A Scalable Video Streaming Approach using Distributed B-Tree

by

Shantanu Khan Gramsci

B.Sc., Bangladesh University of Engineering and Technology, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in
The Faculty of Graduate Studies
(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)
April 2011
© Shantanu Khan Gramsci 2011
Abstract

Streaming video comprises the most of today’s Internet traffic, and it’s predicted to increase more. Today millions of users are watching video over the Internet; video sharing sites are getting more than billion hits per day. To serve this massive user base has always been a challenging job. Over the period of time a number of approaches have been proposed, mainly in two categories - client server and peer to peer based streaming. Despite the potential scalability benefits of peer to peer systems, most popular video sharing sites today are using client server model, leveraging the caching benefits of Content Delivery Networks. In such scenarios, video files are replicated among a group of edge servers, clients’ requests are directed to an edge server instead of serving by the original video source server. The main bottle neck to this approach is that each server has a capacity limit beyond which it cannot serve properly.

Instead of traditional file based streaming approach, in this thesis we propose to use distributed data structure as the underlying storage for streaming video. We developed a distributed B-tree, and stored video files in the B-tree which runs over a cluster of computers and served from there. We show that system throughput increases almost linearly when more computers are added to the system.
# Table of Contents

Abstract ........................................................................................................... ii

Table of Contents .......................................................................................... iii

List of Tables ................................................................................................... v

List of Figures ................................................................................................... vi

Acknowledgements ........................................................................................ vii

1 Introduction .................................................................................................. 1
  1.1 Motivation ............................................................................................... 1
  1.2 Thesis Contribution .................................................................................. 4
  1.3 Thesis Organization ................................................................................ 5

2 Background and Related Work ................................................................. 6
  2.1 Internet Video .......................................................................................... 6
    2.1.1 Downloading .................................................................................... 6
    2.1.2 Streaming ......................................................................................... 7
  2.2 Types of Streaming .................................................................................. 8
  2.3 Streaming Methods ................................................................................ 9
    2.3.1 Client Server Model ......................................................................... 9
    2.3.2 Peer-to-Peer Video Streaming .......................................................... 14
  2.4 Scalable Coding for Streaming Video .................................................... 16
  2.5 Distributed Data Structure for Streaming Video ..................................... 16
  2.6 Summary ................................................................................................. 17

3 System Design ............................................................................................. 19
  3.1 System Architecture .............................................................................. 19
  3.2 System Components .............................................................................. 20
    3.2.1 Sinfonia ........................................................................................... 21
    3.2.2 Distributed B-Tree .......................................................................... 24
List of Tables

3.1 B-tree common functions ...................................... 26
3.2 B-tree object types ............................................. 28
3.3 Fields used in B-tree node structure ....................... 29
3.4 B-tree transactional API ...................................... 30
## List of Figures

2.1 Video from streaming server ........................................ 11
3.1 System architecture ................................................... 20
3.2 Components of proposed approach .................................... 21
3.3 Sinfonia components .................................................... 22
3.4 B-tree conceptual diagram ............................................. 27
4.1 Single server performance when it is over loaded ............... 36
4.2 B-tree scalability for lookup operations ............................ 37
I am grateful to my supervisor Dr. Charles 'Buck' Krasic. Without Buck’s careful guidance and patience, this work would not be possible. I would also like to thank Dr. Son T. Vuong for his valuable feedback and comments as the second reader. I got a lot of help from my friends in the department and the lab, especially from Mahdi, Primal and Aiman. Big thanks to them. I am grateful to Ma, Shakuntala, Desdemona, Lenin, Filistin, Shejoabba, Chutoabba, Shahjahan and Nirban. Finally, Raha deserves special thanks for bearing with me.
Chapter 1

Introduction

Streaming video is increasingly popular these days. A recent study showed Canadians are spending more times on the Internet than watching TV [17]. YouTube [20], one single video sharing web site, gets more than two billion hits per day [13]. Increased Internet speed, widespread availability of Internet to mass people and to many devices, and overall easiness of streaming video make it more suitable to reach this massive audience. It is expected to increase even more [19]. However, handling this large user base has been a challenging job. A streaming system is usually fine for a certain number of users; it does not support beyond that, in some cases service degrades. There is always need to support more and more users. The requirement is if more resources are added, whether the system can handle more number of users. This capability of a streaming system is generally known as scalability of the system. In this thesis, we propose a way to improve scalability of video streaming system. Our approach is complementary to other approaches in this field, and can be used in conjunction with of the existing systems.

1.1 Motivation

Streaming video, in its simplest form, has two main components - servers and clients. Clients requests for specific video, and servers provide them. With streaming support, videos can be watched as soon as they start arriving, without being fully downloaded. This has the advantage over traditional file downloading, since we don’t need to wait for the full file to be downloaded which might take long time depending on the file size and network connection. Also, the Internet is built on top of a best effort model, i.e. when a file has been requested, servers just sends the file as fast as it can, which can congest the network, and saturate the CPU power. So, separate streaming server was the preferred way of streaming for many years. Streaming servers, unlike web server, don’t just dump bit streams to clients connection. It can take care clients preference like which part of video is requested, what is clients available network bandwidth, what is servers condition etc. and based on all these parameters it can stream the video. Because of this
1.1. Motivation

Intelligent feature, it’s capable of handling more number of users. But still, this approach has its own issues in practical usage. And, at the end, there has been a steady shift towards HTTP download based streaming in recent years, which is discussed in Chapter 2.

Traditionally, and most widely deployed streaming systems follow client-server model. A bunch of powerful machines operate as servers, and clients usually have much less capacity. However, the capacity of the servers is not infinite, and there is a limit of number of concurrent clients it can handle. For example, assuming a video file has been encoded at 200 KBPS, when a client requests this file, server will send this file at this speed though total file size may be few megabytes. Server may need to send this bit stream to multiple clients at the same time, each of which requires 200 KBPS. That means, even if servers have gigabit connection, there is a limit in number of clients a server can handle at one time. Most of the time, when a server is overloaded, all the subsequent requests are discarded; or the performance starts to degrade.

Content Delivery Network (CDN) provided a more efficient way of video streaming for client-server model. In CDN, a bunch of content delivery servers are usually placed in data centers connected with high speed Internet connection. When a server publishes a video, it is first replicated among a set of content delivery servers. This content delivery servers are usually placed strategically in geographically distant data centers. And a load balancing scheme is applied so that requests from users close to a specific data center is forwarded to that server. For example, a streaming system might have two different data centers one in East Coast and one in West Coast of United States. Visitors from San Francisco are more likely to be served from data center in the West Coast. Similarly, visitors from New York are likely to be served from East Coasts data center. Usually some videos become more popular than others, and they get the most hits. So those videos are replicated among multiple servers even in same data center. Thus, in a CDN based approach, clients can get the video from a nearby content delivery server, instead of original streaming server which might be too far. This effectively reduces the traffic imposed on streaming server allowing it serving more users. Also, it gives better viewing experience to users as there is low startup delay, and streaming is smooth. But still, capacity of any of these servers is not unlimited, and servers placed in a data center can only handle a limited number of clients. Even after CDN based approach and load balancing, a data center can get the maximum hit it can handle. So currently, the major challenge of client server based streaming system is its scalability.
1.1. Motivation

IP level multi-cast was also proposed to solve this kind of scalability problem. The idea was streaming server sends one video packet to a group of participants, rather than sending the same packet to each of them. In a IP multi-cast session, IP routers form a multi-cast tree with all the participants and the server is connected to the tree. However, this approach has never become very popular, and never been widely deployed. Mainly due to the fact that, managing multi-cast group pushes a lot of overhead to routers, and managing multi-cast session becomes quite complex. Instead, multi-cast session like functionality has been implemented in application layer, which essentially lead to Peer to Peer (P2P) systems. This way video servers and clients form an application level overlay networks to distribute video contents.

P2P file sharing was popular, and there were works towards P2P video streaming. P2P is a decentralized approach. Rather than putting all the loads on streaming servers, it allows client to serve other clients, thus off-loading some loads from server. This way one can find the video most likely from his/her neighbor’s computer. One important motivation behind P2P streaming is that client computers don’t usually use all the available network bandwidth completely. On the contrary, this is the opposite case in server. It is very easy to get overloaded when a lot of users watch video from the same server. But clients network bandwidth is usually good for watching own video and usually good enough other serving others. Thus P2P streaming provides an alternate and feasible way to reach large user base.

P2P systems have the potential to scale well to a very large user base, but it got its own issues to be efficiently deployed as video streaming system. Such video streaming system can be classified in two categories based on their overlay network structure: tree based and mesh based. In a tree based system, there is a well-organized overlay tree structure, and video server is placed at the root. Video is pushed actively from the parents to its children, and eventually it reach to all the peers. The major drawback to this approach is peer churn. A peer departure will temporarily disrupt the video delivery to all the peers in the sub tree rooted at the departed peer. In a mesh based, the peers are not confined with any static structure. Instead, paring relationship is dynamic; established based on the content availability and available recourses among the peers, and terminated when not needed. Peers periodically exchange information about what resource they have, and are willing to share. This way one peer can pull video contents from neighbors who have the same content. Since multiple peering relationships can be maintained, a peer can get from others in case of peer churn, which makes
it a highly robust approach. However, the dynamic peering relationship makes the video distribution to be unpredictable. Different video packets may travel different paths, which can add more delay and jitter in video playback. Consequently, users may face poor video quality from low bit rates, long startup delays and frequent playback freezes.

Currently, most popular and successful streaming systems are based on client server approach using CDN. YouTube is broadcast over CDNs. Also there has been a steady switch to streaming server based approach to HTTP progressive download based approach. In all cases, the capacity is limited by the server scalability. Though a bunch of powerful servers are employed in data center to serve more users, the reality is it does not scale well, at least linearly, when more servers are added. As the high speed Internet speed became available to more and more users, with the widespread availability of recording device, there are more video contents in the Internet than ever before, and video streaming continued to be more popular. To handle this enormous user base in such large scale application has been a challenging job. Even if we add more machines to the system, it does not scale proportionally, and in fact it stops performing well after certain limit. The situation is such that, if the system scales well with added computers, even if it’s not that much, but it scales linearly, that is desirable.

1.2 Thesis Contribution

Scalability issues have been of interest for industry and academia for long time. In fact, today’s smooth video streaming is the result of a combination of works done in this field. Recently, there has been a trend in designing scalable data structure mainly to be used in data center like environment to handle enormous amount of data. This is especially useful for large scale Internet applications like managing shopping cart for millions user, holding search results or managing the back end of multi-player online games. Such data structures include Dynamo [7], BigTable [5] which are custom data structure but can scale well across thousands of computers. Recently there has been works on distributed B-tree [1], which is a more standard data structure that is distributed, and also scales well. As far as we know, none of this works were directly applied to video streaming. In this thesis, we implemented Sinfonia [2], which is a distributed data sharing service for cluster environment. On top of Sinfonia, we implemented a distributed B-tree, and used this B-tree to store streaming video files and stream from there. We found that this scheme scales almost linearly with the increased
system size. So the main thesis contribution is applying a novel approach of using distributed data structure as storage for streaming video. Our approach is complementary to most of the existing approaches, and can be used in conjunction both CDN and P2P based system, though it is mainly meant for data center environment.

1.3 Thesis Organization

This thesis consists of 5 chapters. Chapter 2 provides background details of video streaming technologies in place. It covers from the very basics of video streaming to latest techniques, a comparative analysis among them, and finally how they relate to our approach. In chapter 3, we provide an in depth analysis of our developed system with description of each component and layer. We analysis why our system scale well in Chapter 4, and finally we conclude in Chapter 5 with directions to some possible future works.
Chapter 2

Background and Related Work

In this chapter we discuss background details of watching Internet video, various approaches that has been taken over time and how they evolved to currently most successfully deployed ones, and finally we discuss where our proposed approach stands among other approach. As we will see, our approach is complementary most of this, and can be used together to enhance scalability performance.

2.1 Internet Video

The amount of Internet video content is exploding. Cisco system predicted by end of 2013, 90 percent of Internet traffic will be video content [19]. Over the time of Internet technologies changes, video delivery methods also had different phases. In general we can classify the methods used to deliver Internet video in two groups - downloading and streaming.

2.1.1 Downloading

Downloading is the simplest form of delivering video over the Internet. It is based on the same model that Internet was designed on - best effort model. It works the same way every web pages work. Video files are placed in a web server along with other supported files needed for web site, and web servers serve them like other regular files. For this reason it is easy to set up and use on almost any website, without any additional complexity. All that needed is to provide a simple hyper link to the download-able video files. A slightly more advanced method is to embed the file in a web page using special HTML code. Web browsers requests for video files the same way they request for text and image files. Once requested, we servers send the whole file content to the client. Whole file is then saved on clients computer, usually in a temporary folder, which users then open and view. The advantage to this approach is that users can navigate to different parts
2.1. Internet Video

of the file quickly, and can watch it later since it’s stored in users computer. However, the obvious disadvantage is user has to wait for longer time if the file size is too big, and Internet bandwidth is limited. If the file is too small, it may not be too much of an inconvenience, but for relatively larger files this is not real time at all. The root problem of this approach is the way Internet was designed. It’s by nature best effort model, so web servers try to send data as soon as it was requested and as fast as it can. So, for larger files, servers can easily saturate clients Internet connection, which leads to other problems.

This approach of video delivery is known as HTTP Streaming or HTTP Delivery. HTTP stands for Hyper Text Transfer Protocol which is the underlying protocol to deliver web pages over the Internet. In the early days of Internet video streaming this was the most commonly used approach which still being used for smaller files and small scale user base. Later there was a trend of using separate video streaming server on top different protocols other than HTTP. However, in recent years, there has been steady switch again towards HTTP based video delivery with some modifications.

2.1.2 Streaming

With streaming approach, clients can watch the video as soon as it starts arriving to users computers from server, without waiting to have downloaded the full file. In fact, the file is sent users in a stream, and users watch them as they arrive. This is analogous to water drops flowing in the pipe. When water is poured in one end of pipe, it comes out of the other end as stream water. Same way small chunks of video files flow across the net from server to users computer ahead of they are being played. If video chunks cannot arrive in time, ahead of users view them, because of congestion in network traffic or other reasons, users experience momentarily pause or breakups in the video playback.

In this approach, video is delivered from a specialized streaming server. Unlike web servers, it only sends portion of video file over the connection to client, instead of sending full file. So this kind of servers does not saturate the network connection. In fact, this approach can takes care of a bunch of parameters like available network connection, CPU power in both in servers and clients, and user preference for video quality. Because of this negotiation, this streaming approach required new protocol \(^1\) other than HTTP. On the client side, since video can be watched without being fully downloaded,

\(^1\)discussed later
it needed special client video player too. Some of them are standalone application, and some of them are plug-ins, to enable viewing video inside the web browser. Because of its adaptive nature, this approach allows more users to be served from the same server.

The obvious advantage to this is there is not much waiting time involved, which is way better comparing to file based download. Streaming players usually maintain a buffer between the playback position and streaming position from the server. Streaming position is ahead of playback position, and video content between this two positions are held in the playback buffer. In case the video does not arrive in regular rate, playback position will move faster than streaming position. In case playback position crosses streaming position, users experience pause in the video playback. But if streaming recovers before playback position crosses streaming position, users will not notice any breakup, which is the sole purpose of using buffer.

Streaming approach also provided unique benefits over traditional television broadcasting. With traditional broadcasting, everyone watch the same content at the same time. This is good for live events, but not everyone can watch the show when its broadcasted. With streaming technologies, it is possible for many users to watch same video, and to be in different positions of the same video. It truly made it possible to watch video at any point at any time. With the widespread availability of high speed Internet this became very common these days.

**Progressive Downloading**

There is also a hybrid method known as progressive download. This is basically an enhanced HTTP delivery. In this method the whole video clip is downloaded but begins playing as soon as a portion of the file has been received. Most widely used Internet video sharing sites like YouTube are broadcasted using this approach. There has been also a number of standardization works in this direction.

### 2.2 Types of Streaming

Video streaming can be used in a number of scenarios - from covering live events to multimedia conferences to watch TV shows. This usage can be classified in two categories - live and on-demand. In live streaming, video content is sent to users in real time. So the video playbacks on all users are synchronized. This is especially useful to cover live events like games, shows or for holding video conference. On the other hand, video-on-demand
2.3 Streaming Methods

Any streaming method, in its very basic form, consists of two parts - clients and servers. But when it comes to deployment, they can use different architecture. Depending on the deployment architecture streaming methods can be broadly classified in two categories: client server based and Peer-to-Peer based video streaming. Client server based solutions in conjunction with Content Delivery Networks, are used in most successful video streaming sites. P2P solutions have the inherent benefit of scalability, and are being considered as a viable alternative, though in practice most successful ones are based on client server model.

2.3.1 Client Server Model

This model was introduced at the beginning of Internet video streaming, and after many improvements and trends are still the dominant way of streaming video. It started with file based media delivery from web server; then separate streaming server was introduced using a streaming protocol like Real Time Streaming Protocol (RTSP) [8] which was prominent for quite long time. Recently, there has been a lot of interest in applying file based delivery over HTTP.

Using Streaming Servers

Video streaming using separate streaming server has been traditional method for many years. Streaming server has the intelligence of adaptive streaming which web servers could not provide. As mentioned earlier, streaming servers dont just dump the whole file to users connection, it opens a conversion with the media player in the client side first. It then sends video in chunks such that it does not overload network, but can still provide good viewing experience to users. Streaming servers know about the format, bandwidth, and structure of video files being served, and in many cases, pays attention to the performance of the player that’s receiving the video. Thus streaming
servers deliver just the amount of data that can be handled efficiently by the user. Because of this negotiation feature new streaming protocol was also introduced, RTSP being the most common one. With RTSP clients can open different session to the server, and server maintains the session which maintains clients state until it is disconnected. The client can open several connections in the same session to request data. Once a session is established, the client can send commands to server which is similar to video playback like PLAY, PAUSE etc. Thus RTSP clients can act like network remote controller for streaming server. Once the session is established between servers and clients, they continue to exchange control messages, which helps streaming servers to adjust to changing network conditions as the video plays, improving the viewing experience.

There has been quite a number of streaming technologies based on separate streaming server. Examples include Real Media, QuickTime, Flash Video, Windows Media etc. Each of them used their own proprietary format to store video, which means there is corresponding streaming player for each of them. There are both standalone application and web browser plug-in version of streaming players. A common use case is when users click on a video link in a web page video starts playing. The underlying operation is, when users clicks on the video, it triggers a connection with media server, and media streaming on clients computer. Thus media servers are commonly used with web server. Texts and graphics are placed in web server, and video files are in streaming server. By using certain HTML tag users can trigger and control playback from the streaming server. Each of the streaming technologies has their own streaming HTML tag to embed video.

Technically, streaming servers can use HTTP and TCP to deliver video streams, but by default they use protocols more suited to streaming, such as RTSP and UDP. RTSP provides built-in support for the control messages and other features of streaming servers. UDP is a lightweight protocol that saves bandwidth by introducing less overhead than other protocols. It’s more concerned with continuous delivery than with being 100 percent accurate a feature that makes it well-suited to real time operations like streaming. Unlike TCP, it doesn’t request resends of missing packets. With UDP, if a packet gets dropped on the way from the server to the player, the server just keeps sending data. The idea behind UDP is that it’s better to have a momentary glitch in the audio or video than to stop everything and wait for the missing data to arrive.

Using streaming server has some obvious advantages. Since the server sends video data only as it’s needed and at the rate its needed, it allows
2.3. Streaming Methods

Despite of all such benefits, there has been a steady switch towards HTTP based progressive download based video delivery. One of the main problems of streaming server based delivery is it used UDP as the underlying transport protocol. Regular web pages are delivered over TCP using HTTP. So most routers and firewalls know about this protocol, and do not impose any limitation. However, for UDP and proprietary streaming protocols it is not the case. In many networks UDP is blocked as an added security. With so many firewalls and network devices available from different vendors, UDP can get blocked in many scenarios, which leads streaming video to be not working. UDP network and firewall traversal is still an ongoing research
2.3. Streaming Methods

and standardization issue in IP communication like Voice over IP. Separate streaming server is still being used especially for delivering live contents. However, for on demand video streaming current trend is to use classic web page delivery scheme with some enhancement. This approach is simple, proven to work. Basically, if users can view web page, they should be able watch video.

**Streaming from Web Server**

Video delivery from web server was used at the very beginning of Internet video streaming. This uses the same approach used to deliver web pages, and does not require any additional infrastructure support. With the widespread availability of high bandwidth network connection, and some improved streaming methodologies, the difference between video delivery from streaming server and web server is blurred. In fact, there has been a steady switch from classic media delivery approach to plain HTTP based progressive download for media delivery in past several years. Almost all of the popular video sharing websites like YouTube, Vimeo etc. are based on Http progressive download. Its basically file download from web server. The term progressive refers the fact that the content can be displayed without being fully downloaded. Recent major shifts towards http based progressive download mainly because of the fact media protocols often use UDP as transport media which gets blocked at router and firewall for many users, thus reducing scalability. On the other hand, HTTP connections can be established from most of the Internet connections, and its already proven to work. It does not require any special proxy since most firewalls and firewalls allows HTTP traffic. That way media delivery becomes regular web download. In fact, though media protocol was designed for streaming in mind, whole Internet was designed and optimized for HTTP, hence a shift towards HTTP based streaming makes sense.

The original HTTP based video delivery scheme used to send the whole file to clients as fast as it can, which was the main source of all its problems. However, there has been some recent works [6] [16] which overcame this problem, and make http based video a viable alternative of using streaming servers. All of these approaches share the same generic idea- rather than sending the full video file, video files are split into multiple files of small chunks. When requested, those small files are being served, rather than the big full file. Client requests for the chunks linearly, and plays them sequentially. So chunks are delivered and displayed in such a way that it gives users an impression of streaming video. Since split chunks are served
2.3. Streaming Methods

form a web server, video delivery essentially becomes regular web download. Clients requests for video chunks like any other web object which is proven to work if web pages can be viewed. Thus this approach allows reaching more users, allowing scaling well.

One shuttle advantage of using streaming server over web server was streaming server could handle adaptive streaming. At the beginning of streaming, there is a conversation established between servers and clients. Based on the negotiation, streaming server can decide at which bit rate it should deliver video, which essentially lead to better viewing experience for end users. However, web servers being designed to work on best effort model, delivers video data as fast as it can. To achieve this adaptive streaming benefit in HTTP based approach, video files are encoded at multiple quality level at different bit rates, and adaptive logic has been moved to client side. When clients request for video chunks, it can choose different quality level that it can handle based on several other factors like network condition, CPU usage and other necessary resources. Thus rather than servers choosing the appropriate quality level for each client, this approach allows clients to choose the desired quality level.

With this approach, same files are encoded at different quality level and depending on the length of chunk and total duration one single file can generate huge number of chunks which can be management problem for CDN operators. For example, if file is split along each 2 seconds, and there are 5 different quality levels, that means 10 files for each 2 seconds, which means for a movie of 3 hours, this will be 54000 files. However, with the advance in technology in this area, HTTP based video delivery proven quite successful alternative of streaming servers.

In next sections we discuss two recent technologies that uses web server to stream video using appropriate client side software.

Smooth Streaming Smooth Streaming is Microsoft implementation of adaptive streaming using regular HTTP connection as a form of web based media delivery, introduced as an extension to Internet Information Server 7 (IIS 7). Smooth Streaming compatible clients need to have Microsoft Silverlight plug-in installed. Instead of delivering video as full file, it is delivered as a series of fragments. Smooth Stream compatible clients request for the chunks in special order, and web server delivers them as regular web downloads. These chunks are played in such a way that creates a smooth video effect. Such clients use special heuristics to dynamically monitor current network and local PC conditions and seamlessly switch the
video quality when requesting subsequent fragments. As clients play the fragments, network conditions may change or video processing may be impacted by other applications that are running. Clients can immediately request that the next fragment come from a stream that is encoded at a different bit rate to accommodate the changing conditions. Thus adaptive streaming has been implemented in Smooth Streaming.

Smooth Streaming also provided a better way of creating and managing contents. It defines each chunk or group of picture (GOP) to be a MPEG-4 movie format, though it does not keep them in a separate file. It keeps all them in a special type of single continues file, MP4. When a video files is placed in IIS7, one Mp4 file is generated for each quality level. When a client request a specific chunk, the web server finds that chunk within the Mp4 file, and sends it over the network as a single standalone file. In other words, Smooth Streaming allows file chunks to be created virtually upon client request though actual video is stored as a full length video on server.

Http Live Streaming
Http Live Streaming [16] is a recent proposal from Apple, which has been submitted as a draft to IETF. Similar approach has been used. Video files are also chunked into small files, and kept in web server. While producing small chunks, it also produces a play list file, which is basically an index file for all the generated chunks. Streaming clients fetch the playlist file first, it then retrieves the URL for each chunks from the play list. It then requests for the chunks as it appears in the index file. Once it has enough chunks are downloaded, it plays the client plays back video.

2.3.2 Peer-to-Peer Video Streaming
Peer-to-Peer (P2P) networking has recently emerged as new paradigm of developing large scale distributed system. Especially for file sharing applications using P2P networking have been deployed widely and gained widespread popularity. More recently, there has been works on streaming video contents using P2P networks. In P2P approach, a peer work as both client and server. That means a peer not only get download data from other computer, it serves to other computer as well. Instead of serving from the original streaming server, a peer gets video data from neighboring peers. Thus P2P utilizes a peers uploading bandwidth efficiently which essentially reduces the bandwidth burdens on the servers.

For live video streaming, all the participant peers are interested in the

\[\text{for example, bandwidth may decrease}\]
2.3. Streaming Methods

same video contents; so peers are synchronized in terms of playback. However, for this work, we are interested in on-demand video streaming which is asynchronous by nature. Although a large number of users may be watching the same video, different users may be watching different portions of the same video at any given moment, thus it imposes different challenge in on demand video streaming using P2P. P2P video streaming systems can be broadly classified into two groups tree based and mesh based streaming system. In general the asynchronous behavior of on demand streaming has been addressed by caching contents at users.

In a tree based streaming system, peers form an application level overlay tree structure, where the video source server is placed at the root. Each user joins the tree at certain level, and receives video from the parent peer one level above of its position. Similarly it serves the children peers which are one level below its current level in the tree hierarchy. However, as mentioned before, main problem is different users may be at different position watching hopefully the same video. A number of approaches have been proposed to handle such asynchronous nature of on demand video streaming [11]. The main idea is to cache certain amount of video at the peer level, and which is used to serve other peers. Users can be grouped into sessions based on the arrival time. A threshold time can be maintained, so that users that come within the threshold period will be in the same session. Together with the video source server and users belonging to the same session form an application level multi-cast tree known as base the tree. There can be many such base trees involving the video server. Server streams the entire video file to its children in the base trees, known as base stream. Once a peer receives video from the server, it then broadcasts it to the whole sub tree. When a new user joins the tree based on arrival time, it first gets the base stream, but it may miss the part of the video. Users who have watched it cache the patches and serve the late comer. Thus using the base stream forwarding, caching certain amount of video and serving new peers synchronization issues is successfully addressed.

In a mesh based system peers dont form the static structure; instead they maintain a dynamic paring relationship among the neighbors. Peers periodically exchanges information about what contents they have to share. Based on that paring is established and terminated when not needed. In mesh based file sharing approach, a file is broken into smaller chunks, and these chunks are distributed among the peers. Peers download the data blocks that they dont currently have. If the data blocks at different users are different, its good for improving systems because users will use their upload bandwidth, and get the complete file from each other. This diversity
2.4 Scalable Coding for Streaming Video

Adopting HTTP based video delivery helped to reach a large number of audience, increasing scalability. In all the previous approach, video files are split into smaller chunks and served by web server. For different quality levels, multiple versions of the same file are used. That means there can be multiple files based on bit rate, spatial and temporal quality levels. It has the simplicity that clients will select appropriate bit stream according to its desire, and clients also have to decode just one video quality layer. However, from server management side, this can be huge memory and disk space usage based for large video repositories. In recent years there has been a lot of attention towards scalable video coding [3, 18], which solves the problem with multiple file for each video. In this method, video is encoded once but different quality levels, and stored on web server. When the encoded bit stream is sent to client side, they can only decode at the quality level they are willing to. This approach simplifies the overall approach, and helps to achieve better quality.

2.5 Distributed Data Structure for Streaming Video

All of the above described approaches are similar in the sense that all use a file based approach. This can be single file or multiple files; they are stored on a server, and served from there. But in reality there is a limit for the maximum number of users a server can handle. Generally, CDN providers use data centers located geographically dispersed with replicated contents. So clients are not directly server by the original video source server, instead of served by edge servers which have replicated content. This way the service can scale to a very large user base. However, each single server will be limited
to maximum capacity it can handle, beyond which the performance starts to degrade. Even if the number of servers is increased, performance does not increase proportionately.

There has been a number of works in designing distributed data structure that scales well. Google uses Bigtable [5], Amazon uses Dynamo [7], Facebook also uses a similar data structure called Cassandra [14]. All of these are custom data structures and serve the purpose they are used for. There has been works on more standard data structure like Distributed B-Tree [1] which has been proven to scale well. Scalability issues in video streaming have been addressed in various level and different ways. All of them uses file based approach. In this thesis, we proposed to use distributed B-tree as the underlying storage for streaming video. Instead of keeping as regular file, video files are broken into layers and kept in a b-tree data structure spread over a bunch of well-connected clustered servers. As far as we know this is the first work that uses distributed B-tree as the storage for streaming video.

We first implemented Sinfonia [2], a distributed data sharing service meant to work in data center environments. It has been proven to scale linearly among the cluster of well-connected servers. We then implemented a distributed B-tree on top of Sinfonia. At the time of this work, there was no open source implementation of Sinfonia and distributed B-tree. In our approach, we keep video files in a special B-tree form in the distributed B-tree which is spread over several inter-connected machines. We assumed this will be used in data center environment where network bandwidth is not an issue. Rather than keeping files in disk, we keep in them in a shared memory service distributed environment using a distributed B-tree data structure, which has been proven to scale linearly with the system size. The intuition behind this work is, since the underlying data structure is proven to scale linearly with the increase in system size, video streaming system on top of this should scale accordingly as well. The evaluation of our system also revealed that system throughput increases linearly when more number of servers added.

2.6 Summary

In this chapter we presented background details of streaming video. We explained some common terms used frequently in video streaming area, then discussed commonly used approaches, the trends used in video streaming, and how it shifted back to original http based download approach. We
2.6. Summary

focused that despite of recent success of large scale video streaming sites, there is a certain limit a server can handle, and that is still a bottle neck in increasing overall system scalability. Then we presented our distributed B-tree based streaming approach, which we believe can be a potential approach to break this bottle neck.
Chapter 3

System Design

In this chapter we describe our proposed system video streaming using distributed B-tree. Our work was inspired by a number of previous works on distribute data structure, and was implemented using QStream Streaming Framework (QSF) [9]. We used Sinfonia [2] as the underlying data sharing service on top of which we built a Distribute B-tree [1]. At the time of this writing, there is no open source implementation of Sinfonia or Distribute B-tree. So we developed both from scratch for this work.

In next few sections we describe our proposed architecture, its layers and components in details.

3.1 System Architecture

Our proposal is to use a distributed B-tree for streaming video. Rather than serving by a single server\(^3\), we propose to serve video contents from a group of servers. These servers are connected in a way that overall system throughput increases when more servers are added to system.

For this system, we developed a B-tree which is distributed over a bunch of servers. Unlike traditional B-tree in a computer, the tree is not in the same address space. It is a conceptual tree whose root is one computer, children nodes can be in other computers, thus spread over network. However, one single node is always in any single server. To achieve high scalability, we relied on recent works on data sharing service, Sinfonia. We first implemented a Sinfonia service. Using Sinfonia a group of computers, called memory nodes, allow others to keep their data and to get from there. Application nodes, the computers that put to and get data from memory nodes, use a client side library to communicate with memory nodes.

Our proposed streaming system is shown in Figure 3.1. A bunch of computers in a data center environment collectively serve video contents to the users. Some are part of memory nodes, and some are application nodes. Application nodes create a distributed B-tree structure over the

\(^{\text{3}}\) video source server or the server with a replica content
memory nodes, though memory nodes have no idea what data is being put in their memory or what data structure is being formed from the data. When video is uploaded to the system\(^4\), it is stored over the B-tree. When this video is requested, B-tree client computers first get the content from B-tree, then serves to user. Later we show that with this approach, overall system throughput increases as we add more servers to the B-tree.

We heavily relied on QStream for this work. QStream is an experimental video streaming system which has been used in research for quite few years now. We used mainly two QSF components - Paceline [12] and Qvid. Paceline is the underlying TCP transport library. Qvid is the video streaming component of QStream. We extended Qvid to store video files over the B-tree, and stream from there when requested.

### 3.2 System Components

Our implementation can be divided into layers as shown in the Figure [3.2](#). We used Paceline for all network communication below Sinfonia layer. On top of Sinfonia, we implemented B-tree, which, in turn, is being used from

---

\(^4\)
either by administrator or users
3.2. System Components

Qvid. Thus, this work is surrounded by QStream components. In next few sections, we describe the two main components - Sinfonia and B-tree, their features, implementation details and discuss how video is stored in the B-tree.

![Component Diagram](image)

Figure 3.2: Components of proposed approach

3.2.1 Sinfonia

The goal of our work is to use a distributed data structure as the storage support for streaming video. However, building a distributed data structure that is highly scalable is tough. Most distributed systems are designed with message passing paradigm to maintain distributed state, which turns out to be complex to manage and becomes error prone. This is where Sinfonia is helpful.

Sinfonia is a data sharing service, distributed over a bunch of networked computers. It allows applications to share data in a fault-tolerant and consistent way which is highly scalable. Using Sinfonia, developers can design a distributed application just by designing data structure within the ser-

---

5We did not modify Paceline, hence not described here
vice, rather than dealing with complex message passing protocols. Sinfonia provided the underlying scalable infrastructure support needed to develop distributed data structure we intend to use for video storage.

**Sinfonia Nodes**

Sinfonia service consists of two main components: memory nodes and application nodes. Memory nodes are the computers that host data and application nodes keep data in memory nodes. Application nodes use a client side library to communicate with memory nodes. In terms of implementation, memory nodes are the server processes we keep running that keep users data. User library communicates with memory nodes using Paceline. Each memory node opens a linear address space where application nodes can keep their data. Sinfonia does not impose any structure on data being kept which helps to reduce complexity. One of the key element to achieve scalability is to decouple things as much as possible, and Sinfonia does it by opening a fine-grained address space without imposing any structure like type, schema etc. Thus application nodes can host data in Sinfonia relatively independently of each others.

![Figure 3.3: Components of Sinfonia](image-url)
Minitransaction

Sinfonia introduced a concept of minitransaction, which allows multiple operations to be transactional. Transactions are atomic, so it either executes completely or not at all. For example, it is possible for an application node to read from three different memory nodes, compare some memory locations on another different nodes, and write to possibly different memory nodes, all within the same transaction. If any comparison fails, everything will be aborted, and the advantage is, users do not need to deal with this. In a traditional way of distributed computing, developers need to maintain the distributed state using message passing which added more complexity. Besides the simplicity, minitransaction is also useful for improving performance. When multiple operations are batched together in a transaction, it essentially reduces network round trip time.

A minitransaction consists of three sets of transaction items, namely read items, write items, and compare items. Read items are the data to be read, compare items are the data to be compared with, and write items are data written to memory nodes when the transaction is executed. Each of this items specifies a memory node, and address range that is needed in the corresponding operation. Compare items also provide the data that is needed to compare with the data in the specified memory node within the specified address space. Write items also include data that is to be written in the same location in the specified memory nodes at the given location. Applications using minitransaction create a minitransaction instance first using the Sinfonia client side library. It then adds transaction items to the appropriate set and finally, when it is ready it executes the transaction. All items are added before transaction starts executing, and once it started, no more items are allowed to add.

Minitransaction executes in two phases. In first phase, application nodes send the minitransaction to participant memory nodes. It first goes through all the transaction items in the minitransaction, and prepare the list of memory nodes involved in this transaction. It then sends specific command to each of participant memory nodes in this transaction, and wait for response from each of these memory nodes. This message basically tells the memory node to be prepared to execute a transaction. It does not mean to execute the transaction right away.

Upon receiving the message to be prepared, each memory node then tries to lock the addresses mentioned in the items without blocking. It then compares the data mentioned in each of compare item sets against data available in the memory node at corresponding positions. If the comparison
3.2. System Components

is succeeded for each of them, it returns the read items, buffers the write item and sends a vote to the originating application nodes indicating that it is ready to commit. If the comparison finds any difference in the compare items, the memory node sends no vote to the originating application node.

In the second phase, the application node gathers response from all the participant memory nodes, and if everyone votes positive for commit, application node then sends another commands to participants. Upon receiving this command, memory nodes then write the buffered write items to memory, and complete the transaction. If any of the memory node votes negative, application nodes sends message to abort the transaction. In either case, this phase completes the transaction. Thus, compare items has the ability to control transaction, read items determine what data transaction will return and write items specify what items to be updated.

Optimization

Sinfonia can be optimized to get even better result, especially for video streaming. Our overall goal is to store video files in a distributed data structure developed on top of Sinfonia. For streaming video, most common feature is to read from the structure, which essentially creates a lot of minitransaction with only read items. In those cases since there is no compare and write items involved, there is no need to wait for memory nodes to decide on vote, and then issuing a separate command for actual commit. This means for read only items, Sinfonia transaction can be accomplished on just one round trip time. Because memory nodes return the read items in the first phase of minitransaction, we can essentially get rid of second phase. This is one of the key elements of improving performance.

Sinfonia provides a number of features to make scalable data sharing in distributed environment, minitransaction being the most important one. We implemented a very basic form of Sinfonia needed to make video streaming prototype working. We focused on building the B-tree on top of Sinfonia which we used to store video files, and omitted adding advanced feature in Sinfonia like fault tolerance, recovery from crashes etc.

3.2.2 Distributed B-Tree

The main theme of our work is to store video files in a distributed data structure and serve from them there. There has been recent work on this

\[^6\] whether it will abort or commit
3.2. System Components

[1], however there was no open source implementation of the proposal. So for this work, we implemented a distributed B-tree first.

B-Tree Overview

A B-tree is a special kind of tree data structure which is internally a balanced tree. It stores a set of key value pairs. B-tree supports standard dictionary operations like Insert, Lookup, Update, Delete and enumeration operations like GetNext, GetPrev. Our version of B-tree is actually B+ tree, a variation of B-tree where values are only stored in leaf nodes. Internal nodes only keep pointer to next key. In each level, keys are stored in sorted way. So the dictionary operations like GetNext and GetPrev is easy with B-tree.

For lookup operation, we start from the root, and follow appropriate pointers to its children until we find the leaf node. Recall, our B-tree is a B+ tree, so only leaf nodes contain both key and values. Inner nodes only contain keys, not values. So while doing lookup operation, we might encounter the key we are looking for but to find the corresponding value, we need to continue until we hit leaf node. Update is similar. We need to find the appropriate leaf node first, then we change corresponding value for that key. For insert, we first find the leaf node where this key should go and whether there is an empty slot. If there is any place to insert, we insert the key-value pair there. But if there is no space left in that node, we split it into two children nodes, and modify the parent appropriately. However, this insertion may break the B-tree property because the new parent may just exceed its key-value pair limit, which means we may need to do it recursively. Delete operation has similar approach. We look up the key first; once found we remove the key-value pair, and merge nodes if there are less than half full [7]. GetNext and GetPrev are almost identical to lookup. We find the key first with lookup. Since all the key value pair is stored in leaf nodes, once we find the key in leaf node, next pointer basically means next node. GetPrev is very similar. These operations are summarized in Table 3.1.

B-tree functionalities are very useful for building storage systems like file systems and database systems. B-tree is basically a generalization of binary search tree, so searching a key can be done in logarithmic time. Also keys are stored in sorted order, so both sequential access and random access relatively easier using B-tree. On top of that, it is optimized to handle large block of data. Because of all these storage friendly features B-tree has been quite popular for building file systems and database systems. Projects like MySql, BerkeleyDB or CouchDB all are based on B-tree.

7or any defined size
3.2. System Components

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lookup($k$)</td>
<td>Lookup key $k$</td>
</tr>
<tr>
<td>Update($k,v$)</td>
<td>Update key $k$ with value $v$</td>
</tr>
<tr>
<td>Insert($k,v$)</td>
<td>Insert key $k$ with value $v$</td>
</tr>
<tr>
<td>GetNext($k$)</td>
<td>Returns the next key of $k$</td>
</tr>
<tr>
<td>GetPrev($k$)</td>
<td>Returns the previous key of $k$</td>
</tr>
</tbody>
</table>

Table 3.1: B-tree common functions.

B-Tree in Distributed Computing

Traditional B-tree is a data structure in a single computer’s address space. We call it centralized B-tree. All the pointers to its children and parent nodes in a centralized B-tree are within same address space, and can be effectively denoted by any pointer notation of programming languages like C/C++. However, for distributed B-tree, the tree is formed over a bunch of computer connected with network connection. It is actually a logical tree where different components are distributed across several machines. Figure 3.4 shows it in details. In this context, the next or previous pointers can be on completely different machines and different address space. Hence, regular pointer notation found in programming languages will not work in such implementation. Instead we need to define an application level structure to denote such pointer operation, described later.

Distributed Environment Consideration

We needed the same benefits from distributed B-tree found in centralized B-tree. Logically all the operations are similar. However, because of distributed nature, all of these operations need to traverse network computers to complete the operations. This traversal can be a performance issue. However using Sinfonia as the underlying data sharing service, and some other optimization technique this approach has been proven to work well. Like centralized B-tree, all the lookup operation starts from the root node. From root node, applications know where to go next. It then fetches that node, which can be on a different computer. Once fetch is done, it checks whether this node has the key its looking for. This process is continues until it fetches the leaf node from some computer.

The same lookup operation is done first to update a node. Once found, the node is updated. Insert also finds the leaf node first where it should insert the new key. However, this insert may cause split in the tree nodes as we have seen in centralized approach.
3.2. System Components

![Diagram of B-tree conceptually spread over servers. Clients only cache inner nodes. Leaf nodes are only available in server. Logical B-tree is shown in each server, though altogether it makes a single B-tree.]

In centralized B-tree all the splits we did recursively were in the same computer, i.e. same address space, and we could do it recursively. However, for distributed B-tree, since nodes are distributed over networked computers, so changing in tree structure might involve changing in a number of computers. This means insert operations might need to read from a bunch of computers, and potentially writing to a group of computers as well. And of course, there is comparison involved in most of these operations. For smaller approach this is probably okay, but for larger-tree this becomes a problem to maintain the state distributed over network. Because, a B-tree nodes in the network might be serving someone else while some user tries to insert. To manage such state, traditionally message passing protocol has been used, which creates complexity for large scale application, and more error-prone. Sinfonia was designed to isolate this part of distributed application. As mentioned in earlier section, Sinfonia provides a data sharing service, in a fault tolerant and scalable way. So using Sinfonia, we don’t need to manage...
3.2. System Components

<table>
<thead>
<tr>
<th>Object ID</th>
<th>Used to uniquely identify objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree Node</td>
<td>Stores key value pairs</td>
</tr>
<tr>
<td>Bit Vector</td>
<td>Used to indicate free and empty memory</td>
</tr>
<tr>
<td>Meta Data</td>
<td>Holds server list and root node ID</td>
</tr>
</tbody>
</table>

Table 3.2: B-tree object types

Distributed state when tree structure is changed across multiple computers. Sinfonia provided powerful primitive, minitransaction. All the B-tree read, write and comparison operations can fit it to a Sinfonia minitransaction transaction items. When the transaction executes, all the operations are executed across multiple computers. Thus other than the logical operation of B-tree, underlying distributed state management can be done easily using Sinfonia.

Implementation Details

Sinfonia memory nodes provide the address space where user put their data. All the Sinfonia operations are done through minitransaction. Sinfonia does not impose any data structure requirement for the data being kept in its memory node. But to design a distributed data structure on top of Sinfonia, applications need to maintain the data structure notion in client side. Sinfonia memory nodes have no idea what structure is being used to keep data. Its the application nodes that handle the logical complexity of managing data structure, and defines application specific data structure. For the distributed B-tree, the authors proposed using some objects to maintain the B-tree structure. We followed the same approach as the original authors since it was proven scale well.

Though the logical distributed B-tree algorithm is quite similar to centralized B-tree, implementation-wise it was quite different since we could not use the well-defined notion of pointer. Pointers in centralized approach points to a memory location on the same computer, but here this is another computer in most cases. So pointer based structure could not be used in such implementation. Instead we defined some object types that helps define the node structure, to manage B-tree properties and also to navigated nodes over the tree spanned across many computers. We describe the object types we used in next few sections, and also in Table 3.2.

- **Object ID** are used as the replacement of pointers in centralized approach. Its basically a C programming structure with the fields to
3.2. System Components

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isLeaf</td>
<td>Indicates whether a node is leaf</td>
</tr>
<tr>
<td>height</td>
<td>Distance from root</td>
</tr>
<tr>
<td>numKeys</td>
<td>Number of keys stored</td>
</tr>
<tr>
<td>keys[1...numKeys]</td>
<td>Array of keys</td>
</tr>
<tr>
<td>values[1...numKeys]</td>
<td>Array of values</td>
</tr>
<tr>
<td>children[1...numKeys]</td>
<td>Children pointers in inner nodes</td>
</tr>
</tbody>
</table>

Table 3.3: Fields used in B-tree node structure

identify memory nodes, and what address is being pointed to. Object ID can uniquely identify an object in the overall system. It also includes a version number field, which is increase each time an object is changed. This is helpful because when we look for whether whole object has changed, we can just compare the version number of the same object instead of comparing whole object. This reduces network bandwidth, and contributes to overall fast response. Each of the other tree objects we introduced (discussed later) has own object ID.

- **Tree Node** describes each node in the tree. Its the most commonly used object type that takes the most memory in the B-tree space. From root to inner node to leaf node, every node is represented using this structure. It provides all the fields to accommodate all these uses. Tree Node has a field to indicate whether this is root or leaf node. It maintains an array of key-value pair that is supposed to in this node. Keys are sorted, so finding them is easy. In case of inner node, value field is empty. But for leaf node, both key and value are present as pair. There is also another array of object ID which basically points to children nodes. This is equivalent to pointers to children in centralized approach. When tree nodes are traverses, this is how application gets to know which node to fetch next. This array is empty for leaf nodes and marked accordingly. Tree node also holds a field to indicate the height of this node in the tree, i.e. distance from this node to root node. All the B-tree operation starts from root, and root node might get changed just like root pointer changes in centralized approach. But new operations or new users of distributed B-tree need to find root to start any operations on it. Hence there is a requirement of keeping special data about the tree information. Metadata data structure was introduced to server this purpose. It can be distributed to all clients of B-tree in a number of ways. In our implementation, we just used a
3.2. System Components

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BtreeCreateTx()</td>
<td>Creates a transaction handle</td>
</tr>
<tr>
<td>BtreeRead(txn, n)</td>
<td>Reads object n within transaction txn</td>
</tr>
<tr>
<td>BtreeWrite(txn, n)</td>
<td>Writes object n within transaction txn</td>
</tr>
<tr>
<td>BtreeTxCommit(txn)</td>
<td>Commits transaction txn</td>
</tr>
<tr>
<td>BtreeTxAbort(txn)</td>
<td>Aborts transaction txn</td>
</tr>
<tr>
<td>BtreeEndTx(txn)</td>
<td>Ends transaction txn</td>
</tr>
</tbody>
</table>

Table 3.4: B-tree transactional API

A bunch of fixed server addresses that was passed to all clients as part of the configuration file. Basically, this metadata information is replicated to a bunch of servers to help achieve better start time, i.e., scalability. Other than the root node location, metadata also holds the list of participant servers upon which this tree is distributed. Once clients received metadata, clients may configure Sinfonia to use these servers as well for future transactions.

- **Bit Vector** data structure has been used to indicate which object slot is currently free for each memory node. So, there is a separate bit vector for each participant server in the B-tree. When creating a new tree node, clients first choose a server, found in metadata for the tree. Then it receives the bit vector for that server. Once bit vector is fetched, it allocates a new slot and puts the bit vectors in the write object list of transaction. When the transaction is executed, along with other changes, it writes back the bit vectors in the server as well.

**Transactional API**

B-tree operations like insert and delete in distributed environment may change the tree structure. Also, there can be multiple clients working on the same tree concurrently. To synchronize in these scenarios, we use transactions in B-tree operations. This also helps to group together multiple operations on multiple B-trees and perform everything atomically. As mentioned before, we use Sinfonia minitransaction to handle transactions in B-tree operations. B-tree transactional APIs are depicted in Table 3.4. Sinfonia minitransaction only deals with linear address space. However, for B-tree transactions use fixed length object data structures. Any B-tree operation has to be done with a context of transaction. So a B-tree transaction is created first, and that context is passed in all later operations. B-tree trans-
actions maintain a set of read and write objects. As the operation continues within that transaction, each object read is added to read set, and object that needs to be written are added to write set. So when the client needs to change a tree node, for example due to insertion of a key value pair, it puts the tree node to local write set fist. It does not write to the server directly.

When the transaction is committed, all the read objects are validated against the current copy in the server, and write objects are actually written. An important point to note, all the B-tree read operations that read from the server use a read only minitransaction. This kind of transaction is done in one round trip time. It can be even less, since we cached inner nodes. So not all read operations actually need to read from the server, most of the time they are read from local machine.

When B-tree transaction is actually committed, all the objects read so far during this transaction are validated for consistency. If there is any difference, transaction aborts, and read set gets the latest copy of objects, so that it can be used next time. However, for write set it is different. As the transaction continues, whenever it needs to write an object, i.e. adding a new key value pair or allocating a new node in bit vector, these objects are first added to write set. They are not directly written back to memory node server using a write only transaction. Once the transaction commit executed, its writes all the objects in the write list if all the read objects in the read list are consistent. Thus we do not lock the objects for the whole period of transaction operation. B-tree transaction only locks objects when it executes commit operation.

When B-tree transaction commit is called, internally, is transformed into a Sinfonia minitransaction. B-tree read objects are used to generate the compare transaction items in Sinfonia minitransaction. However, we only check the version number of the object ID of read objects. Since every time the object gets changed, the version numbers also incremented, and object IDs are unique, just by checking version number we can decide whether the object has been changed or not. Of course, this reduces network traffic and some memory copy operations. B-tree transaction only writes the objects if the read sets are valid. Write sets are transformed to Sinfonia write transaction element. We use a serialization scheme to convert these objects into transaction list to be used with Sinfonia linear address space. Thus B-tree provides transactional API to handle its operations.

\footnote{some are also read from cache}
3.2. System Components

Optimization

To improve performance, our implementation of B-tree client node caches the objects it reads from servers. Most of the B-tree operations require reading a lot of objects, and writing to few objects. For example, for inserting a new key value pair, clients start with reading root node first. Then it fetches the corresponding child node, until it gets the leaf node where the key pair should be put. Thus for this kind of operation, B-tree transaction will require reading many read and only a few write objects. Most the objects read already might be needed to read again for other transaction. For example, object like root node is always read first for all operations. Certainly, there are some objects that are more frequently fetched comparing to others. Caching these objects reduces network traffic, speeds up transaction execution and improves overall performance.

While objects are cached in client side, the same object can be changed by other clients in the original location at server. This makes the cached object to be out of date. But clients are not notified right away about the changes of the cached object. Clients also dont update them right away. Instead, clients update the replica objects lazily, i.e. when needed. Since version numbers are increased each time an object is changed in the server, transactions involving out of date cached objects will abort eventually. The client then discards current local copies. When these objects are read again, it fetched fresh copies from server, and caches them again.

Caching objects on B-tree clients does not actually puts much overhead. We only cache inner nodes, which are relatively smaller comparing to leaf nodes. Recall, our B-tree is a B+ tree, where key value pair is only kept at leaf nodes. Inner nodes only contain pointer to children nodes. Thus inner nodes are quite small, and hence size of our replicated data is also small.

3.2.3 Layered Video Storage over Distributed B-Tree

The main intuition behind our work was to use a scalable data structure to achieve high scalability for video streaming, and we designed our system by keeping video files in the distributed B-tree we developed. We could have tried other distributed data structures, but B-tree provided the most standard and intuitive features required for streaming video. For example, it is quite useful to have built-in support of dictionary operation like GetNext and GetPrev. These types to dictionary operations are very common for video streaming servers.

B-tree has been used in many projects as underlying storage architecture.
Previous works [10] have shown it can be used for video storage for adaptive streaming to enhance interactivity and quality scalability. Traditional video storage method is avi (.avi), QuickTime movie (.mov), flash video (.flv) etc. These file formats are good for distribution, are widely used for streaming, and also provide for random access. Streaming client can request to start from any random point, and server serves from that point. However, all of these storage formats assume the playback will be sequential. It can be from the beginning or from any random point. These formats don’t consider the fact that subsequent access may not be sequential, which is quite important for adaptive streaming servers. An adaptive server depending on the network condition, resource status and adaptation policy tries to adapt gracefully, and the response is usually by dropping parts of video during playback. In the above mentioned storage schemes, such partial access to video file from the upper layer eventually include neighboring data, which consumes high storage bandwidth.

QStream uses a layered video storage scheme called SPEG which allows storage bandwidth can be adapted in coordination with layered video. Layered video scheme is quite common for scalable video encoding, where video data in an encoded streaming is conceptually divided among multiple layers. A base layer can be decoded with minimum complexity. With the cost of additional complexity, extended layers are decoded.

In layered approach, QStream divides video files into multiple layers. In terms of implementation, given a .avi file, it generates many B-trees which conceptually represents this layer information. These trees can be stored in file, or can be stored in data structures distributed across cluster of computers. This is where our approach comes to play. Originally QStream used Berkeley db, which is internally stored as b-tree. All the B-trees QStream generated were stored in disk as a single file, which contained all the trees. These trees all together represent the layered approach. Basically there are three layers Root layer, Metadata layer and Data layers.

- **Root layer** provides the global information about the file. This includes number of audio and video streams, length and types of each stream etc. There is one metadata layer for each of the video stream mentioned in the root layer.

- **Metadata layer** contains information about each video frame; it does not contain the actual video data. The information is about how important the video frame is in terms of temporal scalability, and where to find spatial enhancement layers. Adaptive streaming servers
use this information to decide the importance of a frame, and adjust
guality level accordingly.

- **Data layer** contains actual audio and video data chunks.

When client requests for an encoded frame, server does not access data
layer directly. Rather it uses meta data layer to determine the importance
of this chunk of data. This importance can be based on a combination
of factors such as playback speed, network condition together with possibly
other adaptation policy. So, if the importance is below a certain level, servers
can skip that part of the video without accessing the data chunks. This
actually makes adaptation response faster, and enhances overall viewing
experience. Other file based approaches mentioned earlier don’t have any
direct support for such scalability.

### 3.3 Summary

In this chapter we described our proposed video streaming system using dis-
tributed B-tree. We first implemented a Sinfonia service, and a distributed
B-tree on top of Sinfonia. Both Sinfonia and B-tree are implemented on top
of QSF’s event model, and uses Paceline as the TCP transport library. Then
we modified QStreams streaming component, Qvid, to store video files over
our distributed B-tree instead of keeping the file on disk. We also modified it
to stream from there. Using this approach, video contents can be delivered
by a group of servers, rather being delivered by a single server. We found
this approach is highly scalable.
Chapter 4

System Analysis and Evaluation

In next few sections, we first evaluate our prototype video streaming system. Then we analyze why it scales well.

4.1 Evaluation

Video is stored in the distributed B-tree, and lookup is the most common form of B-tree operation that is performed when users requests video. So evaluated our streaming system based on the lookup performance of underlying B-tree.

4.1.1 Test Setup

Our test bed includes 4 powerful computers with AMD Phenom(tm) II X6 1055T Processor, each connected with giga-bit network connection. For comparison purpose, we first evaluated system performance when both B-tree server and client are running on single machine. By system performance, we mean overall throughput of the system. The tree was populated with 1000 random keys, and each client lookups up for each of them sequentially. The clients issue a lookup request to the B-tree, and as soon as the lookup operation returns, it issues another one. Thus we ensure that the system is always busy for maximum throughput. We increased both the number of servers and clients, and recorded overall throughput of the system in each case. Our version of B-tree uses 10 KB node size. Size of each key is 100 bytes, and value of each key is 1 KB. This bigger key is more representative for video streaming since Qvid uses video chunk size of 1 KB.

For our system evaluation we used all 4 computers. For the tests involving system size more than 4, we ran multiple servers and clients on same computer, but did not increase the number up to the point which overloads the system. So this test is limited to small number of servers and clients. However, since the test bed computers had 6 cores each, and our application...
4.1. Evaluation

takes the advantage of multiple cores using thread affinity, we believe this test is ideal for proving system scalability, within small range.

4.1.2 Results

For the single PC test, we found that system throughput increases up to a certain level, beyond which it does not increase anymore. In fact, when the server gets overloaded, average throughput drops which leads to reduced user experience in terms of watching video. Our implementation is based on QSF which takes the advantage of multi-core processors by assigning each thread to each processor. So the performance increased linearly up to a certain level. We also tested using a single core processor computer. Performance dropped sharply in this case with increased system size. This has been illustrated in Figure 4.1.

![Figure 4.1: B-tree clients and servers running on the same PC. B-tree is used as an example application to show what happens when system is overloaded. Since our implementation takes multi-core processor advantage, overall throughput increased to a certain point. Beyond this, throughput does not increase but average throughput drops.](image)

For the distributed B-tree, overall throughput increased almost linearly with the system size, as shown in Figure 4.2. Also average throughput at
each B-tree client did not drop with the system size.

![Distributed B-tree scales almost linearly](image)

Figure 4.2: Distributed B-tree scales almost linearly

### 4.2 Analysis

We now conceptually analyze why the system scales well.

**Minitransaction Primitive**

We believe use of Sinfonia handled the fundamental problem of achieving linear scalability. In traditional distributed systems, developers have to maintain state of the system using message passing protocol. As the system becomes large, this state becomes complex, and managing such state becomes error-prone. For many years distributed applications like cluster file system were considered hard [2]. But the authors of Sinfonia showed how this can be written in much less time and simpler code using Sinfonia. This is just an example how Sinfonia paradigm can make complex programming model simpler.

We think the fundamental contribution of Sinfonia is that it separated a complex component of traditional distributed programming, and represented
4.2. Analysis

it as a simpler primitive in Sinfonia, called minitransaction. The key design was to decouple as much as possible. Sinfonia does not impose any data type or structure requirements for the data being kept in memory nodes. Not having any dependency helps to decouple things. So more components can work independently, and collectively contribute to the overall system performance.

B-Tree Optimization

Our implementation of B-tree used Sinfonia as underlying data sharing service. Being a generic and simple service, Sinfonia did not provide some of the options that were specific to B-tree. We used a couple of approaches which jointly helped to improve performance.

Sinfonia did not provide caching for application nodes; however we implemented object caching in the client side to reduce network round trip time. Applications read all the inner nodes as it traverses. Since each tree object has unique Object ID, they were easily identifiable in the application node cache. We only cached inner nodes, which are quite small comparing to leaf node. Recall that our B-tree is actually a B+ tree which keeps key value pair only in the leaf nodes. Inner nodes only hold pointers. Thus modern computer with moderate memory can cache entire B-tree inner nodes in its memory. This means for most of the operations, application nodes can read the object from own memory. This saves network round trip time drastically.

Our implementation of B-tree used an optimistic transaction approach. Clients accessing B-tree operations first create a transaction handle, and all subsequent operations in that transaction use the handle as a reference. Once the client is done with all the B-tree operations, it calls transaction commit, which then creates Sinfonia minitransaction internally. That means, all the objects referenced in the transaction set are only locked during the commit operation. Such optimistic operations increase the option for decoupling, which in turn allows supporting more concurrent operations to be performed on the tree. This is mainly useful for uploading video content to the B-tree.

Datacenter Assumption

One of the original considerations behind this work is to improve server scalability running in clustered environment. Computers in the data center are usually connected with gigabit network connection, and network and
power outage are rare in such environment. Also data center computers usually have low latencies, and variation in latencies is very small too. We believe that the assumption of being used in ideal network condition keep things simple, and helps to achieve the goal of high scalability. If we run the same distributed B-tree where network conditions are poor, surely we will get different result.

**QSF Components**

We used QSF components thoroughly in our implementation. In fact whole B-tree and Sinfonia are implemented using various components of QSF. QSF has been used in research for quite few years now, and became quite mature over the years.

We used Paceline as our underlying network transport library. Paceline improves TCP performance by applying some application level techniques. By using Paceline, we not only got rid of the traditional downside of TCP like delay due to retransmission, but also found improved performance. Qvid is Qstreams streaming component which has been tuned to adapt based on a number of factors like network speed, available bandwidth, CPU power etc. Qvid uses a layered approach to store its video files for which B-tree structure is an ideal choice. Qvid uses Berkeley DB to store all the trees generated for each video into one file. We modified Qvid to store all the generated trees in our distributed B-tree and to serve from there. Using QSF event model, the implementation was easier than what it would be otherwise using multiple threads and synchronizing states among them.

### 4.3 Summary

In this chapter we evaluated our approach, and provided an in-depth analysis of why this approach works better. Our test results show that on demand video streaming using distributed B-tree can scale linearly, and can be potentially used with existing system to improve overall scalability.
Chapter 5

Conclusions and Future Works

5.1 Conclusions

In this thesis, we presented a novel approach of using distributed B-tree for streaming video, and showed that this can scale almost linearly with the system size. Our work was inspired by a number of other works on distributed data structure [2] [1], and extended an existing streaming system [15]. We implemented a basic Sinfonia service, on top of which we developed a distributed B-tree. We then modified QStreams streaming component to store video files over the distributed B-tree and to stream from there. We assumed this will be used only in a data center environment. A bunch of participating servers will be used to form the distributed B-tree. These servers allow streaming servers to keep video files over the data structure they provide and to stream from there. Streaming servers, when requested by clients, get video data from the B-tree distributed over the network, and send video to clients. The underlying Sinfonia service provides linear address space for application programmers to host data. It does not impose any specific requirements for the data type being hosted there. This kind of decoupling feature at lower level provided the foundation to achieve high scalability. In our B-tree implementation, we cached inner nodes at the client computers, which helped to reduce network round trip times. We evaluated our prototype system, and found that system throughput increases almost linearly when more servers are added.

5.2 Future Works

This thesis constitutes an initial step towards video streaming using scalable data structure as storage. A number works can be carried out further to our work.

We only tested our system in a prototype environment. It’s worth testing
5.2. Future Works

in a real world Internet video streaming traffic.

In our B-tree implementation we cached only inner nodes at the B-tree clients. Leaf nodes are usually bigger, and may take huge memory in B-tree clients if cached there. However, especially for video streaming these nodes can be cached as well. For video streaming, most of the operation is B-tree lookup, and also video is displayed sequentially. That means if a video frame is accessed from a leaf node, most likely subsequent access will be on the same node. Different caching strategy can be used. We did not have enough time to investigate further.

In our current implementation of B-tree, when new nodes are allocated, we choose a random server from the group of available servers. We could have applied a more intelligent approach here. For an actual deployed system, some servers may be more powerful, may have comparatively more memory or at least may be less busy comparing to others. Choosing those servers for new nodes would have improved overall performance. Also, there can be business specific policy set up for where the new tree nodes should go.

In fact, the policy for choosing new server may lead to new research possibilities. For example, by choosing new nodes appropriately, it is possible to form a peer to peer overlay network among the servers participating in distributed B-tree. The servers might be running on geographically distant data centers, but might be utilized to get the benefit of both P2P systems and client server model. Despite widespread popularity of P2P systems, there are still some issues that is blocking P2P system to be a valuable building block for robust and large scale Internet applications [4]. We believe its worth considering using distributed B-tree in conjunction with P2P systems which might resolve some of the current problems that P2P systems have.
Bibliography


Bibliography


