A Hybrid P2P Pre-Release Distribution Framework for Flash Crowd Avoidance in P2P Video on Demand Streaming

by

Stanley Kai Him Chiu

B.Sc., The University of British Columbia, 2005

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

(Vancouver)

July, 2008

Abstract

In recent years, the high maintenance cost of centralized video on demand systems has led to the development of peer to peer video on demand systems. These peer to peer systems help to remove the cost and bandwidth limitations of a centralized group of servers. In a peer to peer scenario, the publisher and a small set of peers who were published to must handle all video requests. If many peers request a video after it is released, the small set of peers with the video cached may not have enough bandwidth to satisfy all requests. This situation is known as a flash crowd. We propose a hybrid peer to peer framework that allows publishers to publish videos before release time. For marketing purposes, it is common for videos that are ready for distribution to be kept from being released until a preset release time. By distributing a video before the release time, more peers will have the video at release time, thus allowing more requests to be handled. A hybrid peer to peer encryption management system is used to prevent users from viewing videos before release time. In order to determine who to distribute a video to before users are allowed to view the video, we design a hybrid peer to peer subscription system. In this system, users may specify interest in sets of videos and are notified of new videos matching the interest so that retrieval may start. Finally, we modify an existing peer to peer video on demand framework to better handle concurrent streaming and downloading. Our experiments show that this framework can greatly increase a peer to peer streaming system's ability to handle flash crowd situations.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisor, Dr. Son T. Vuong, for his support and encouragement. I would also like to express my gratitude for my second reader, Dr. Charles Krasic, for his support and suggestions.

I would also like to thank all the members of the NIC lab, who made the lab a great environment to work in. Their comments and suggestions are greatly appreciated.

Finally, I express thanks to my family and my relatives who have supported me throughout the years.

Chapter 1

Introduction

1.1 Motivation

Video on demand systems have gained popularity in recent years. These systems allow users to watch any part of any video available on the system whenever they want. In the past, these systems were implemented by having a central server or group of servers provide videos for all other users of the system. However, this approach is limited by the high cost and bandwidth required to maintain the central servers.

The emergence of the peer to peer computing model promises to solve this problem by removing the need for a central server. This model has proven successful in solving file sharing problems [3, 5]. Many researchers have attempted to create peer to peer systems suitable for video on demand streaming [10, 20, 28]. In peer to peer systems, each peer acts as both a server and a client, allowing for the sharing of resources amongst peers without a centralized server.

Without an expensive server with large amounts of bandwidth, the quality and smoothness of the video playback is often limited by the limited upstream bandwidth available to peers on the internet. These issues are especially apparent during flash crowd situations, where a great number of users attempt to watch the same video or small set of videos in a short amount of time. This often occurs when a popular video is released. In peer to peer approaches, only

the publisher and the few peers published to have the video at release time. Since these few peers have limited bandwidth, it is unlikely and often impossible for these few peers to smoothly stream the video information to all peers who requested the video.

Commercial peer to peer video streaming services, such as Joost [4] and BabelGum [1], alleviate this problem by maintaining a set of servers to help distribute the video. However, like with a central server solution, this method may be costly to maintain.

In recent years, centralized schedule systems have been introduced to a few commercial systems, including Veoh [7] and BBC iPlayer [2]. These systems display schedules of videos for users and allow them to subscribe to them. Streaming quality may be improved in situations where only small numbers of users are subscribed to or watch the video after release. However, in the case of popular videos, large number of subscribed viewers may automatically attempt to download the video after release, increasing the demand on the senders. This may cause video startup delay and video quality to become worse.

In this thesis, we attempt to alleviate the problem of flash crowds without requiring the maintenance of a set of costly servers for video distribution. Popular videos, such as television series and movies, often have a preset release for marketing and advertising purposes, even if the video data itself is ready to be released. For example, a season of a television series that has been edited and is ready to be viewed will often be released slowly over the duration of a season. Movies that are ready for digital distribution are also delayed normally in order to synchronize with the distribution of physical media such as DVDs, which take time to produce and transport.

We propose allowing the distribution of videos before release time to alleviate flash crowd problems. To support this, the system must decide: (1) how to

prevent users from watching videos before release time, (2) who to distribute early releases of videos to, and (3) how to distribute video. We design a hybrid peer to peer encryption management system, hybrid peer to peer subscription system, and modify an existing peer to peer video on demand system, Bitvampire [20], to solve these problems. By using a hybrid peer to peer approach, the load on the central server can be kept to a very low level while supporting a large number of users in the system.

1.2 Thesis Contributions

This thesis attempts to solve bandwidth and flash crowd problems by allowing pre-release distribution of videos using information provided by the user about their video interests. This is achieved with the following contributions:

- Design a suitable hybrid peer to peer video subscription and notification system that allows users to: (1) view schedules, (2) subscribe to sets of videos, and (3) receive notification when a subscribed video is released.
- Design a hybrid peer to peer distributed encryption management system that allows publishers to publish encrypted videos with a set date to allow viewing. Decryption keys are automatically published onto the distributed system upon release time, allowing streaming peers to retrieve it and decrypt the video.
- Design a retrieval system that prioritizes requests from streaming peers over downloading peers to allow smoother streaming of videos and more efficient use of aggregate bandwidth for a video in the network.
- Design a system architecture for implementing the system and describe implementation details. We also implement a simple GUI prototype to demonstrate publisher and user interactions with the system.

- Evaluate the proposed systems and algorithms both theoretically and through simulations.

1.3 Thesis Organization

This thesis is divided into six chapters. Chapter 2 describes the background information for peer to peer video streaming technologies and details related works in the field. Chapter 3 presents the design for the proposed framework. In Chapter 4, we translate the design into an architecture and describe implementation details for implementing the architecture. We also implement a simple prototype with a graphical user interface (GUI) to show how a user interacts with the proposed system. Chapter 5 evaluates the framework both theoretically and through simulations. Lastly, we conclude the thesis and discuss possible research directions for further improvement of the framework in Chapter 6.

Chapter 2

Background Information and Related Works

This chapter provides some background information concerning video streaming and peer to peer technologies.

2.1 Background Information

2.1.1 Traditional Television

Video streaming was originally made popular by television. Television video streaming allows for a centralized authority to distribute multiple channels of video to users. Users may select a channel and view what the centralized channel administrators have chosen to be displayed at the time. This streaming of video information is achieved by multicasting the video data over certain preset frequencies either over the air or through a physical cable.

These systems offered the users little flexibility. The only choice the user may make is what channel he/she wants to watch. Over time, personal video recorders (PVRs) emerged. These new systems offer the user greater flexibility. While the channel administrators get to control at what time an episode is available, users can now see a schedule beforehand and mark videos for recording. By recording a video, a user is able to watch it at any time after it has been

released. Pausing, fast forwarding, and jumping to specific parts of videos is also possible now since the entire video is stored.

2.1.2 Internet Video Distribution

Because traditional television multicasting methods required specific frequencies in the airwaves or specialized cables to be reserved for transmitting video information, the number of video streaming companies in an area is often limited. These same limitations also make it hard for companies to operate world wide. With the emergence of the internet, a new distribution method capable of transferring videos became available. The internet is a global network in which any connected computer can communicate with any other connected computer. With this new communications method, information such as videos can now be transferred to anyone in the world as simple files.

2.1.3 Internet Video Streaming

Centralized video streaming services [6, 8] gained popularity and demonstrated the feasibility of distributing video over the internet. A central server or a set of centralized servers stored the videos, allowing users to access and stream them directly. For the consumers, these systems offer great flexibility. Unlike traditional television streaming, it is possible to watch videos that have been released in the past without having recorded it. Watching any video ever released onto the central server and skipping to any part of the video are often allowed on such systems. Services offering these capabilities are commonly known as video on demand services.

2.1.4 Peer to Peer Streaming Systems

The central server approaches required large amounts of bandwidth, storage, and computing power. These requirements meant that it is often expensive to maintain a popular video on demand service. As hardware technologies improved, the bandwidth available to consumers also increased. This allowed another video streaming technology, peer to peer video streaming, to emerge. By allowing the viewers, also known as peers in peer to peer systems, to use their upstream bandwidth to share videos with each other, the central server can be eliminated. However, without the costly central servers, these systems are susceptible to problems such as flash crowds, in which many peers want to watch a video in a short amount of time. This is because bandwidth capacity is often asymmetric, with each peer having more download bandwidth than upload bandwidth. In this situation, peers can easily consume more bandwidth than they contribute.

2.2 Related Works

This section describes pervious works attempting to solve video streaming distribution problems.

2.2.1 Proxy-based Systems

Early video on demand systems attempted to alleviate server demand by using proxies [9, 11, 22, 25]. Two approaches for implementing these systems are (1) prefix-based caching [22, 25] and (2) segment-based caching [9, 11]. Proxies in prefix-based caching cache the beginning parts of popular videos. When viewers request these videos, the proxies will send the cached frames while requests are concurrently sent to request the central server to retrieve remaining parts of

the video. These proxies help to decrease start up latency for viewing videos and also may decrease the load on the central server. Segment-based caching systems split a video into parts called segments. These segments are distributed to various proxies, which can help serve viewers, decrease the load on the central server, and decrease startup latency.

2.2.2 Peer to Peer Multicast

Peer to peer video multicasting is another category of video streaming research [10, 12–14, 27]. These systems are similar to a peer to peer version of traditional television, allowing senders to stream specific video streams to large numbers of viewers. One such system, SpreadIt [14], operates by creating and maintaining a single tree. Video is streamed by having nodes in the tree relay video streams to its children nodes. However, this approach does not handle node failures very well, especially if the failure is at the source. Narada [12, 13], NICE [10], and ZIGZAG [27] focus on multiple sender and multiple receiver streaming applications. In these systems, trees are formed when a sender wants to stream video to multiple receivers.

2.2.3 Peer to Peer Video on Demand

Peer to peer video on demand systems are yet another category of research [20, 28]. In these systems, viewers have greater flexibility. Viewers are generally allowed to start watching any part of any video existing on the system. Mohamed et al [28] proposes a system in which a set of relatively stable peers, named seed peers or super peers, store all videos initially. As peers stream videos from seed peers, they cache the video. Caching the video allows the peers to help the super peers in further distributing videos. However, these systems are vulnerable to flash crowds. In this situation, only the seed peers have the video,

making it difficult to stream to all peers at a rate that allows smooth playback of the video.

BitVampire [20] improves peer to peer video on demand streaming by allowing viewers to aggregate bandwidth from multiple senders and splitting videos into smaller parts called segments.

PALS [23] uses layered coding for peer to peer streaming. When congestion occurs, PALS may still be able to display a lower quality version of the video utilizing the bandwidth still available from senders.

2.2.4 Commercial Video Scheduling Systems

In recent years, commercial peer to peer video on demand systems [2, 4, 7] have appeared. Very recently, scheduling systems have been introduced to Veoh [7] and BBC iPlayer [2]. These systems allow retrieval of schedules and allow users to subscribe to video series or centralized video lists. Clients periodically poll the server, and thus are able to automatically retrieve videos upon release.

Chapter 3

System Design

This chapter presents the design of the video on demand system. By allowing the publication and distribution of videos before release time, we can increase the video quality and increase the smoothness of the video playback.

3.1 Introduction

As described in Chapter 1, peer to peer video streaming technologies are vulnerable to flash crowds. Commercial products generally attempt to solve this problem by setting up large amounts of servers with high bandwidth, but maintenance of the servers can be costly. We propose another solution. By allowing publishers to publish and distribute videos before release time, we increase the duration of time the video can propagate through the peer to peer network before any peer watches the video. As it propagates through the network to subscribers, it is cached by these subscribed peers, thus increasing the availability of the video. Having cached the video or parts of the video, this set of subscribers will need to retrieve less data or no data at all when viewing of the video occurs after the video release time.

In order for this framework to improve video streaming performance, videos must be published before release time. Fortunately, this seems to be the case for many types of videos, such as television series and movies. For television series, there is normally a set time for each episode to be displayed during a

season, even if the series is ready beforehand. There also often needs to be at least a few hours of time between having the video ready and the airing of the video on television, due to physical preparation and transportation delays. Since publishers normally synchronize the release time for various distribution methods, this means that there is often time before the release time in which the movie is ready and may be distributed in our framework. For movies, digital release dates are also synchronized with the distribution of physical mediums such as DVDs. The creation and transportation of physical media often takes time, which may be used for pre-release distribution of the media in our proposed framework.

Our design consists of three major parts, which are (1) the subscription system, (2) the encryption management system, and (3) the peer to peer streaming system. The subscription system allows peers to specify the videos they are interested in and automatically retrieve these videos. By allowing subscribed peers to automatically retrieve videos, published videos will automatically be propagated through the peer to peer network even before release time, increasing availability. To prevent peers from watching distributed videos before the release time is reached, the encryption management system does not distribute decryption keys for encrypted videos before release time. The actual distribution of peer to peer videos is based on Bitvampire [20]. However, by allowing peers to automatically download videos, situations where downloading peers and streaming peers compete for bandwidth might occur. We modify the algorithms described in Bitvampire to handle these situations by prioritizing streaming peers.

3.2 System Operation

When a peer publishes a video, information about the video and decryption information is sent to a light weight central server. The publishing peer may then start distributing an encrypted version of the video data onto the peer to peer video on demand system. After receiving the publication information, the central server first stores the video information locally. The central server now updates the video schedule to include the published video and stores this information onto the hybrid peer to peer storage system described in Section 3.3. When the release time for the video is reached, the central server publishes the decryption information onto the hybrid peer to peer storage system.

Schedule information is accessible by all peers in the network through the hybrid peer to peer storage system. A peer may subscribe to a set of videos or watch any released video. When a peer watches a video, a modified version of the Bitvampire [20] streaming algorithm is used to retrieve the video. Before displaying the video, which is only allowed after release time, the decryption key is retrieved from the hybrid peer to peer system and used to decrypt the video.

By periodically polling the hybrid peer to peer storage system, peers have access to up to date schedules. When a new video in the schedule matches a subscription, the peer will automatically retrieve the video using a modified version of the Bitvampire streaming algorithm described in Section 3.6.

3.3 Hybrid Peer to Peer Storage System

We first describe the storage system used in this thesis. Hybrid peer to peer frameworks [21, 28, 29] use a mix of different centralized or peer to peer architectures to improve performance. B. T. Loo et al. [21] investigate the performance

of various hybrid peer to peer infrastructures. These hybrid peer to peer architectures often increase performance by choosing between various algorithms such as searching through flooding or searching through structured peer to peer searches dynamically depending on query. By supporting multiple infrastructures, the best features of different algorithms may be combined.

We propose a simple hybrid peer to peer storage system which is easy to implement and is suitable for our system. Information stored in our system, such as decryption information, should never be lost. In our system, we use a central server to create schedules and control publishing. This allows the administrators of the central server to have local access to information about all videos, and allows them to control the publication of videos. The central server also manages decryption keys before publication to ensure normal untrusted peers do not get access to this key before release. Since a central server already exists, it is logical to use the central server for retrieval of this information. However, we do not want to cause great processor and bandwidth load on the central server. Therefore, we combine a peer to peer infrastructure with the central server for distribution of information, using the central server only as a backup of information. When information is lost in the peer to peer infrastructure, a request is sent to the central server. The central server not only responds to this request, but also republishes this information onto the peer to peer infrastructure. By using this hybrid architecture, we hope to achieve greater stability by restoring the values on the distributed system when a value is lost, and by allowing viewing to work using the distributed system even when the central server is down. We also hope to decrease both the bandwidth and processor requirements of the central server by making peer streaming and scheduling requests use the peer to peer system unless problems occur.

Distributed hash tables (DHTs) [24, 26] are well suited to become the peer

to peer infrastructure for our system. DHTs allow the storage of $< key, value >$ pairs and allows lookup of a *value* by supplying a *key*. DHTs have the advantage of having fast lookups, being scalable, and being fault tolerant. One main disadvantage of DHTs is that queries have to be exact key lookups. Complex queries such as wildcards are hard to support over DHTs. Fortunately, all queries in this thesis can be designed to be lookups of exact keys without any lost of features.

3.4 Video Subscription

A video subscription system allows users to specify interest in sets of videos, which is referred to as subscription. It also allows the client program to automatically detect when a video matching a subscription is published. This allows videos to be downloaded before viewing of the video occurs. It alleviates the flash crowd problem by propagating the video throughout the peer to peer network as subscribers cache downloaded videos, and by allowing subscribers to have time to retrieve the video data before streaming.

3.4.1 Subscription Granularity

In order to allow simple and intuitive specification of sets of videos, a method of specifying subscriptions needs to exist. Personal video recorder systems have been popular in recent years. Such systems normally allow users to see the schedule before the release date. Users are then able to either select a specific video to record, or select a recurring time slot to record. Advanced PVR systems can also allow the automatic recording of a television series.

Based on this popular system, this thesis proposes to allow 3 types of subscriptions: specific video, series, and channel. Subscribing to a specific video allows the video to be retrieved before viewing. This method of subscription is

especially helpful for viewing of long high quality videos, such as movies. The second type of subscription, series subscription, allows subscription to the set of videos belonging to a specific series. This is useful for television series with multiple episodes. By using this subscription type, the user does not have to wait for an episode to be announced before subscribing. Instead, the user only needs to select a series to be recorded and new episodes will be automatically queued for retrieval. The third type of subscription, channels, allows the subscription to sets of videos belonging to a specific channel. A channel contains sets of videos with features users might be interested in. For example, short humorous videos or movies produced by a specific producer can be grouped into one channel, allowing users to simply subscribe to that to automatically retrieve those videos.

3.4.2 Subscription Information Storage

In order to support the subscription system described above, video information must first be given by the publisher to the central server. When a publisher publishes a video, the publisher must specific a time for when the video will be released. This allows schedules to be created and updated. The publisher must also specify the channel, series, episode number, and video name of the video. If the channel, series, or episode number are not specified, it is automatically set to a special keyword *notapplicable*. The storage of channel and series numbers allows users who have subscribed to them to be notified properly, while the specification of episode number allows the retrieval of only one copy of the video should multiple publishers publish different encodings of the same episode on different channels. The name and episode numbers are also information the user is likely to find useful when searching for a video. After receiving the information, the central server assigns the video a unique video identifier (VID)

Name	Description
Video Identifier	Unique identifier for the video.
Name	The name of the video.
Release time	The time after which users may watch the video.
Series name	The series this video belongs to.
Episode name	The episode this video belongs to.
Channel name	The channel this video belongs to.

Table 3.1: Video Information

in order to distinguish between different videos with the same name. Table 3.1 shows the schema that the central server stores to allow this system to operate.

With this information, the central server can maintain a schedule containing the information about videos and when they will be released. This is basically a list of the video information for a set of videos, which will be referred to as the schedule in the rest of this thesis.

We can now use the hybrid peer to peer storage system to store the schedule. By putting this information onto the hybrid system, peers may periodically poll to find new schedules. In order to store the schedule on our hybrid peer to peer storage system, we must map the information to a $< key, value >$ pair. Since there is currently only one large schedule containing all videos, any universally known key word, such as "schedule" may be used as the *key*. The *value* for this *key* will be the schedule for all videos in the system.

However, each key is often managed by only one node or a small set of nodes in DHT algorithms [24, 26]. With one node managing the schedule information for all videos and handling all user requests for the schedule, the process and bandwidth load on this peer will be very high. Reliability issues might also become a problem, since peers on a DHT are often unreliable. Even though the hybrid peer to peer approach allows the central server to be used as a backup, the central server will spend great amounts of bandwidth sending the full schedule to a new peer every time a node handling the schedule leaves the network.

Key	Value	Description
"Schedule" + s	$schedule_s$	DHT pair allowing storage of a schedule containing information for videos between s and $s+d$.

Table 3.2: Storage Schema for Schedule Information

Therefore, we optimize the design by splitting the schedule with information about all videos into schedules with information about all videos in a certain time range. The time range, r, controls the size of the schedules. If r is set to a very high value, certain peers on the DHT may have to handle great amounts of requests. If r is set to a very low value, in order to get the schedule of videos in a certain time period, such as a day, many peers might need to be contacted since each peer is only responsible for a very short time range.

To map this to a $< key, value >$, we can simply concatenate "schedule" with the start time, s, for the time range, r, of the schedule. We will refer to a schedule storing information for videos between s and $s+d$ as $schedule_s$. The value of r and one value of s, such as 0, must be globally known so users may map any time to a the correct schedule. For example, if it is known that 0 is a valid s and we want find the schedule containing the time 100005 to a schedule, we may simply search the DHT using the key "schedule" concatenated with the integer value of $floor(100005/r) * r$ to find the appropriate schedule. Table 3.2 shows the $< key, value >$ pair stored onto the DHT.

3.4.3 User Notification System

Automatic retrieval of videos matching subscriptions by users allows videos to propagate through the network. One simple approach is to have users poll the hybrid peer to peer storage system periodically for updates to all schedules. Periodically matching updated videos in the schedules to a locally stored list of

subscriptions allows a list of videos matching subscriptions to be stored. The videos on this list may then be automatically retrieved.

Unfortunately, this simple approach is infeasible in large networks since peers must retrieve all schedules periodically, which will greatly increase the overhead introduced by the subscription system. To solve this, we propose that users only poll and retrieve schedules for a range of time, *pollrange*, before and after the current time. This is suitable for our system since users will often only need schedules of recently released and upcoming videos for normal operation of the framework. Only those users who specifically want to see older videos will have use for very old schedules. This time range is the period where videos just released or soon to be released are located, and will be the time period that affects flash crowds most. A high value for *pollrange* allows the automatic retrieval of videos with release times further in the future, while a low value for *pollrange* decreases the amount of overhead introduced by the system.

This optimization greatly decreases the overhead and feasibility of the system, but introduces a new problem. If a user has not run the program in a time period greater than *pollrange*, schedules for the time period before $currenttime - pollrange$ will not be retrieved. Without knowledge of the videos released in that period, videos that match the user subscriptions might not be found. As previously mentioned, retrieving schedules for every time period since the start of the system to retrieve information for all videos requires large amounts of bandwidth and processor overhead and is likely infeasible.

Recall that this thesis supports three granularities for subscribing to videos, which are specific video subscriptions, series subscriptions, and channel subscriptions. Specific video selections are unique in that a user must have found it on a schedule, which means the video information is already known from the schedule. Therefore, users may not be notified of subscribed videos only for

Key	Value	Description
"channel" + *channelname*	*List < videos >*	DHT pair allowing storage and retrieval of a list of videos in channel *channelname*.
"series" + *seriesname*	*List < videos >*	DHT pair allowing storage and retrieval of a list of videos in series *seriesname*.

Table 3.3: Storage Schema for Video Information of Channels and Series

channel and series subscriptions. With the hybrid peer to peer system already available, we propose simply putting lists of video information for channels and series directly onto the storage system. We map these lists of videos to the storage system by generating unique keys, which are *"channel"* + *channelname* for channels, are *"series"* + *seriesname* for series. Table 3.3 shows the new information stored by the storage system.

With this information available, users may simply retrieve updates of video information for channels and series they have subscribed to when they start the program. There is no need to periodically poll for updates to these lists, since upcoming and current schedules are periodically retrieved.

Program 3.1 illustrates the pseudocode for this algorithm. The program first retrieves a list of videos for series the user has subscribed to. The list of videos matching series subscriptions is then added to the *subscribedVideos* list. This is repeated for channels. Periodically, schedules for the time period between $currentTime - pollRange$ and $currentTime + pollRange$ are updated. The videos in these schedules are then matched to the subscriptions, and matching videos are added to the *subscribedVideo* list.

Program 3.1 Algorithm for retrieving schedules and finding subscribed videos.

```
// @param pollRange    The number of time units before and after
//                     the current time to poll for
// @param pollRate     The number of time units between polls for
//                     updated schedules
// @param seriesSubscriptions    List of series subscriptions
// @param channelSubscriptions   List of channel subscriptions
// @param videoSubscriptions     List of video subscriptions
// @param subscribedVideos       List of videos matching
//                               subscriptions
public void receivedBlockEncryptionHandler(long pollRange, long
    pollRate, List seriesSubscriptions, List channelSubscriptions,
    List videoSubscriptions, List subscribedVideos) {
  List seriesVideoList = getSeriesVideoList(seriesSubscriptions);
  updateSubscribedSeriesVideos(subscribedVideos, seriesVideoList,
      seriesSubscriptions);

  List channelVideoList = getChannelVideoList(channelSubscriptions)
      ;
  updateSubscribedChannelVideos(subscribedVideos, channelVideoList,
      channelSubscriptions);

  while (!quit) {
    long startTime = currentTime - pollRange;
    long endTime = currentTime + pollRange;
    List schedules = getSchedules(startTime, endTime);

    List matchedVideos = findSubscribedVideos(schedules,
        seriesSubscriptions, channelSubscriptions,
        videoSubscriptions);
    subscribedVideos.add(matchedVideos);
    Thread.wait(pollRate);
  }
}
```

3.5 Encryption Management System

Without encrypting videos, early distribution of videos allows users to view videos before release time. This is obviously undesirable behaviour and is likely unacceptable to publishers. In order to prevent viewing of videos before release time, we introduce a simple encryption management system.

3.5.1 Encryption

When a video is published, we encrypt the video before distributing it to prevent viewing of the video. Encryption allows the transformation of the video data into normally unusable data. However, with the correct decryption key, a decryption algorithm can be used to transform this unusable data back into the original video data. Our design allows the use of any encryption algorithm that uses only a decryption key for decryption.

In Bitvampire [20], videos are split into smaller parts, segments, before publication occurs. These segments are further split into smaller pieces, blocks. By choosing a random key and an encryption method, the publisher may encrypt these small blocks before distribution. Users who retrieve these encrypted blocks will not be able to watch it. Since these blocks are very small, it generally does not affect the startup delay of the video stream.

3.5.2 Encryption Information Storage

Normal operation of the system requires that users be able to watch the video after release time. In order to support this, we propose that the publisher sends the decryption key and the encryption method name to the central server during publication. The central server does not distribute this information to anyone before release time, thus ensuring that viewing of the video does not occur. When the release time is reached, the decryption key may be put onto

Key	Value	Description
"key" + VID	decryption key	DHT pair allowing storage and retrieval of the decryption key for a video with unique identifier VID.
"method" + VID	encryption method name	DHT pair allowing storage and retrieval of the encryption method name for a video with unique identifier VID.

Table 3.4: Storage Schema for Encryption Information

the hybrid peer to peer storage system for users to retrieve. Table 3.4 shows how this information may be mapped into $< key, value >$ pairs using the unique video identifier VID.

3.5.3 Encryption Information Retrieval

When a user chooses to watch a video after release time, the user first retrieves the decryption key and encryption from the storage system. As each block arrives through the underlying peer to peer streaming application, it can simply be decrypted before being displayed by the peer to peer streaming application.

Using this approach, peers will decrypt the video every time they view the video. If the encryption method is very demanding on processing power, an optimization might be useful to shift when the video is decrypted and decrease the number of times a video is decrypted. This allows for smoother playback of the video even by users who have older and slower computers.

We optimize the algorithm by having users store the decrypted version of the video whenever a video is downloaded or viewed after release time. When we store pieces of the movies, segments, we also store whether or not the segment is encrypted.

When senders distribute segments to receivers, the sender must specify

whether or not the segment is encrypted. Before release time, it will always be encrypted. However, after release time, if it is encrypted, the user may decrypt it before storage. By using this optimization, copies of the videos on the network will become decrypted versions over time and will propagate as decrypted versions. Each viewer will still only need to decrypt each block at most once, with some users viewing after release time not having to decrypt at all.

Program 3.2 presents the pseudocode for the retrieval of blocks. After the retrieval of a block, the user first checks to see if the video block is encrypted. If it is not, the block may simply be stored as a decrypted block. If it is encrypted and the video has not been released, the user stores the encrypted version of the block. If it is encrypted and the video has been released, the user retrieves the decryption information from the hybrid peer to peer storage system and uses it to decrypt the block. This block is then stored in its decrypted form.

Program 3.2 Algorithm for handling the decryption of blocks.

```
// @param releaseTime The release time of the video
// @param vid         The unique identifier for the video
// @param block       The block of video data
// @param encrypted   True if the transferred block is encrypted,
//                    else false
public void receivedBlockEncryptionHandler(long releaseTime, int
    vid, Block block, boolean encrypted) {
  long currentTime = getCurrentTime();
  if (!encrypted) {
    storeDecryptedBlock(vid, block);
  } else {
    if (currentTime < releaseTime) {
      storeEncryptedBlock(vid, block);
    } else {
      byte[] decryptionKey = getDecryptionKey(vid);
      String decryptionMethod = getDecryptionMethod(vid);
      Block decryptedBlock = decrypt(block, decryptionKey,
          decryptionMethod);
      storeDecryptedBlock(vid, decryptedBlock);
    }
  }
}
```

3.6 Video Retrieval

With the help of the subscription and encryption systems, videos may now be distributed onto the peer to peer video on demand streaming network. This section investigates optimizations to Bitvampire's peer to peer streaming algorithm for allowing downloading requests and streaming requests to coexist without affecting streaming quality.

3.6.1 Video Downloading

There are now two kinds of data transfers, video streams and video downloads. Video streams occur when a user plays a video, while video downloads occur when videos match user subscriptions. The video streams are handled by the peer to peer streaming algorithm. The download transfers can use traditional peer to peer file transferring algorithms or peer to peer streaming algorithms.

Peer to peer file transferring algorithms generally work by sending blocks out of order. If a user decides to watch a subscribed video that has not completed downloading, out of order blocks already cached may not help in lowering the bandwidth requirements for smooth playback of the video.

For example, imagine a situation where a peer has downloaded the 10 minutes of a 20 minute video with a bitrate of 100KBps. The peer now decides to start watching the video. If the peer has downloaded the last 10 minutes of the video, the data downloaded does not help in streaming the first 10 minutes of the video. The peer requires a constant 100KBps for smooth playback. If the peer has downloaded the first 10 minutes, the user only requires $(20minutes - 10minutes) * 100KBps/20minutes = 50KBps$ for smooth playback of the video. This is because the peer already has the video information required to display the first 10 minutes, and can use the 20 minutes of playback time to download the last 10 minutes while maintaining smooth video playback.

As illustrated by the example, in order video transfers are preferable for our design. Therefore, our design makes use of peer to peer streaming algorithms for both streaming and downloading.

3.6.2 Stream Prioritization

With both downloading and streaming transfers occurring at the same time, downloads may compete with video streams for bandwidth when there is not enough bandwidth in the system. Slow video transfer in video download situations simply means the download will go slower. In the case of video streaming, slow video transfer can cause video playback to stutter and pause. We therefore optimize the system to prioritize video streams.

Receiver Side Prioritization

On the receiver side, if a user chooses to stream a video while downloading subscribed videos, the receiver may not have enough download bandwidth to support both. As previously described, the user will likely prefer a smooth video stream over faster video downloads. Thus, we modify the Bitvampire algorithm to prioritize video streaming.

Videos in Bitvampire [20] are split into smaller pieces called segments. Retrieval of videos is achieved through location of peers with a specific segment and requesting the segment from these peers. Peers in Bitvampire keep track of their download bandwidth. If the incoming rate from sender s decreases for a period T or is notified by sender s to switch, it will switch the sender s by sending its unfinished part of the request to another peer with the segment cached.

By keeping track of whether a video transfer is a stream or a download, the retrieval algorithm can be improved to prioritize streaming. Using the algorithm

described by Bitvampire to monitor download speeds, we can determine whether the video stream is being retrieved at the requested speed. If it is, then the downloading of other videos must not be affecting the speed of the stream, thus requiring nothing to be done. However, if the speed of the stream transfer is lower than the speed requested from the suppliers, the downloading of other videos may be affecting the retrieval speed of the stream, possibly causing pauses in the video playback.

We modify the Bitvampire supplier switching algorithm to solve this issue. When it is detected that the incoming rate for the video stream transfers are lower than they should be for a period T, instead of switching senders, any video transfers for downloading videos are first stopped. The receiver then waits another period T, and if the incoming rate is still too low, the supplier is switched. After the user finishes watching the video stream, video downloading is resumed.

Supplier Side Prioritization

In Bitvampire, each peer sets the maximum upload bandwidth, $band_{max}$, that it can handle. When peers retrieve videos, they retrieve a list of peers with the segment they want to retrieve. The scheduling algorithm in Bitvampire schedules amongst peers with the segment using information about each peer's maximum upload bandwidth, $band_{max}$ and available bandwidth, $band_{avail}$. Without modification of this algorithm, the supplier would treat downloads and streams of videos the same way. Video playback smoothness is directly affected by the bandwidth of video streams, but not by the bandwidth of video downloads. We therefore modify this algorithm to prioritize video streams.

We allow peers retrieving videos to send a flag to video suppliers, specifying whether the video transfer is a download or stream. In order to allow suppliers to replace downloaders with streamers when it is unable to serve a stream request,

each supplier keeps track of the bandwidth reserved by each downloader. A supplier also maintains information about the amount of bandwidth reserved for downloads, $band_{downloads}$, along with the previously mentioned $band_{max}$ and $band_{avail}$.

When a peer attempts to stream a video, it tries the original algorithm by finding suppliers with the segment, and reserving bandwidth from suppliers using knowledge of $band_{max}$ and $band_{avail}$ for each supplier. If it is unable to find enough suppliers to reserve the required bandwidth, the peer now assumes suppliers have $band_{avail} = band_{avail} + band_{downloads}$ and reserves bandwidth amongst suppliers using this new information. Program 3.3 illustrates this algorithm. When there is not enough bandwidth for smooth streaming, the peer starts requesting that suppliers free bandwidth by stopping transfers to downloaders. This repeats until there is enough bandwidth to stream the video smoothly or the list of suppliers is depleted.

Program 3.4 presents how a supplier can free bandwidth for streaming requestors. The supplier stops transfers to downloaders one by one until it has saved the requested amount of bandwidth. The supplier also updates the available bandwidth, bandwidth reserved for downloads, and the list of downloaders.

Program 3.3 Algorithm for requesting the stream of a video segment.

```
// @param vid              The unique identifier for the video
// @param segmentNum       The segment of the video that is being
//                         requested
// @param requiredBandwidth The bandwidth required to stream
//                          smoothly
public void streamSegment(int vid, int segmentNum, int
    requiredBandwidth) {
  Supplier [] suppliers = findSuppliers(vid, segmentNum);

  int totalAvailBandwidth = 0;
  for (int i = 0; i < suppliers.size(); i++) {
    totalAvailBandwidth += supplier[i].availBandwidth;
  }

  if (totalAvailBandwidth > requiredBandwidth) {
    requestSegment(vid, segmentNum, requiredBandwidth, suppliers);
  } else {
    for (int i = 0; i < suppliers.size(); i++) {
      int missingBandwidth = requiredBandwidth -
          totalAvailBandwidth;

      if (missingBandwidth <= 0) {
        streamSegment(vid, segmentNum, requiredBandwidth);
        return;
      }

      if (missingBandwidth < supplier[i].reservedDownloadBandwidth)
          {
        supplier[i].freeFromDownloaders(missingBandwidth);
        totalAvailBandwidth += missingBandwidth;
      } else {
        supplier[i].freeFromDownloaders(supplier[i].
            reservedDownloadBandwidth);
        totalAvailBandwidth += supplier[i].
            reservedDownloadBandwidth);
      }
    }

    streamSegment(vid, segmentNum, requiredBandwidth);
  }
}
```

Program 3.4 Algorithm for freeing bandwidth by stopping transfers to downloaders.

```
// @param requiredBandwidth The extra bandwidth required by the
//                          streaming peer.
// @param downloaders       The list of downloaders being
//                          transferred to.
// @param numDownloaders    The number of downloaders
public void freeFromDownloaders(int requiredBandwidth, List
    downloaders, int numDownloaders) {
  for (int i = 0; i < numDownloaders; i++) {
    if (requiredBandwidth <= 0) {
      return;
    }

    reservedDownloadBandwidth -= downloaders.get(i).
        reservedBandwidth;
    availBandwidth += downloaders.get(i).reservedBandwidth;
    requiredBandwidth -= downloaders[i].reservedBandwidth;
    stopTransfer(downloaders.get(i));
    downloaders.remove(i);
  }
}
```

Chapter 4

System Implementation

This chapter describes the implementation details of the system. In section 4.1, we describe the general architecture of the system. Details for implementing this architecture using existing technologies are described in section 4.2.

4.1 Architecture

4.1.1 Framework

Our system is composed of multiple layers based on a simplified version of the RTG framework presented in BitVampire [20], which is inspired by the JXTA framework [16]. The RTG framework allows the decoupling of each layer, thus separating the system logic and allowing algorithms to be replaced with relative ease. Table 4.1 describes each layer and lists components that belong to each layer.

4.1.2 Component Details

At the highest level of this architecture, interaction with the user is supported through the use of a graphical user interface (*GUI*). Initiation and interaction with the *GUI* triggers the *Application Logic* component, which processes the user's requests.

In order to handle these requests, the *Application Logic* component calls the

Layer	Description	Components
Application	This layer contains application specific components, which often includes the GUI.	GUI, Application Logic
Abstraction	This layer provides a simple interface that allows the application layer to access the service layer. Service layer modules can thus be swapped easily without changes to the application layer.	Controller
Service	The service layer contains common services. Different algorithms to provide a service may be implemented here to affect the performance of the system.	Encryption, Subscription, Automatic retrieval, Peer to peer streaming
Core	This layer contains the high level peer to peer communication models.	Hybrid peer to peer storage system
Communication	This layer contains the low level peer to peer communication details.	DHT, Sockets

Table 4.1: Framework Layers

abstraction layer *Controller* component. This component is normally only an adaptor, containing no real logic. By calling the right service layer component, the controller can delegate the logical computations to the correct service layer components.

In this system, there are four main service components, which are *Encryption*, *Subscription*, *Automatic Retrieval*, and *Peer to Peer Streaming*. Encryption and decryption of video data is handled by the *Encryption* component. The *Subscription* component handles user subscriptions, detection of videos matching user subscriptions, and updating of schedules. When a video matching a user's subscriptions is detected, it is handled by the *Automatic Retrieval* component. The *Automatic Retrieval* component maintains a list of videos that match subscriptions and its download status. The role of this component is to provide the *GUI* with the a list of downloading videos, as well as automatically starting and stopping downloads to control the number of videos being downloaded concurrently. The last major component, *Peer to Peer Streaming*, allows for the streaming of video data. In this system, this is provided by Bitvampire.

Communication with other peers is required for the operation of many of these components. The core layer abstracts this in order to facilitate the sharing and reuse of the core layer components by higher levels. In this system, the major component in this layer is the *Hybrid Peer to Peer* component, which allows data to be retrieved from the DHT if information is on the DHT, or the central server if the DHT has lost the information requested.

The DHT and central server communication is supported by the final layer, communication. The *DHT* component allows the storage of $< key, value >$ pairs on the network, while the *Central Server* component allows communication between the central server and the client.

4.2 Implementation Details

4.2.1 Related Software

This prototype makes use of 2 other software frameworks: BitVampire [20] and FreePastry [15]. Bitvampire provides the peer to peer streaming algorithms while FreePastry provides the distributed hash table algorithms. Since both FreePastry and BitVampire are built using the Java programming language, Java is also used to implement this prototype.

4.2.2 System Flow

This section details the flow of high level requests with which a user may interact with the system. Viewing of videos is not shown here because it is handled by the underlying peer to peer video streaming architecture, Bitvampire.

Publication

Figure 4.1 shows the UML sequence diagram for the publication of a video. When a user publishes a video using the GUI, the logic in the application layer *GUI* classes checks the input information for errors. If there are no errors, the central server is informed of the publication through the communication layer *CentralServerHandler* class. As part of the core layer, this class handles the communication between the central server and the client. A response containing a unique video identifier for the video is expected from the central server, and is passed back to the application logic in the *GUI* when it is received. To publish the actual video data, the abstraction layer *Controller* class is now called in order to gain access to a service. The *Controller* class passes the request to the appropriate service, *MediaService*. *MediaService* is part of the Bitvampire code and provides peer to peer streaming services to the applicaton. One of its

functions, *publish*, is now called to handle the distribution of the video data. We modify *MediaService* to call the *Encryption* service class for encryption of data before distributing the data.

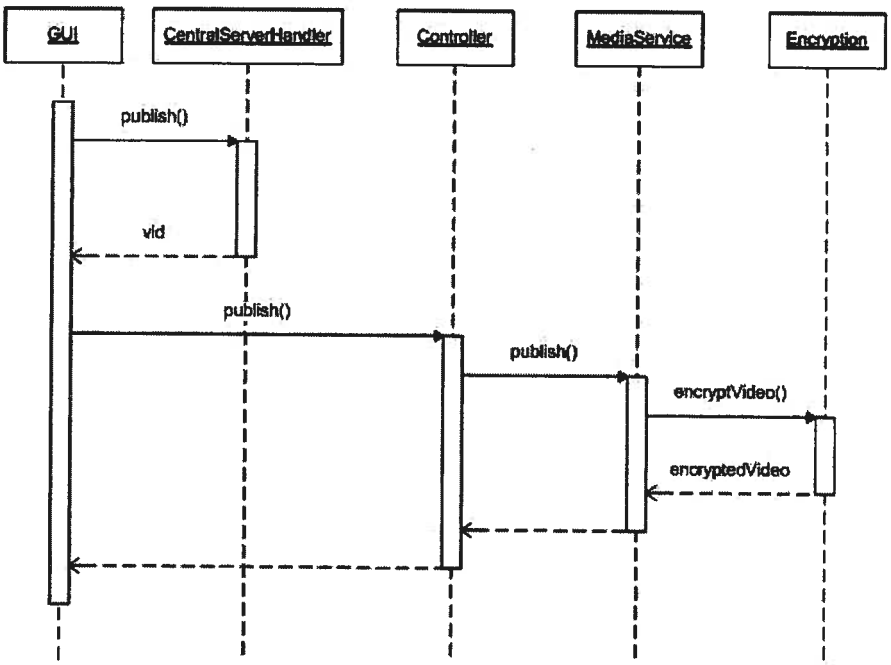

Figure 4.1: Sequence diagram of video publication

Schedule Updates and Automatic Video Retrieval

Figure 4.2 shows the sequence for updating schedules and downloading videos. The service layer *SubscriptionManager* class calls the communication layer *HybridStorage* class to retrieve updated video lists for channels and series the user has subscribed to. The service layer *RetrievalManager* class is called to match this updated video list with its current list of video downloads. If new videos to download are found, they are queued by the *RetrievalManager* and automatically downloaded using the *MediaService* class. Updates for re-

cent schedules are then retrieved by the *SubscriptionManager* using the the *HybridStorage* class. This list of videos is sent to the *RetrievalManager*, which checks for matches to subscriptions and queues any matched videos for download. Periodic polling for recent schedules is done by the *SubscriptionManager*.

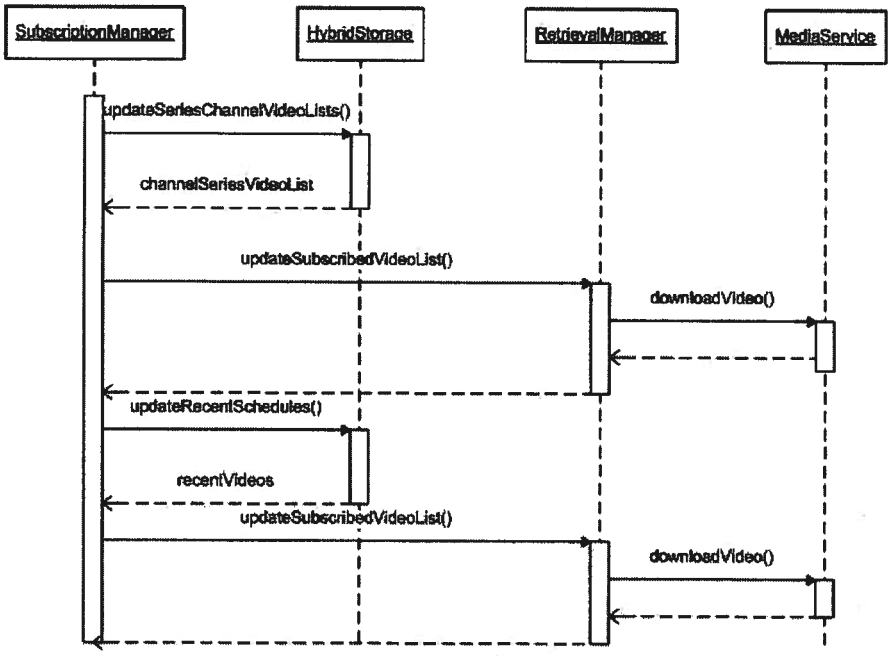

Figure 4.2: Sequence diagram of schedule maintanence

View Schedules and Downloads

The flow for viewing schedules and viewing downloads is illustrated in 4.3. After a user decides to view the schedule using the application layer *GUI*, the *GUI* simply retrieves the schedules from the service layer *SubscriptionManager* and the download list from the service layer *RetrievalManager*. Since both *SubscriptionManager* and *RetrievalManager* keeps these lists up to date, there is no need to contact the network.

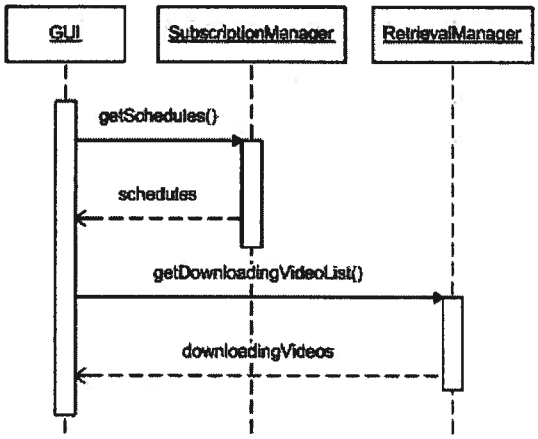

Figure 4.3: Sequence diagram of schedule and download display

4.2.3 Graphic User Interface

We implement a prototype with a Graphical User Interface (GUI) to demonstrate how publishers and clients may interact with this system. The GUI for input of publication details and display of the schedule information are shown in Figure 4.4 and Figure 4.5 respectively.

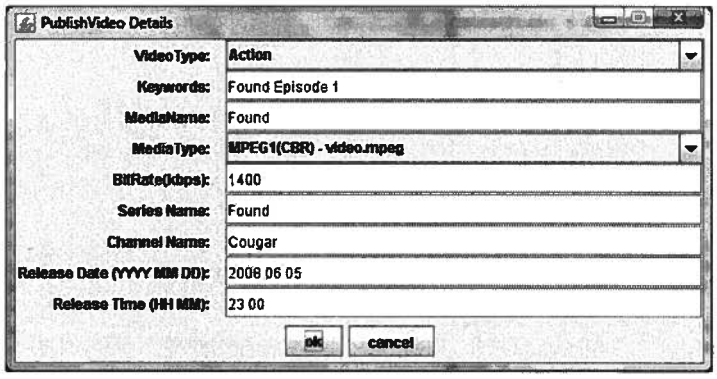

Figure 4.4: GUI for publication of a video

In Figure 4.4, the publisher is asked to input information which our frame-

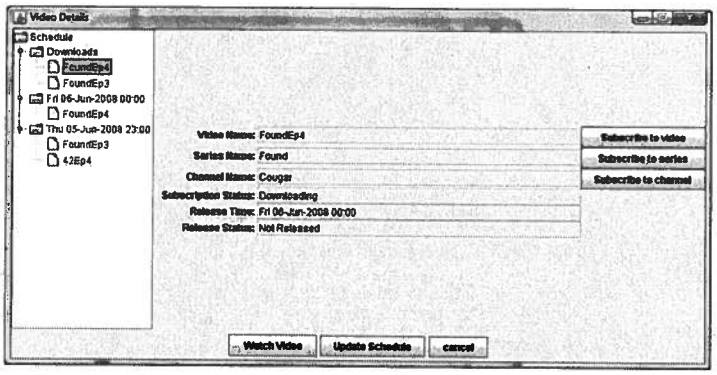

Figure 4.5: GUI for displaying schedules and downloads

work requires, such as the title ("Keywords" in the Bitvampire GUI), series ("MediaName" in the Bitvampire GUI), release date, and release time. Other information, including bitrate, is required by the Bitvampire.

Figure 4.5 displays a user's GUI after he has subscribed to a series named "Found". This is done by first clicking on any video in the schedule, shown in the left side of the interface. On the right side, various information about the video such as the channel name and series name are displayed, along with buttons for each type of subscription. These buttons allow the user to subscribe to sets of videos. The screenshot shows that both the released episode 3 and the unreleased episode 4 are automatically detected as subscribed videos and are displayed in the download area. The statuses of both episodes are also set to "Downloading", signifying that automatic retrieval has started.

Chapter 5

Evaluation

This chapter evaluates the proposed system both theoretically and through simulations.

5.1 Theoretical

We evaluate the hybrid peer to peer approach by comparing it to a central server approach.

In a traditional central server system, each request goes directly to the central server, introducing $bandwidth_t$ units of bandwidth overhead, where

$$\begin{aligned} bandwidth_t &= requests_p * messagesize_p \\ &\quad + requests_r * messagesize_r \\ &\quad + numpollschedules * pollRate * averagemessagesize_s \end{aligned} \tag{5.1}$$

$requests_p$ represents the number of publish requests and $requests_r$ represents the number of retrieval requests. $messagesize_p$ is the size of a publish message, while $messagesize_r$ and $averagemessagesize_s$ are the sizes of video information messages and schedule messages respectively. $numpollschedules$ represents the number of schedules within the $pollrange$ and $pollRate$ is the rate at which users poll for updates to schedules.

In the hybrid peer to peer storage system, the system overhead is defined as

$$\begin{aligned} bandwidth_h &= requests_p * messagesize_p \\ &\quad + lostrequests_r * (2 * messagesize_r) \\ &\quad + lostrequests_s * (2 * averagemessagesize_s) \end{aligned} \tag{5.2}$$

where $lostrequests_r$ is the number of retrieval requests not retrievable from the DHT and $lostrequests_s$ are the number of schedule requests not retrievable from the DHT. Note that we assume the DHT overhead is negligible, since the high frequency and size of viewing requests and schedule requests should far outweigh the DHT overhead in systems like Freepastry.

In the hybrid system, we assume each lost request on the DHT results in $2 * message$ because the central server answers the request and also restores the information onto the DHT.

We evaluate each part of the equation independently. For publishing messages, we see that both systems require $requests_p * messagesize_p$ units of data transfer. The DHT is not used for publishing requests and thus there is no difference in bandwidth incurred. However, publishing messages are likely to be the least common since publishing only occurs once per video.

We now evaluate the retrieval related cost. We first find the difference between the hybrid system and the traditional system, as shown below:

$$\begin{aligned} lostrequests_r \quad & *(2 * messagesize_r) - requests_r * messagesize_r \\ &= (2 * lostrequests_r - requests_r) * messagesize_r. \end{aligned} \tag{5.3}$$

We see that if $(2 * lostrequests_r) < requests_r$, then the hybrid peer to peer system requires less bandwidth. In other words, if 50% or more of the DHT requests are not lost, then the hybrid peer to peer system performs better. The

schedule related cost comparison is equivalent to the retrieval related cost. It is easy to see that if 50% or more of the DHT requests are not lost, then the hybrid peer to peer system has an advantage. We also conclude that the percentage of lost messages in the DHT is proportional to the amount of bandwidth required by the hybrid storage system.

Simulations done by Rowstron et al. have achieved greater than 95% success rate in many cases with feasible parameters using PAST, which is a storage system built on top of pastry [24]. At 95%, the hybrid peer to peer system only uses $requests_r/(2 * 0.95 * requests_r) * 100\% = 52.6\%$ of the bandwidth used by a traditional central server approach for video information and schedule distribution.

5.2 Simulation

We implement a simulator with which we evaluate the performance of the subscription and pre-release distribution system. The simulator used is a modified version of the original Bitvampire [20] simulator.

5.2.1 Simulation Setup

We simulate a network with 4500 peers and 6 seed peers. Seed peers in Bitvampire are peers who are more stable, have more bandwidth available, and have more storage space.

Most parameters in the simulation are equivalent to the Bitvampire simulation parameters. Peers have a maximum bandwidth ranging between 512 kbps to 2 Mbps and a maximum delay ranging from 4ms to 10ms. While peers only allow a maximum of 128 kbps to 1 Mbps to be used for upload, seed peers' have maximum upstream bandwidth ranging from 1 Mbps to 2 Mbps.

We allow seed peers to store a maximum of 1000 to 3000 segments while peers

may store 24 to 36 segments. In Bitvampire's simulator, each segment itself is 5 minutes long and roughly 19MB. Each video consists of 12 segments and is 60 minutes in length. The bitrate of each video is 512kbps. In Bitvampire's simulation, 500 movies exist in the system. In this simulation, we choose to use only 50 movies. This allows us to simulate a flash crowd over a small set of movies more easily. We will vary this parameter in one of the simulations.

Peers in peer to peer networks often leave and rejoin. To properly simulate this, the Bitvampire simulator simulates 20 peers leaving per minute, with peers rejoin the network after 15 minutes to 3 hours.

Three different video request arrival patterns are used to simulate different kinds of user behaviour in the real world: constant arrival, flash crowd arrival, and periodic flash crowd arrival. To simulate a constant arrival pattern, 20 requests arrive per minute constantly. This kind of behaviour can occur for popular old videos, attracting a constant stream of interested viewers. Another kind of arrival pattern, flash crowds, often occurs when a new video is released and great numbers of people want to view it at the same time. To simulate this, we first simulate peer requests at a rate of 20 requests per minute, then increase the rate to 120 requests per minute, then decrease it back to 20. The third kind of traffic, periodic flash crowds, can occur when different popular videos are released one after another. We simulate this by alternating between high request rates of 120 requests per minute and low request rates of 20 requests per minute. Figure 5.1 and Figure 5.2 show the flash crowd pattern and the periodic flash crowd pattern respectively.

In this simulator and the Bitvampire simulator, the popularity of videos follow a Zipf-like distribution. In a Zipf distribution, the i^{th} most popular video has a popularity proportional to $1/i^a$, where a is a parameter. This is chosen because Cherkasova et al. [19] collected statistics of video access frequencies in

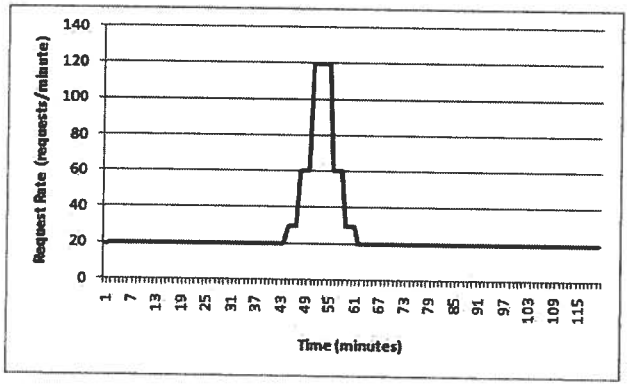

Figure 5.1: Flash crowd request pattern

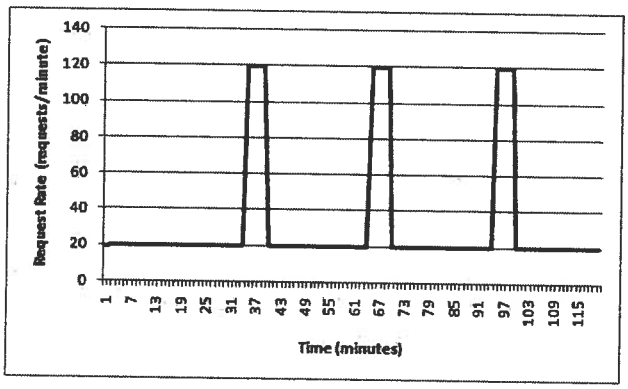

Figure 5.2: Periodic flash crowd request pattern

HPLabs media server and HP corporate media solutions server over a period of time and found that file access frequencies closely resembled a Zipf-like distribution with a value of a between 1.4 and 1.6. Following Bitvampire's settings and the results of Cherkasova et al., a is set to 1.4 in the simulator.

5.2.2 Simulation Methodology

In our simulation, we have 3 phases: publishing, subscribing, and watching. Following BitVampire's simulation methodology, we first enter the publishing

phase and allow all videos to be published. In the subscription phase, a percentage of the peers are chosen to be subscribers. These subscribers subscribe to videos and automatically start the downloading process. After a preset period of time, video releases are simulated by allowing peers to watch videos. Requests are generated following the request crowd patterns previously discussed. Each pattern generates video requests for 2 hours.

5.2.3 Simulation Results

This section describes the results from the simulations. We first simulate and evaluate the effect of subscriptions on streaming capacity. We then examine the effects of our stream prioritization algorithm. Next, we vary the percentage of peers who watch subscribed videos and measure its effects. The streaming quality difference between subscribers and non-subscribers is investigated next. Finally, we investigate the effects of varying the number of movies involved in the flash crowd.

Subscription Effects on Streaming Quality

We first explore the effects of allowing subscriptions on the quality of video streams. To measure the effect, we copy Bitvampire's methodology and measure the segment rejection ratio for viewers. The rejection ratio is defined as the percentage of segment requests for streaming videos which do not succeed in reserving bandwidth. The rejection ratio is a good measure of the performance of the streaming system because a high rejection ratio means a high percentage of peers are likely to experience pauses in the video, while a low rejection ratio means few peers are likely to experience pauses.

Figure 5.3, Figure 5.4, and Figure 5.5 show the simulation results for constant crowd, flash crowd, and periodic flash crowd request patterns respectively. For

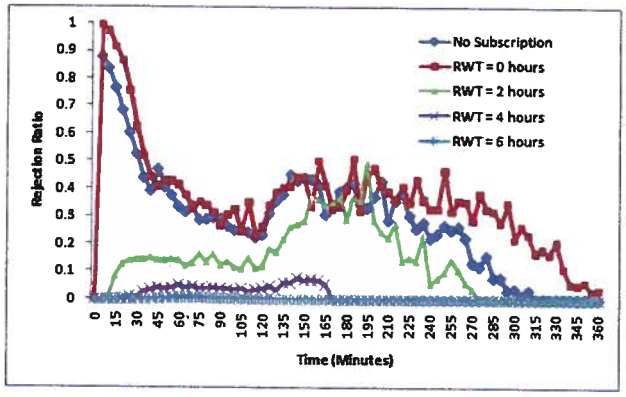

Figure 5.3: Constant crowd rejection ratio

each crowd, data is collected for the original Bitvampire algorithm and the Bitvampire algorithm with subscriptions. We use 4 different parameters for the time period between publication and release time, which we will call release wait time (RWT) in the rest of the thesis.

All three crowd patterns are affected by subscriptions in the same way. With $RWT = 0$, which is the case in a subscription system without any pre-release distribution capabilities, the subscription system increases the rejection ratio. This means users are more likely to experience pauses and long wait times. This occurs because downloaders who would have been idle in a system without subscriptions now compete with streaming peers for bandwidth.

However, as the RWT is increased to 2 hours, the capacity of the system becomes greater than the original Bitvampire system without subscription support. In this situation, peers have 2 hours to download the video before release time, and therefore increase the number of copies of segments on the network. These peers have also cached some segments, and do not need to request them after release time. As RWT is increased to 4 hours and 6 hours, the rejection ratio becomes lower and lower until it is almost 0.

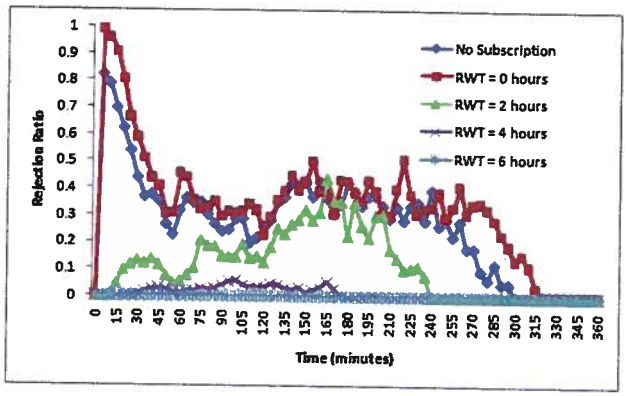

Figure 5.4: Flash crowd rejection ratio

Stream Prioritization Effects on Streaming Quality

The next set of simulations evaluates the effects of the stream prioritization algorithm. In the previous section, we saw that the subscription system has similar effects on all three crowd patterns. In all future experiments, we will use only the flash crowd pattern. Like the previous simulations, we will use the rejection ratio as a measure of the streaming quality experienced by users.

Figures 5.6, 5.7, 5.8, and 5.9 show the rejection ratios for 0 hours, 2 hours, 4 hours, and 6 hours of time before videos are released after publication respectively. In all settings, the optimized algorithms maintain a lower rejection ratio than the unoptimized version. At $RWT = 0$ hours, we see that the stream prioritization algorithm is able to perform almost as well as the original Bitvampire even though many downloaders are attempting to reserve bandwidth. All following simulations will use the optimized version of the algorithm.

Subscription Rate Effects on Streaming Quality

With the following set of experiments, we evaluate the effects of the subscription rate on streaming quality. We define the subscription rate as percentage of peers

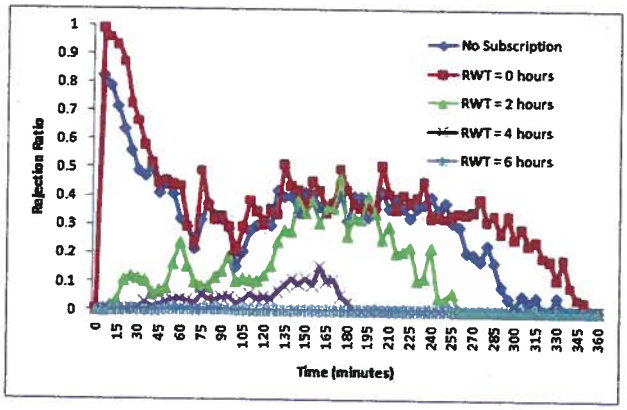

Figure 5.5: Periodic flash crowd rejection ratio

who watch subscribed videos in the simulation.

Figure 5.10 shows that with 0 hours between publication and release time, the rejection rate for all percentages of subscriptions are similar. Peers with subscriptions to videos have not had a chance to download the video before other peers start watching the video. Since watching peers have priority and there is not enough bandwidth in the network for all requests, only the watching peers succeed in requesting videos. This means that all of the bandwidth available from suppliers is used for streaming peers, so subscriptions and downloaders do not affect the system or the streaming rejection ratio.

With 2 hours between publications as shown in Figure 5.11, we see a result that might appear strange. While having subscriptions results in a significantly lower rejection ratio than not having subscriptions, a lower percentage of subscriptions also results in a lower rejection ratio. The cause for this is that with only 2 hours between publication and release time of videos, a high subscription percentage results in a flood of download requests that do not finish before watching starts. As a result, the later segments in each video have not had a chance to propagate through the network and get cached.

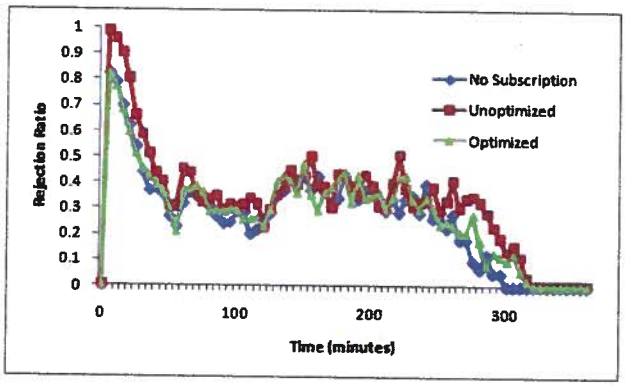

Figure 5.6: Stream prioritization effects with RWT = 0 hours

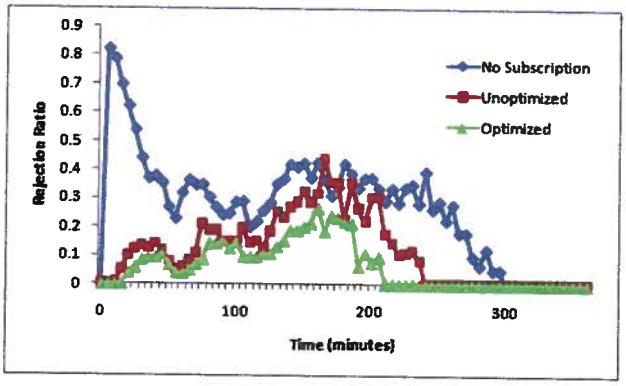

Figure 5.7: Stream prioritization effects with RWT = 2 hours

Figure 5.12 shows the rejection ratios with 4 hours between publication and release time of videos. We see that in this situation, 25% subscription rate results in the lowest rejection ratio. This shows a subscription rate of 25% is the optimal point in this situation where many downloading peers are able to complete downloading videos before the videos are released. As the subscription is increased or decreased from this optimal point, the rejection ratio increases.

As we increase the RWT further to 6 hours, we see that the optimal point is at 100% subscription ratio. This is shown in Figure 5.13. In this case, the

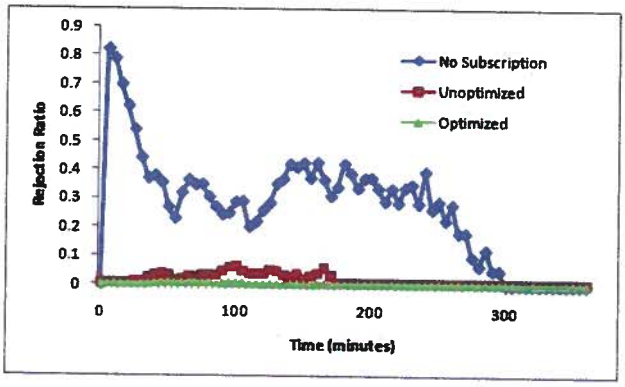

Figure 5.8: Stream prioritization effects with RWT = 4 hours

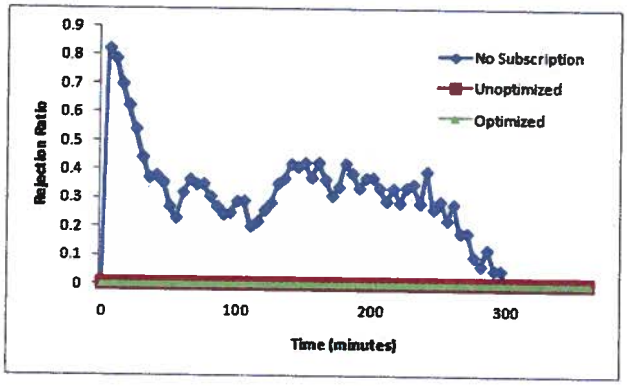

Figure 5.9: Stream prioritization effects with RWT = 6 hours

RWT is long enough that subscribers are almost all capable of downloading the complete video before release.

From these results, we can conclude that there is an optimal percentage of subscribers depending on the network capacity and number of requests. If there is not enough bandwidth for subscribers to download significant amounts of the video before release time, then a lower subscription rate may result in a lower rejection ratio. However, we note that with $RWT >= 2$, rejection ratio is significantly lower than the system without subscription capabilities.

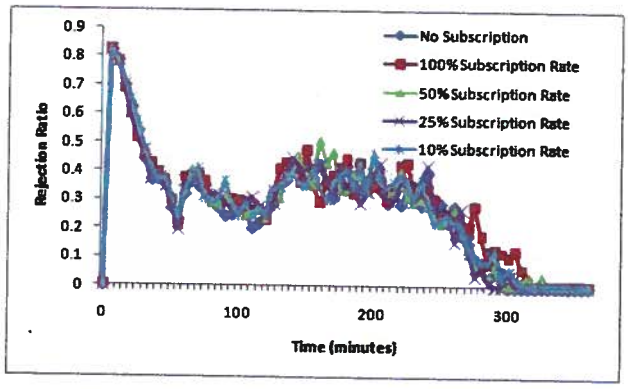

Figure 5.10: Subscription rate effects with RWT = 0 hours

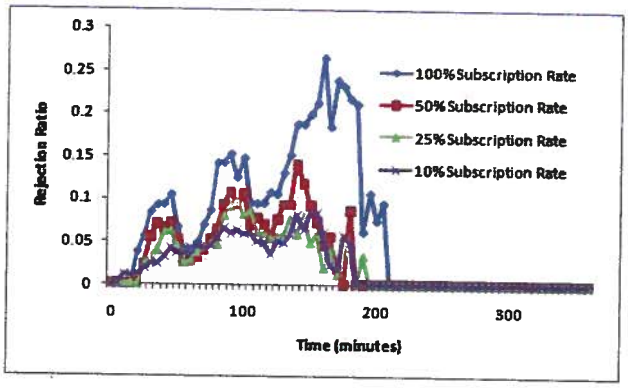

Figure 5.11: Subscription rate effects with RWT = 2 hours

Incentives for Subscribing to Videos

We examine the streaming quality difference between subscribers and non-subscribers. In order to evaluate this, we run the simulation and measure the percentage of total segments transferred for subscribers and non-subscribers using various parameters for RWT. The percentage, which we will refer to as the complete rate, is calculated using $percent_t = requests_t / requests_{end}$, where $requests_i$ is the number of requests at time i and end is when all peers have stopped streaming videos. A higher percentage of total segments transferred

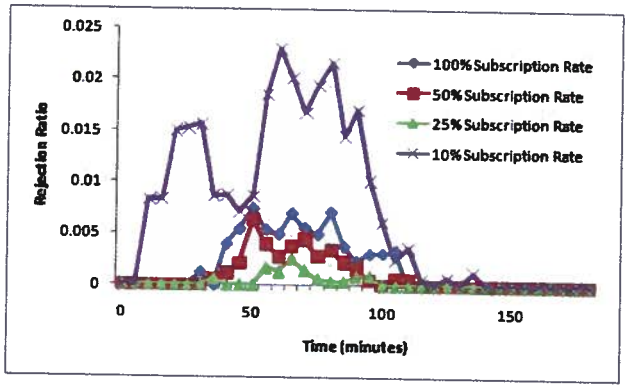

Figure 5.12: Subscription rate effects with RWT = 4 hours

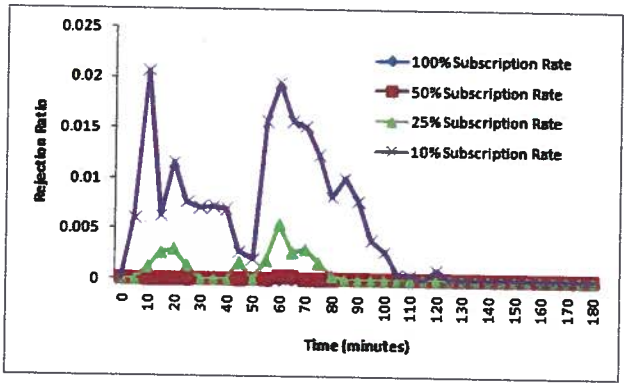

Figure 5.13: Subscription rate effects with RWT = 6 hours

signifies that the set of peers have been more successful in retrieving requested videos.

Figures 5.14, 5.15, 5.16, and 5.17 show the simulation results for RWT settings of 0 hours, 2 hours, 4 hours, and 6 hours respectively. In all of the results, subscribers have a higher percentage of total segments transferred at all points in time. This shows that subscribers are more likely to finish streaming videos earlier with fewer disruptions in the video playback.

We also see that with a RWT setting of 0 minutes, the difference is negligible.

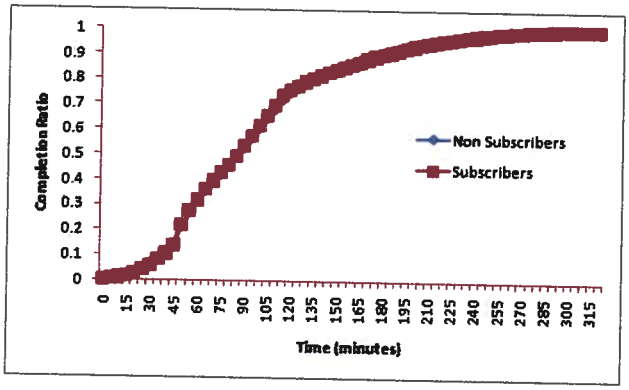

Figure 5.14: Completion ratio for RWT = 0 hours

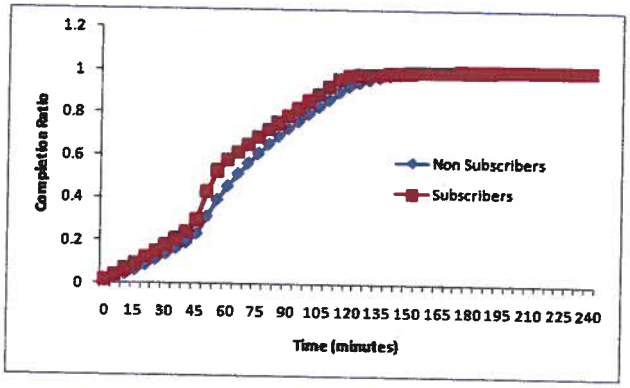

Figure 5.15: Completion ratio for RWT = 2 hours

As previously mentioned, in this situation with stream prioritization in effect, subscribers' download requests do not get accepted and there is little difference between subscribers and viewers. However, as RWT increases, we note that there is a noticeable difference between subscribers and non-subscribers. The smoother playback of video gives peers a natural incentive to subscribe to videos.

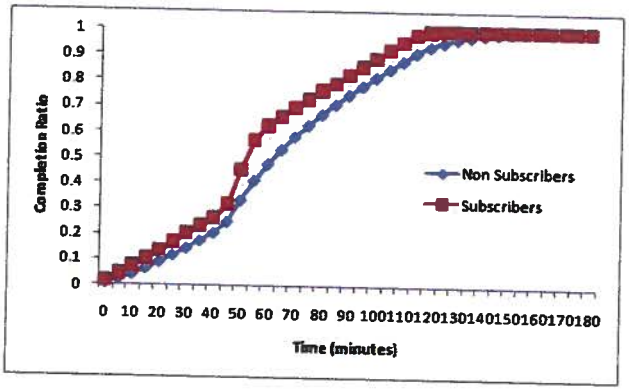

Figure 5.16: Completion ratio for RWT = 4 hours

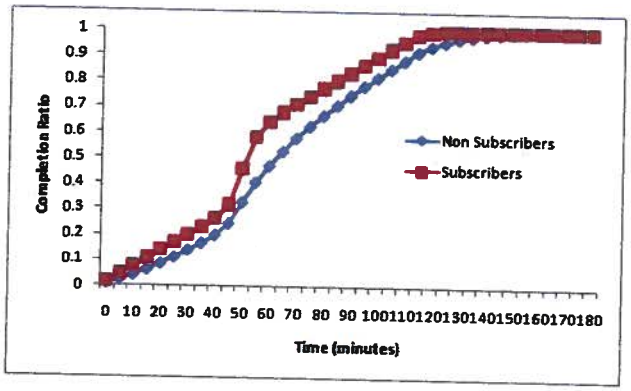

Figure 5.17: Completion ratio for RWT = 6 hours

Effect of Varying the Number of Movies

We measure the rejection ratio for 1, 50, and 100 movies to evaluate the effect of varying the number of flash crowd movies on streaming quality. It is important to note that the number of movies in this setting is the number of movies which receive a large amount of requests after they are released. This number does not include movies which few people request but exists on the network, which is not simulated since we only want to see the effects of subscription on flash crowds.

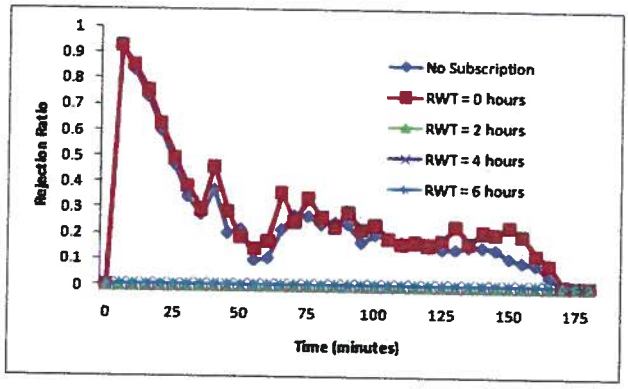

Figure 5.18: Rejection ratio for 1 movie

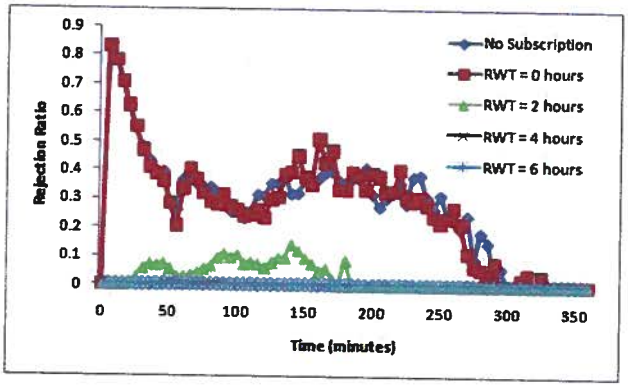

Figure 5.19: Rejection ratio for 50 movies

The rejection ratio for 1, 50, and 100 movies are shown in Figures 5.18, 5.19, and 5.20 respectively. With 1 movie, the rejection ratio is decreased to 0 when $RWT >= 2$. This shows the pre-release system is most effective with a flash crowd consisting of few movies. With few movies, the segments quickly propagate through the network and get cached by peers, thus increasing the bandwidth available for streaming of these segments. With 100 movies, we see that the rejection ratio is higher than the rejection ratio of 1 movie. With a larger number of movies, segments do not propagate and get cached as quickly

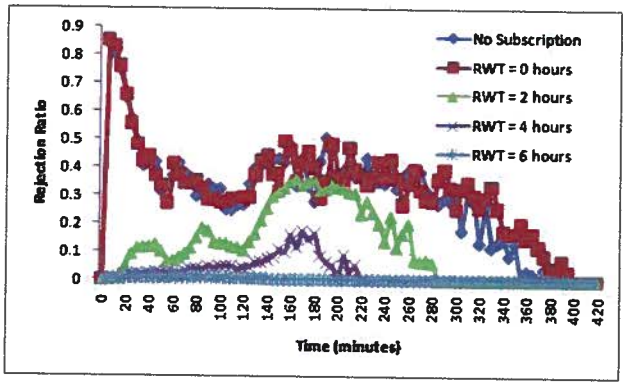

Figure 5.20: Rejection ratio for 100 movies

since the requests are split amongst a greater number of segments.

In all of the results, rejection ratios for no subscription and $RWT = 0$ are very close. As RWT is increased, the rejection ratio is lowered in all three simulations.

Chapter 6

Conclusion and Future Work

6.1 Conclusions

In this thesis, we propose a hybrid peer to peer framework for a subscription and a pre-release distribution system. This framework enhances peer to peer video on demand streaming by allowing publishers to publish videos before release time. This feature allows video playback to remain smooth even in flash crowd situations. We design a subscription system to allow peers with unreleased videos to know who to distribute the videos to. An encryption management system is proposed to prevent users from watching videos before release time. Finally, we modify the streaming algorithm in a peer to peer video on demand framework, Bitvampire [20], to support downloading of videos. We optimize this algorithm by prioritizing streaming peers over downloading peers.

We show the viability of this system by describing implementation details for creating a prototype using existing frameworks. These frameworks are Bitvampire, a peer to peer video on demand framework, and FreePastry, a DHT framework. A GUI prototype is implemented to investigate how users may interact with the system.

In order to evaluate our system, we modified the simulator used by Bit-

vampire to support subscriptions. The simulation results show that the stream prioritization algorithm increases the streaming quality experienced by peers. The results also show that allowing the pre-release distribution of videos can significantly increases the quality of video playback in flash crowd situations. This remains true as the percentage of subscribers and the number of movies affected by the flash crowd is varied. We also observe that subscribers on average experience better playback quality than non-subscribers, creating a natural incentive for users to subscribe.

6.2 Future Work

There are several areas that can be interesting to investigate. The first one is the investigation of peer to peer continuous query systems, such as P2P-DIET [17]. Continuous query systems generally allow more complex queries to be made. For example, a user might be able to subscribe to all series starting with the letter A, or all comedies made by a certain director. Investigation into the performance difference, whether or not users want this level of subscription capability, and how to make a user friendly interface for it can lead to interesting results.

Scalable video is another interesting area of research. Priority drop [18] and other scalable video approaches allow a lower quality version of the video to be downloaded first. By combining more video data with the lower quality version, a higher quality version of the video may be displayed. Using this approach, as the bandwidth available varies, the playback of the video can still be smooth, albeit at a lower quality when the bandwidth of the transfer is low. Scalable video approaches can be integrated into the video download retrieval algorithm. Instead of always downloading the blocks in order, a scalable video approach would allow the downloading of a lower quality layer first. The advantage of

this is that if there is little or no bandwidth available for streaming the video and the download of the video is not complete, the user would be able to play more of the movie using a scalable video approach.

Another possible area of exploration is supporting authentication and permission control. In traditional television systems, administrators are able to control what channels each peer has access to, allowing the traditional business model of charging for access to videos to be used. In the proposed architecture, the administrator is unable to prevent peers from transferring data to each other and is unable to prevent peers from modifying information on the DHT. One possible solution is for a trusted central server to generate a public key and a private key using asymmetric cryptography algorithms such as RSA. Asymmetric cryptography can be used to allow the central server to sign data with its private key. With knowledge of the central server's public key, peers can check whether or not the central server has signed the data, and thus can determine if data came from the central server. The algorithms in this framework can be modified to only accept video information and decryption keys signed by the central server, thereby preventing unauthorized peers from publishing videos by modifying the DHT. To allow permission control, each peer receives a signed certificate from the central server stating his/her permissions, name, and public key. Whenever a peer, $PeerA$, requests a video from another peer, $PeerB$, he/she must send their certificate to prove they have permission to access the video. $PeerA$ also sends a personally signed message containing the current time and the name of $PeerB$, which $PeerB$ can check using the public key from $PeerA's$ certificate. This message allows $PeerB$ to know the request was sent by $PeerA$, and not an old request repeated by a malicious peer. This system allows the central server to control who has access to what channels, while requiring very little bandwidth from the central server. The central server only needs to

be involved in the creation of user accounts and changing of permissions.

Finally, abuse prevention is another interesting direction to investigate. For example, in algorithms such as stream prioritization, the requester is trusted and assumed to be telling the truth. However, a malicious or greedy peer may always send stream requests instead of download requests. A malicious peer may also flood the network with reservation messages or $freeDownloadBandwidth$ messages which causes suppliers to drop downloaders. Peers may also abuse the system by continuously downloading while only allowing very little upstream bandwidth to be utilized for sharing. Incentive mechanisms may be used to prevent this.

Bibliography

[1] Babelgum. `http://www.babelgum.com`.

[2] Bbc iplayer. `http://www.bbc.co.uk/iplayer/`.

[3] Bittorrent. `http://www.bittorrent.com`.

[4] Joost. `http://www.joost.com`.

[5] Napster. `http://www.napster.com`.

[6] Netflix. `http://www.netflix.com`.

[7] Veoh. `http://www.veoh.com`.

[8] Youtube. `http://www.youtube.com`.

[9] S. Acharya and B. Smith. MiddleMan: A Video Caching Proxy Server. *Proceedings of NOSSDAV*, 2000.

[10] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 205–217, 2002.

[11] Y. Chae, K. Guo, MM Buddhikot, S. Suri, and EW Zegura. Silo, rainbow, and caching token: schemes for scalable, fault tolerant stream caching. *Selected Areas in Communications, IEEE Journal on*, 20(7):1328–1344, 2002.

[12] Y. Chu, S. Rao, S. Seshan, and H. Zhang. Enabling conferencing applications on the internet using an overlay muilticast architecture. *Proceedings of the 2001 SIGCOMM conference*, 31(4):55–67, 2001.

[13] Y. Chu, SG Rao, S. Seshan, and H. Zhang. A case for end system multicast. *Selected Areas in Communications, IEEE Journal on*, 20(8):1456–1471, 2002.

[14] H. Deshpande, M. Bawa, and H. Garcia-Molina. Streaming Live Media over a Peer-to-Peer Network. *Submitted for publication*, 2002.

[15] P. Druschel, E. Engineer, R. Gil, Y.C. Hu, S. Iyer, A. Ladd, et al. FreePastry. *Software available at http://www. cs. rice. edu/CS/Systems/Pastry/FreePastry.*

[16] L. Gong. Project JXTA: A Technology Overview. *Sun Microsystems*, 2001.

[17] S. Idreos, M. Koubarakis, and C. Tryfonopoulos. P2P-DIET: an extensible P2P service that unifies ad-hoc and continuous querying in super-peer networks. *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 933–934, 2004.

[18] C. Krasic, J. Walpole, and W. Feng. Quality-adaptive media streaming by priority drop. *Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, pages 112–121, 2003.

[19] M. Gupta L. Cherkasova. Charactering Locality, Evolution, and Life Span of Accesses in Enterprise Media Server Workloads. *Proceedings of NOSSDAV*, 2002.

[20] X. Liu. *BitVampire: A Cost-Effective Architecture for On-Demand Media Streaming in Heterogeneous P2P Networks.* PhD thesis, The University of British Columbia, 2005.

[21] B.T. Loo, R. Huebsch, I. Stoica, and J.M. Hellerstein. The case for a hybrid P2P search infrastructure. *Proceedings of the 3rd IPTPS*, 2004.

[22] S. Ramesh, I. Rhee, K. Guo, ST Microelectron, and CA San Jose. Multicast with cache (Mcache): an adaptive zero-delay video-on-demand service. *Circuits and Systems for Video Technology, IEEE Transactions on*, 11(3):440–456, 2001.

[23] R. Rejaie and A. Ortega. PALS: peer-to-peer adaptive layered streaming. *Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, pages 153–161, 2003.

[24] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, 2001.

[25] S. Sen, J. Rexford, and D. Towsley. Proxy prefix caching for multimedia streams. *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 3, 1999.

[26] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the 2001 SIGCOMM conference*, 31(4):149–160, 2001.

[27] DA Tran, KA Hua, and T. Do. ZIGZAG: an efficient peer-to-peer scheme for media streaming. *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, 2.

[28] D. Xu, H.K. Chai, C. Rosenberg, and S. Kulkarni. Analysis of a Hybrid Architecture for Cost-Effective Streaming Media Distribution. *Proceedings of SPIE*, 5019:87, 2003.

[29] B. Yang and H. Garcia-Molina. Comparing Hybrid Peer-to-Peer Systems. *The VLDB Journal*, pages 561–570, 2001.