Simba: A Distributed Video Fileserver

Satyam Vaghani and Pranav Kantawala
svaghani@cs.stanford.edu, pranavk@cs.stanford.edu
Computer Science Department
Stanford University

Abstract

This report describes the design and implementation of Simba, a scalable, fault-tolerant and low-cost video delivery system constructed from a collection of lightweight servers running on a network of workstations (NOW). Video files are striped into very small fragments on the network attached storage or the local file system of each workstation. The fragments are streamed across to the Simba client, which seamlessly integrates all these fragments in real-time to reproduce the entire movie without incurring high buffering costs. The system does not store persistent state or scheduling information at any delivery servers. Consequently, Simba scales with the addition of delivery servers without compromising consistency or fault-tolerance. The ideas presented in this report can also be applied for designing peer-to-peer streaming media delivery systems.

1. Introduction

Real-time video delivery systems have evolved over the past few years to stream video to large audiences with disparate connection speeds. The delivery servers used for this purpose usually require large processing power, main memory and secondary storage. Real-time video delivery over IP networks also requires a high aggregate bandwidth at the server side to support a large number of concurrent users. We have designed and implemented Simba, a distributed video server that amortizes the cost of real-time video delivery by distributing the workload and storage requirements over a network of workstations.

Video files that are served over Simba are assigned universally unique identifiers (UUID) and striped into a number of small fragments of the order of 1-4 MB. These fragments are stored on network-attached storage, or locally on the workstations. Computers that intend to participate as Simba video delivery servers listen for commands from a central server at a unique multicast address. The delivery servers are known as workers and constitute a dynamic set that changes with time. We shall refer to the central server as the master. The master occasionally sends a multicast message directing the workers to send load information. A scheduler on the master chooses a good worker to handle an incoming request for a video fragment from the client and hands off the request to the worker. Simba supports a single master and an unlimited number of workers.

The use of networks of workstations as media delivery systems has been a topic of active research and many innovative solutions have been suggested in the past. The Microsoft Tiger Video Fileserver [1] uses video striping, ATM switching fabric and DMA transfers from disk to network interfaces to achieve a significant speedup in video delivery. Servers are organized into a ring and the delivery schedule is passed from one server to the next. The Tiger architecture uses specialized hardware and customized servers to achieve scalability and fault tolerance. The Zeno distributed video fileserver [2] proposes the use of networks of workstations where each workstation acts both as a client as well as a server. The use of workstations to serve potentially long clips of video can

increase the probability of errors or overloads during the service.

The motivation behind the Simba architecture comes from the fact that networked workstations have enough *residual* bandwidth, computing power and storage capacity to store and deliver small fragments of video files reliably. A highly available video delivery system can be constructed by distributing the workload over these localized NOWs. To illustrate the point, we do a back-of-the-envelope calculation for the Stanford Campus Network, which consists of approximately 45000 hosts spanning over 350 active subnets. A conservative estimate of 50MB of residual storage space on each machine would yield an aggregate storage capacity for approximately 2250 full-length movies. Similar arguments apply to processing power (in MIPS) and network bandwidth (in Mbps). We shall present a detailed discussion of scalability later in this report.

The report is organized into six sections. The next section describes the functional components, protocols, scheduling mechanisms and file formats used in Simba. Section 3 covers the implementation details. We have documented the performance gains achieved through Simba in section 4. We propose some useful enhancements to the current implementation in section 5 and conclude in section 6. We have restricted finer details about the software usage and the Simba API in Appendix A and B.

2. The Simba Architecture

This section describes the three-tier architecture of the Simba video delivery and viewing system followed by a summary of the Simba file formats. We also discuss the communication protocols used and the design goals that influenced the choice of the communication model. We identified three distinct entities in the Simba video server setup: the Simba client, master and the worker. The relationship between them is illustrated in Figure 1.

2.1 The Client

This is a generic MPEG-1 and MPEG-2 program stream decoder and viewer. The client also contains an Apple QuickTime plug-in to view QuickTime videos. The Simba client differs from normal video clients in that it is aware of the fact that videos are composed of multiple fragments. Hence it can request a video on a fragment-by-fragment basis and play the fragments in order without any intermediate gaps. The client requests fragments from the central server using a control connection. The client request contains its address and a port number at which it constantly listens for incoming media transfer connections.

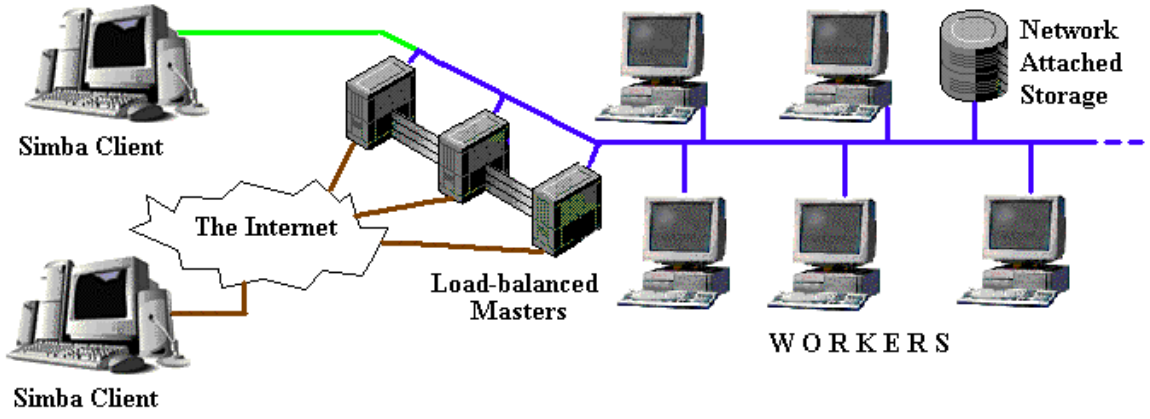

Figure 1. The Simba Architecture

2.2 The Master

This is the main entry point and the central coordination authority in the Simba server system. All clients direct their request to watch a video to the master. The clients do not need to know about workers or their addresses. The master forwards client requests to lightly loaded workers. The forwarding algorithm uses worker load information, refreshed at regular intervals, to make forwarding decisions.

The master is critical to the proper functioning of the Simba video server. We can ensure a high availability of the master by hosting the master on tandem non-stop machines or other fault tolerant assemblies. The scheduling algorithms on the master are kept as simple as possible to minimize computation and prevent it from being a bottleneck in the overall performance of the system.

2.3 The Worker

This component receives a forwarded video fragment request from the master and establishes connection to the client to serve the data. The workers can be thought of as file transfer agents that are activated on a command from the master. Workers are designed to be lightweight processes since they run on commodity workstations. Workers can join and leave the Simba server assembly without producing a noticeable change in the Quality of Service as perceived by the user.

2.4 Simba Video Files

Simba was designed to support existing video coding standards. Consequently, we do not impose restrictions on the encoding format of the video files (provided a suitable plug-in is supplied), the number of fragments a video should be striped into or the length of each fragment. The system assigns a Universally Unique Identifier (UUID) to each video for identification purposes. This scheme makes the resolution of video names unambiguous. Fragments of the same video are identified by a three character long fragment number ranging from 001 to 999. The filename format for a fragment is:

<UUID>.<fragment number>

We currently use a conventional file splitter to split video files into multiple fragments of fixed length. Videos can also be split into fragments of equal running length by using a MPEG file editor. We have verified this approach using a software package called MyFlix. However, the former approach was faster and can be automated easily. MPEG file editing also stands the risk of violation of copyrights since the frame sequence is modified at fragment boundaries.

In our prototype, we have used network storage to export a uniform view of the video repository to all the workers. All video fragments reside on the AFS. This is a convenient arrangement since any worker can serve arbitrary movies and fragments. The master scheduler becomes simpler in this scenario.

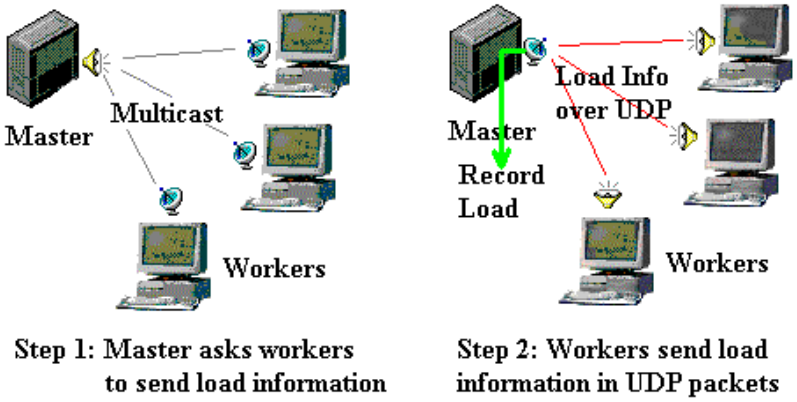

Figure 2. The master gathers load information from the workers periodically

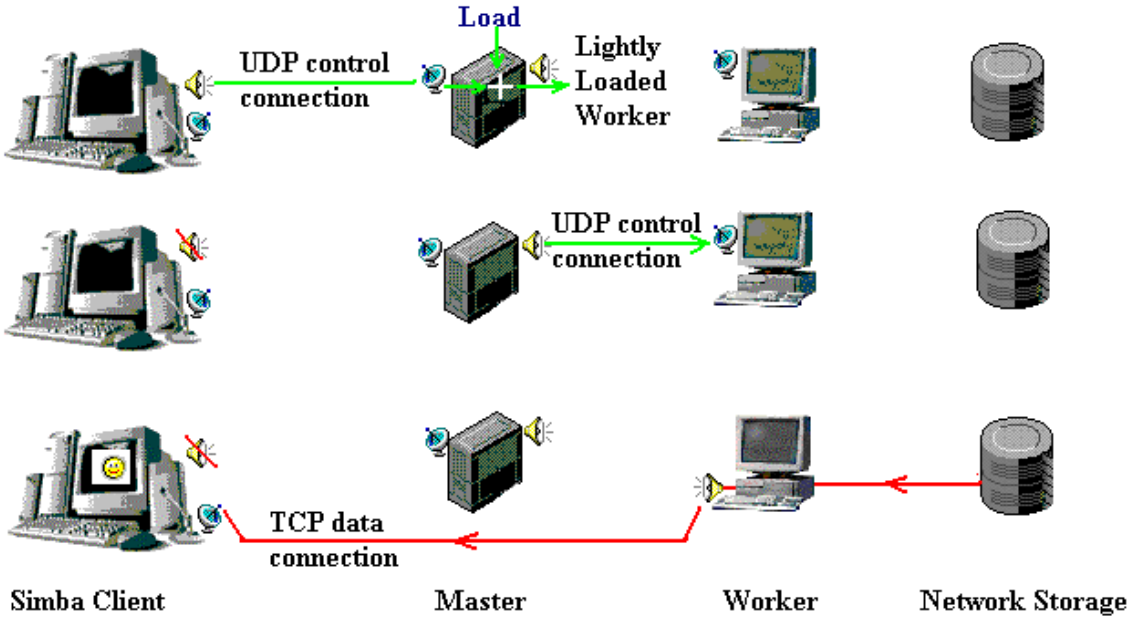

Figure 3. The three steps involved in a video fragment transfer

Step 1. The Simba client sends a UDP control packet to a master. The packet contains client IP address, port number of the TCP listener, video UUID and the requested fragment number. The server looks up the worker load information list to select a lightly loaded worker. Step 2. The server forwards the client request to the worker. Step 3. The worker establishes a TCP connection with the client and the fragment transfer goes through.

2.5 Communication Protocols

We use UDP packets for all control connections and TCP packets for video data connections. The control primitives have been kept simple and idempotent so that the inherent unreliability of the UDP protocol does not have an adverse effect on the functioning of the system. UDP allows the clients, masters and workers to exchange messages without incurring a connection setup and termination overhead. The master and workers are located on the same subnet hence the probability of UDP packet loss is very low. The TCP-based data connection, on the other hand, guarantees reliable delivery of video data. The connection setup and termination overhead is negligible relative to the time it takes to transfer a single fragment. The Simba master uses multicast to send commands to all the active workers in the system.

A one-way control connection, from the client to the master, allows the client to send video fragment requests to the server. The reverse path from the master to the client is not implemented for a number of reasons: after sending the fragment request, the client will have to listen to incoming data transfer connections anyway. Blocking on an acknowledgement is a redundant step since and acknowledgement and an incoming data connection both imply that the client request has been scheduled for service. The acknowledgement originates from the master while the incoming TCP data connection originates from the worker. Hence, the latter is a more authoritative indication of successful response to the worker request. The relaxation of the acknowledgement requirement reduces the request processing time on the master. Finally, the acknowledgement would use an unreliable protocol while the data connection uses a reliable substrate.

The master uses UDP to forward client requests to workers. The communication channel is assumed to be more reliable relative to the client-master interaction because the master usually operates on the same subnet. The master sends out a multicast message to all the workers to ask for load information. The use of multicast allows the Simba server assembly to scale along two axes. Workers can join and leave multicast groups at will. A newly instantiated worker can subscribe to the multicast address to receive commands from the master. A worker who left the multicast group will not send load information to the server in the next cycle and hence will automatically lose contact with the master. Secondly, the master does not need to address the workers individually to receive the load information. Hence, the number of active workers in the system does not affect the performance of the master. The choice of multicast currently requires the existence of a master in every subnet that contains active workers. This restriction can be relaxed in the near future with the deployment of routers and switches transparent to multicast [4], [5], [6], [7].

Simba supports a one-way data connection from the worker to the client. The data connection is similar to the active connection setup in FTP [8]. This achieves the desired level of transparency in Simba; the client is unaware of the existence of workers in the system.

2.6 Scheduling

The master schedules the allocation of client requests among workers, so that over a large period of time, the load is distributed evenly across all workers. It periodically gathers load information from workers. Based on this information, it decides which worker should serve the next client request.

When a worker process is initiated, it subscribes to a pre-defined multicast address. On this address it listens for load information requests from the master. It maintains a count of the number of clients it is serving at any point of time. When the master asks for load information, it sends this information to the master.

We implemented a round-robin scheduler for Simba. The master maintains a linked-list of workers along with their load-information. It starts off with an empty list. It then sends a multicast message to all workers to provide load information. It builds the linked-list based on replies from workers. The new client requests are then allocated to workers in a round-robin fashion. After a certain pre-defined interval, the master sends out the next multicast message. The linked-list is now re-built for this interval. We observed that if the time period of the multicast messages is approximately equal to the average fragment run-time, the server does not accumulate stale load information. This also prevents lightly loaded workers from being bombarded with a lot of forwarded client requests. This would also cause rapid fluctuations in loads experienced by the worker since all the requests would be temporally crowded over a small time-period.

3. Implementation

This section describes the Simba code components and APIs. We shall not provide source code listing in this report because of the prohibitive size of the source code repository. However, the source code is available for download and we can provide the details. While this section describes the structure of the code, the appendix details the main functions and APIs.

3.1 Programming Notes

The Simba code is organized into directories and subdirectories according to the functionality.

- All the system-wide header files reside in `include/`
- Client sources are in `src/`. In particular, the video viewer interface source resides in

`src/interface/` while the MPEG stream handlers reside in `src/input/`

- The server and worker sources are in `server_side/`
- The audio and video plugins are in `plugins/`.
- Many code modules are compiled and archived in the directory `libs/`

The source code is written in Ansi C. Functions contain the name of their container module. The Makefile in the source root can be used to compile all the targets. The targets for the Simba client, worker and server are `vlc`, `worker` and `server` respectively.

We have developed and tested the code on the Solaris/Sparc platform. The code can be ported to Solaris/Intel and Linux/Intel platforms. The user may have to run the configuration script, `configure`, and edit the options in `Makefile.opts` to resolve minor platform or file system related differences.

3.2 Shared Libraries

The client and the worker code modules share the thread management and formatted output libraries.

We decided to use POSIX threads as opposed to process creation to make the Simba components faster, scalable and simplify synchronization mechanisms. The code for the master runs in a single thread. The thread management library contains portable functions for thread creation, destruction and synchronization. These functions are essentially wrappers to the POSIX thread API and they take care of minor variation in implementation across platforms.

We cannot use printf() in the threads because it is known to cause indeterministic behavior in threads. The formatted output libraries provide thread-safe functions to produce formatted output in case of a normal run, an error condition, warning or debugging. The library maintains a queue of output messages at run-time and prints them out in FIFO order. Alternately, the library functions can synchronize the output using a print locks.

3.3 Client

We used the VideoLan client [3] developed by a group of students at École Centrale, Paris. The client is a standalone MPEG-1 and MPEG-2 player designed to view movies broadcast over the campus LAN from a VideoLan server. The client source code is available for download and modification under the GNU Public License.

The VideoLan client cannot handle MPEG program streams directly from the network. We have developed a new module that downloads and buffers the program stream from the Simba worker into the program stream decoder. In addition, we added the necessary logic to handle videos as fragments.

The client code uses modules and plug-ins. A module is a group of compiled-in C source files that are linked against the main application at build time. This feature enables developers to choose between distributing the source code of new developments and distributing only the object files. The client contains the following modules:

- Interface. This module is the entry point to the client program. It manages all user interactions and thread spawning.
- Network. It is responsible for sending fragment requests to the master and downloading and buffering the video data from the workers. This thread implements all the communication protocols required to interact with the Simba server system.

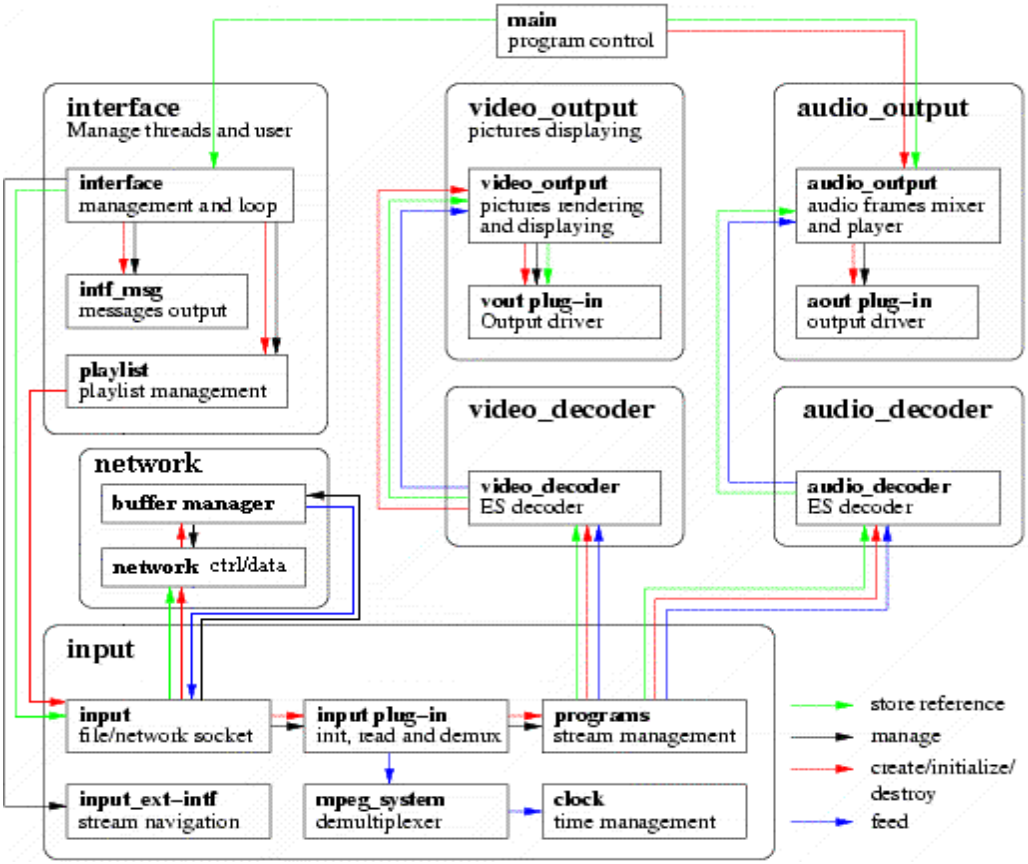

Figure 4. Relationship between the different modules in the Simba client

- Input. It opens and input descriptor, reads packets, parses them and passes reconstituted elementary streams to the decoder(s). We pass a descriptor the
- Video output. It initializes the video display, gets all pictures from the decoder(s), optionally converts them to RGB format (from YUV), and displays them.
- Audio output. It initializes the audio mixer, i.e., finds the right playing frequency, and then re-samples audio frames received from the decoder(s). This module is not functional for the Solaris build of the Simba client.
- Misc. Miscellaneous utilities used in other modules. This is the only module that will never launch a thread.

The client is known to suppress audio output when run on Solaris. However, a fix for this problem is due in the near future. The client receives the encoded audio from the workers; hence it is suitable for the purpose of our project.

The Simba client is heavily multi-threaded and the various modules either use shared data structures or semaphores for synchronization. The relationship between the different modules is illustrated in figure 4. Documentation about the interface module, video, and audio modules is available on the VideoLan website. We shall only describe the changes to the input module and the newly added network module.

The Network Module

The network module operates in an independent thread in conjunction with the input module. It hides the actual source of the

video data and the protocols involved in retrieving the video data from the input. It presents the video data stream as a seekable file descriptor to the input module. This abstraction allows the input module to work on the Simba network as well as local video files.

The network thread is instantiated by a call to `network_CreateThread`. The thread manages three sockets at a time: the UDP control socket to the master, the TCP listener for incoming data connections and (possibly) the actual video data connection from a worker. The network thread initialization routine creates the UDP control socket and the TCP listener.

The thread sends a fragment request to the server and waits for an incoming data connection on the TCP listener. On receiving a connect, the thread starts downloading the video data. Once it downloads enough data to ensure that the buffer does not underflow when the input module starts processing the data, it signals the input thread to start reading. For example, the low-water-mark in our implementation is two fragment lengths. In case a fragment download fails, the thread removes the downloaded portion from the local buffer and resends the fragment request to the server. This makes the Simba client immune to worker failure. The network thread also ensures that the downloaded data does not exceed the upper limit of the buffer capacity. If this upper threshold is reached, the thread postpones sending the next fragment request to the master until the buffer occupancy falls significantly below the high water mark. This fragment download is repeated until all the fragments of the video are downloaded. The fragments are downloaded in order of increasing fragment number.

The network thread does not accept concurrent data connections from multiple workers because this would increase the algorithmic complexity for no significant performance gain. At best, the multiple connections would cause the buffer occupancy to hover dangerously close to the high water mark at all times. The multiple download threads would then have to sleep, thus decreasing throughput.

The Input Module

The idea behind the input module is to read in video data packets with the minimum knowledge of their contents. The reader takes a packet, reads its ID, and delivers it to the decoder at the right time indicated in the packet header (SCR and PCR fields in MPEG). All the basic browsing operations, i.e. seeks into the input stream are implemented without looking at the content of the elementary stream.

An input thread is spawned at the beginning of the program by calling the `input_CreateThread` function. While the initial VideoLan implementation re-spawned the thread in case of a change of media, we retain the thread since the media specifics for all the fragments of a single movie are the same. This reduces the thread state initialization overhead and the inter-fragment processing delay.

Once the input thread is active, it looks for a suitable plug-in to send the data for decoding. The plug-in exports a function, which when applied to the input stream, returns the relevance (weight) of the module for decoding the stream. In the case of MPEG program stream files which are used in the Simba setup, the MPEG program stream handler announces itself as the most relevant plug-in.

The module calls the initialization routine for the plug-in and waits for the network module (running as another thread) to signal the availability of data. As soon as the network module exceeds a ‘safe buffer occupancy’ threshold, the input thread reads in packets and calls the program stream demultiplexer in a loop till it encounters the end of the stream. The plug-in is responsible for initializing the stream structures, managing packet buffers, reading packets and demultiplexing them. Once the input thread

encounters the end of a fragment, it waits for the network thread to signal the availability of the next fragment. Under normal operation, the network thread leads the input thread by at least a single fragment. Hence the input thread should not experience a delay between fragments.

3.4 Master

The master sits in the middle tier of Simba architecture. It talks to clients at one end and to workers at the other end. There is only one master in the system. It contains following functional modules:

Multicast Agent

This module is responsible for collecting load information from workers. At predefined regular intervals of time, it sends out a multicast message to the workers requesting load information. It creates a linked-list of worker responses that is then used by the scheduler for worker selection.

Scheduler

This module is responsible for deciding which worker should server the next client request. It has been currently implemented as a round-robin scheduler. However, since the code is modular, any other scheduling algorithm can easily replace the current implementation.

Request Forwarder

This module listens for client requests. Upon receiving a valid request, it asks the scheduler to provide information about the worker that should serve the client request. It then forwards the request to that worker.

3.5 Worker

The worker provides load information to the master and serves video fragments to the client. There can me multiple worker instances in the system. The worker contains the following functional modules:

Load Dispatcher

This module listens for worker load requests from master. Upon receiving such a request, it sends the count of clients it is serving at that moment. It also receives requests from the master to server a particular client. The requests are forwarded to Fragment server module that does the actual fragment transfer.

Fragment Server

This module is responsible for reliably transmission of video fragments to the client. When it receives client information from the master, it creates a new thread and opens a TCP connection to the client. After connecting to the client, it transfers the fragment to it. On completion of fragment transfer the thread dies.

4. Performance

The load on a video delivery server varies with the popularity of video content hosted on it. Simba alleviates the load presented to a single machine by fragmenting the video into very small clips. Different workers can serve the different fragments of a popular video and hence reduce the probability of overloading a single worker. Fragmentation also removes the port restriction; that is, more than 65000 users can concurrently watch different parts of a single video from different machines.

The Simba architecture reduces state maintenance at the worker nodes so that the inherent unreliability of these machines does not become a performance bottleneck. Simba scales as new workers are added to the system and is immune to failure of existing ones.

We performed experiments to test the key design parameters of Simba: scalability, fault-tolerance, speedup in serving video data and uniform load distribution. All the machines used in the experiment were SunUltra60Creator3D workstations with 256MB RAM and 1GB swap space. These machines are a part of the Stanford comput-

ting cluster, which is open to all Stanford graduate and under-graduate students. A 491MB video was striped into 161 fragments. The total running time of this VCD quality video was 48 minutes. The fragments were stored in a network mount. We could run upto sixteen clients that were trying to view the same video. We used four workers to handle this load. The fact that we were using a public computing infrastructure prevented us from expanding our client set.

4.1 Scalability

The fundamental limit of scalability in the Simba system is the amount of worker state information that the master maintains. The master needs to maintain 12 bytes of information for each worker in the system. Thus, the number of workers that can be supported by the system depends on the amount memory available on hardware that runs the master. The master does not suffer from performance degradation as the number of workers or clients increase since it uses a round-robin scheduling algorithm, and the time taken to select the worker is always *O(1)*. On the other hand, workers use the TCP protocol to reliably transfer video files to clients. So the number of clients that a worker can serve with high fidelity will be limited by the disk-access speed and number of TCP connections. The system provides the flexibility to add new workers to the system at any time as load on the system increases. Thus, if the master runs on hardware with sufficient memory, and if there are sufficient workers to serve client requests, the system is extremely scalable. Besides, the memory footprint of master or workers does not increase over a period of time. This makes it possible to deterministically gauge the hardware requirements of the system.

4.2 Fault-tolerance

A major goal of the Simba system was to make the system resilient to failure of any single entity in the system. The aim was to avoid any prolonged degradation in the system due to the failure at a single point. The master and workers are primary points of failure in the system. We assume a level of file system abstraction in that the file storage system is resilient to disk failures, and that there is sufficient replication of video fragments in the file system.

While serving client requests, if a worker process fails, the clients will request the same video fragment again. The master will now allocate the request to a new worker based on its scheduling algorithm. This may result in the client receiving a part of video a second time, but since video fragments tend to be of 3-minute durations, this network overhead may be acceptable at times of failure. As far rendering the video is concerned, this glitch is transparent to the client.

Since the master maintains load information of workers and decides the allocation of client requests to workers, it may be considered a single point of failure in the system. However, the master maintains the state only during the interval between requests for load information from workers. So if the master fails at any time, a new master process can be started to take its place. This new master can initiate the process of gathering load information by sending out a multicast message. The system would fail to cater to client requests during the time in which master is not functioning. A mechanism can be devised to restart the master in case of failure, which results in a system that is completely transparent to its failure.

5. Future Work

Considering the time duration for this project, we designed and implemented the basic distributed video file server and modified an MPEG client to work with our servers. But for the time constraints, there are many interesting issues that we would have liked to look into. We discuss some of them below.

Experiments with Scheduling Algorithms

Our implementation of master uses a round-robin scheduling algorithm to distribute client requests among workers. We can ex-

periment with different scheduling algorithms to study the impact on load distribution among workers and overall performance of the system. For instance, it may be obvious that the most lightly loaded worker should server the next client request. So, the master can maintain a linked-list of workers sorted by the worker load and allocate client requests based on this information. However, the scalability issues should be kept in mind while selecting a scheduling algorithm. For instance, if the time to select the next worker to server a client request is *O(n)*, the performance may degrade if there are many workers in the system.

Peer-to-Peer (P2P) Architecture

We can extend the current three-tier architecture to a fully scalable P2P architecture. This would provide more flexibility to end-users since the storage and distribution of files would no longer be in control of a single entity. MPEG video files tend to be large, they could be of the order of 1 MB per minute of video content. So striping the video content across several peers reduces the load on a single peer, both in terms of storage capacity and network bandwidth. However, such a system would demand more advanced scheduling and data management techniques.

TCP for Control Messages

The current implementation uses UDP packets to exchange control messages among the clients, the master and workers. We use TCP only for reliable transmission of video file fragments. Since all our processes were running on the same local area network, we have assumed reliable delivery of UDP packets. However, in most environments, the network may not be 100% reliable and may drop some client requests packets on the way. To ensure complete reliability of information exchange, we can use TCP for all purposes.

6. Conclusions

Simba utilizes the residual bandwidth, computing power and storage capacity of networks of workstations to construct highly available video delivery server aggregates. The use of existing computing infrastructure reduces the cost of deployment and maintenance of video delivery systems.

Simba fragments videos to store it across multiple workstations and also distributes the load of serving very popular videos across several workstations. The uniform distribution of workload over all the active servers enables Simba to support more concurrent viewers per video.

We found that our implementation scales well with the addition of workers and clients in the system. The Simba client is resilient to worker failure as long as there is at least a single active worker in the system. The master server should be protected from faults by employing load-balancing and mirroring techniques.

Acknowledgements

We would like to thank Ciro Noronha for guiding us through the implementation of the system. We would also like to thank Manish Godara and David Hole for helping us with the project logistics.

References

[1] W. Bolosky, J Barrera III, R. Draves, R. Fitzgerald, G. Gibson, M. Jones, S. Levi, N. Myhrvold, R. Rashid. The Tiger Video Fileserver. In the proceedings of the Sixth International Workshop on Network and Operating System Support for Digital Audio and Video. IEEE Computer Society, Zushi, Japan, April 1996.

[2] The Zeno distributed video fileserver, http://www.cs.cornell.edu/Info/Faculty/Brian_Smith.html.

[3] The VideoLan project, MPEG and DVD for every OS, http://www.videolan.org.

[4] Cisco, Pragmatic General Muticast (PGM). http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120t/120t5/pgmscale.pdf

[5] Floyd, S., Jacobson, V., Liu, C., McCanne, S., and Zhang, L., A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing, IEEE/ACM Transactions on Networking, December 1997, Volume 5, Number 6, pp. 784-803.

[6] Talarian, Smart PGM. http://www.talarian.com/products/smartpgm

[7] RFC 1112. Host Extensions for IP Multi-casting. http://www.ietf.org/rfc/rfc1112.txt

[8] RFC 959. File Transfer Protocol. Section 3.2 Establishing Data Connections. http://www.ietf.org/rfc/rfc0959.txt

[9] R. Stevens . Unix Network Programming. Vol. 1, Prentice Hall, N.J.

Appendix A. A Note on Usage

The Simba client is designed to run in an X11, Gnome or GTK environment. We explain the most common option to invoke the Simba client:

```
fable19:~/Simba-1.3.0524> vlc [-w <width>] [-d <height>] [-n
<streamtype>] –f totalfragments –m masterIP videoUUID
```

`-w`	set the width of the video viewer window. The default is 250 pixels
`-d`	set the height of the video viewer window. The default is 250 pixels
`-n`	specify the stream type. Simba supports only MPEG program streams, ps.
`-f`	total number of fragments the current video is divided into
`-m`	the IP address of one of the masters in the Simba system
`videoUUID`	the identifier of the requested video

The master may be invoked using the following command:

```
Fable20:~/Simba-1.3.0524> master
```

The worker may be invoked using the following command:

```
Fable21:~/Simba-1.3.0524/mpegs> worker
```

The worker should be invoked from the directory where video fragments are stored. The client, master and worker should be executed on different machines. Besides, the worker and server should be executed on machines within the same subnet.

Appendix B. The Simba API

The Simba client, master and worker source code is heavily commented. In this section, we present the interfaces exported by the main modules in the Simba system. This listing is not exhaustive and does not document the functions internal to each module.

Shared Libraries

Environment Variable Management - these methods are used to get default values for some threads and modules.

```
void   main_PutIntVariable( char *psz_name, int i_value )
void   main_PutPszVariable( char *psz_name, char *psz_value )
```

Store a name, attribute pair as an environment variable so that it can be accessed from any thread within the client. `main_PutIntVariable` converts an integer into a string before storing it.

```
int    main_GetIntVariable( char *psz_name, int i_default )
char * main_GetPszVariable( char *psz_name, char *psz_default )
```

Read back the values of environment variables. These functions return a default value of an environment variable specified in `psz_name` does not exist or is empty.

Formatted output functions: These provide a thread-safe means to output normal, warning and debug messages onto the console

```
p_intf_msg_t intf_MsgCreate ( void )
void intf_MsgDestroy ( void )
```

Create and destroy the queue of messages. In case the queue does not exist, the library uses print locks to synchronize console output between different threads.

```
void intf_Msg ( char *psz_format, ... )
void intf_ErrMsg ( char *psz_format, ... )
void intf_WarnMsg ( int i_level, char *psz_format, ... )
void intf_IntfMsg ( char *psz_format, ... )
```

intf_Msg and intf_IntfMsg are used to print out messages to the consule during the normal functioning of the program. Intf_WarnMsg is used as an early warning system to a non-fatal error. Intf_ErMsg is used to print out an error, typically following the failure of a function call.

Thread management functions. These functions are wrappers to the POSIX thread API functions and are portable across platforms. Their behavior is analogous to the pthread library functions.

```
static __inline__ int  vlc_thread_create( vlc_thread_t *p_thread,
char *psz_name vlc_thread_func_t func, void *p_data )
static __inline__ void vlc_thread_exit  ( void )
static __inline__ void vlc_thread_join  ( vlc_thread_t thread )

static __inline__ int  vlc_mutex_init    ( vlc_mutex_t *p_mutex )
static __inline__ int  vlc_mutex_lock    ( vlc_mutex_t *p_mutex )
static __inline__ int  vlc_mutex_unlock  ( vlc_mutex_t *p_mutex )
static __inline__ int  vlc_mutex_destroy ( vlc_mutex_t *p_mutex )

static __inline__ int  vlc_cond_init    ( vlc_cond_t *p_condvar )
```

```
static __inline__ int  vlc_cond_signal  ( vlc_cond_t *p_condvar )
static __inline__ int  vlc_cond_wait    ( vlc_cond_t *p_condvar,
vlc_mutex_t *p_
mutex )
static __inline__ int vlc_cond_destroy  ( vlc_cond_t *p_condvar )
```

Simba Client

The Interface Module

Interface module maintains the most complicated data structures in the entire system. The module can access data from any thread audio, video, input or network thread once they are instantiated.

```
intf_thread_t * intf_Create ( void )
void intf_Destroy ( intf_thread_t * p_intf )
```

Create and destrop the interface. The interface, in turn creates the GUI, and loads the input module, the network module, audio module and the video module into separate threads. The `intf_thread_t *` points to a structure that contains the complete state of the Simba client. Many state variables are uninitialized at thread creation time since they are managed by other threads.

```
void intf_AssignKey( intf_thread_t *p_intf, int r_key, int f_key,
int param)
keyparm intf_GetKey( intf_thread_t *p_intf, int r_key)
void intf_AssignNormalKeys( intf_thread_t *p_intf)
int intf_ProcessKey ( intf_thread_t * p_intf, int i_key )
```

Keyboard input management functions for the video viewer GUI. The user can control the viewer through keypresses instead of using a mouse.

The Input Module

```
input_thread_t *input_CreateThread ( playlist_item_t *p_item, int
*pi_status )
```

This function creates a new input, and returns a pointer to its description. On error, it returns NULL. If pi_status is NULL, then the function will block until the thread is ready. If not, it will be updated using one of the THREAD_* constants.

```
void input_DestroyThread(input_thread_t *p_input, int *pi_status)
```

This function should not return until the thread is effectively cancelled.

```
static void RunThread( input_thread_t *p_input )
```

Thread in charge of processing the network packets and demultiplexing.

```
void input_FileOpen( input_thread_t * p_input )
```

This function uses the file descriptor abstraction provided by the network thread and opens the file descriptor for reading. The state of this open video data stream is stored in the main input thread structure. This function contains the logic to maintain pace with the network thread so prevent buffer overflow and underflow.

```
void input_FileClose( input_thread_t * p_input )
```

Close the open video data stream and signal the event to the network thread.

The Network Module

`network_thread_t *network_CreateThread ( playlist_item_t *p_item, int *pi_status )`
This function creates a new network thread, and returns a pointer to its description. On error, it returns NULL. If pi_status is NULL, then the function will block until the thread is ready. If not, it will be updated using one of the THREAD_* constants. The network thread can be created only by the input thread.

`void network_DestroyThread( network_thread_t *p_network, int *pi_status )`
Mark a network thread as zombie. This function should not return until the thread is effectively cancelled.

`static void RunThread( network_thread_t *p_network )`
Thread in charge of processing the network packets and buffering. This function is the main body of the network thread. RunThread is also in charge of working with the input thread to prevent buffer overflow and underflow.

`void network_Open( network_thread_t * p_network )`
Initializes all the control and data sockets to talk to the master and accept incoming data connections from any worker.

`void network_Close( network_thread_t * p_network )`
This function shuts down the control socket and the TCP listener on the client.

The video and audio module API is available at the VideoLan website, http://www.videolan.org. We have not made any changes to these modules, hence the API documentation available on the website does not require any revision.

The Simba master

`static int open_connections( int* client_sockfd, int* worker_sockfd )`
This function opens two UDP sockets. The server uses client_sockfd for listening to client requests and worker_sockfd for sending multicast messages and forwarding client requests to workers.

`static void run_main_loop( int client_sockfd, int worker_sockfd )`
This function has the main server loop. It's an infinite loop in which it listens for client requests and forwards them to workers. It also periodically sends out multicast messages to workers to get load information, and receives replies from workers.

`static void process_client_request( int client_sockfd, int worker_sockfd )`
This function is called when a client request is received. It reads the client request and forwards it to the worker that is selected to server the client request.

`static void request_worker_load( int worker_sockfd )`
This function sends a multicast message to workers requesting load information. It is called at regular intervals of time. This time period is configurable, and is currently set to 3 minutes.

`static void process_worker_load( int worker_sockfd )`
This function processes the replies of workers to multicast messages.

`static int add_worker( in_addr_t ip_addr, unsigned int client_count )`
This function adds a worker node to the tail of linked-list of workers.

`static in_addr_t get_worker_address( void )`
This function returns the 32-bit IP address of worker who is selected to serve the client request. The scheduling algorithm is coded inside this function. It provides an abstraction to the end-user in that the algorithm can be replaced without changing any other code fragments.

`static void delete_worker_list( void )`
This function clears the linked-list at regular intervals so that fresh list can be rebuilt with the most recent load information from workers.

The Simba Worker

`static int initialize( int* server_sockfd, int* request_sockfd )`
This function opens two UDP sockets. The worker uses server_sockfd for listening to multicast messages from server, and request_sockfd for listening to requests from server to serve a particular client request.

`static void run_main_loop( int server_sockfd, int request_sockfd )`
This function has the main worker loop. It's an infinite loop in which it listens for multicast messages from server, and for requests from server to serve a particular client request.

`static int create_thread( void )`
This function is called when the worker receives a request from server to transfer a video fragment to client. It creates a thread that is a path of execution within worker process. As the thread is created it calls a function that manages the transfer of video fragment to the client.

`static void transfer_file( void )`
This function is associated with the thread that gets created to server a client request. It provides an interface for transferring video file to the client. It internally calls functions that establish connection to client and do the actual file transfer.

`static int open_client_connection( request_packet_t* server_request, int* file_fd, int* client_sockfd )`
This function establishes a TCP connection with the client for the transfer of video fragment. It gets the client IP address, port, UUID and fragment number (part of the filename) from server_request. It opens the file that is later referenced using file_fd and it opens a socket for talking to client that is later referenced using client_sockfd.

`static int transfer_file_data( int file_fd, int client_sockfd )`
This function does the actual file transfer to the client. It reads data from the file referenced by file_fd and transfers it to client by referencing the socket client_sockfd.