `RemoveVideo()`. The `Load()` method currently remains unimplemented but will eventually compute Scheduler's work-load, thus making way for admission control.

As mentioned above Scheduler is event-driven. The primary event Scheduler can handle is a SIGALRM signal. During each call of Scheduler's alarm function each video in the service list is inspected. All (secondary) events for a video will be handled by passing them to their respective event handlers of name `HandleEvent()`. Such events are:

- Messages arrived on the pipe from the child process.
- Packets arrived from the meta server on respective TCP socket.
- Control packets arrived from video clients on UDP sockets.

The events are processed according to the Scheduler's finite state machine. Please refer to table "Server finite state machine (scheduler)" on page 11.

`Schedule()` serves isochronous streams statistically at any real frame rate between 0 and 25 frames per second. In order to do so, it inspects each video during each scheduling cycle and decides whether a frame becomes due during the cycle. Thus frames may be scheduled early but never late. The duty to send the next frame to a video client is then passed to the respective object of class ScheduledVideo.

`Schedule()` uses Scheduler's methods `SendDiskManMsg()` and SendServerMetaPDU to communicate with the disk manager and the meta server.

```
class Scheduler {
public:

  char            * name;
  int               runs;
  unsigned long     time_slice;
  float             workload;        // 0 <= workload <= 1
  float             freetime;        // 0 <= freetime <= 1
  ScheduledVideoList video_list;
  int               number_videos;
  int               max_fps;
  StreamManager   * stream_manager;
#ifdef __STATISTICS
  TallyVariance     service_time;
  FILE            * scheduler_stat_file;
#endif

  Scheduler (StreamManager*, int, int);
  ~Scheduler();
  void              Schedule();
  void              AddVideo (ScheduledVideo *);
  void              RemoveVideo (ScheduledVideo *);
  float             Load();
  int               SwitchOn();
  int               SwitchOff();
  void              ReSchedule(int);
  void              HandleEvent(PipeBuffer *);
  void              HandleEvent(ScheduledVideo *, int, unsigned long);
  void              HandleEvent(MetaPreparePDU *);
  void              HandleEvent(MetaCommitReadyAbortPDU *);
  void              SendDiskManMsg(PipeBuffer *);

protected:
  long              checkpoint[3];
  struct itimerval  itimer;
  int               Listen, Speak;
  SELECT          * pipe_select;
  TIMEVAL         * select_timer;
  void              SendServerMetaPDU (char*, int);
};
```

4.1.2.2 ScheduledVideo Class Description

Upon request from the meta server the server creates an instance of class ScheduledVideo. The class contains all vital information regarding one video service. Such information are:

- the associated shared memory buffer identifier, shared memory address, and state
- the associated instance of class SubStream
- play-back information such as the initial frame number, the current frame number, the skip factor for fast forwarding, the deadline for starting play and various audio information currently unused.

```
class ScheduledVideo : public Video {
public:
  class SubStream  * sub_stream;        // interface to object sub_stream
  int                shmid;
  int                active_segment;
  int                frame_number;
  int                initial_frame;
  int                fforward;
  int                timeout;
  long               time_stamp;
  SHMBufferInfo      buffer[2];
  int                volbase;     // Baseline for volume control
  int                audioslice;  // Number of audio bytes per frame
  Audio_hdr          audio;        // Audio header from input file
                     ScheduledVideo ();
                     ~ScheduledVideo();
  int                Due();
  void               SendFrame(long, int = 1);
};
```

The class is inherited from class Video (in server.h) like disk manager's class ServicedVideo. The parent class Video contains common fields like

- the video identifier unique to every stored video
- the unique stream identifier issued by the meta server
- the video's default frame rate
- how many servers the video is stored on
- the video's state (refer to table "Server finite state machine (scheduler)" on page 11).

```
class Video {
public:
  int                vid;              // VID from disk manager
  int                sid;              // Stream ID from meta server
  int                fps;              // play speed [fps]
  int                n_frames;         // contains n frames
  int                n_server;         // number of servers
  int                status;           // reserved, playing, on hold, done
  unsigned long      position;
  int operator<(Video&);
  int operator==(Video&);
};
```

Besides its constuctor and its destructor which detaches from shared memory, class ScheduledVideo contains two methods. The method `Due()` decides whether a frame becomes due during the current scheduling cycle, using the field `time_slice` in class Scheduler.

Method `SendFrame()` is used to send one frame to a video client using. It is called with a time-stamp and an optional flag indicating whether the video should be advanced by one frame after sending (default is true). The method inspects the shared memory buffers and calls SubStream's method `SendFrame()` if it finds the next frame. Eventually finds a shared memory buffer empty and requests the disk manager to refill it by issuing

a READ_BUFFER request and declares the other shared memory buffer as active. The condition of no more frames being buffered in either one buffer as a result of severe disk manager overload can be triggered in `SendFrame()` as well.

If `-D__NOAUDIO` is specified as compile option `SendFrame()` explicitly extracts all audio information from the video data before sending to the video client. Only raw parallax JPEG data contained in a parallax movie frame is sent. However, with the option not set all the complete parallax movie frame as stored in a parallax movie file is sent to the video client. It is therefore left up to the client to parse through the video frame extraction of video and audio data.

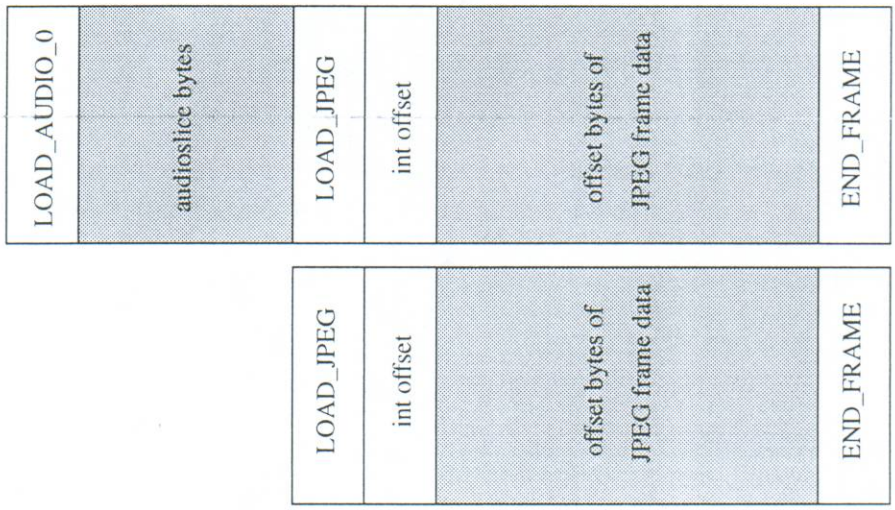


Figure 1: Video frame format with audio data contained (top) and without audio data (bottom).[2]

ScheduledVideo's method `SendFrame()` treats video frames regardless of whether audio data is originally contained. If not, the extraction by the switch `__NOAUDIO` makes a difference of six bytes (`LOAD_JPEG + sizeof(int) + END_FRAME`). Otherwise the extraction will cause the dropping of six + `audioslice` bytes.

4.1.2.3 StreamManager Class Description

In general, class StreamManager is responsible for all network i/o. This includes reading and writing on the TCP socket to the meta server, reading at writing on all UDP sockets to video clients, and frame fragmentation and reassembly.

---

2. Please note that the movie frame format described in Parallax's "Software Developer's Guide" is incorrect. It does not correspond to the format of movie files written by Parallax's 'videotool'.

The class makes explicit use of class SubStream which will be described below. It also uses the C++ network API developed at Eurecom for addressing transparency. The member `MetaStream` of class SOCKINET-STREAM provides the TCP stream to the meta server.

Class StreamManager is initialized with the meta server port number, its hostname and the maximum number of UDP datagrams per frame (defaults to 20). In the constructor it establishes a TCP connection with the meta server. The connection is closed in the constructor, thus remaining persistent for the video server's lifetime.

StreamManager's method for sending frames to video clients, `SendFrame()`, fragments a video frame into pieces of `PDUDATASIZE` (defined in pduC.h) bytes or less each and sends them as UDP datagrams. Its counterpart `ReceiveFrame()` reads video data contained in a datagrams and can be used to receive a video stream for storage.

`SendControlMsg()` and `ReceiveControlMsg()` are used to issue control pdus like an END to the video client. They are also carried in UDP datagrams. `SendMetaMsg()` and `ReceiveMetaMsg()` communicates with the meta server via TCP.

`CreateSubStream()` creates a new instance of class SubStream. Its counterpart `DeleteSubStream()` deletes it. Since the video client's address and port number are unknown at the point of time of creation of individual instance of class SubStream (during the two phase commit with the meta server), this information has to be carried into the object later on. To do so one has to call the method `InitRemoteAddr()` with the video client's hostname and its port number.

```
class StreamManager {
 public:
  // network connection stuff for Meta Server communication
  SOCKADDRINET     MetaLocalAddr;
MetaRemoteAddr;
  SOCKINETSTREAM*  MetaStream;
  char             HostName[32];
  // select object neccessary for listening on several file descriptors
  SELECT*          SelectStream;
  TIMEVAL*         Timing;
  // time stamp information
  long             sendtime;
  // segmentation  buffer
  DataPDU*         SegBuf;
  int              MaxSize;
  StreamManager(int, char*, int);
  ~StreamManager();
  SendFrame(SubStream*, char*, int, int, unsigned long);
  ReceiveFrame(SubStream*, char*, int);
  SendControlMsg(SubStream*, int, unsigned long = 0, long = 0, int = 0);
  ReceiveControlMsg(SubStream*, int*, unsigned long* = NULL, long* = NULL,
    int* = NULL);
  SendMetaMsg(char*, int);
  ReceiveMetaMsg(char*, int);
  SubStream* CreateSubstream();
  DeleteSubstream(SubStream*);
  InitRemoteAddr(SubStream*, char*, int);
  LookUpId(SOCKINETSTREAM*);
};
```

4.1.2.4 SubStream Class Description

An instance of class SubStream is created for each video client during the two phase commit protocol. Class SubStream connects to server to its video client by using UDP sockets (class SOCKINETDGRAM, for addresses MYSOCKADDRINETATM and SOCKADDRINETATM).

SubStream is created by passing an object of class SELECT to which its net descriptor will be added. Later on one can use the SELECT object for polling the socket.

SubStream's method `AssembleFrame()` generates UDP datagrams from video frames corresponding to section "Video transmission protocol" on page 8, and writes the datagrams to the corresponding socket. Video frames are fragmented into pieces of `PDUDATASIZE` (defined in pduC.h) bytes or less each.

```
typedef struct StoredVideoInfo {
  int             vid;
  int             position;
  int             size_in_blocks;
  int             n_server;
  JPEGHeader      jpeg_info;
}
```

The member `position` specifies the video's starting position on the raw disk as expressed in blocks of size `DISK_MAX_BUFFERS` (also defined in vstorage.h). The video is assumed to be stored sequentially, occupying `size_in_blocks` blocks of size `DISK_MAX_BUFFERS` bytes.

In its dispatch loop `Serve()` DiskManager performs a blocking read from the message pipe to the scheduler process. It handles all incoming requests, events and errors on a first-come-first-serve basis. Please refer table "Server finite state machine (disk manager)" on page 12 for complete reference of the disk manager - scheduler IPC protocol. When requested to open a new video DiskManager searches its library for the video identifier passed from the scheduler. If found, a new ServicedVideo instance is created and shared memory allocated as described. If all goes well the video is added to DiskManager's service list `video_list`.

DiskManager's methods for reading from and writing to the disk file descriptor are straight forward and need not be further explained. It should be made clear that class DiskManager is a best-effort approach without any performance optimization.

```
class DiskManager {
public:
  DiskManager  (char *, int, int);
  ~DiskManager ();
  int               rawdisk;
  int               ShutDown;
#ifdef __STATISTICS
  TallyVariance     service_time;
  FILE            * diskman_stat_file;
#endif
  int               Add (ServicedVideo *);
  int               Add (StoredVideo *);
  int               Remove (ServicedVideo *);
  int               Remove (StoredVideo *);
  int               Read  (off_t, int, char *);
  int               Write (off_t, int, char *);
  void              Serve();
  ServicedVideoList video_list;
protected:
  int               DID;
  StoredVideoList   library;
  int               Listen, Speak;
  int               number_videos;
  void              SendSchedulerMsg (PipeBuffer *);
};
```

4.1.3.2 StoredVideo Class Description

Class StoredVideo is used to store information on one video stored on the video server's disk. It contains the struct StoredVideoInfo shown in the previous section and new operators == and < to allow a StoredVideo to be sorted in ascending order by its unique video identifier.

```
class StoredVideo {
public:
  StoredVideoInfo   info;
  StoredVideo(StoredVideoInfo *);
  int operator==(StoredVideo&);
  int operator< (StoredVideo&);
};
```

4.1.3.3 ServicedVideo Class Description

Class ServicedVideo is used to store information on one video currently serviced or ready to be serviced by the disk manager. It differs from class StoredVideo in the fields determining the attached shared memory, and the operators == and < to allow a ServicedVideo to be sorted in ascending order according to the video's position on the disk as stored in `info.position`. If two serviced videos hold one same disk position (i.e. the video server plays the same video to different clients) the video's unique stream identifier is taken into the metric.

```
class ServicedVideo : public Video {
public:
  int               shmid;
  char            * shmaddr;
  int               shmsize;
  StoredVideoInfo   info;
  ServicedVideo (int, StoredVideo *);
  ~ServicedVideo();
  int operator==(ServicedVideo&);
  int operator< (ServicedVideo&);
};
```

4.2 Video Client

4.2.1 General Architecture

The video client is designed to play several video streams at the same time. The mechanism for administration of several streams is already included, yet the scheduling of videos with different frame rates remains unimplemented. A class Scheduler is also provided but not yet implemented. At present the client runs as one process. The architecture of the client is fully asynchronous and event-driven.

The functionality can be divided into two parts:

- Handling of metaserver communication, user interface communication and the communication with the servers, especially reading frames from the ATM network. These tasks are accomplished by a dispatch loop in `main()` of the client. The connection to the meta server is established using Ethernet TCP, the transmission of frames by the servers is based upon ATM UDP. For both techniques a transparent C++ network API developed at EURECOM has been used. The communication with the user interface is handled via standard i/o pipes.
- Display of frames in an X window with the correct frame rate. In order to guarantee real-time behavior an interval timer is set to 1 divided by the frame rate of the video. At each SIGALRM delivery the client process calls the `ScheduleFrames()` function and displays the frames. The realization of the display was also supported by a C++ Parallax API developed at EURECOM.

Those two parts are bound together by an instance of class SORTLIST, a list of all videos to be served by the client. The list is sorted by a unique stream identifier issued by the meta server and a priority code currently unused. The information stored in class Video allows the client to address the ATM network and the meta server, as well as to draw video data on the screen (refer to section "Video class description" on page 30).

4.2.1.1 Main Dispatch Loop

One of the main tasks of the client is the communication with the servers, the user interface and the meta server. Before entering the main loop the client creates instances, opens files for statistical tracing and debugging, and builds a **TCP** connection to the meta server. If no meta server is found or the connect command fails the client exits. After successfully connecting the client the connection is preserved for the client's lifetime. In the main loop a select call with well-defined waiting time is made. Then the following steps are periodically taken:

- **Browse through a list of Video objects to be served by the client:** For one Video stream the client creates one **UDP** socket for each server. For each Video object the corresponding **sockets** are now **checked** if readable. In case data is encountered on the socket it is peeked into (by using the MSG_PEEK command) and the client decides which kind of PDU arrived.

For each data PDU a reassembling function is called that can rebuild the frames that have been fragmented by the sending processes, respectively the servers.[3] While in the state VIDEO_NEGOTIATE the arrival time of the frames is used for an adjusting calculation of the servers start time as expressed in its own system time; finally the frames are dropped.

For each Video object **X event handling** is then performed. `Expose` and `Resize` events are caught and handled. `Expose` events switch the Parallax display to `PLX_VIDEO` that provokes a redraw of the image. `Resize` events are handled with Parallax's resize function.

Finally, the **states** of each Video are **checked**, state transitions are performed and time-outs are controlled and updated. We implemented one time-out for an empty frame queue and one for retransmission and negotiation while calculating the starting points of the servers as UDP does not guarantee the arrival of the packets. A time-out provokes Video object deletion and its removal out of the service queue.

- Checking user interface input:
  If data arrives on the standard input pipe it will be read and evaluated. User commands are executed in conformance with the protocol described in section "Finite state machines" on page 9.

- Checking meta server input:
  If the file descriptor for the meta server socket in the select object is set a look-ahead function is called to retrieve the identifier of the arrived pdu. The client then performs state transitions as described in section "Finite state machines" on page 9.

4.2.1.2 Alarm Function

Depending on the frame rate of the current video an interval timer is set. It calls the SIGALRM signal handler `ScheduleFrames()` which provides for the accurate display of the video. As at present only one video stream is supported by the client. For future multi-stream support a fixed frame rate should be chosen. Each video object should be decided on having a frame due to be displayed or not.[4] Browsing through the video list is however already implemented and performed during run-time.

3. The average size of a frame exceeds the maximum transfer unit of UPD. Hence it is necessary to transmit the frames in smaller units. At present a unit size of 4096 bytes is used.
4. This same mechanism has been implemented in the video server. It would have to be ported to the video client.

The function evaluates each video's state. Two states that are most important for the scheduling function:

- **VIDEO_READY:**
  In this state the size of the frame queue, i.e. the number of playable frames, is checked. Once encountering a frame queue half filled a transition to state VIDEO_PLAYING is performed. Thus we can guarantee that at start-time of the video and in the case of slow servers the frame queue gets partly refilled before any playback may take place. This strategy optimizes the use of our buffer.

- **VIDEO_PLAYING**
  In the state VIDEO_PLAYING the function also checks the size of the frame queue. If found empty it follow the transition to VIDEO_PLAYING and sets a timer. To guarantee an accurate playback the client follows the following strategies:

  It calculates the difference between the sequence number of the frame that is currently due and of the first frame in the queue. A negative difference indicates that the frame is late: it is discarded and removed from the queue. On a positive distance a frame will be displayed if the difference is either zero or greater or equal to the number of servers involved. This makes it possible to react in real-time to frame loss and server shutdowns while being able to continue displaying the video. If a server shuts down for any reason the video will continue to play back with the same frame rate, yet with every *n*th frame missing. For a great number of servers this is hardly noticeable to the user. A mechanism for stream recovery has to be developed in the near future.

  The client can be blocked temporarily by the X server when the user moves or places other windows. It may also be blocked by launched processes. In these cases the socket buffer overflows and arrived packets are dropped. The client detects packet drop by calculating the distance between two consecutive alarm function calls. With this value a new due frame number is calculated. If the distance between the current frame number and the calculated new due frame number is greater than the buffer space currently free the frame queue and the socket buffer are cleared. The current frame counter is set to the new value. The expected frame numbers for the various sockets are updated. This mechanism can react to large interruptions while

smaller ones are buffered.

4.2.2 Class Description

4.2.2.1 FrameBuffer and Buffer Class Descriptions

The Buffer class contains the video and playback data for one frame received. `FrameNum` is used for play-back as mentioned above. `status` can be either one of

- `BUFFER_FREE`
- `BUFFER_RESERVED`
- `BUFFER_FULL`

After a complete frame has been read a new buffer slot is allocated for the next expected frame (`BUFFER_RESERVED`). A playable buffer is set with `BUFFER_FULL`.

`lTime` is used to calculate the delay between subsequent frames in the queue. This mechanism can for example be used for detecting the delay of certain servers. `data` is a pointer to the continuous media data, respectively JPEG compressed images.

```
class Buffer {
public:
  int        FrameNum;
  int        FrameLength;   // absolut length in bytes
  u_int      status;        // reserved, free, ...
  long       rTime;         // remote time, means send time
  long       lTime;         // local time, means receive time
  char*      data;          // frame data

  Buffer ();
  ~Buffer();

  int operator<(Buffer&);
  int operator==(Buffer&);
};
```

The Buffer constructor sets `status` to `BUFFER_FREE` and allocates an amount of memory sufficient for storage of one frame (defined in `MAXVIDEO`). The maximum size of one frame is not known in advance. The memory is freed by the destructor.

The FrameBuffer class makes use of the Buffer class. It contains an array of Buffer elements called `BList` and a sorted list of pointers to Buffer slots called `PList`. `PList` is the actual frame queue. The queue is sorted in ascending order by the frame number.

While calling the FrameBuffer constructor with the requested buffer size

a new `BList` object is created. Initially, the frame queue `PList` is empty. The `GetFreeBuffer()` function returns an index to a free buffer slot in `BList`. This index can be used to get access to the slot by using `GetBuffer(int)`.

`AddFrame()` inserts a buffer slot into the frame list and increases the number of playable frames by one. During the insertion SIGARLM is blocked to avoid simultaneous access in the alarm function and in `main()`. The counterpart `RemoveFrame(int)` removes either a specific frame number or the first frame from the queue when called without an argument. Specific frame numbers must be removed during the negotiation process between client and servers because adjust frames are not displayed. `QueueSize()` returns the number of playable frames in the queue, `QueueEmty()` returns `TRUE` on an empty frame queue.

```
class FrameBuffer{
 public:

  FrameBuffer(int);
  ~FrameBuffer();

  GetFreeBuffer();
  Buffer* GetBuffer(int);
  BufferSize();

  // playable frame list sorted by framenumber
  FrameQueue       PList;

  AddFrame(Buffer*);
  RemoveFrame(int = 0);
  QueueSize();
  QueueEmpty();

 protected:
  // specifys the amont of playable frames in the buffer
  int       PlayableFrames;
  int       buffer_size;

  // buffer array provides elements to be inserted in PList
  Buffer*          BList;
};
```

4.2.2.2 SubStreamHandler Class

The SubStreamHandler class realizes the communication with the servers that deliver together one video stream. Hence each video stream contains an own SubStreamHandler object. It uses the communication API developed at EURECOM. An array of UDP sockets is created, one for each server delivering a sub-stream of video data. We expect frame number zero arriving on array element zero, frame number one on array element one, etc. Since the location of each video are known by the meta server it will distribute the correct destination ports to all involved servers during the two-phase-commit protocol. The remote and local addresses are

stored in SOCKADDRINETATM objects. Furthermore the SubStreamHandler contains variables for frame reassembling, statistics and control of the sub-streams. And it finally contains a FrameBuffer object for frame storage as described in section "FrameBuffer and Buffer class descriptions" on page 24.

```
class SubStreamHandler {
 public:
  // ********************************************************************
  // network connection stuff for server communication
  // ********************************************************************
  FDSOCKINETDGRAM*   ServerStream;     // UDP-Socket for server substreams
  MYSOCKADDRINETATM* LocalAddr;        // Address Client      (INET-Server)
  SOCKADDRINETATM*   RemoteAddr;       // Address Server      (INET-Client)

  // ********************************************************************
  // Variables for frame reassembling
  // ********************************************************************
  int*               actual_slot;      // actual buffer slot
  int*               pdu_number;       // expected PDU number
  int*               frame_number;     // expected frame number
  DataHeader*        header;           // PDU header
  long*              receive_time;     // stores receive time
  long*              server_timeout;   // catch server shutdown

  ReassembleFrame(int);
  ResetReInfo(int);

  // ********************************************************************
  // variables for statistics & control
  // ********************************************************************
  int                NumberServer;
  unsigned long*     ScheduledPeriod;
  long*              ReceiveTime;
  long*              SendTime;
  int*               ExpNum;
  int*               last_lost;        // last lost frame
  int*               lost_frames;      // number lost frames
  int*               last_unexpected;  // store last unexp for calculation
  int*               status;           // substream status
  long*              interframe_delay; // control serverspeed
  long*              expected_delay;   // might be different on frame loss

  CalculateExpectedDelay( int, int, int = 0);
  DisplayReport(FILE*);
```

is given in our environment for the reason that Fore's UDP relies on connection-oriented ATM with Fore's SPANS. `ReassembleFrame()` may return one the following values:

- `RE_FAILED`
- `RE_AWAIT`
- `RE_SUCCESS`

If the last pdu of a frame has been successfully read into the buffer `RE_SUCCESS` is returned. The buffer is inserted into the frame queue, the expected delay is calculated and written into the debug file, and all buffer variables are set. If further pdus are expected `RE_AWAIT` is returned. The `ResetReInfo()` function resets all reassembling variables after a `RE_FAILED`.

`CalculateExpectedDelay()` calculates the expected delay between the currently arrived frame and the last one. This value increases when frames are lost. Regularly however it stores a multiple of the frame rate in milliseconds.

`DisplayReport()` writes information about the number of lost frames, the last lost frame number and the total number of lost frames into the debug file.

`InitRemoteAddr()` is called after a COMMIT from the meta server. The server destination addresses are initialized with hostname and port number.

The synchronization of the servers is performed by the `AdjustServerLast()` function. It is called with the current server index and the frame rate for the video. After having received a CONTROL_ADJUST each server sends an adjust frame with a local time stamp to the client. When all frames have been arrived the function calculates the individual starting time of each server. The latest frame is taken as reference; all other servers are adjusted relative to this frame. The starting times are calculated in each server's own system time as follows:

The expected distance to the reference frame is added to the scheduled time of the frame. This value is corrected by the actual arrival distance to the reference. Finally, a time buffer is added. It equals of the maximum round-trip time for all servers plus a buffer defined in `TIME_BUFFER_IN_SLICES`. Even if frames are out of phase, i.e. the expected frames are not received in descending order, the function per-

forms a correct calculation for each server, calculating the expected distance using the variable `ExpNum`.

CONTROL_START pdus are then sent to all involved servers. The adjust frames are removed from the frame queue. This strategy minimizes the delay between a play request of the user and the actual starting point of the video. As tests show servers are well-synchronized. This mechanism is also used to restart playing after a pause.

`LookUpId()` performs a message peek into a UDP packet. It returns the pdu identifier or -1 on failure.

`SubStreamEnd()` checks the states of each server stream and returns `TRUE` if all server's states are `SUBS_END`.

`AbortSubStreams()` sends an CONTROL_ABORT to all servers but the one who provoked the abort. It returns `TRUE` on success.

4.2.2.3 StreamManager Class Description

The StreamManager is generally responsible for all network i/o. It offers almost the same functionality as the video server's StreamManager (refer to section "StreamManager class description" on page 18). The client's StreamManager contains additional functions to connect and disconnect to the meta server. The functions `CreateSubStreamHandler()` and `DeleteSubStreamHandler()` correspond to `CreateSubStream()` and `DeleteSubStream()` in the video server's StreamManager. Though perhaps unnecessary these functions provide transparent access to the network.

```
class StreamManager {
 public:
  // network connection stuff for Meta Server communication
  SOCKADDRINET     MetaLocalAddr;     // Address Server          (INET-Server)
  SOCKADDRINET*    MetaRemoteAddr;    // Address Meta Server     (INET-Client)
  SOCKINETSTREAM*  MetaStream;        // TCP socket Server       (INET-Server)
  char             HostName[32];      // Local hostname
  int              PortNumber;        // Local Port

  // select object neccessary for listening on several file descriptors
  SELECT*          SelectStream;
  TIMEVAL*         Timing;

  StreamManager();
  ~StreamManager();

  SubStreamHandler* CreateSubStreamHandler(int);
  DeleteSubStreamHandler(SubStreamHandler*);

  SendMetaMsg(char*, int);
  ReceiveMetaMsg(char*, int);

  LookUpId(SOCKINETSTREAM*);
  ConnectMetaServer(int, char*);
  DisconnectMetaServer();
};
```

4.2.2.4 Video Class Description

As mentioned above the Video class binds the two parts of the client together. Video streams are administered in a sorted list of Video objects. Both the main loop and the alarm function browse through the list to obtain the necessary information.

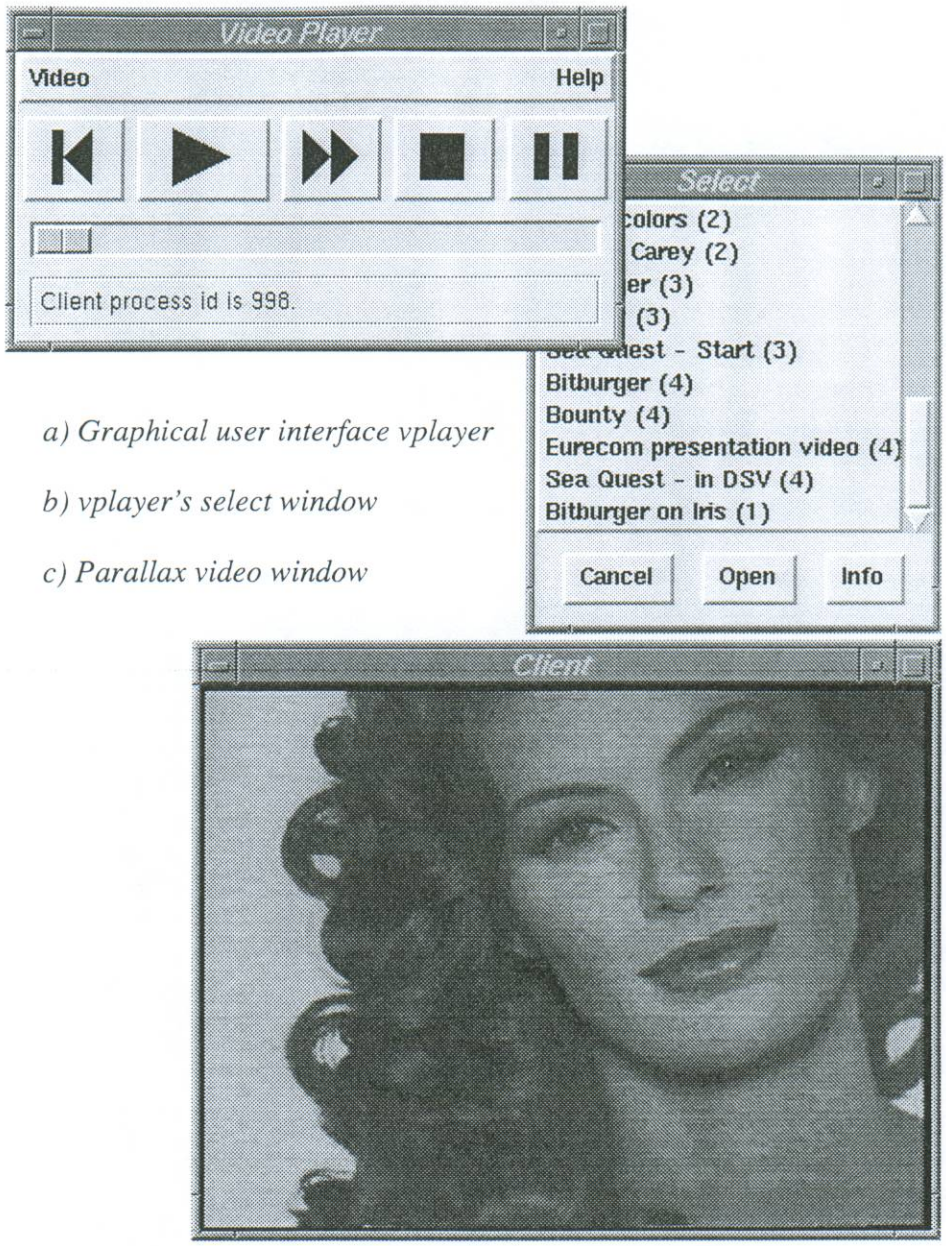
Network access is rendered by a pointer to a SubStreamHandler object which is explained in section "SubStreamHandler class" on page 26. Furthermore a Video contains a PRLX object, a CIMAGE object to store JPEG data in a common parallax format and some X stuff for event handling. The actual state of a Video is stored in `status` which is used for control of the stream during its lifetime. For the protocol machine refer to section "Finite state machines" on page 9.

```
class Video {
public:
  int                       sid;              // unique stream id
  int                       vid;              // VID from disk manager
  int                       priority;         // VIV (Very Important Video)
  long                      currentposition;  // next frame to be played
  int                       fps;              // play speed [fps]
  int                       status;           // idle, adjust, playing, ...
  class SubStreamHandler*   substreamhandler; // related sockets and buffer

#ifdef __DISPLAY
  PRLX*                     parallax;         // parallax object for output
  MYCIMAGE*                 cmdata;           // data structure for parallax
  int                       new_expose;       // catch X event
  XEvent*                   xevent;
  HandleXEvent();
#endif

  int                       retransmitted;    // number of retransmissions
  int                       end;              // indicates end of video
  long                      play_timeout;     // timout during playback
  long                      neg_timeout;      // timeout on lost frames
                                              // during negotiation

  Video(int, int, int, int, class SubStreamHandler*, int, int, int);
  ~Video();

  int operator<(Video&);
  int operator==(Video&);
};
```

Each Video contains a time-out for playback and one during negotiation with the severs. After a certain number of retransmissions of the CONTROL_START the negotiate time-out is executed. The number of retransmissions is defined in `RETRANSMISSIONS`.

4.3 Graphical User Interface

The Tcl scripting language and the Tk toolkit have drastically reduced development time of our graphical user interface (GUI) to the video client. We used Extended Tcl and standard Tk widgets for the script file `vplayer` containing all GUI code. Vplayer executes the C++-coded client described in the previous section with command pipeline for standard input and output. The IPC between vplayer and client is based upon user signalling and works completely asynchronous. Please refer to section "Vplayer finite state machine" on page 10 for complete reference.



a) Graphical user interface vplayer

b) vplayer's select window

c) Parallax video window

Figure 2: Graphical user interface

The GUI may post as many as five windows excluding client's Parallax window, three of which are shown in figure "Graphical user interface" on page 31.

- The main window "Video Client" provides a menu bar for video selection, exit and a help facility, and several buttons for video control. At the very bottom it displays a line of state information like "No selection." when no video has yet been selected.
- The selection window "Select" provides a listbox with scrollbar and buttons to accept the selection, cancel or show more information. Note that only one video can be selected for playback at a time but multiple

videos may be selected for further information.

- The "Video Info" window displays all available information (i.e the data passed from the meta server to client and from client to vplayer) on selected videos, e.g. their approximate play time, their frame rate, title, abstract etc.
- When the user chooses "About" in the help menu vplayer displays a notice about the authors as contained in the file `COPYRIGHT`; "Readme" displays the current directory's file `README`.

Vplayer enables its main window buttons according to the video's state. This way false user interaction can be avoided. When no video is selected neither are the buttons. A successful "Open" on a video causes the play button to be enabled. A successful video start causes the pause and stop buttons to be enabled. A video playback without failure or abort from either video client or video servers causes the leftmost) "Back" button to be enabled. It is in fact an abbreviation for again selecting the same video in the video list, causing the client to issue a new VODRequest to the meta server.

4.4 Meta Server

The meta server is the leanest process of our architecture, yet perhaps the most important one for successful video playback. Without it the video client will fail to request videos from the servers for their location will remain secret.

The meta server's stream manager works very much like the video server's and the video client's stream managers but keeps an additional sorted list for all client and server connections. This way it may at any time decide which servers are currently running to perhaps cut short on unsuccessful video requests. The server list is sorted by the server host-names, the client list is sorted by the unique connection identifier. Unique connection identifiers are used to be able to serve multiple sessions with one single client at a time, e.g. an info request while in the process of negotiating a video play. The connection identifier is unique to the video client.

A unique connection identifier is not needed for the video servers because their hostnames together with the stream identifier used during negotiation are unique.

All video-on-demand requests to the meta server are administrated in a

list sorted by their stream identifiers. The complete two-phase-commit protocol is handled on every entry of this list while meta server's directory service is handled completely separate.

4.4.1 Control Flow

In its main() function meta server subsequently does this:

- It accept new client connections on socket `META_CLIENT_PORT` and new server connections on socket `META_SERVER_PORT`. The socket numbers are hard-coded for the moment and obviously chosen at random.
- It handles events from clients in `HandleClientEvent()`.
- It handles events from servers in `HandleServerEvent()`.
- Finally, it updates all changes in its job list and performs state transitions. The state transitions are performed conforming to table "Meta server finite state machine" on page 9. Please refer for complete reference.

4.4.2 Class Description

4.4.2.1 VideoEntry Class Description

The meta server stores all meta information in instances of class VideoEntry. An entry contains all necessary information for both directory service and play-back negotiation.

The VideoEntry class is inherited from class VideoInfo. Besides its empty constructor and destructor it contains operators for < and == to enable the creation of a sorted list with VideoEntry objects.

```
class VideoInfo {
 public:
  int         vid;
  char        name[64];
  char        subject[64];
  char        abstract[255];
  int         fps;
  int         width;
  int         hoehe;
  int         bandwidth;
  int         qfactor;
  int         nbitspixel;
  int         number_frames;
  int         greatest_frame;
  int         smallest_frame;
  int         number_server;
  HostInfo    source[MAX_SERVER];
};
```

The entries are initially retrieved from the file MetaServer.Data in the

meta server directory. The file's format is straight forward. Each information entry, e.g. the frame rate, is followed by a line feed character. This makes a total of 14 lines for all meta information, plus the hostnames of all video servers the video is stored on. Please note that the fields 'title', 'abstract' and 'subject' must not contain line feed characters.

The first line of MetaServer.Data is expected to contain the total number of entries in the file. A wrong number may cause unexpected results.

The video list is contained in a instance of class VideoDataBase. This class contains several methods for video information retrieval.

```
class VideoDatabase {
 public:
  int          listsize;
  VideoList    videolist;

  Init(char* = NULL);

  VideoEntry* RetrieveInfo(int);
  VideoEntry* RetrieveInfo(char*);

  VideoEntry* FirstVideo();
  VideoEntry* NextVideo();
};
```

4.4.2.2 Job Class Description

As stated above, the Job class handles the two-phase-commit protocol and contains the protocol machine. Each client is identified uniquely by its connection identifier but may occupy multiple stream identifiers. Several client sessions are thus multiplexed through one TCP connection.

The fields `client_status` and `server_status[]` contain the job's state. The field `VideoInfo` points to the meta data structure containing all meta information regarding the negotiated video. `server_portinfo` and `client_portinfo` contain port numbers at the servers' and client's sites. They are buffered until issuing a commit at the end of the two-phase-commit protocol, informing client and servers of their peer port numbers. `prepare_timeout` holds a timer to enable a time-out for negation.

Job class' constructor is called with the client's connection identifier. The method `CheckServerConnections()` examines whether all servers needed for the requested play-back currently run. `CheckReadyVote()` returns true if all parties involved in the two-phase-commit have sent their ready vote. `GetServerByName()` returns the index of server in the VideoInfo structure or -1 on failure.

```
class Job {
 public:
  int          sid;
  int          cid;
  int          vid;
  int          client_status;
  int          server_status[MAX_SERVER];
  VideoInfo*   videoinfo;
  PortInfo     server_portinfo;
  PortInfo     client_portinfo;

  long         prepare_timeout;

  Job(int);
  ~Job();

  CheckServerConnections();
  CheckReadyVote();
  GetServerByName(char*);

  HandleServerEvent(int, char*, char*);
  HandleClientEvent(int, char*);

  int operator<(Job&);
  int operator==(Job&);
};
```

4.5 Stripetool

We have developed a small tool that can be used to stripe (distribute) videos onto the video server's disks in our computer network. It opens a video server's disk and writes the video's share of information intended for the very server.

Since we were short of time stripetool has several disadvantages. First of all, it does not operate on the networks but works only on the local machine's hard disks. In order to stripe a video onto several video servers one must use the tool more than once. This causes the process of striping a video to be rather complicated. We intend to feature striped write-back in our video server. Once this is done stripetool is no longer needed.

Furthermore, the issue of a net-wide unique video identifier for a newly striped video at run-time, consistent throughout all video servers, remains unresolved.

4.5.1 Retrieving Storage Information

Stripetool can be used in two ways. It can retrieve all video storage information from all video servers by executing with no arguments (`stripetool`). The tool will then open the files `VODServer.<hostname>` and show for each video available on each server

- its video identifier
- the starting position of the disk as expressed in `DISK_BUFFERSIZE` bytes and the size in disk blocks
- how many servers it has been striped onto
- its default frame rate and how many frames it contains
- width, height, parallax compression factor and the size of the audio portion contained in each frame.

```
---> Video server ficoide
        VID 1000, position 100, size 190, 1 server
        fps:    16
        frames: 6719
        width:  384
        height: 288
        qfac:   100
        audio:  500
        VID 1001, position 290, size 153, 1 server
                [...]
VID 1001, position 133, size 42, 1 server
        fps:    16
        frames: 1642
        width:  384
        height: 288
        qfac:   130
        audio:  500
---> capucine stores 6 videos. Next available vid is 4005.

===> Next globally unique vid is 4006
```

For each server it computes the next available video identifier. Given the next free video identifier for each server stripetool now computes a globally unique video identifier that may be used during the next striping of a video.

4.5.2 Storing a Video

Stripetool can be used alternatively for storing a video on a video server's hard disk. It stripes the video according to the command line arguments

```
stripetool <file> <n_server> <this_server> <vid>
```

- `file` specifies the path to a valid Parallax movie file.
- `n_server` specifies the total number of servers the video will be striped onto
- `this_server` may range from 0 to `n_server`-1, indicating the order of servers during play-back.
- `vid` indicates the video identifier that is used for this newly stored video. It can be computed using stripetool with no arguments accord-

ing to the previous section.

Stripetool parses through the Parallax movie file's header and all frames that follow. It extracts exactly the portion of video information needed by the video server of the machine it runs on, according to the command line arguments. All header information is preserved for future use in the `StoredVideoInfo` structure.

```
typedef struct StoredVideoInfo {
  int             vid;
  int             position;
  int             size_in_blocks;
  int             n_server;
  JPEGHeader      jpeg_info;
};
```

Stripetool collects a maximum number of video frames into memory buffer of size `VIDEO_BUFFERSIZE` before writing the whole buffer to the disk. Please note that the number of frames per disk block may vary. We experienced numbers between 30 and 50 frames per block. The video storage block format is shown in figure "Video storage block format" on page 34. Note that every on video frame corresponds to figure "Video frame format with audio data contained (top) and without audio data (bottom)." on page 18.

Stripetool writes as many disk blocks as necessary, in sequential order.

Figure 3: Video storage block format

Both meta server and video server must obtain information on the newly stored video. The video server gets all information contained in the `StoredVideoInfo` structure, including the video identifier, the storage position and the length in blocks. Stripetool appends the structure *directly* to the file `VODServer.<hostname>` of the local host. After a re-start the video server will know the new video and be able to serve it to video clients.

```
<command line argument vid>
Fill in title of <command line argument filename>
Fill in subject here
Fill in abstract here
<frame rate>
<width>
<height>
0
<q-factor>
<colors>
<frames>
0
0
<command line argument n_servers>
Add the server names here
```

The portion of meta information has to be added manually to the meta server's file `MetaServer.Data`. This is quite reasonable because the administrator might want to add title, subject, abstract etc. to the meta entry. Stripetool provides a little aid, however. It creates a generic entry and appends it to the file `MetaServer.Data.<hostname>`. Once the video has been striped onto all video servers the administrator may fill in any one generated entry and append it to the file `MetaServer.Data` on which the meta server relies.

Please refer to section "Meta server" on page 32 for complete reference of `MetaServer.Data`'s file format.

A Video Server Architecture Based on Server Arrays

Development of a Prototype Client Server VOD Architecture

Johannes Dengler, Werner Geyer

Institut EURECOM
2229, Route des Cretes
Sophia Antipolis
F-06560 Valbonne

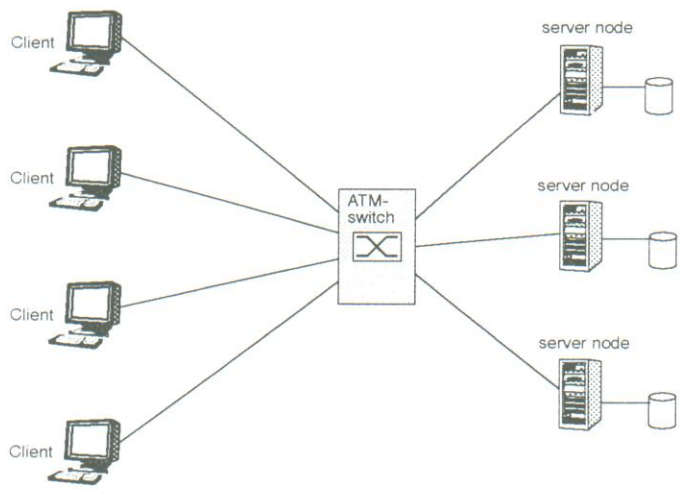
{dengler, geyer}@eurecom.fr

Overview

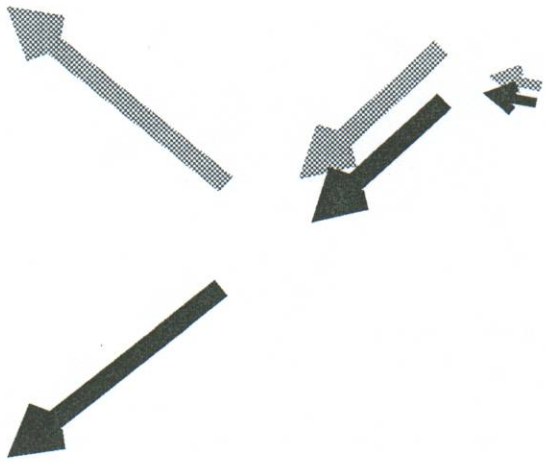
- Introduction
- Architecture
- Protocols
- Implementation
- Summary
- Future work
- Demonstration

Requirements for video server

- Isochronous stream
- Efficient usage of storage capacity and bandwidth
- Scalability and load balancing

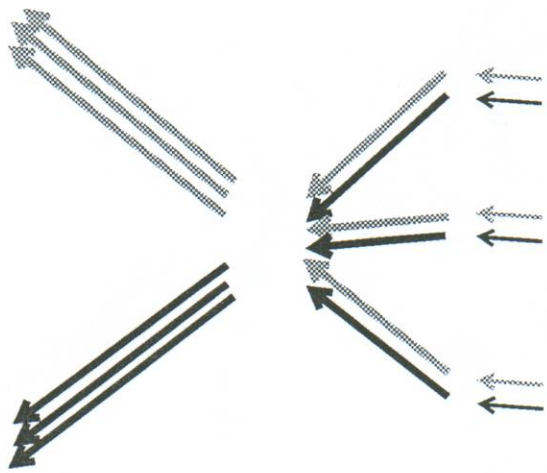


The autonomous server array



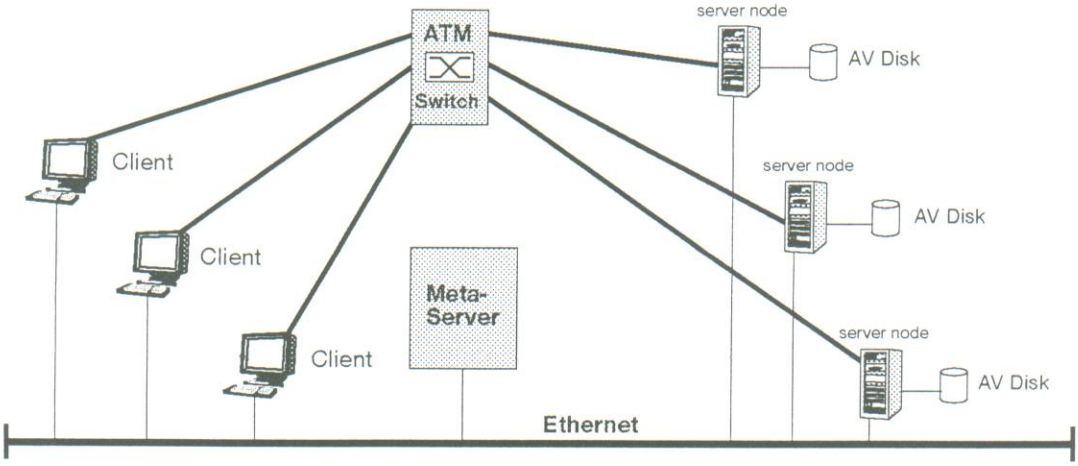
- Load concentrated on one server and its network connection

The striped server array

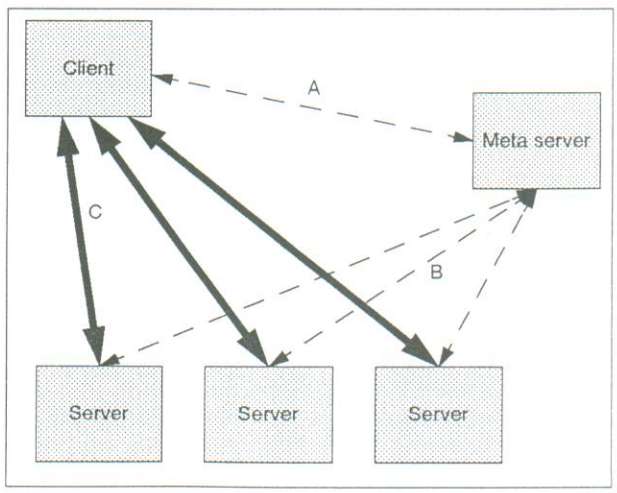


- Load distributed to all servers and all network connections of servers

System architecture of the prototype



- Low-bandwidth for control streams, high-bandwidth for video streams
- At present: no dedicated servers and clients



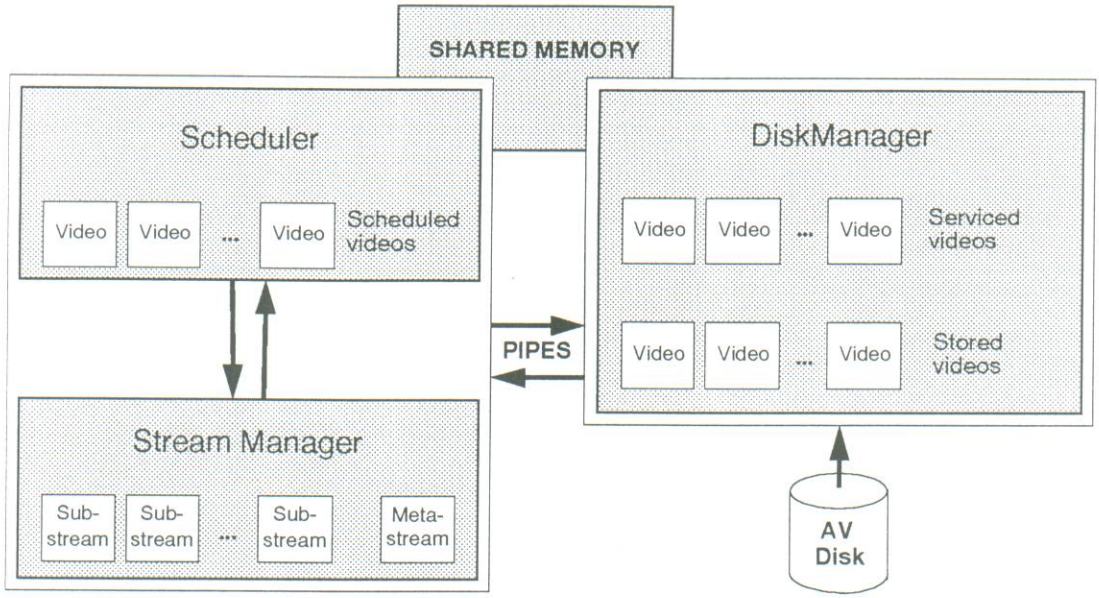
Meta control protocol (A/B)

- Directory service
  - Info Req/Rsp
- Session Establishment
  - Play Req/Rsp
  - Two phase commit
    - ➥Prepare
    - ➥Ready/Abort
    - ➥Commit

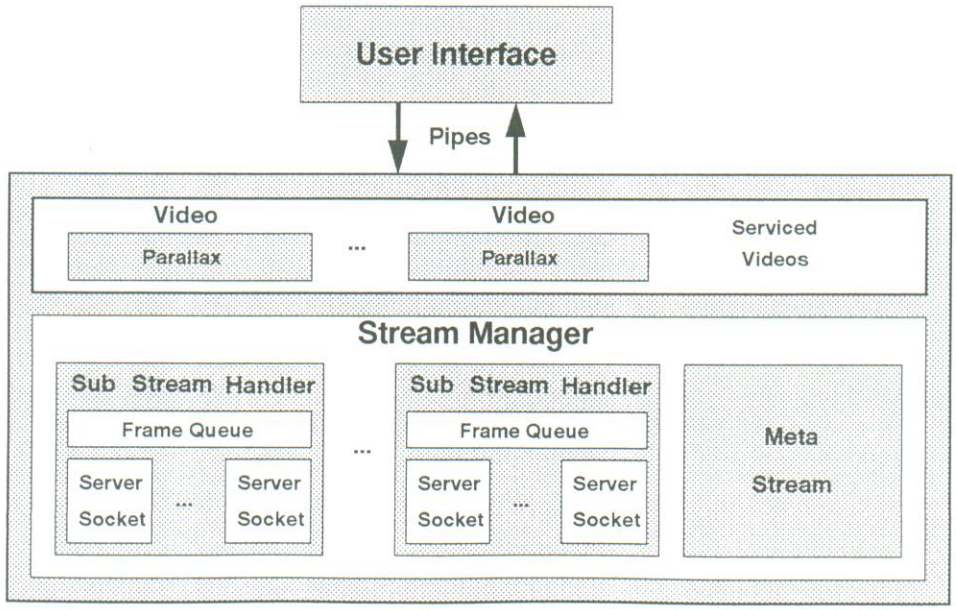
Video transmission protocol (C)

- Data PDUs
- Control PDUs
  - Adjust
  - Start
  - Pause
  - Stop
  - Abort
  - End
  - Fast forward

Server architecture



Client architecture



Summary

- Rapid prototyping pays off
- Prototypical architecture open for future improvements
- 11,000 lines of code, up to 4 servers and numerous clients
- A server's load drops drastically when striping is used

Future work

- Rate control during play-back
- Fast forward and rewind
- Distributing frames on the servers (recording)
- Client that displays and synchronizes multiple CM streams
- Audio streams
- Optimizing disk access
- Optimizing buffer management

Demonstration

- Let's go.

Video Server Architectures: Performance and Scalability

Extended Abstract

Christoph Bernhardt, Ernst Biersack
Institut Eurécom,
2229 Route des Crêtes,
06904 Sophia-Antipolis — France
Phone: +33 93002626
FAX: +33 93002627
email: {bernhard,erbi}@eurecom.fr

1 Introduction

Emerging high-speed networks give rise to a new class of applications in the domain of *multimedia*. *Multimedia applications* commonly include the use of video and audio to represent information. Video On-Demand, Tele-Shopping, Teleconferencing, and multimedia games are typical examples of such applications. With the exception of purely *conversational* applications like Teleconferencing all other multimedia applications require a storage facility for video and audio data. We will subsequently call this facility a *video server*[1].

The special nature of video and audio data makes it impossible to employ a normal file server design for a video server. The design of a video sever has to meet requirements that stem from the *continuous nature* of video and audio. A video (or audio) *stream* consists of several *frames* (or *samples)* that have to be played within tight temporal constraints. A storage server for continuous media must guarantee to deliver data of a continuous media stream in a timely fashion. Today's file servers are not designed to give such guarantees.

Continuous media streams differ in two more ways from "normal" data. Their access-method is mostly sequential and the amount of required storage is — especially in the case of video — huge. The properties of "normal" datafiles are exactly the opposite.

This paper provides an overview over design issues for video server architectures and gives a qualitative view on the performance and scalability of different architectures. Finally, a new architecture is proposed that has superior scalability properties when compared to other architectures.

2 Video Server Architectures

Video servers are used in a wide range of applications. In some applications the video server is integrated into a single user PC, as e.g. in multimedia games based on CD-ROM; in other applications the video server constitutes the center of a huge network. Clients connect to the server to retrieve video and audio, as e.g. in a Video On-Demand application. All these video servers have to meet some common requirements:

---

[1] The name *video server* does not restrict the server to exclusively storing videos. All other media that have similar properties as video will be stored on the same type of server.

- they must guarantee the timely handling of continuous media data;
- they should be reasonably efficient in their utilization of storage capacity and storage bandwidth (this point is especially emphasized by video data, that imposes high demands on storage bandwidth and capacity).

Since a wide range of applications requires different sizes of video servers, a video server architecture should be scalable. Some architectures are inherently not scalable, like e.g. the single-user PC video server or a super-computer based video server. But the deployment of multimedia infrastructures for local networks or the step-wise introduction of Video On-Demand systems make scalability mandatory.

Figure 1 depicts the basic elements of a video server architecture from a system point of view. Architectural elements are the video server itself, the highspeed network, and the clients

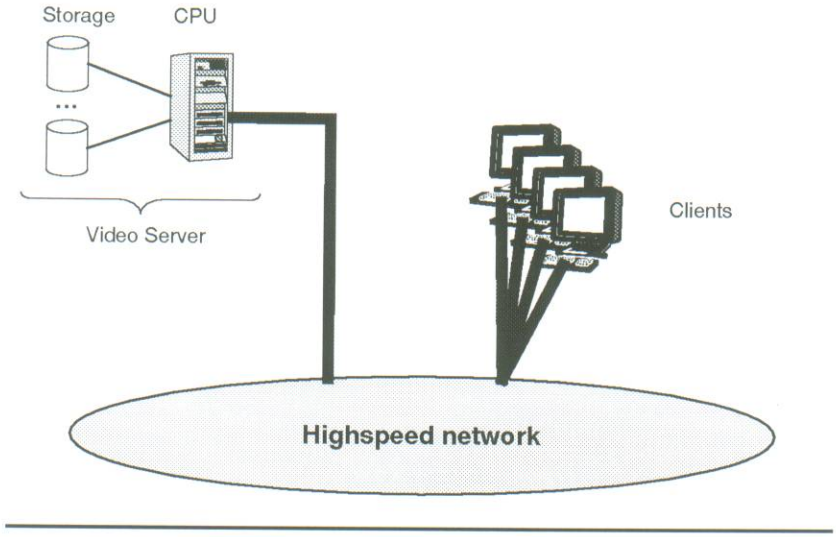


Figure 1: generic video server system architecture

connected to the server. The main technical decisions are how to realize the storage in the video server and how to perform the communication between server and client.

2.1 Storage

The medium is a very important aspect of every storage architecture. There are several possible media for the storage of multimedia information:

- Magnetic tape; tape storage alone is not suitable for the storage of continuous media in a video server. It supports the continuous retrieval of data very well, but the high latency of tape change operations forbids its use in a multiuser environment.
- Optical disks; optical disks have the disadvantage of a relatively low bandwidth and high seek times making it unsuitable as sole storage medium in a multi-user storage server.
- Magnetic disks; magnetic disks are the preferred medium for storage in continuous media servers. They provide a relatively high bandwidth combined with low seektimes and low latency.

- Solid state memory; solid state memory with its high bandwidth and near zero latency has optimal properties for providing storage in a multiuser video server. The only problem is its price.

Some of the above storage media are only viable in a combination with other techniques, e.g. in a storage hierarchy combining tape storage as archival medium with magnetic disks as secondary storage that serves most user requests and solid state memory for the most frequently retrieved data.

This section concentrates on the use of magnetic harddisks as storage medium, since most servers will be mainly based on this medium. To achieve the required high storage capacity and bandwidth (a MPEG I coded video requires 1.5 Mbit/s of bandwidth and approx. 1.2 GByte of storage), a video server combines several disks into a disk array. Each disk in such a disk array generally has enough bandwidth to service multiple users retrieving a video (a typical disk has a sustained data throughput of approx. 24Mbit/s resulting in a maximum of 16 users/disk).

To optimize the utilization of a disk system in a video server some parameters have to be set accordingly:

- *Striping*; the stored data has to be distributed over multiple disks. Striping helps to avoid situations where frequent retrieval of some data leads to *hot-spots* on some disks and under-utilization of others, data of one stream is usually distributed over all or at least a number of the disks of a disk array. This process is called striping.
- *Storage allocation*; to avoid extensive seek movements of the disk heads, disk blocks have to be allocated to video streams in an intelligent way exploiting the continuous property of the stream data.
- *Retrieval method*; the retrieval method determines the way stream data is read from the disks. It is responsible for guaranteeing deadlines to clients and to prevent the *starvation* of clients due to missed deadlines.

There are several proposed schemes in the literature. The references [1][6][9] provide some examples of disk scheduling schemes for video servers. The main problem in selecting a scheduling method is the absence of a unifying framework to compare the performance of different approaches.

2.2 Stream Transfer Server → Client

After retrieving the stream data from the disks, it has to be transferred to the client. There are two different modes of video stream transport between server and client:

- *Real-time*; in real-time mode the server transmits the video information with the natural data rate of the stream. For a video stream this could mean that the server sends one video frame every 1/30th second, for instance. The client would take this data from the network and almost immediately play it or use it otherwise without doing extensive buffering.
- *Fast-load*; as mentioned in the last section the disk array has a much higher bandwidth than required for one video stream and to make efficient use of the disk system the smallest block retrieved from the disk system for one stream is

usually bigger than one frame. The disk utilization can even be improved with increasing blocksize. The resulting large blocks can be sent faster than real-time to the clients where they are buffered until the individual frames of a large block are due for play-out.

From the network point of view the two methods differ in the type of traffic they induce on the network. The real-time method generates a rather smooth traffic, whereas fast-load produces a heavily bursty traffic. To deal with this bursty character Gelman [5] introduces a complicated path reservation protocol. Verbiest [2] relies on the effects of statistical multiplexing. In general the two methods result in different buffer requirements. Using real-time transfer, the buffering is mostly done in the video server itself. For the fast-load case the network and the client have to provide the necessary buffering.

In the context of ATM the work of Golestani [7] and Ferrari [10] shows that, to guarantee a certain QoS, the traffic offered to the network must comply with requirements concerning its smoothness. To be able to accommodate the highly bursty traffic of fast-load, the network has to reserve more bandwidth, thus making the transport more expensive. Therefore, the design of a video server has to take requirements imposed by network constraints into account. Today's video server architectures do not consider communication costs. Even specific cost models developed for Video On-Demand [3] neglect the contribution of communication costs to the overall costs.

3 Scalability

A video server architecture must be scalable with respect to the number of users that use its service and with respect to the amount of stored material. As mentioned earlier one way to achieve scalability in a server architecture is to employ a hierarchy of storage media. Here, tapes and optical disks are used as archival medium whereas magnetic disks and solid state memory are used to store data that is directly accessible to users. In this section we consider a video server architecture consisting solely of magnetic disk storage. This is a viable approach since in many cases a video server will only employ this kind of storage and even if not, the magnetic disk storage will still be an important part of the overall storage architecture.

The next sections investigate the scalability properties of server nodes based on magnetic disks, i.e. a video server in this notion consists of disk arrays connected to a highspeed network. We will show that the way the data is stored on these servers has an important impact on the scalability of the server.

3.1 Autonomous Video Server (Existing Work)

Traditionally, a video server consists of a collection of *autonomous continuous media servers*[2]. Each server is capable of storing **entire** continuous media streams. Disk arrays are used to provide servers with high storage bandwidth. For reasons of load balancing a single stream is striped over multiple disks of a disk array. Since striping is limited to the disks of **one** server, we call it *intra node striping*. To protect a single server against disk failures, redundant data is stored internal to this server so that normal operation can be sustained even in case of a failing disk. Such a disk configuration is known as RAID (see Fig. 2).

---

[2] The number of servers in a collection is variable and may be as small as one.

3.2 The Server Array

In our research architecture, several *server nodes* together build a *server array*. This server array works similar to a disk array. The individual server nodes are implemented using the same hardware as before. As opposed to the autonomous video server, a server node is not storing an entire stream. Instead, a single stream is distributed over several server nodes of the server array. Each server node only stores a *substream* of the original stream. This is analogous to the striped storage on disk arrays. Since this striping occurs over multiple servers, it will be referred to as *inter node striping* (see Fig. 3). The idea of distributed storage of continuous media streams is mentioned in [8] and [4] without providing any detailed description or analysis of its realization.

The striping of a stream over several server nodes makes a server array more vulnerable to failures than an autonomous video server. The failure of only one server node renders substreams stored on the other server nodes useless. To cope with failures, for each stream redundant data are stored on separate server nodes, so that the failure of one or more server nodes can be tolerated. This again is analogous to how RAIDs cope with failures of disks in a disk array. Note however, that it is not necessary any more to have any redundancy internal to a server node. Therefore, it is not necessary to configure the disks of individual server nodes as a RAID.

To make use of a server array, clients must be able to handle multiple substreams. The clients are responsible to split a stream into substreams for storage and to re-combine the substreams during the retrieval of a stream.

3.3 Comparison of Scalability Properties

The major advantage of the proposed new architecture is its superior scalability. We need scalability to adapt the resources of the server to an increase in the demand for service. Critical resources of a video server are the

- capacity of the disk storage in terms of the amount of video streams that can be stored;
- bandwidth of the disks, the CPU, and the system bus that determine the amount of data that can be stored/retrieved concurrently.

Other issues like latency and reliability will not be addressed since the two architectures differ only slightly in these aspects.

There are three scenarios that require scaling of the capacity and/or bandwidth:

- the demand for storage volume grows; necessary, when the number of streams to be stored increases over time;
- the demand for storage bandwidth grows; this might happen if the user community of the server grows or if the existing user community makes heavier use of the storage system;
- the demand for storage bandwidth and volume grows at the same time; an example for this can be easily found in upcoming Video On-Demand trials, where after a first test phase more movies will be offered to a growing number of users.

We compare how the two server schemes adapt to each of the cases.

Growing demand for storage volume

Storage volume can be increased by adding more diskspace to existing servers by either adding new disks or exchanging existing ones with higher capacity disks. There is (almost) no inherent limit to the possible volume increase. Only cost of disk storage will prevent a simple up-scaling. At a certain point, tertiary storage, like tape or optical disk, will be used to provide low-cost, high-volume storage. The two server schemes do not differ with respect to volume scaling. Both can be equally well scaled by adding diskspace or tertiary storage.

Growing demand for storage bandwidth

A growing demand for bandwidth cannot simply be met by adding more disks. New disks provide more raw disk bandwidth, but there are design limits that impose an upper limit on the maximum, concurrently usable disk bandwidth. One of these limiting factors is the system bus of the server; another could be the CPU.

To overcome the limited bandwidth, additional autonomous servers must be added in a traditional video server configuration. The simple duplication of resources creates problems. Since each autonomous node stores entire streams, it must be decided which streams are to be stored on which server. Whenever a new server is added a new distribution across all autonomous servers must be found. Whereby the distribution must be constructed in order to avoid load balancing problems resulting in *hot spot* servers. A hot spot server is a server that stores streams that are more frequently requested than others. It will be higher utilized than other servers storing less popular streams. Eventually, a hot spot server will be overloaded and unable to accept new requests for service while other servers are still available.The only escape from this situation is to duplicate popular streams on more than one server. The problem here is, that precious storage volume is wasted. Another principal problem is that the popularity of streams might not be known in advance. To adapt to the changing demand for individual movies, complex on-line data reorganization is needed.

The server array scales easier in that just another server node is added. The distribution of the substreams must be re-organized once after adding the node to take advantage of the added bandwidth and capacity. During operation of the server load balancing is provided implicitly analogous to disk striping in disk arrays. Each stream is distributed over several server nodes. During storage or retrieval of a stream all server nodes involved are equally utilized. The load balancing is optimal if each stream is distributed over all server nodes. In reality, for ease of maintaining the server array, it might be necessary to store streams only on subsets of server nodes. However, to achieve optimal conditions again, it is still possible to do a complete reorganization off-line. The heuristic for determining subsets of server nodes is still open and for further research.

The following pictures give an example for a situation where two videos are retrieved from a video server. In Figure 2, the traditional video server architecture is used. The two videos happen to be stored on the same server. It is clear that this server and also its net connection are hot spots in the overall server cluster. In Figure 3, our new architecture is used. The two videos are distributed over all nodes of the server array. All servers contribute an equal share to the overall effort of retrieving the videos.

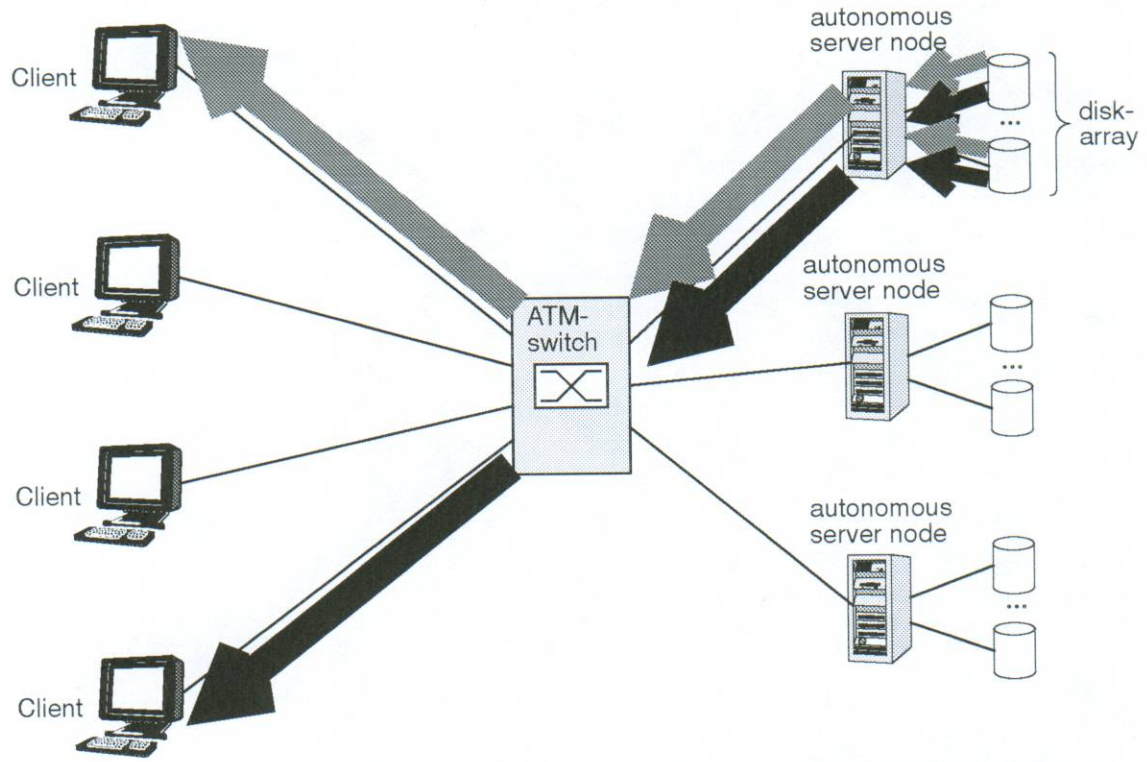


Figure 2: Autonomous servers

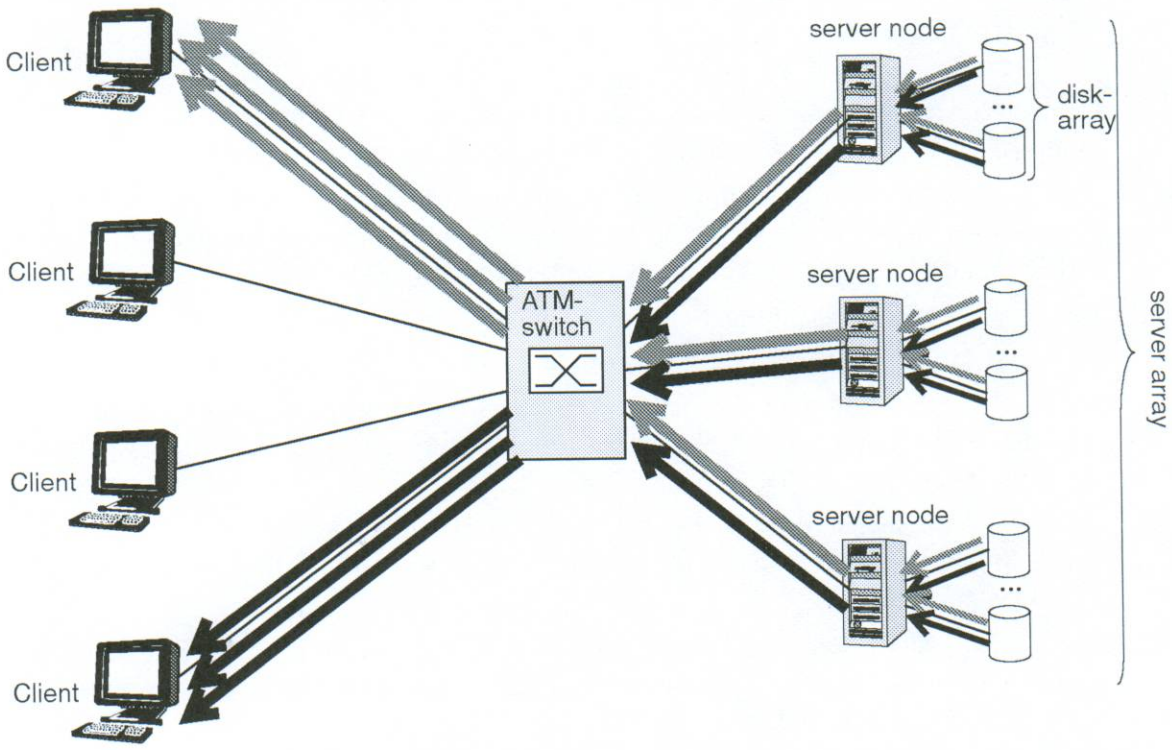


Figure 3: Server array

4 Conclusion

We have shown the basic design parameters of a video server. More work is needed to develop a framework for analyzing the effects of design decisions made. This is especially important for a comparison of different existing video server architectures. We also investigated the network aspects of a video server. This work is closely related to still not settled decisions on how resource reservation will be done in new highspeed networks like ATM. But we believe that a video server architecture must take network issues into account.

We finally introduced a new video server architecture still being under development at our institute. This architecture promises superior scalability properties with respect to storage capacity and bandwidth. It also allows very fine tuning of the network traffic that is generated by a video server. Further work will elaborate on these issues. To gain some hands-on experience with the architecture, a prototype is currently being built. This prototype consists of a number of SUN workstations connected by an ATM switch.

5 References

[1] M.-S. Chen, D. Kandlur, P. Yu, "Optimization of the Grouped Sweeping Scheduling (GSS) with Heterogeneous Multimedia Streams," Proceedings of the 1st ACM International Conference on Multimedia, August 1993

[2] D. Deloddere, W. Verbiest, H. Verhille, "Interactive Video On Demand," IEEE Communications Magazine, May 1994, pp. 82-88

[3] Y. Doganata, A. Tantawi, "A Cost/Performance Study of Video Servers with Hierarchical Storage," IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598

[4] C. Federighi, L. Rowe, "A Distributed Hierarchical Storage Manager for a Video-on-Demand System," Storage and Retrieval for Image and Video Databases II, IS&T/SPIE Symposium on Elec. Imaging Sci. & Tech., San Jose, February 1994

[5] A. Gelman, H. Kobrinski, L. Smoot, S. Weinstein, M. Fortier, D. Lemay, "A Store-And-Forward Architecture for Video-On-Demand Service," Canadian Journal of Electrical and Computer Engineering, Vol. 18, No. 1, 1993, pp. 37-40

[6] J. Gemmell, J. Han, "Multimedia Network File Servers: Multichannel Delay-Sensitive Data Retrieval," Multimedia Systems, Vol. 1, No. 6, April 1994, pp. 240-252

[7] S. J. Golestani, "A stop-and-go queuing framework for congestion management," Proceedings ACM Sigcomm '90: Communication Architectures and Protocols, September. 1990, pp. 8-18

[8] P. Lougher, D. Shepherd, D. Pegler, "The Impact of Digital Audio and Video on High-Speed Storage," Proceedings of the 13th IEEE Symposium on Mass Storage Systems, June 1994, pp. 84-89

[9] Y.-J. Oyang, M.-H. Lee, C.-H. Wen, "A Video Storage System for On-Demand Playback," TR NTUCSIE 94-02, Department of Computer Science and Information Engineering, National Taiwan University, May 1994

[10] H. Zhang, D. Ferrari, "Rate-Controlled Static-Priority Queuing," TR-92-003, Computer Science Division University of California at Berkeley, February 1992