BitVampire: A Cost-Effective Architecture for On-Demand Media Streaming in Heterogeneous P2P Networks

by

Xin Liu

M.Sc., Peking University, 2002
B.Eng., Tsinghua University, 1999

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

September 2005

Abstract

On-demand media streaming has recently gained intensive consideration due to its promising usage in a rich set of Internet-based services such as video on demand, distance learning, media distribution, etc. However, there are still many challenges towards building efficient, scalable, on-demand streaming systems. In this thesis, we propose a novel cost-effective on-demand media streaming architecture for heterogeneous peer-to-peer networks, named BitVampire. The key idea of BitVampire is to aggregate peers' storage and bandwidths to facilitate on-demand media streaming. To achieve this goal, we split published videos into segments and distribute them to different peers. When watching a video, a peer searches the corresponding segments, and then aggregates bandwidths from multiple supplying peers to stream the video. To deploy this architecture in a dynamic heterogeneous peer-to-peer network, three key techniques are used: (1) Given that peers offer different resources and may leave at any time, a media segments distributing algorithm and a caching scheme are proposed, which achieve fast system streaming capacity amplification. (2) An application-level overlay, called Category Overlay, is chosen as the underlying search infrastructure to efficiently find the desired segments. (3) A scheduling algorithm is proposed to aggregate bandwidths from multiple supplying peers and coordinate them to serve one streaming request. We demonstrate the effectiveness of this proposed architecture through extensive simulation experiments on large, Internet-like topologies.

Contents

List of Tables

List of Figures

Acknowledgements

First, I would like to thank my supervisor, Dr. Son T. Vuong, for his guidance, inspiration, and encouragement. I am also grateful to Dr. Charles Krasic for being my second reader and for his useful comments that have improved this thesis.

Thanks to the members in NIC lab, who I was pleasant to work with. In particular, I want to thank Jun Wang. Without his collaboration, this thesis could not be accomplished. Thanks to my lab mates - Juan Li, Mohammad Alam, Anthony Yu, Gary Huang, Kamran Malik, Christian Chita, Sukanta Pramanik, Sergio G. Valenzuela, Ying Su, and Wei Li.

Finally, I'd like to thank my parents and my wife, Bin Zhang, for their endless love and support.

XIN LIU

The University of British Columbia

September 2005

Chapter 1

Introduction

1.1 Motivation

With the proliferation of high bandwidth networks, on-demand media streaming has recently gained intensive consideration due to its promising usage in a rich set of Internet-based services such as video on demand, distance learning, media distribution, etc. However, there are still many challenges towards building efficient, scalable, on-demand streaming systems due to the high bandwidth and delay requirements for media streaming.

The conventional design of on-demand media streaming systems follows the Client-Server model, in which a set of centralized servers store all the video files. Clients directly contact servers and request streaming content from servers. Obviously, this architecture is not scalable since servers become the bottleneck as the requests increase. To alleviate servers' traffic load, several proxy-based architectures have been proposed,

in which a set of proxies are deployed in the network. Clients can request the cached portion of videos from the proxies.

However, in both server-based and proxy-based architectures, servers and proxies are expected to deliver a high-quality streaming service to a large number of clients. Therefore, servers and proxies should be very powerful in terms of computing power, outbound bandwidth, storage, etc., which makes deployment and maintenance very expensive. On the other hand, recent research and experiments reveal that the current Internet has enough resources to support large-scale media streaming in a peer-to-peer fashion [47][37]. Inspired by this fact, we propose BitVampire, a low-cost on-demand media streaming architecture, which exploits the often-underutilized peers' resources to support large-scale on-demand media streaming. Since this architecture does not need powerful servers/proxies, it is much more cost-effective compared to other approaches.

The basic idea of BitVampire is to split published videos into segments and distribute them to different peers. When watching a video, the requesting peer (or receiver) first searches the corresponding segments, then aggregates bandwidths from multiple supplying peers to stream the video. To deploy this architecture in a dynamic heterogeneous peer-to-peer network, three problems need to be addressed: (1) How to distribute and cache segments, taking into consideration that peers offer different resources and may leave at any time. (2) How to efficiently find the desired segments. (3) How to aggregate bandwidths from multiple peers and coordinate them to serve one streaming request. In this thesis, we present the design, implementation, and evaluation of BitVampire, with the emphasis on addressing the three problems mentioned above.

1.2 Thesis Contributions

This thesis proposes BitVampire, a novel architecture that exploits the often-underutilized peers' storage and bandwidths to support cost-effective on-demand media streaming in dynamic heterogeneous peer-to-peer networks. We have implemented a prototype based on the proposed architecture and evaluated the architecture through an extensive simulation study. The simulation results verified the effectiveness of the proposed architecture. The following are the main contributions of this thesis[1]:

- To efficiently find the desired media segments, we choose Category Overlay [26][43] as the underlying search infrastructure. The simulation study in [26][43] shows that Category Overlay can provide an efficient search service in a dynamic peer-to-peer network. However, BitVampire operations are independent of the underlying search infrastructure. Therefore, BitVampire can also be deployed on top of other search infrastructures, as long as these infrastructures provide efficient, keyword-based search services.
- We propose a Media Segments Distributing (MSD) algorithm to distribute segments to peers. Given that peers offer different resources and may leave at any time, MSD tries to distribute media segments to the peers that are more stable, have higher available outbound bandwidth and lower streaming serve frequency, which results in fast system streaming capacity amplification (system streaming capacity is defined as the number of video watching sessions that can be served concurrently).
- We propose a Multiple Suppliers Scheduling (MSS) algorithm to aggregate

[1] Parts of this work have been recently accepted for publication [24][25].

bandwidths from multiple supplying peers and coordinate them to serve one streaming request, which results in a small initial buffering time.

- We developed a general purpose P2P application framework, which separates system architecture from specific service implementation.
- To demonstrate the feasibility of the proposed architecture, we implemented a prototype based on it. The prototype is implemented based on our general purpose P2P application framework, using Java and JMF [17].
- We evaluated our proposed architecture through an extensive simulation study on large, Internet-like topologies.

1.3 Thesis Organization

This thesis consists of six chapters. Chapter 2 provides the background to P2P computing, as well as a detailed description of related work. In Chapter 3, we present the system design. We first introduce Category Overlay, which is chosen as the underlying search infrastructure. We then present and discuss our approaches to distribute, search, and cache media segments, as well as the scheduling algorithm to aggregate bandwidths from multiple supplying peers. Chapter 4 presents details regarding our application framework and prototype implementation. Chapter 5 presents the simulation setup and performance evaluation results. We conclude the thesis and discuss potential directions for future research in Chapter 6.

Chapter 2

Background and Related Work

This chapter introduces background information on Peer-to-Peer (P2P) technology, with the emphasis on P2P content search and P2P media streaming. Then we provide a survey of the related work.

2.1 Peer-to-Peer Computing

Since the success of Napster [28], Peer-to-Peer technology has been receiving intensive attention. It is increasingly becoming an important technique in various areas, such as distributed and collaborative computing both on the Web and in ad-hoc networks. There are lots of industrial efforts in P2P technology, including the P2P Working Group, led by many industrial partners such as Intel, Sun, HP, and a number of startup companies; and JXTA, an open-source effort led by Sun. There are also a number of academic events dedicated to P2P technology. However, the fundamental idea of organizing computers as peers is not new. Actually, the original Internet was designed in a Peer-to-Peer manner. It

encourages sharing information on research and development in scientific and military fields by sending data packets between any two computers. Hence, the current popular P2P computing model, since the first appearance of Napster [28] in May 1999, can be seen as "a renaissance of the original Internet model" [2].

There are several of the definitions of P2P that are being used by the community. The Peer-to-Peer Working Group defines P2P as "the sharing of computer resources and services by direct exchange between systems" [31]. Clay Shirkey from the Accelerator Group defines P2P as "a class of applications that takes advantage of resources - storage, cycles, content, human presence - available at the edges of the Internet" and "peer-to-peer nodes must operate outside the DNS and have significant or total autonomy of central servers". However, we can conclude that a typical P2P application should possess some basic properties: *dual identities (client and server)*, *resource sharing*, and *cooperation*. Peer nodes are usually connected and they cooperate with each other in providing resource sharing services: a node acts as *client* when it is requesting resources, while acts as *server* when it is providing resources.

Dejan S. Milojicic, etc. [27] summarizes the properties that the P2P computing model is able to provide: (1) cost sharing - cost can be shared and distributed to all peer nodes. (2) scalability and reliability - services are provided by many autonomous peer nodes rather than few central servers. (3) resource aggregation - many types of resources, which were originally available only on local machines, can now be shared among peer nodes. (4) increased autonomy - resource and computation locality can be better enforced. (5) anonymity and privacy - users are able to prevent their information from being collected by a particular entity. (6) ad-hoc connectivity - peer is not tied to any particular location in the system.

2.1.1 Peer-to-Peer Content Search

Currently there are mainly two P2P search schemes in the literature. Unstructured P2P systems such as Gnutella [11] and Kazaa [21] use flooding as their essential search techniques. Although flooding is simple and works well in a highly dynamic network environment, it will inevitably generate a huge amount of redundant messages, which makes it not scalable. Structured P2P systems such as Chord [39], CAN [33], and Tapestry [46] use Distributed Hash Table (DHT) based search techniques, which can guarantee to locate content within a bounded number of hops. But these techniques tightly control both the placement of data and the topology of the network, which results in high maintenance costs. Furthermore, they can only support search by identifier and lack the flexibility of keyword searching.

The emergence of recent work on hybrid infrastructures, such as YAPPERS [12] and [23], reveal the possibility of creating a P2P system that combines both the advantages of unstructured P2P and DHT. Inspired by these works, Category Overlay is proposed as the joint research work of this thesis, which can provide an efficient search service with a relatively low maintenance overhead. Section 3.2 covers the details of Category Overlay.

2.1.2 Peer-to-Peer Media Streaming

The first P2P technique for streaming applications was introduced by [38]. This early design, however, did not address the stability of the system under network dynamics. [9] proposed SpreadIt, which builds a single distribution tree of the peers. A new receiver joins the streaming session by traversing the tree nodes downward from the source until

finding one with unsaturated bandwidth. SpreadIt has to get the source involved whenever a failure occurs, thus vulnerable to disruptions due to the severe bottleneck at the source. Additionally, orphaned peers reconnect by using the join algorithm, resulting in a long blocking time before the service can resume.

CoopNet [30] uses a multi-description coding method for the media content. In this method, a media signal is encoded into several separate streams, or descriptions, such that every subset of them is decodable. CoopNet builds multiple distribution trees spanning the source and all the receivers, each tree transmitting a separate description of the media signal. Therefore, a receiver can receive all the descriptions in the best case. A peer failure only causes its descendant peers to lose a few descriptions. The orphaned are still able to continue their service without burdening the source. However, this is done with a quality sacrifice. Furthermore, CoopNet puts a heavy control overhead on the source since the source must maintain full knowledge of all distribution trees.

Narada [7][8] focuses on multi-sender multi-receiver streaming applications, maintains a mesh among the peers, and establishes a tree whenever a sender wants to transmit content to a set of receivers. Narada only emphasizes on small P2P networks. Its extension to work with large-scale networks was proposed in [16] using a two-layer hierarchical topology. To better reduce cluster size, thereby reducing the control overhead at a peer, the scheme NICE [3] and ZIGZAG [41] focus on large P2P networks by using the multi-layer hierarchical clustering idea.

2.2 Related Work

In the following sections, we present the previous work related to our proposed architecture. We start from the conventional central server-based on-demand media streaming systems. Then we describe several proxy-based systems. Finally, an existing P2P-based system is presented, as well as its difference between our proposed architecture.

2.2.1 Central Server-based Systems

A majority of the existing on-demand media streaming systems follows Client-Server model, in which a set of centralised servers store all of the video files and respond to all of clients' requests. However, this architecture is not scalable since servers will become the bottleneck as the requests increase. To save servers' resources and alleviate servers' traffic loads, multicast has been applied and different solutions have been proposed. Batching [44] aggregates multiple client requests into one multicast session. However, the users have to suffer long playback delay since their requests are enforced to be synchronized. Patching [14][5] tries to address this problem by allowing the client to catch up with an on-going multicast session and patch the missing starting portion through server unicast. In merging [10], a client can repeatedly merge into a larger and larger multicast session using the same way as patching. However, in order to ensure smooth playback, these two approaches need the client to be capable of receiving multiple streams simultaneously and buffering large amount of data. In periodic broadcasting [15][42], the server separates a media stream into segments and periodically broadcasts them through different multicast channels, from which a client can choose to

join in. Although more efficient in saving server bandwidth, it shares the same limitations as the approaches mentioned above.

2.2.2 Proxy-based Systems

Another major category of on-demand media streaming systems employs the cooperative proxy caching technique. Existing work in this area includes prefix-based caching [40][35] and segment-based caching [1][6]. In prefix-based caching, proxies store the initial frames of popular clips. Upon receiving a request for the stream, the proxy initiates transmission to the client and simultaneously requests the remaining frames from the server. As the proxy is generally closer to the clients than the origin server, the start-up delay for a playback can be remarkably reduced. In segment-based caching, parts of media content are cached on different proxies in the network and the stream is coordinated to playback from these independent caches.

2.2.3 P2P-based Systems

Mohamed M. Hefeeda, etc. proposed a P2P on-demand media streaming architecture in their work [13]. This is probably the system most like ours by far. However, our architecture is different from theirs in the following ways: (1) They cluster peers into two-level clusters, and cluster super peers are selected from cluster members. Then they rely on these super peers to search the media content. In our architecture, we rely on Category Overlay to search the desired media segments, which can provide an efficient keyword search service. (2) In their architecture, a seed peer introduces a media file into the system. Initially the seed peer holds all of the segments. As the streaming requests come in, the accessed segments will be cached in requesters. However, in our architecture,

once the video is published, it will be split into segments and these segments will be distributed to different peers. The segments distributing process improves the service availability, since the published video is belonging to the whole network, not the single publisher peer. (3) In their work, they do not consider the user behaviour when watching video. Thus, their architecture requires that all the segments of a video be found before the streaming starts. However, this is not a reasonable restriction because most streaming sessions lasts only for a short period. Our architecture removes this restriction, where users can watch any segments if there is enough available resources currently in the network. (4) To aggregate bandwidths from multiple supplying peers, their architecture simply assigns a different portion of the segment to peers proportional to their bandwidths. In our architecture, a more complicated schedule algorithm is proposed to coordinate multiple supplying peers to stream the segment. (5) In their simulation study, the system contains only one video. While in our simulation experiments, the system has 500 videos published, and we also take into consideration the videos' popularity and the user behaviour pattern when watching the video.

Chapter 3

System Design

In this chapter, we present BitVampire system design. We first give an overview of the whole architecture, then present and discuss each key component, including Category Overlay, media segments distributing, searching and caching, and the scheduling algorithm to coordinate multiple supplying peers to serve one streaming session.

3.1 System Overview

This section provides an overview of the proposed architecture. We first identify all entities in the system. Then, we explain how the system works and how the entities interact with each other.

3.1.1 System Entities

Following are the entities in our proposed architecture:

- *Peers*. This is a set of nodes currently participating in the system. Typically these are the computers of clients who are interested in some of the media files offered by the system. We denote $\{P_1, P_2, \ldots, P_n\}$ as the set of peers in the system and P_i as the i^{th} peer. To participate in the system, peers contribute some of their storage and outbound bandwidths. The target environment of our proposed architecture is P2P networks, and it has been known that peers in P2P networks are quite heterogeneous [36]. Thus, we model the peers' heterogeneity as follows. (1) Bw_i (in kbps), the outbound bandwidth peer P_i is willing to contribute to the system and Bw^{avail}_i (in kbps), the available outbound bandwidth peer P_i can provide to the system at a specific time. (2) St_i (in bytes), the storage peer P_i is willing to contribute to the system and St^{avail}_i (in bytes), the available storage peer P_i can provide to the system at a specific time. Initially, $Bw^{avail}_i = Bw_i$, $St^{avail}_i = St_i$ and $Bw^{avail}_i \leq Bw_i$, $St^{avail}_i \leq St_i$ hold at any time.

- *Seed Peers*. To handle the situation in which all the hosting peers of a specific segment leave the system, we introduce seed peers into the architecture. Seed peers always stay in the system, and each segment of published videos has a replica stored in seed peers. Seed peers serve the streaming request only when the request cannot be satisfied by regular peers. Seed peers are almost the same as regular peers, except that they are stable and have large storage capacity. However, in the case of a media centre distributing contents, the seed peers can be dedicated machines with reasonable capacity.

- *Statistical Server*. To provide the popularity of videos, we introduce statistical server into the architecture. Statistical server is a dedicated, well-known server whose responsibility is to gather statistical information on video access and provide

videos' popularity to peers. For a specific video k, if its access count is C_k, and the access count for the most popular video is C_{max}, video k's popularity is defined as C_k/C_{max}. Although the video access information can be gathered in a distributed manner, we choose statistical server for the following reasons: (1) The messages exchanged between the statistical server and peers are very small, and the computation the statistical server needs to perform when receiving a message is quite small too. Thus, the traffic load and computation overhead in the statistical server are within a reasonable range. (2) Gathering video access information in a distributed manner will increase the system's complexity and introduce extra traffic due to the consistency problem. (3) Furthermore, to alleviate traffic load on a single server and to avoid single point of failure, we can deploy multiple statistical servers in the system.

- *Media Files.* This is a set of videos currently available in the system. Every video is assigned a unique ID, called *videoID*, which is generated when the video is published. Every video belongs to a predefined type, such as Action video, Sports video, Comedy video, etc, and is associated with a list of keywords provided by the publishers. We assume each video is encoded with a constant bit rate *Br* (in kbps). A video is split into equal sized segments, and segment is the minimum unit that a peer can cache.

3.1.2 System Operations

In our proposed architecture, when a peer publishes a video, the video will be split into equal sized segments, and these segments will be distributed to peers according to our media segments distributing algorithm (Section 3.3). Once peers receive segments, they

will publish the received segments to the Category Overlay (Section 3.2). Note that during the segments distributing process, every segment will have a replica distributed to one of the seed peers.

When a peer (called a receiver) wants to watch a video, it first searches (Section 3.4) the 1st segment. Then it determines if the streaming request can be satisfied by the peers contained in the search results (including seed peers). If the answer is yes, it sends a message to the statistical server to notify it the requested video has been accessed (this only occurrs for the 1st segment), and then selfishly determines the best subset of supplying peers (Section 3.6.1) and applies the proposed multiple suppliers scheduling algorithm (Section 3.6.2) to aggregate bandwidths from the selected supplying peers and coordinate them to stream the 1st segment; otherwise, the request is rejected. When the streaming of the 1st segment is almost over, the receiver will do the same thing with the 2nd segment, the 3rd segment, and so on. Figure 3-1 shows an example. Suppose peer P_6 wants to watch a video whose playback bit rate is 500kbps. It searches for segment #0 and finds that P_1, P_2, P_3 have segment #0; it then selects P_1, P_2, P_3 as the supplying peers and aggregates bandwidths from them to stream segment #0. Segment #1 and #2 are streamed in the same way.

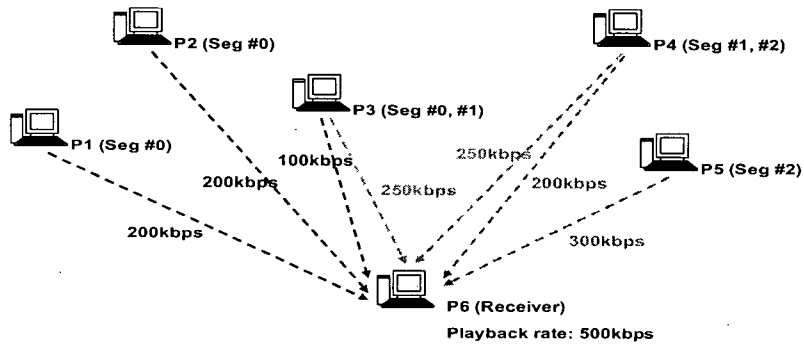


Figure 3-1 Example of watching a video

After the streaming of a segment is over, the receiver will cache the segment in its contributed storage (Section 3.5). For popular segments, as more and more requests come in, more segments will be cached on peers, thus more streaming requests can be supported. While for non-popular segments, since only a few peers may cache the segments, thus it is likely that there not exist peers with enough available outbound bandwidth to support the streaming. In this case, seed peers can offer their bandwidths to help the streaming session (recall that all the segments have a replica distributed to seed peers). Thus, in our architecture, the streaming requests for popular segments are more likely supported by regular peers, while the seed peers will more likely support the requests for non-popular segments. Since the requests for non-popular segments are rare, the traffic load in seed peers is within a reasonable range, which is verified by our simulation study (Section 5.2.2)

The following sections present the key components of our proposed architecture, including Category Overlay, segments distributing, searching and caching, and the multiple suppliers scheduling algorithm.

3.2 Category Overlay

In this section, we briefly introduce Category Overlay, which is chosen as the underlying search infrastructure in our architecture.

The basic idea of Category Overlay is to construct multiple category specific overlays on the unstructured peer-to-peer system and restrict a specific search within the corresponding overlay. In more detail, we first cluster the whole peer group into clusters. Then in each cluster, nodes[2] (called *Agent Nodes*) are selected to take charge of

[2] In this thesis, node and peer are used interchangeably.

predefined categories. The *Agent Node* is responsible for maintaining a keyword table (called *Content Index Table*) for all the content belonging to the categories it is in charge of. For a specific category, all of its *Agent Nodes* (in different clusters) are connected to form a category overlay. Thus, multiple category overlays can be constructed over the clusters.

Figure 3-2 shows an example of Category Overlay. As the figure shows, peers are clustered into three clusters: C_1, C_2, and C_3. In each cluster, nodes are selected to take charge of three predefined categories: Ca_1, Ca_2, and Ca_3. For example, in cluster C_1, node N_1 is in charge of category Ca_3; in cluster C_2, node N_2 is in charge of category Ca_3; and in cluster C_3, node N_3 is in charge of category Ca_3. Since nodes N_1, N_2, and N_3 are all *Agent Nodes* for category Ca_3, they are connected to form the category overlay O_3. Category overlay O_1 (for category Ca_1) and O_2 (for category Ca_2) can be formed in the same way. Thus we have three category overlays sitting on top of the clusters. Note that a node can take charge of more than one category. For instance, in the figure, Node N_1 takes charge of both category Ca_2 and Ca_3, so it participates in both category overlay O_2 and O_3.

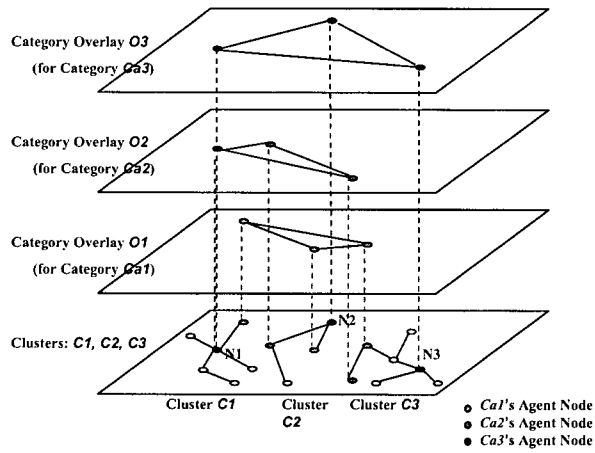


Figure 3-2 Example of Category Overlay

In Category Overlay, every cluster member node maintains a *Category Table*, which stores the Category-to-Agent mappings. It looks like a hash table in which the key is a category and the value is that category's *Agent Node*. When a node publishes content belonging to a specific category, it first looks up its *Category Table* to find that category's *Agent Node*, then it sends a "publish content" message to the found *Agent Node*, along with the keyword list (while the content is still in the owner's storage). Upon receiving this message, the *Agent Node* will store the keyword list in its *Content Index Table*. When a node unpublishes content belonging to a specific category, it first finds the category's *Agent Node*. Then it notifies the *Agent Node* to delete the corresponding record entry in the *Content Index Table*.

When a node issues a query, it should specify a category, as well as a list of keywords. The query will go to the *Agent Node*, which is in charge of that specified category. Then the corresponding *Agent Node* looks up its *Content Index Table* to find the content with the matched keywords, and returns the results to the query initiator. In addition, the *Agent Node* also needs to propagate the query within the corresponding overlay. Each *Agent Node* in this overlay will look up its *Content Index Table* and return the results to the query initiator. Compared to Gnutella [11], in which queries need to go through all the nodes, a query in Category Overlay just needs to be propagated within the corresponding overlay, which is much more efficient.

Note that in Category Overlay, each cluster is tree-based. The links between two cluster members are called *Cluster Links* (tree branches). Two neighbour clusters in Category Overlay are connected through *Inter-Cluster Links*.

To make this thesis self-contained, we briefly describe cluster construction and maintenance, as well as category overlay construction and maintenance in the following

sections. However, more detailed information about Category Overlay and its maintenance mechanisms, as well as the simulation-based performance evaluation, can be found in [26][43].

3.2.1 Cluster Construction

In Category Overlay, the cluster is tree-based, which has a central node (called *Core Node*), and all other member nodes are within *N* hops distance[3] from the *Core Node*. We call this *N* hops distance as the *Cluster_Range_Limit* and the hops distance from each member node to the *Core Node* as the node's *Range*. The simulation study in [26][43] shows that *Cluster_Range_Limit* = 2 results in a reasonable cluster size in a typical Power-Law topology network. Therefore, in this thesis, we discuss our clustering algorithm, assuming that *Cluster_Range_Limit* is set to 2. To describe our algorithm more clearly, we define the following technical terms:

Table 3-1 Definitions used in clustering algorithm

Terminology	**Description**
Core Node	Root of cluster tree (central node of cluster).
Master Node	Child of *Core Node*.
Slave Node	Child of *Master Node*.
Range	The hops distance from current node to *Core Node*.
Cluster_Range_Limit	The maximum hops distance from cluster member to *Core Node*.
Cluster Link	Tree branch, either connecting *Core Node* and *Master Node* or connecting *Master Node* and *Slave Node*.
Inter-Cluster Link	The link connecting different clusters.

Peers are clustered into clusters when they join in the system. Figure 3-3 illustrates

[3] In this thesis, all the hops distances are in the application level.

```
/* The first node of the whole peer group will become the Core Node for the
first cluster. */
/* When a node Nx wants to join the group, it will contact a node Ny, which
is already in the peer group. */

1:  join (Nx, Ny) {
2:      if (Ny's Range < Cluster_Range_Limit) {
3:          Nx joins in Ny's cluster, Ny will be Nx's parent;
4:          link (Nx, Ny) will be a Cluster Link;
5:      }
6:      else {   // Ny's Range >= Cluster_Range_Limit
7:          if (Nx knows other nodes in peer group) {
8:              Nx tries to contact other node to join;
9:          }
10:         else {
11:             create new cluster, Nx will be the Core Node;
12:             link (Nx, Ny) will be an Inter-Cluster Link;
13:         }
14:     }
15: }
```

Figure 3-3 Basic clustering algorithm

the pseudo code of our basic clustering algorithm. Furthermore, to ensure that the generated clusters have a reasonable and similar size, we use the following optimizations:

- *Cluster_Size_Limit:* cluster has a size limit. Once a cluster reaches this limit, it will reject any join request until some members leave. With this parameter, we can restrict the cluster size within a reasonable up-bound.

- *Cluster Size Full_Fraction:* when a node wants to join in a cluster but only knows boundary nodes (*Slave Nodes*), instead of being forced to create a new cluster, a boundary node can forward this request to its parent. If the cluster size is less than the *Full_ Fraction*, the node can join in this cluster. With this parameter, we increase the probability of a node joining in the existing cluster, thus decreasing the possibility of generating a small cluster. Our simulation result suggests that 0.9 is a good setting for *Full_Fraction*.

- *Core_Qualification:* a node that wants to be a *Core Node* should satisfy some

qualifications, such as powerful computing ability, high bandwidth, long stay period in the system, etc.

The simulation study in [26][43] shows that with the optimizations mentioned above, our clustering algorithm can produce reasonable and similar sized clusters.

3.2.2 Cluster Maintenance

When a participating peer of a cluster leaves or fails, the cluster is maintained as follows:

- *Peer Leave.* Leaving of a node N_Y may or may not affect other nodes, depending on its role in the cluster. If N_Y is a *Slave Node*, it can leave by only notifying its parent node. If N_Y is a *Master Node*, it should notify its parent node as well as all its children nodes. Upon receiving the notification, every child node picks up another *Master Node* in the cluster as its new *Master Node*. In each cluster, *Core Node* has several backup nodes. If the leaving node N_Y is a *Core Node*, it has to select a successor from the backup nodes before it leaves. The successor then notifies all the *Master Nodes* and confirms its new role. It also notifies its children nodes and converts them to *Master Nodes*.

- *Peer Failure.* To detect peer failure, every node in the cluster (except *Core Node*) periodically sends "alive" messages to its parent. If a parent node does not receive "alive" messages from its child for a period T_{alive}, that child node is identified as failure. The *Core Node* periodically sends "alive" messages to the backup nodes. If the backup nodes do not receive "alive" messages from the *Core Node* for a period T_{alive}, the *Core Node* is identified as failure. Once the peer failure is detected, the same actions as described in peer leave are performed, except that it is now the

parent node's duty to do the notification.

3.2.3 Category Overlay Construction

Before we discuss Category Overlay construction, we describe some of the technical terms and data structures used in Category Overlay as follows:

Table 3-2 Definitions used in Category Overlay

Terminology	Description
Category Agent Node	Node in charge of a certain category. It maintains the *Content Index Table* and *Neighbour Agents List* for that category. In this thesis, we simply call it *Agent Node*.
Category Overlay	Overlay network that consists of all the *Agent Nodes* of a certain category, as well as links among them.

Table 3-3 Data structures used in Category Overlay

Data Structures	Description
Content Index Table	Data structure that stores the keyword lists for all the contents of certain categories, within a cluster. Each entry in this table is a tuple: <*Category, Keyword list, Owner node*>. An entry <C_A, *KL*, N_X> means that node N_X has the content with the keyword list *KL* belonging to the category C_A. *Content Index Table* is only maintained at *Agent Nodes*.
Category Table	Local table that maps categories to their *Agent Nodes*. Each entry in this table is a tuple: <*Category, Agent Node, Timestamp*>. An entry <C_A, N_X, T_i> means that at time T_i, node N_X is believed to take charge of category C_A. Note that every category has a corresponding entry in this table and every node has this table.
Neighbour Agents List	For a specific category, this list stores all the neighbour clusters' *Agent Nodes*. An *Agent Node* and all the nodes contained in its *Neighbour Agents List* forms a *category overlay*. Each entry in this table is a tuple: <*Category, Agent Node List*>. An entry <C_A, {N_1, N_2, … , N_m}> means that nodes N_1, N_2, … , N_m are the neighbour clusters' *Agent Nodes* for category C_A. *Neighbour Agents List* is only maintained at *Agent Nodes*.

Category Overlay is constructed as nodes joining in the system. When a node N_X wants to join in the system, it first performs the clustering algorithm (described in Section 3.2.1) to find the cluster and parent to join in. There are three cases: (1) N_X is the first node of the system. In this case, a new cluster is created. N_X will be the *Core Node* and take charge of all of the categories. (2) N_X contacts node N_Y (in cluster C_Y), and finally it creates its own cluster C_X. As the case 1, node N_X will be the *Core Node* and it will take charge of all of the categories. Furthermore, N_X and N_Y will exchange their *Category Tables* through the *Inter-Cluster Link*. These two *Category Tables* will be propagated in cluster C_Y and C_X respectively. In more detail, *Category Table* from N_X will be propagated to all the *Agent Nodes* in cluster C_Y; *Category Table* from N_Y will be propagated to all the *Agent Nodes* in cluster C_X. Thus the *Agent Nodes* in cluster C_Y and C_X can update their *Neighbour Agents Lists* according to the propagated *Category Tables*. (3) N_X joins in an existing cluster. Once N_X joins in the cluster, the parent node will send back its *Category Table* to N_X. N_X will use this *Category Table* as its own *Category Table*. Furthermore, N_X's parent node will determine if it should migrate some of its categories to N_X (based on its current traffic load and N_X' bandwidth, computing power, etc.). If yes, N_X's parent node will migrate some of its categories to N_X, which means that N_X takes charge of these categories.

3.2.4 Category Overlay Maintenance

In Category Overlay, when the *Agent Node* of a specific category leaves or fails, the Category Overlay is maintained as follows:

- *Agent Node Leave.* Before leaving, the leaving *Agent Node* selects a stable yet under-loaded cluster member node as the successor and migrates all of its

categories to that node. Then, the leaving *Agent Node* follows the steps described in Section 3.2.2 to leave.

- *Agent Node Failure.* In Category Overlay, the *Agent Node* maintains *Content Index Table* and *Neighbour Agents List*, which are crucial data structures for a search. If an *Agent Node* fails, these data structures are lost, which decreases search efficiency. To cope with this, each *Agent Node* selects a cluster member node as its backup node. The data structures on backup node are updated based on the frequency of contents publishing. When the failure of an *Agent Node* is detected, its backup node takes over the responsibility and selects another node to be the backup node.

Besides *Agent Node* leave and failure, another issue to be addressed in Category Overlay is the *Category Table* inconsistency problem. Recall that in Category Overlay, when a node joins in the system or an *Agent Node* leaves the system, category migration will occur. However, only the nodes participating in the migration know the change, while all other cluster members do not know. Thus, inconsistency between nodes' *Category Tables* is inevitable. If the environment is very dynamic, then the inconsistency level could quickly rise to a point where looking up *Category Table* may even slow down the searching.

To solve this inconsistency, we introduce a periodical aggregation report scheme, in which each node periodically sends a category update report to its randomly selected neighbour. This report contains the latest N updates (or category migration events) known to the reporter, as well as M random entries in the reporter's *Category Table*. Upon receiving the report, a node needs to update its own *Category Table*, based on the accompanied timestamps. The time interval between two reports is a local decision and

depends on the updating frequency. The simulation study in [26][43] shows that this periodical aggregation report scheme can maintain the *Category Table* consistency at a relatively high level with an acceptable overhead.

3.3 Media Segments Distributing

In BitVampire, when a peer publishes a video, the video will be split into equal-sized segments and distributed to different peers. Distributing segments to different peers has several advantages, including: (1) After segments are distributed to peers, the published video is stored by the network, not the single publisher peer. Thus if the publisher peer or some hosting peers of segments leave or fail, the rest of the segments can still be accessed, which increases service availability. (2) Since the segments of a published video are hosted by different peers, the streaming request of a video is supported by different peers at different stages, which reduces the streaming burden on a single peer.

As mentioned in Section 3.1.1, the target environment of our proposed architecture is P2P networks, which are quite heterogeneous. Typically, participating peers offer different resources and may leave at any time. Taking all of these into consideration, we propose a Media Segments Distributing (MSD) algorithm to distribute segments to peers. In the following sections, we first present and discuss our media segments distributing algorithm in detail, then describe a design improvement.

3.3.1 Media Segments Distributing Algorithm

In our proposed architecture, every participating peer contributes some of its outbound bandwidth and storage to the system. The outbound bandwidth and storage peer P_i

contributes are denoted as Bw_i and St_i, and the available outbound bandwidth and storage peer P_i can provide at a specific time are denoted as $Bw^{avail}{}_i$ and $St^{avail}{}_i$. Initially, $Bw^{avail}{}_i = Bw_i$, $St^{avail}{}_i = St_i$ and $Bw^{avail}{}_i \leq Bw_i$, $St^{avail}{}_i \leq St_i$ hold at any time. In addition, peer P_i estimates its stay time in the system by computing the smoothed weighted average as follows and uses this value to represent its stability.

$$EstimatedStay_i = \alpha \times EstimatedStay_i + \beta \times CurrentStay_i \quad (3\text{-}1)$$

where $EstimatedStay_i$ is the estimated stay time of peer P_i, taking into account all the stay history of P_i, and $CurrentStay_i$ is the time period peer P_i participated in the system since its last leave or failure. $\alpha + \beta = 1$, α is between 0.8 and 0.9, and β is between 0.1 and 0.2. Besides, peer P_i also maintains the average usage ratio of its contributed bandwidth since it participated in the system, called $R^{usage}{}_i$, and the frequency it serves streaming requests in the recent period, called $Freq^{serve}{}_i$.

When a peer wants to publish a video, it will split the video into equal-sized segments. The workload analysis of today's enterprise media server [14] found that most clients only watch the first several minutes of media files. To benefit from this fact, we let the first segment have several replicas. The reasons why we choose only the first segment to be replicated are as follows: (1) In our simulation, the length of a segment is set to 5 minutes (We believe 5 minutes or longer is a reasonable setting for the length of segment, because too small segments will result in too many contents published to the Category Overlay, thus increasing the maintenance overhead and search traffic of the system), and the analysis study in [4] reported that more than 60% of streaming sessions last less than 5 minutes. Therefore, the possibility of peers requesting the rest of segments (except the first segment) is small. (2) Recall that every segment of published videos has

a replica being distributed to seed peers; thus we can always find a specific segment even if this segment does not have replicas and its hosting peer leaves or fails.

Suppose the video is split into N_s segments, and the first segment has N_f replicas, then in total N_s+N_f segments need to be distributed to peers. The publisher will broadcast a "publish video" message to its cluster members through *Cluster Links*, and this message will be propagated to other clusters through *Inter-Cluster Links*. After receiving this message, the peer will send an "accept segment" message back to the publisher, along with its *EstimatedStay*, *Bw*, R^{usage}, and $Freq^{serve}$. The publisher waits for $Timeout_p$ to collect the sent-back "accept segment" messages. After receiving such a message, it marks the sender as a candidate and collects information contained in the message. After $Timeout_p$, the publisher will assign segments to candidates.

Before discussing our media segments distributing algorithm, we define $G^{St}{}_i$, the goodness of candidate peer P_i to store a segment/replica as a function of its $EstimatedStay_i$, Bw_i, $R^{usage}{}_i$, and $Freq^{serve}{}_i$. Suppose there are m candidate peers: $\{P_1, P_2, \ldots, P_m\}$, $G^{St}{}_i$ is defined as follows:

$$G^{St}{}_i = \alpha_{St} \times \frac{EstimatedStay_i}{\max_{1\le i\le m}\{EstimatedStay_i\}} + \beta_{St} \times \frac{Bw_i \times (1-R_i^{usage})}{\max_{1\le i\le m}\{Bw_i \times (1-R_i^{usage})\}} - \gamma_{St} \times \frac{Freq^{serve}{}_i}{\max_{1\le i\le m}\{Freq^{serve}{}_i\}} \quad (3\text{-}2)$$

where α_{st}, β_{st}, γ_{st} are the factors to give $EstimatedStay_i$, $Bw_i \times (1-R^{usage}{}_i)$, $Freq^{serve}{}_i$ different weights and $\alpha_{st}+\beta_{st}+\gamma_{st}=1$. The values of α_{st}, β_{st}, and γ_{st} depend on the application environment. If the P2P networks are quite unreliable, with peers leaving or failing very frequently, then a bigger value should be assigned to α_{st}; if every peer contributes only a few of its outbound bandwidth, thus the total bandwidth capacity of the system is limited, then assigning a bigger value to β_{st} would be appropriate; finally, if the streaming requests are frequent, it should be better to set γ_{st} to a bigger value. In our simulation, α_{st}

is set to 0.35, β_{st} is set to 0.25, and γ_{st} is set to 0.4. Given this formulation, the more stable candidate peer with a higher average available bandwidth and lower streaming serve frequency will have a greater G^{St}.

Figure 3-4 is the pseudo code of our Media Segments Distributing (MSD) algorithm. We first compute each candidate's G^{St}, then sort the candidates by G^{St} in descending order and store the results in the *candidateList*. The media segments distributing algorithm will take this list as its input. Note that the algorithm tends to assign media segments to the candidate peers that have higher G^{St}, which means these peers will take more responsibility to serve streaming requests. However, their $Freq^{serve}$ will increase as the streaming requests come in, thus decreasing their G^{St}. When another video is published, it is likely that their G^{St} will be exceeded by others, so that video's

```
/* In this algorithm, we do not differentiate between the original segment and
   its replica; they are referred same as segment. */

Input:
    candidateList :     the candidate list sorted by G^St in descending order;
    num_candidates :    number of candidates;
    num_segs :          number of segments;
    num_replicas :      number of replicas for the first segment;

Assigning:
 1:  j = 1;
 2:  for (i = 0; i < num_segs + num_replicas; i++) {
 3:        select j^th node in candidateList, suppose the selected node is N_k;
 4:
 5:        if (i < num_replicas + 1)
 6:              assign segment 0 to node N_k;
 7:        else
 8:              assign segment (i - num_replicas) to node N_k;
 9:
10:        if (j == num_candidates)
11:              j = 1;
12:        else
13:              j++;
14: }
```

Figure 3-4 Media segments distributing (MSD) algorithm

segments will be distributed to other peers. In the long term, this could result in load balance in peers to some extent.

Once the segments assignment is done, the publisher will send segments (in the rest of this thesis, we do not differentiate between the original segment and its replica; they are referred same as segment) to peers. When a peer receives a segment, it checks if there is enough storage available ($St^{avail} \geq seg\_size$, where *seg_size* is the size of a segment). If yes, it stores the received segment and decreases its St^{avail} as follows: $St^{avail} = St^{avail} - seg\_size$; otherwise, it uses LRU (Least Recently Used) algorithm to select a victim segment to replace.

3.3.2 Distributing Algorithm Analysis

This section gives a time analysis of our media segments distributing algorithm. As illustrated in Figure 3-4, the algorithm has a loop (line 2) and the loop body (line 3-13) will be executed (N_s+N_f) times, where N_s denotes the number of segments and N_f denotes the number of replicas for the first segment. Within the loop body, the j^{th} node in the *candidateList* is selected. Since the *candidateList* is sorted and j is always less than or equal to (N_s+N_f), the time for selecting the j^{th} node from the *candidateList* is bounded by $O(N_s+N_f)$. Thus the time for assigning segments to candidate peers is bounded by $O((N_s+N_f)^2)$. The algorithm needs the *candidateList* to be sorted. Suppose that there are totally m candidate peers and quick sort is used to sort them by their G^{St}; the time for sorting is $O(m \cdot \log m)$. Thus the total time of our media segments distributing algorithm is $O(m \cdot \log m + (N_s+N_f)^2)$, where m is the number of the candidate peers, N_s is the number of segments of the publishing video, and N_f is the number of the first segment's replicas.

3.3.3 Design Improvement

The algorithm analysis in the previous section does not consider the communication cost. However, during the video publishing process, the communication cost could be high. Because the "publish video" messages will be broadcasted to every cluster member and be propagated to other clusters, and every node receiving this message will send back an "accept segment" message, which (1) imposes lots of communication traffic on the system; (2) requires the publisher to wait a long period to receive enough "accept segment" messages and collect information of candidate peers; and (3) increases the traffic load on the publisher, since it will receive a large number of messages sent back by candidate peers. To cope with this, we revise our approach to collect information of candidate peers.

Recall that in the cluster maintenance (Section 3.2.2), to detect peer failure, every peer periodically sends "alive" messages to its parent. We let every peer send its *EstimatedStay*, *Bw*, R^{usage}, and $Freq^{serve}$ along with the "alive" message. The parent collects information contained in the received "alive" messages and periodically sends an aggregate report to its parent, along with the "alive" message. Thus, eventually, *Core Node* will have recent information of every cluster member. *Core Node* sorts the cluster members by their G^{St} in descending order and stores the result in a sorted candidates list. *Core Node* periodically maintains the sorted candidates list based on the renewed information of cluster members. When a peer publishes a video, it sends a "publish video" message to its cluster's *Core Node*, and this message will be propagated to the *Core Nodes* of other clusters. After receiving this message, the *Core Node* will select the first N_c ($N_c > N_s + N_f$) peers from the sorted candidates list and send the information of these peers back to the publisher. The publisher waits for $Timeout_p$ to receive the messages sent

back by the *Core Nodes* and collects information of the candidate peers. Then it follows the steps mentioned in Section 3.3.1 to distribute segments to peers.

Our revised approach assigns more responsibilities to *Core Nodes* during the video publishing process. However, *Core Nodes* are typically the most powerful and stable nodes in the clusters; thus it is appropriate to assign more responsibilities to them.

3.4 Media Segments Searching

After segments are distributed to peers, those peers will publish received segments to Category Overlay. As mentioned in Section 3.2, to publish content in Category Overlay, a node should specify the category the content belongs to, as well as a keyword list. In our proposed architecture, we define the categories as follows: (1) we first predefine the video types, such as Action video, Sports video, Comedy video, etc; (2) then we combine the video type and segment number as the category, such as Action-0, Action-1, Sports-0, etc. So all the first segments of the Action videos belong to the Action-0 category; all the second segments of the Action videos belong to the Action-1 category, and so on.

Note that when a peer publishes a video, it should specify the video type and provide a list of keywords. When the publisher distributes segments to peers, the specified video type and keyword list will be sent to peers as well. When a peer publishes the received segment, it will use the combination of video type and segment number as the segment's category, and use the received keyword list as the publishing keywords. In addition, each published segment has a *videoID* to specify which video it comes from (Recall that every video has a unique *videoID*); thus, when searching, we can use this *videoID* to ensure that the found segments come from the same video. After the segments

have been published, we can search the desired segments in the same way described in Section 3.2.

3.5 Media Segments Caching and Seed Re-Distributing Mechanism

In BitVampire, once a peer finishes watching a segment, it will cache this segment in its contributed storage. We use this cache policy because we believe that the possibility of a peer re-watching this segment is relatively higher than others. However, the storage contributed by a peer is limited, so we adopt LRU (Least Recently Used) as the cache replacement algorithm, in which the least recently used segment will be chosen as the victim if there is not enough available storage for the new cached segment. Note that when a segment is cached in peer P_i, peer P_i will publish the cached segment into the Category Overlay, while when a segment is chosen as the cache replacement victim, its hosting peer will unpublish it from the Category Overlay.

As mentioned in Section 3.1.1, when the streaming requests cannot be satisfied by regular peers, seed peers will offer their bandwidths to help serve the streaming session. To alleviate the streaming traffic load on seed peers, we propose a Seed Re-Distributing (SRD) mechanism, in which when the seed peer offers help to stream a segment, it will distribute a replica of that segment to peers, thus decreasing the future demand on seed peers. However, two issues need to be addressed to make this mechanism feasible: (1) Which segment served by seed peers should be re-distributed to peers? Should all the segments served by seed peers be re-distributed, or should it be selective? (2) Which peer should the segment be re-distributed to?

For the first issue, the simplest solution is to re-distribute all the segments served by seed peers. However, this is not a good approach. As discussed in Section 3.1.2, most of the segments served by seed peers are non-popular segments. The possibility of peers re-watching these segments is small. If these segments are re-distributed to peers, it does not provide much help to increase the streaming capability of the whole system, but wastes lots of seed peer bandwidth. To cope with this, we classify the segments by their importance. If a segment's importance exceeds a threshold Th_{dist}, that segment will be re-distributed to peers by seed peer. For a segment that belongs to video k, suppose its segment number is i (the $(i+1)^{th}$ segment of the video), its importance Imp^k_i is defined as follows:

$$Imp^k_i = Pop_k \times 1/\alpha_{Im}^{\sqrt{i}} \tag{3-3}$$

where Pop_k is the popularity of video k, which can be acquired from the statistical server. α_{Im} is the factor to give segment different weight based on its position in the video. In our simulation, α_{Im} is set to 1.5. Given this formulation, the segment that comes from the popular video and is split from the initial portion of the video will have a bigger importance, which means it is more likely to be re-distributed to peers. The value for threshold Th_{dist} could be pre-set or dynamically adapted based on the load on seed peers. In our simulation, we use pre-set and the threshold Th_{dist} is set at 0.6.

For the second issue, we use our proposed media segments distributing (MSD) algorithm (Section 3.3.1) to re-distribute the segment, if that segment is decided to be re-distributed. Our simulation study (Section 5.2.2) verified the effectiveness of this mechanism, which can reduce traffic load on seed peers.

3.6 Media Segments Streaming

In BitVampire, when a peer (called receiver) wants to watch a video, it will search the 1st segment, then aggregate bandwidths from the selected supplying peers and coordinate them to stream the 1st segment. When the streaming of the 1st segment is almost over (enough time should be left for searching and generating schedule for next segment), the receiver will do the same thing with the 2nd segment, the 3rd segment, and so on. Aggregating bandwidths from multiple supplying peers has several advantages, including: (1) Since peers are quite heterogeneous, a single peer may not have enough bandwidth to support a streaming session. In this case, aggregating bandwidths from multiple supplying peers is necessary. (2) Aggregating bandwidths from multiple supplying peers increases the robustness of a streaming session, since if some of supplying peers leave or fail, other supplying peers still contribute their bandwidths to the session.

In the following sections, we first describe how to select supplying peers from the candidates that are returned by searching. Then, we present and discuss our multiple suppliers scheduling algorithm in detail.

3.6.1 Supplying Peers Selection

When a peer (receiver) searches the desired segments for watching, the size of the results could be large. Thus we need a scheme to select supplying peers from the search results. We let the receiver selfishly determine the best subset of supplying peers. Details of our scheme are presented below.

After receiving the search results, the receiver will send an enquiry message to each peer contained in the results. Upon receiving this message, a peer will send a reply

message back to the receiver, along with its Bw^{avail} and *EstimatedRTT*, where *EstimatedRTT* is the estimated round trip time between the peer and the receiver. The receiver waits $Timeout_e$ to get the reply messages and collect information contained in the messages. After $Timeout_e$, the receiver will select the subset of supplying peers based on their G^{Sp}, the goodness of the peer to become supplier. Suppose there are m candidate peers: $\{P_1, P_2, \ldots, P_m\}$, the $G^{Sp}{}_i$ for a peer P_i is defined as follows:

$$G^{Sp}{}_i = \alpha_{Sp} \times \frac{Bw^{avail}{}_i}{\max_{1 \le i \le m}\{Bw^{avail}{}_i\}} - \beta_{Sp} \times \frac{EstimatedRTT_i}{\max_{1 \le i \le m}\{EstimatedRTT_i\}} \qquad (3\text{-}4)$$

where α_{Sp}, β_{Sp} are the factors to give $Bw^{avail}{}_i$, $EstimatedRTT_i$ different weights. Setting a bigger value to α_{Sp} gives preferences to peers which have higher available bandwidth, while setting a bigger value to β_{Sp} tends to bias selection towards nearby peers. We believe giving a bigger weight to $EstimatedRTT_i$ is more appropriate. Because usually $EstimatedRTT_i$ reflects the distance between the receiver and the candidate peer; a smaller $EstimatedRTT_i$ indicates savings in the backbone bandwidth and less susceptibility to network congestion, since traffic passes through fewer routers. Therefore, in our simulation, α_{Sp} is set to 0.4, and β_{Sp} is set to 0.6. Given this formulation, candidate peer which is nearer to the receiver, has a higher available bandwidth will have a greater G^{Sp}.

The receiver will select M (in our simulation, $M = 3$) candidate peers that have the greatest G^{Sp} as the suppliers, as long as the aggregated available bandwidth from these peers is bigger than or equal to the video playback bit rate. Otherwise, more than M peers will be selected to meet the playback bit rate requirement. The unselected candidate peers will be kept in a *standby* set, from which substitute peers can be selected in case suppliers leave or failure. If the aggregated available bandwidth from all of the candidate peers is less than the playback bit rate, the segment watching request will be rejected.

After supplying peers have been selected, the receiver will reserve bandwidths from them. Suppose M supplying peers (P_1, P_2, ..., P_M) are selected, and the video playback bit rate is Br. The receiver will reserve bandwidth Bw^r_i (the reserved bandwidth should be in multiple of bandwidth reservation unit Bw^r_u. In our simulation, Bw^r_u is set to 64kbps.) from supplier P_i in proportion to its G^{Sp}, and satisfy the following condition:

$$\sum_{i=1}^{M} Bw^r_i = Br \tag{3-5}$$

Then the receiver will send a "reserve bandwidth" message to each supplier. After receiving this message, supplying peer P_i will decrease its Bw^{avail} by Bw^r_i. When the streaming session supplied by peer P_i is over, P_i will increase its Bw^{avail} by Bw^r_i.

Note that by "reserve bandwidth", we do not mean that the bandwidth Bw^r_i from peer P_i is actually reserved and can not used by other applications. The current Internet does not provide resource reservation service, thus the bandwidth contributed by supplying peer P_i may fluctuate during the streaming session. In our architecture, a receiver reserves bandwidth Bw^r_i from supplying peer P_i, it only means that peer P_i decreases its Bw^{avail} by Bw^r_i. In another word, bandwidth reservation process in our architecture only controls whether and how much a peer contributes to a streaming session, it does not guarantee that the bandwidth is actually reserved.

3.6.2 Multiple Suppliers Scheduling Algorithm

To fully use the aggregated bandwidths from multiple supplying peers, we want different suppliers to send different portion of a segment to the receiver at the same time. So we further divide each segment into equal sized blocks, in which each block contains T_{blk}

seconds of video content. Thus the receiver can parallel download different blocks from different supplying peers in real-time model. The value for T_{blk} depends on the bit rate at which the video is encoded. For a video which is encoded at bitrate 512 kbps, $T_{blk} = 1$ is an appropriate seeting, since 1 second vidoe content has about 64KB data, which can be sent by a few of UDP packets. In our simulation, T_{blk} is set to 1.

Given a set of supplying peers $\{P_1, P_2, \ldots, P_M\}$ and the reserved bandwidths from these peers $\{Bw^r_1, Bw^r_2, \ldots, Bw^r_M\}$, the problem is how to assign blocks to these supplying peers to send. A possible solution is Round Robin (RR), where blocks are assigned to suppliers in a round robin fashion. However, RR treats each supplier equally, no matter how much bandwidth it contributes to the streaming session. Thus some bandwidth contributed from the powerful peers could be wasted. To cope with this, another possible solution is assigning blocks to suppliers in proportion to their contributed bandwidths. Thus supplier P_i sends Bw^r_i/Br blocks, starting from whatever P_{i-1} ends. This approach fully used the bandwidth from each supplier. However, for the blocks at the beginning of a segment, only P_1 contributes its bandwidth, which inevitably results in a long initial buffering time. Taking all of these into consideration, a good schedule should be the one in which blocks are assigned to suppliers in a roughly round robin manner, and also in proportion to the bandwidths contributed by suppliers.

Based on the discussion above, we propose a Multiple Suppliers Scheduling (MSS) algorithm, which assigns blocks to different suppliers to send. The algorithm is executed by the receiver to generate the schedule. Figure 3-5 illustrates the pseudo code of the algorithm. The suppliers are sorted by their Bw^r in descending order. For a supplier *supplier[i]*, *time_left[i]* indicates the time left for it to send blocks. Initially, *time_left[i]* is set to the *deadline* for suppliers to cooperate to finish sending all of the blocks. We

assign blocks to suppliers starting from the last block to the first block. To assign a block *blocks[curr_blk]*, we first find the maximum *time_left* value across all the suppliers and store it in a variable *max_t*. Then we iterate through the suppliers in the sorted order and check if current supplier *supplier[i]*'s *time_left[i]* is equal to *max_t*. If yes, *blocks[curr_blk]* is assigned to *supplier[i]* and we subtract blk_size/Bw'_i (the time for *supplier[i]* to finish sending a block) from *time_left*[i]. At this point, *blocks[curr_blk]* is assigned. We repeat the same procedure to assign *blocks[curr_blk-1]*, *blocks[curr_blk-2]*, and so on, until we finish assigning the first block.

Figure 3-6 illustrates an example of assigning 8 blocks to suppliers using MSS.

```
Input:
    num_suplliers :    number of suppliers;
    supplier[i] :      the suppliers sorted by Bw' in descending order;
    Bw'[i] :           reserved bandwidth at supplier[i];
    num_blks :         number of blocks;
    blocks[i] :        blocks;
    blk_size :         block size;
    deadline :         deadline for suppliers to cooperate to finish sending all of the blocks;

Scheduling:
 1: for (i = 1; i ≤ num_suppliers; i++) {
 2:     time_left[i] = deadline;   // time_left[i]: time left for supplier[i] to send blocks;
 3: }
 4: curr_blk = num_blks-1;
 5: while (curr_blk ≥ 0) {
 6:     max_t = max {time_left[1], ... , time_left[num_suppliers]};
 7:     for (i = 1; i ≤ num_suppliers; i++) {
 8:         if (time_left[i] == max_t) {
 9:             assign blocks[curr_blk] to supplier[i];
10:             time_left[i] = time_left[i] – blk_size / Bw'[i];
11:             curr_blk --;
12:         }
13:         if (curr_blk < 0)
14:             break;
15:     }
16: }
```

Figure 3-5 Multiple suppliers scheduling (MSS) algorithm

Suppose *Br* (the playback bit rate of the video) is 512kbps and T_{blk} is 1 (a block contains 1 second of the video content). There are 3 suppliers contribute their bandwidths, in which P_1 contributes 320kbps (5/8·*Br*), P_2 contributes 128kbps (1/4·*Br*), and P_3 contributes 64kbps (1/8·*Br*). The *deadline* for suppliers to finish sending blocks is set to 11 second (how to set the *deadline* is detailed in next section).

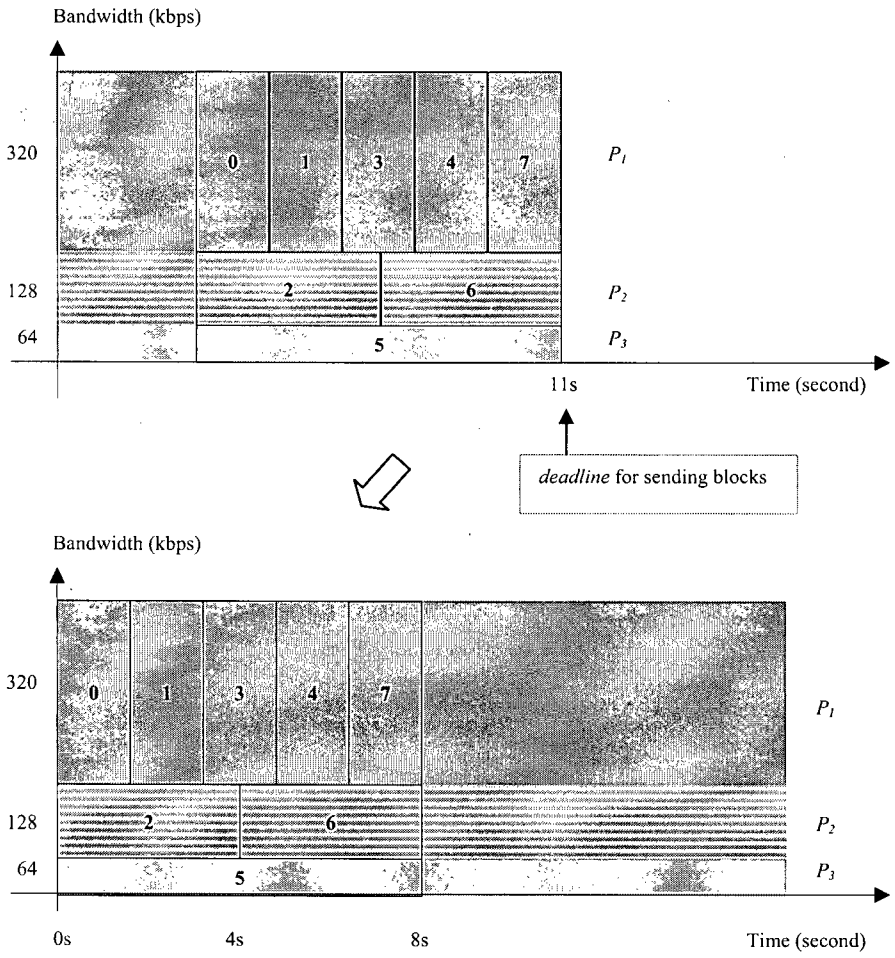


Figure 3-6 Example of assigning 8 blocks to suppliers using MSS

As the figure shows, using MSS, blocks are assigned to suppliers in proportion to their contributed bandwidths (5 blocks are assigned to P_1, 2 blocks are assigned to P_2, and 1 block is assigned to P_3), thus the bandwidth from each supplier is fully used. Totally it takes 8 seconds for the suppliers to cooperate to finish sending all of the 8 blocks. Compared to this, RR takes 16 seconds to finish sending the blocks (illustrated in Figure 3-7). RR treats each supplier equally, thus P_1 gets 3 blocks to send, P_2 gets 3, and P_3 gets 2. Obviously, some of the bandwidth contributed from P_1 is wasted and P_3 should take less blocks.

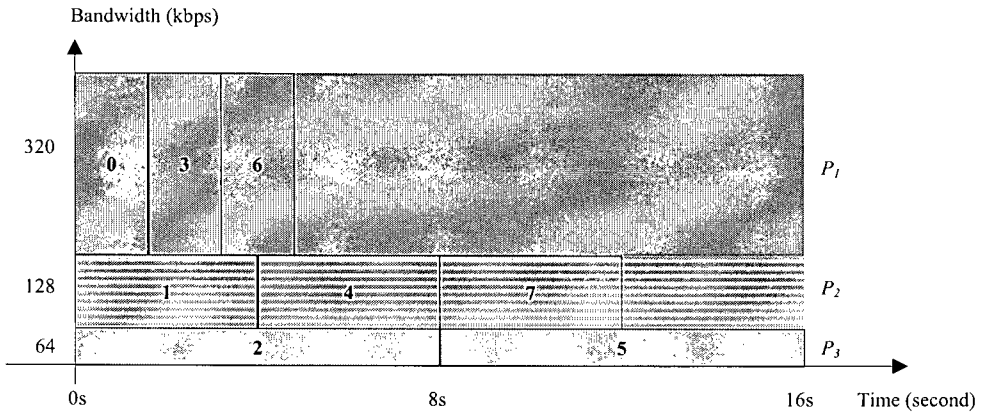


Figure 3-7 Example of assigning 8 blocks to suppliers using RR

3.6.3 Scheduling Algorithm Discussion and Optimization

In this section, we first discuss how to set the *deadline* for suppliers to finish sending blocks. Then, we explain why we assign blocks starting from the last block to the first block. Finally, we describe an optimization on MSS algorithm, which results in small finish time for sending blocks in some scenarios.

Before discussing the approach to set the *deadline*, we first describe the concept of *initial buffering*. During the streaming session, some of the suppliers may leave or fail at any time, and the incoming streaming rates from the suppliers may decrease due to the network congestion. In these cases, the receiver will select a substitute supplying peer from the *standby* set to replace the leaving/failing supplier or the supplier whose sending rate is decreasing. We call this *supplier switching*. During the *supplier switching* period, the aggregate received rate is less than the required playback bit rate, thus the receiver may experience buffer underflow. To cope with this, we require the receiver to buffer at least $S_{initBuff}$ blocks before the playback starts. This is called *initial buffering*. Given $S_{initBuff}$, the *deadline* is set as follows:

$$deadline = (S_{initBuff} + num\_blks) \times T_{blk} \tag{3-6}$$

where *num_blks* is the number of blocks needed to be assigned to suppliers to send.

As illustrated in Figure 3-5, MSS algorithm assigns blocks to suppliers starting from the last block to the first block. Another possible approach is to assign blocks starting from the first block, which is illustrated in Figure 3-8. However, if the blocks are assigned starting from the first block, some initial blocks may be assigned to the suppliers that contribute little bandwidths, thus inevitably increases the *initial buffering time* (defined as the time to finish downloading the initial $S_{initBuff}$ blocks). Figure 3-9 shows the example of assigning blocks starting from the first block. As shown in the figure, block 2 is assigned to supplier P_3, which contributes only 64kbps bandwidth. Suppose $S_{initBuff}$ is 3, then the *initial buffering time* is 8 seconds. However, as shown in Figure 3-6, if the blocks are assigned starting from the last block, the *initial buffering time* is only 4 seconds. Reducing *initial buffering time* is important, because long *initial buffering time* means that the receiver will suffer long waiting period before the video playback starts.

Input:

num_suplliers : number of suppliers;
supplier[i] : the suppliers sorted by Bw' in descending order;
$Bw'[i]$: reserved bandwidth at *supplier[i]*;
num_blks : number of blocks;
blocks[i] : blocks;
blk_size : block size;

Scheduling:

```
1:  for (i = 1; i ≤ num_suppliers; i++) {
2:      time_start[i] = 0;   // time_start[i]: the time at which supplier[i] can start sending blocks;
3:  }
4:  curr_blk = 0;
5:  while (curr_blk < num_blks) {
6:      min_t = min {time_start[1], … , time_start[num_suppliers]};
7:      for (i = 1; i ≤ num_suppliers; i++) {
8:          if (time_start[i] == min_t) {
9:              assign blocks[curr_blk] to supplier[i];
10:             time_start[i] = time_start[i] + blk_size / Bw'[i];
11:             curr_blk++;
12:         }
13:         if (curr_blk ≥ num_blks)
14:             break;
15:     }
16: }
```

Figure 3-8 Algorithm for assigning blocks starting from the first block

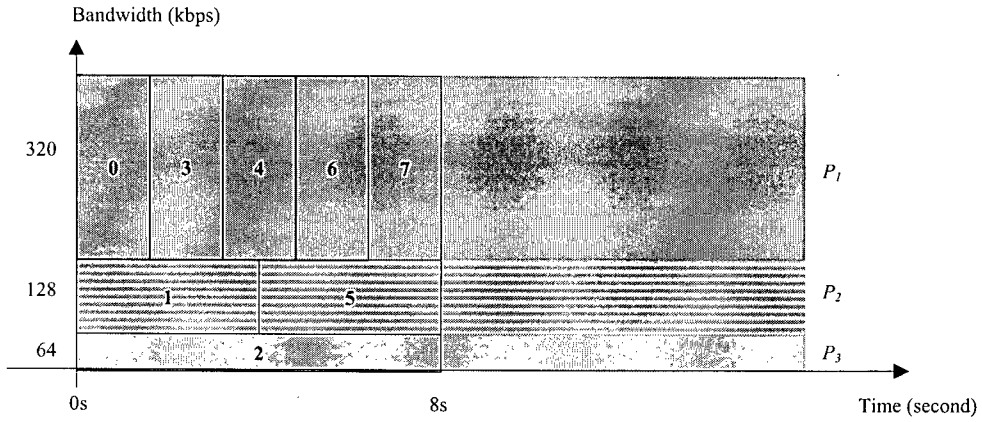


Figure 3-9 Example of assigning 8 blocks starting from the first block

Consider the following schedule scenario, in which 11 blocks need to be assigned to suppliers and the *deadline* for suppliers to finish sending blocks is set to 17 second ($S_{initBuff}$ is set to 6). Other settings are same as the example in Section 3.6.2. Figure 3-10 shows the schedule generated from MSS algorithm.

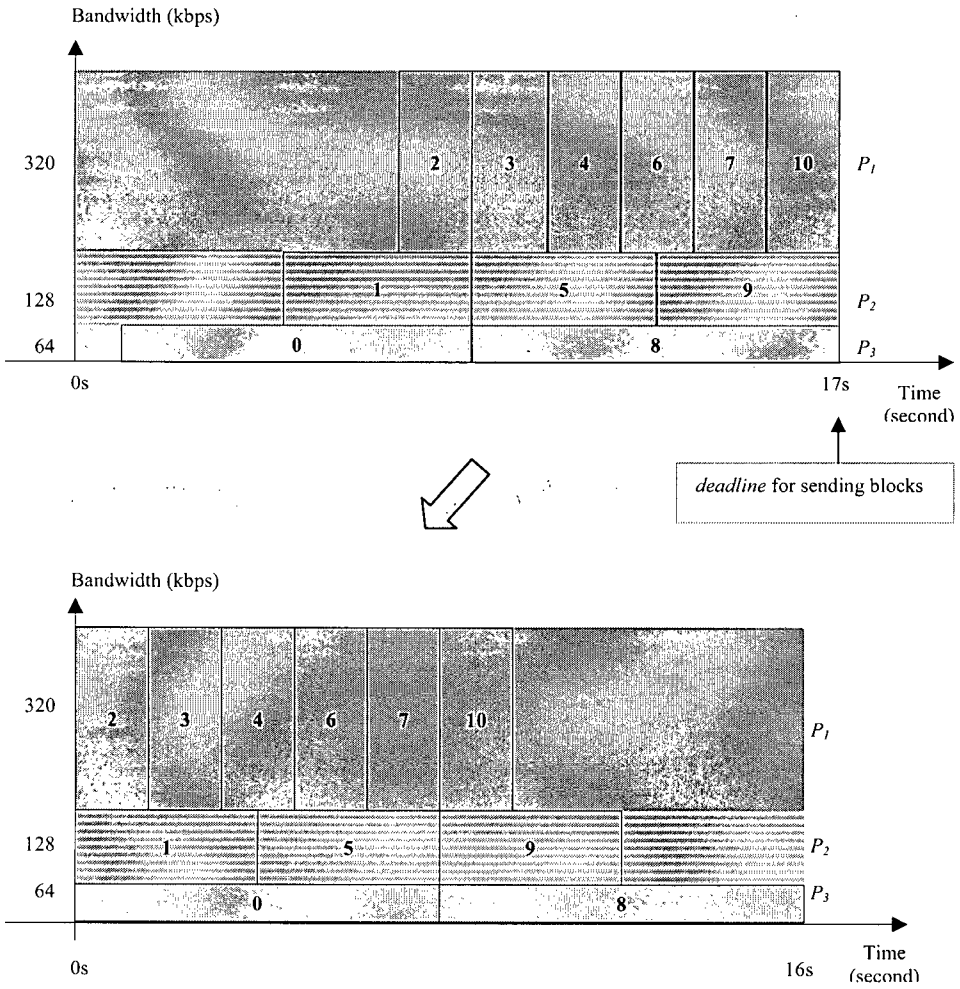


Figure 3-10 Example of assigning 11 blocks to suppliers using MSS

Obviously, if block 0 is assigned to P_1, the total sending time will be reduced from 16 seconds to 12 seconds, which is illustrated in Figure 3-11. Based on this observation, we revised our MSS algorithm. Figure 3-12 illustrates the pseudo code of the revised algorithm. We add an optimization stage, at which the algorithm checks whether moving a block from the supplier that contributes less bandwidth to the supplier that contributes

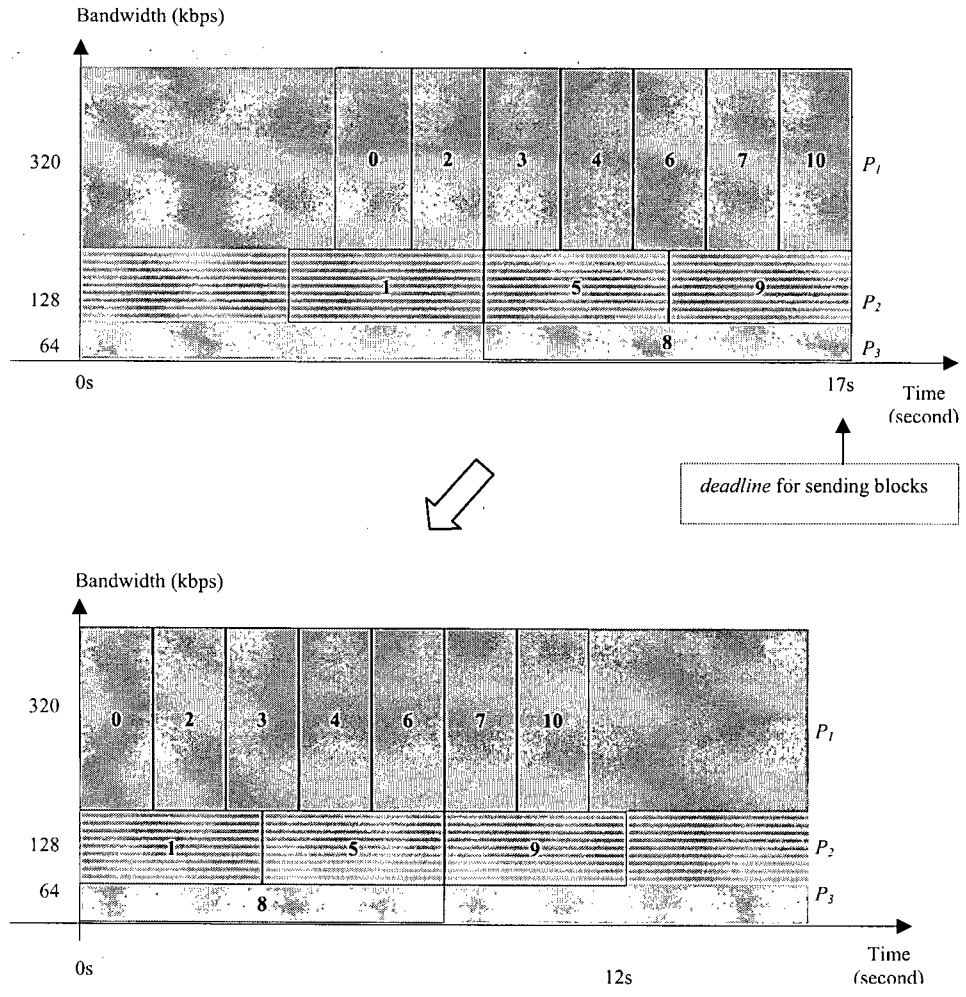


Figure 3-11 Example of assigning 11 blocks to suppliers using revised MSS

more bandwidth will reduce the sending time. If yes, the block is assigned to the supplier that contributes more bandwidth.

```
Input:
 num_supllier's :   number of suppliers;
 supplier[i] :      the suppliers sorted by Bw^r in descending order;
 Bw^r[i] :          reserved bandwidth at supplier[i];
 num_blks :         number of blocks;
 blocks[i] :        blocks;
 blk_size :         block size;
 deadline :         deadline for suppliers to cooperate to finish sending all of the blocks;
Scheduling:
 1: for (i = 1; i ≤ num_suppliers; i++) {
 2:    time_left[i] = deadline; // time_left[i]: time left for supplier[i] to send blocks;
 3: }
 4: curr_blk = num_blks-1;
 5: while (curr_blk ≥ 0) {
 6:    max_t = max {time_left[1], ... , time_left[num_suppliers]};
 7:    for (i = 1; i ≤ num_suppliers; i++) {
 8:       if (time_left[i] == max_t) {
 9:          assign blocks[curr_blk] to supplier[i];
10:          time_left[i] = time_left[i] – blk_size / Bw^r[i];
11:          curr_blk --;
12:       }
13:       if (curr_blk < 0)
14:          break;
15:    }
16: }
Optimizing:
17: touched = true;
18: while (touched) {
19:    touched = false;
20:    min_t = min {time_left[1], ... , time_left[num_suppliers]};
21:    min_pos = k; suppose time_left[k] is the min time_left across all the suppliers.
22:    max_tadj = min_t;
23:    max_pos = min_pos;
24:    for (i = 1; i ≤ num_suppliers; i++) {
25:       if (i == min_pos)
26:          continue;
27:       tmp_t = time_left[i] – blk_size / Bw^r[i];
28:       if (tmp_t > max_tadj) {
29:          max_tadj = tmp_t;
30:          max_pos = i;
31:       }
32:    }
33:    if (max_pos != min_pos) {
34:       touched = true;
35:       assign supplier[min_pos]'s first block to supplier[max_pos];
36:       time_left[min_pos] = time_left[min_pos] + blk_size / Bw^r[min_pos];
37:       time_left[max_pos] = time_left[max_pos] - blk_size / Bw^r[max_pos];
38:    }
39: }
```

Figure 3-12 Revised multiple suppliers scheduling (MSS) algorithm

3.6.4 Scheduling Algorithm Analysis

This section gives a time analysis or our multiple suppliers scheduling (MSS) algorithm. In the scheduling stage, loop (line 1-3) will be executed M time, where M is the number of suppliers. For the nested loop (line 5-16), each round of the outer loop will assign a block to a supplier, thus the outer loop will be executed N_b times, where N_b is the number of blocks. Within the outer loop, finding the maximum *time_left* value across all the suppliers needs M time, and the inner loop (line 7-15) runs at most M times. Thus the time for scheduling is $O(M+N_b \cdot 2M)$, which is bounded by $O(N_b \cdot M)$. The scheduling needs suppliers sorted by their Bw^r. Suppose quick sort is used, then the sorting time is $O(M \cdot \log M)$. Thus, the total time for scheduling is bounded by $O(M \cdot \log M + N_b \cdot M)$.

In the optimizing stage, the outer loop (line 18-39) runs at most M times. Within the outer loop, finding the minimum *time_left* across all the suppliers needs M time, and the inner loop (line 24-32) runs M times. Thus the time for optimization is bounded by $O(M^2)$. In total, the running time of our MSS algorithm is bounded by $O(M \cdot \log M + N_b \cdot M + M^2)$, where M is the number of suppliers and N_b is the number of blocks.

3.6.5 Streaming Session

Once the receiver generates the schedule, it will send the schedule to the suppliers. When a supplying peer receives the schedule, it will send the assigned blocks to the receiver in order using UDP and perform TCP-friendly congestion control over the UDP connection (e.g., RAP [34] or TFRC [29]). As mentioned in Section 3.6.3, during the streaming session, some of the suppliers may leave or fail at any time, and the incoming streaming

rates from the suppliers may decrease due to the network congestion. In these cases, the *supplier switching* will happen, in which the receiver selects a substitute supplying peer from the *standby* set to replace the leaving/failing supplier or the supplier whose sending rate is decreasing. After that, the receiver will generate a new schedule to assign the rest not-received blocks to the new set of suppliers. Taking this into consideration, generating schedule for all of the blocks contained in a segment at the same time is not a good approach, since the schedule may change during the session. Thus, we divide a segment into several equal length *schedule sections*, in which each *schedule section* contains N_{sche} blocks. During the streaming session of a segment, the receiver first generates the schedule for the 1st *schedule section* and coordinates the suppliers to stream the blocks contained in the 1st *schedule section*. Once the receiver almost receiving all of the blocks contained in the 1st *schedule section* (enough time should be left for the receiver to generate and send schedule for the next *schedule section*), it will do the same thing with the next *schedule section*, and so on. Choosing the value for N_{sche} (which specifies the length of a *schedule section*) is a trade-off. If N_{sche} is too small, the computation overhead and communication traffic will increase, since the schedules will be generated frequently. If N_{sche} is too big, most portion of a schedule is useless, since it is very likely that the schedule will be changed latter. In our simulation, N_{sche} is set to 60, thus a segment has 5 *schedule sections*.

In the client side, the receiver maintains a *ring buffer*. The size of the *ring buffer* is $\alpha_{buff} \times N_{sche} \times blk\_size$, where $\alpha_{buff} > 1$ (in our simulation, α_{buff} is set to 1.5) and *blk_size* is the size of a block (in bytes). Once the receiver receives a block, it will write this block to the right position of the *ring buffer*. As mentioned in Section 3.6.3 and above, to absorb all transient effects because of streaming packets arriving late, selecting new suppliers in

case of suppliers leave or fail, we require the receiver to buffer at least $S_{initBuff}$ blocks before the playback starts (*initial buffering*). After the *initial buffering time*, the receiver will continuously read data from the *ring buffer* and render video frames on the player window.

During the streaming session, the receiver monitors the incoming rate from each supplier. If the receiver detects that the incoming rate from a supplier is decreasing for an enough long period T_{dec}, or it is notified or detects the leave or failure of a supplier, *supplier switching* will happen. The receiver will select substitute supplying peers from the *standby* set and reserve bandwidths from them (the total new reserved bandwidth should be bigger than or equal to the bandwidth provided by the supplier which is substituted). Then it will generate a new schedule to assign the rest not-received blocks to the new set of suppliers, and sends the schedule to the suppliers. Once receiving the schedule, the suppliers will send the assigned blocks to the receiver in order.

The receiver also monitors the status of the *ring buffer* and tracks the received blocks during the streaming session. Every block should be received by the receiver T_{adv} seconds before the playback. Otherwise, the block is identified as lost, and the receiver will ask the corresponding supplier to re-send it.

Chapter 4

Prototype Implementation

To demonstrate the feasibility of BitVampire, a prototype has been implemented using Java and Java Media Framework (JMF) [17]. This chapter discusses the methodology and details of the prototype implementation. We first describe a general Peer-to-Peer Application Framework, based on which the prototype was developed. Then we present the prototype's system architecture, its core classes, and GUI design.

4.1 Implementation Methodology

BitVampire relies on Category Overlay to search media segments; thus, to implement it, a prototype of Category Overlay, called CoolSearch, has been implemented first. As a joint research project of this work, Jun employed a general Peer-to-Peer Application Framework to implement the prototype of Category Overlay [43]. We have followed his approach during the prototype implementation. The next section briefly introduces that framework.

4.1.1 A General Peer-to-Peer Application Framework

Probably the most widely used general Peer-to-Peer Application Framework is JXTA [20] (Figure 4-1). JXTA focuses on providing a network computing platform and a set of open protocols to allow any connected device on the network to communicate and collaborate in a P2P manner [19]. Its goal is to develop the basic building blocks and services to enable innovative applications for peer groups.

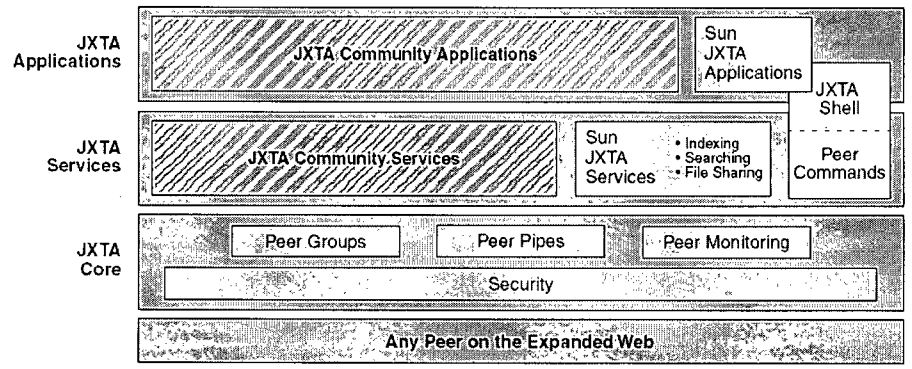


Figure 4-1 JXTA layers [18]

However, JXTA architecture also includes some specific components to provide various features such as security, ubiquity, and platform independence, which are beyond our focus in the prototype implementation. Therefore, we need a lightweight P2P application framework, which provides system level support, as well as a few basic building blocks and services, to ease the task of creating a single application with different peer connection schemes.

Inspired by JXTA, a general Peer-to-Peer Application Framework, called RTG (Ready-To-Go) [43] was designed, which can be easily customized for different application domains. The design goals of RTG are as follows: (1) Simplify service algorithm replacement and extension, (2) Separate peer architecture from system services

and application logics. Figure 4-2 illustrates the architecture of RTG, and Table 4-1 describes the basic functionalities for each layer in RTG.

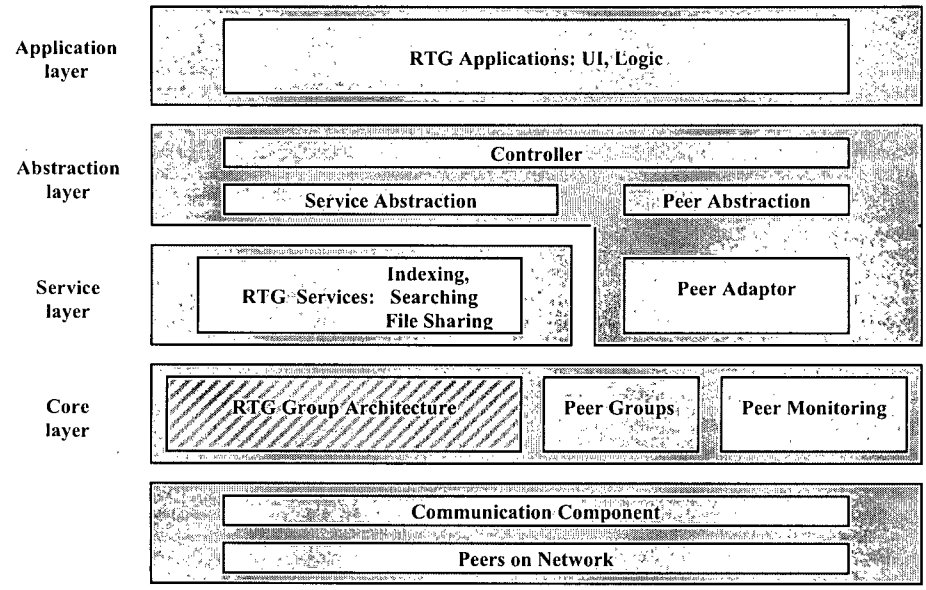


Figure 4-2 RTG (Ready-to-Go) layers [43]

Table 4-1 RTG layer descriptions [43]

Layer	*Description*
Application	The application layer is where application specific components should be placed, such as user interface (UI) and application logic.
Abstraction (Controller)	This layer separates the application domain from the system architecture. Specifically, abstractions and adaptors are used to decouple application logic from any specific service or peer group implementation. A controller provides 3-way coordination (among services, peer groups, and logics).
Service	The service layer contains various service modules, such as searching and indexing. Different service algorithms can be implemented here to influence system performance.
Core	This layer finalizes the actual P2P model as applied to the current system.
Communication	A Peer-to-Peer specific communication model is provided at this layer, with the purpose of removing the necessity to implement this low-level component for most P2P systems.

The controller in the RTG Abstraction layer is the key to the decoupling of application logics, services, and peer architecture. More specifically, it not only controls how an application uses various services to achieve a certain goal but also bridges the gap between service components and peer architecture. Under this design, either service algorithms or peer architecture can be replaced without generating many modification requirements to other parts of the system.

4.1.2 System Architecture

Based on RTG framework, we present our prototype's system architecture in Figure 4-3.

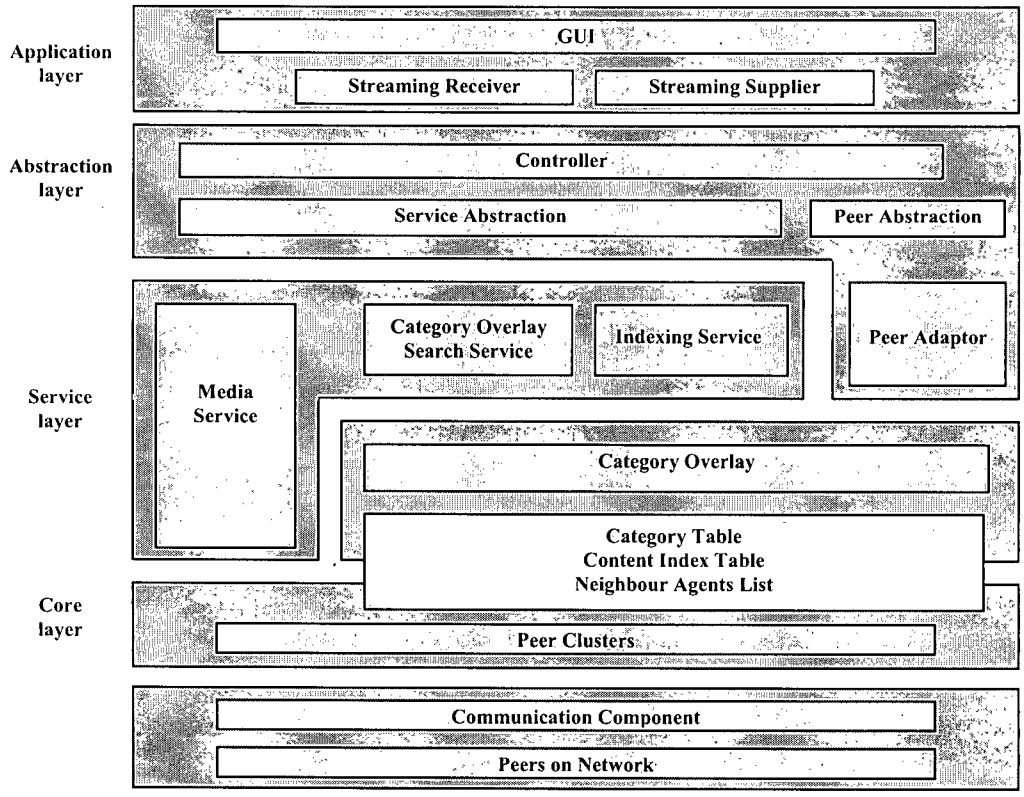


Figure 4-3 Prototype's system architecture

As Figure 4-3 shows, in the core layer, peers are grouped into clusters according to our clustering algorithm (Section 3.2.1), and Category Overlay is constructed over the clusters. In the service layer, *Category Overlay Search Service Component* provides media segments search service, and *Media Service Component* provides a set of services related to media processing, including publishing media, distributing media segments, caching media segments, watching media, etc. In the application layer, the application logic is divided into two parts: (1) *Streaming Receiver Component*, which includes scheduler, streaming packets receiver, buffer manager, and media frame render; and (2) *Streaming Supplier Component*, which includes steaming packets sender and packets sending rate controller.

Table 4-2 provides a summary description of Java packages in our prototype implementation.

Table 4-2 Packages for prototype implementation

Package	***Description***
communication	Classes in this package provide basic implementation for *Communication Component* in Figure 5-3, including message formatting, marshalling and unmarshalling, communication channel creation, message delivery, and message processing.
communication.coolsearch	Customized to provide CoolSearch specific message types and to satisfy special delivery requirements.
communication.bitvampire	Customized to provide BitVampire specific message types and to satisfy special delivery requirements.
service	Service abstraction layer.
service.coolsearch	This package contains classes that implement various services, including *category overlay search*. Other classes in this package implement construction and maintenance algorithms for clusters and Category Overlay.
service.bitvampire	This package contains classes that implement various services related to media processing, including publishing media, distributing media segments, caching media segments, watching media, etc.
kernel	This package includes classes that implement the *controller component*, a generic P2P structure layer (peer abstraction layer) as well as a system processing model.

kernel.coolsearch	It provides CoolSearch specific extensions to generic classes in kernel package, such as peer node identification, peer adaptor, and message processing scheme.
kernel.bitvampire	It provides BitVampire specific extensions to generic classes in kernel package, such as peer resource representation, peer adaptor, and message processing scheme.
vod.bitvampire.receiver	This package contains classes that implement various application logics of the *Streaming Receiver Component*, including scheduler, streaming packets receiver, and media frame render.
vod.bitvampire.supplier	This package contains classes that implement the various application logics of the *Streaming Supplier Component*, including steaming packets sender and packets sending rate controller.
ui.coolsearch	GUI implementation for CoolSearch.
ui.bitvampire	GUI implementation for BitVampire.
resource	Resource package provides an abstract layer for resource storage, maintenance, and sharing. Implementation is given for a generic resource type.
resource.coolsearch	Implementation for a CoolSearch based resource.
resource.bitvampire	Implementation for a BitVampire based resource.
util.coolsearch	Utility package that contains general purpose data structure, constants, and other utility functions related to CoolSearch.
util.bitvampire	Utility package that contains general purpose data structure, constants, and other utility functions related to BitVampire.
property	This package stores and maintains generic system parameters.
property.coolsearch	Stores and maintains CoolSearch specific parameters, such as clustering parameters, predefined categories, etc.
property.bitvampire	Stores and maintains BitVampire specific parameters, such as segment length, timeout settings, etc.

4.2 Implementation Details

As mentioned before, BitVampire is implemented using Java and Java Media Framework (JMF) [17]. In the prototype, control packets are sent using TCP, and streaming packets are sent using UDP. To ensure TCP-friendly congestion control over the UDP connection, RAP protocol [34] is used to adjust the UDP packets sending rate. To provide some details of the prototype implementation, we list the core Java classes and present Graphic User Interface (GUI) design in the following sections.

4.2.1 Core Classes

Our prototype implementation consists in total of 109 Java files, 21 packages, 133 classes, and about 20,000 lines code. Thus, we only list some of the core classes that provide important functionalities to the prototype as follows. Each entry is composed of a full class name and a description.

Table 4-3 Core classes for prototype implementation

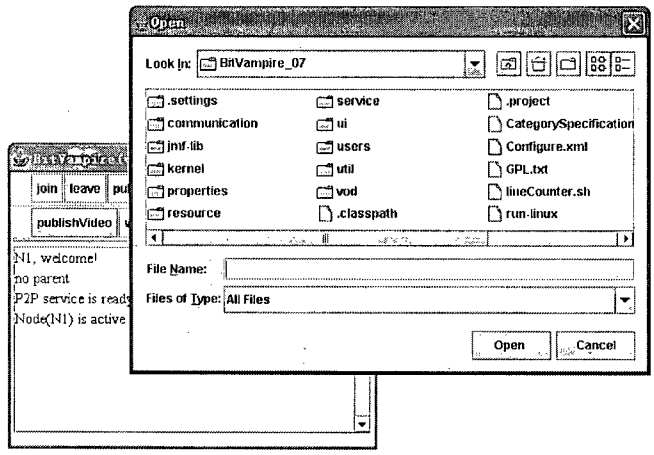
Class	*Description*
communication.MessageImpl	Providing generic marshalling and unmarshalling services.
communication.coolsearch.CoolSearchClient	Providing a communication layer interface for peers at core layer to deliver CoolSearch specific messages.
communication.bitvampire.BitVampireClient	Providing a communication layer interface for peers at core layer to deliver BitVampire control messages.
service.coolsearch.CoolSearchStrategy	*CoolSearch Service Abstraction* to decouple *controller* from specific service implementations. Combined with *Peer Abstraction*, they provide common interfaces to facilitate the 3-way communication among *controller*, *peers*, and various *services*.
service.coolsearch.GroupManager	Implementing clustering algorithm and cluster maintenance mechanisms.
service.coolsearch.ContentIndex	Providing *Content Index Table* management and indexing service.
service.coolsearch.CategoryManager	Constructing and maintaining Category Overlay.
service.coolsearch.SearchService	Providing *category overlay search* service.
service.coolsearch.RetrieveService	Providing file sharing service.
service.bitvampire.BitVampire Strategy	*BitVampire Service Abstraction* to decouple *controller* from specific service implementations. Combined with *Peer Abstraction*, they provide common interfaces to facilitate the 3-way communication among *controller*, *peers*, and various *services*.
service.bitvampire.MediaManager	Managing and maintaining peer's contributed resources, including storage, outbound bandwidth, etc.
service.bitvampire.MediaService	Implementing a set of media services, including publishing media, distributing media segments, caching media segments, etc.
service.bitvampire.MediaViewServant	A servant thread to view a specified media.
service.bitvampire.MediaSegViewProcessor	Implementing the logic to view a specified media segment.

kernel.LocalController	Implementing core functionalities in *controller* component at abstraction layer.
kernel.LocalNode	*Peer Abstraction* representing node object on current peer, providing a common interface to *controller* for functionalities and services available to local peer.
kernel.RemoteNode	*Peer Abstraction* representing node object on remote peer, providing marshalling and unmarshalling services to *controller*.
kernel.IncomingMsgServant	Dispatching various incoming requests to different parts of the system.
vod.bitvampire.receiver.ScheduleProducer	Implementing multiple suppliers scheduling algorithm.
vod.bitvampire.receiver.UDPReceiver	Receiving UDP stream packets, tracking packets receiving status, and reporting lost packets.
vod.bitvampire.receiver.BufferManager	Managing and maintaining the ring buffer.
vod.bitvampire.receiver.MediaPlayer	Rendering the media frames in the player window.
vod.bitvampire.supplier.UDPSender	Sending UDP steam packets according to the schedule.
vod.bitvampire.supplier.SendingRateController	Implementing RAP protocol [34] to control the UPD packets sending rate.
ui.coolsearch.CoolSearchUI	Graphic user interface for CoolSearch.
ui.bitvampire.BitVampireUI	Graphic user interface for BitVampire.
resource.ResourceDBManager	Managing local content database.
resource.coolsearch.CoolSearchGeneralResource	Representing a generic CoolSearch resource type, providing resource specific marshalling and unmarshalling services.
resource.bitvampire.MediaSegmentResource	Representing a BitVampire media segment resource type, providing resource specific marshalling and unmarshalling services.

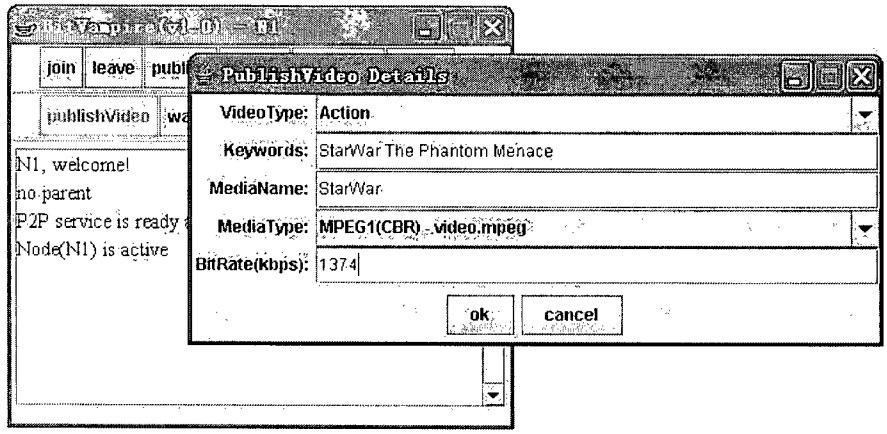
4.2.2 Graphic User Interface

This section presents the Graphic User Interface (GUI) of our prototype. Figure 4-4 shows the GUI for publishing videos in BitVampire, and Figure 4-5 shows the GUI for watching videos in BitVampire. When publishing a video, the user should specify the local path of the video file (Figure 4-4(a)), then specify the video's type, provide a list of keywords, and specify the video's playback bit rate (Figure 4-4(b)). When watching a video, the user should specify the video type and provide keywords (Figure 4-5(a)). After

the initial buffering period, a media player window (Figure 4-5(b)) will show up and the video frames will be rendered in the window. Figure 4-6 shows a snapshot of a more complicated running scenario of our prototype, in which three nodes are running and two of them are watching videos.

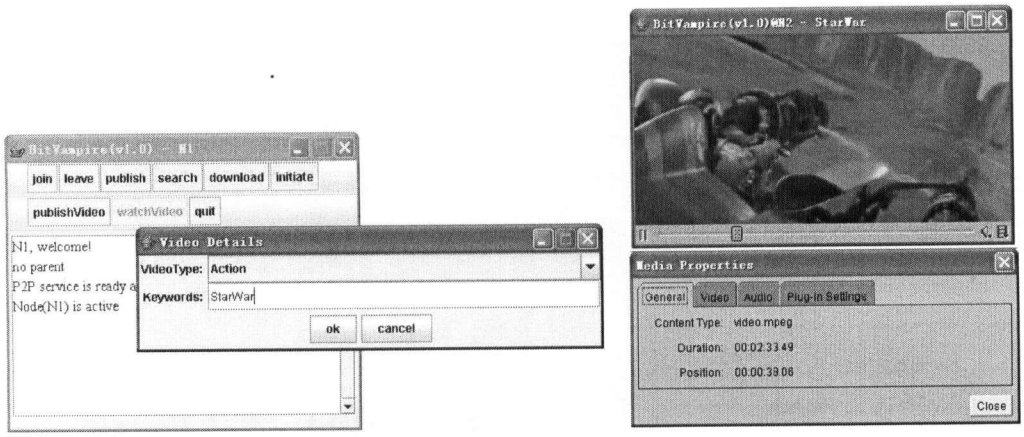


(a) Specifying local path of the publishing video file



(b) Specifying the publishing details of the video

Figure 4-4 GUI for publishing video



(a) Specifying searching details (b) Media player

Figure 4-5 GUI for watching video

Figure 4-6 Snapshot of the prototype running

Chapter 5

Evaluation

In this section, we evaluate the performance of our proposed architecture through extensive simulation experiments. We first describe the simulation setup and then present the results.

5.1 Simulation Setup

5.1.1 Simulation Topologies

In all of the simulations, we use large hierarchical, Internet-like topologies. All of the topologies have three levels. The top level consists of several Transit domains, which represent large Internet Service Providers (ISPs). The middle level contains several Stub domains, which represent small ISPs, campus networks, moderately sized enterprise networks, etc. (Each Stub domain is connected to one of the Transit domains). At the bottom level, end hosts (peers) are connected to Stub domains. Figure 5-1 shows a part of

the topology used in the simulation. The first two levels (router-level) contain transit routers and stub routers, which are generated using the GT-ITM tool [45]. We then randomly attach end hosts (peers) to stub routers with uniformed probability. Each experiment was run on 10 different topologies, and the results presented in this thesis are the average results of experiments running in these 10 topologies. Unless otherwise specified, the topologies used in the simulations consist of averagely 10 transit domains, 200 stub domains, 2050 routers, and a total of 3010 end hosts (peers), in which 6 hosts are selected as seed peers.

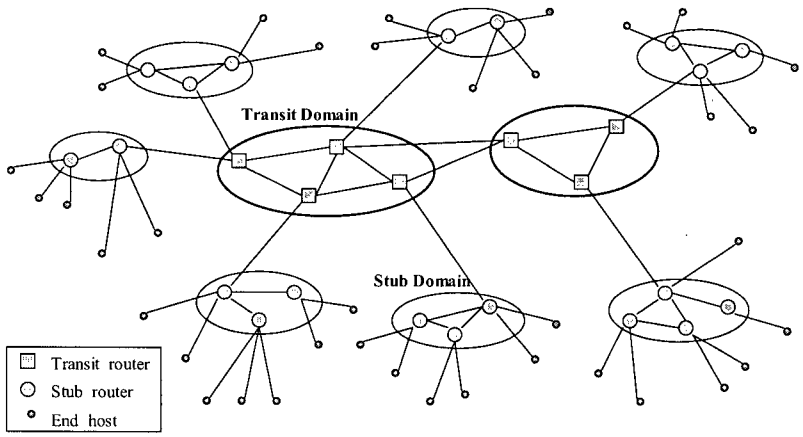


Figure 5-1 Part of the topology used in the simulation

5.1.2 Simulation Parameters

All of the experiments use the following parameter settings, unless otherwise specified.

During the simulation, there are totally 500 videos published in the network, each with 512 kbps constant playback bit rate (CBR) and 1 hour length. Each video is split into 12 segments. The length of each segment is 5 minutes, and the size is about 19 MB.

We let the first segments have 2 replicas ($N_f = 2$), and by default, the receiver will select 3 ($M = 3$) supplying peers, if these peers have enough available outbound bandwidths.

We assign bandwidths and delays to the network links as follows: (1) Each link between two routers has a bandwidth ranging from 6 Mbps to 20 Mbps, and a delay ranging from 5 ms to 40 ms. (2) Each link between end hosts (peers) to routers has a bandwidth ranging from 512 kbps to 2 Mbps, and a delay ranging from 4 ms to 10 ms. The routing between two routers in the network follows the shortest path.

The contributed outbound bandwidths and storage from peers are configured as follows: (1) Each peer contributes an outbound bandwidth ranging from 128 kbps to 1 Mbps, and storage ranging from 2 segments (38 MB) to 5 segments (95 MB). (2) Each seed peer contributes an outbound bandwidth ranging from 1 Mbps to 2 Mbps (on the average, each seed peer contributes 1.472Mbps bandwidth in the simulations), and storage ranging from 1000 segments (19 GB) to 3000 segments (57 GB). Note that the configuration of peers in the experiments represents a typical equipment setting for current desktop PCs connected to the Internet. From the simulation results presented in the following sections, we can see that based on these usual, low-cost PCs, our proposed architecture can support large-scale on-demand media streaming service.

To reflect the dynamic nature of peer-to-peer networks, we let 20 peers leave the system per minute. Each leaving peer will stay off-line for a period ranging from 15 minutes to 3 hours, and then rejoin the system. We evaluate the performance of the system under 3 different video request arrival patterns: (1) Constant arrival, where every 3 seconds, a peer initiates a video watching request (request rate: 20 requests/min). (2) Flash crowd arrival, where at the beginning, peers request videos at the rate of 20 requests/min, and then suddenly increase to the rate of 120 requests/min. Figure 5-2

shows this arrival pattern. (3) Periodic flash crowd arrival, where the flash crowd requests (request rate: 120 requests/min) occur periodically. Between two flash crowd arrivals, the video requests arrive at a low and constant rate (20 requests/min). Figure 5-3

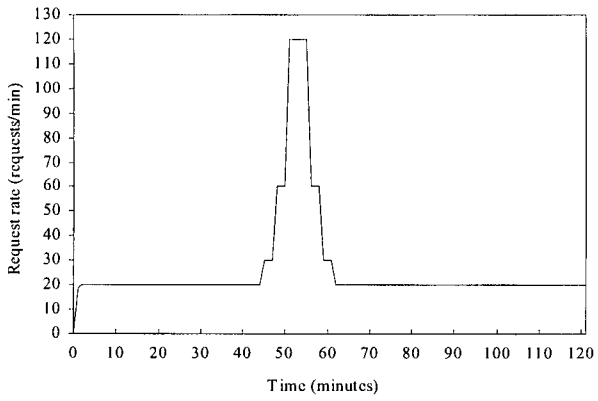


Figure 5-2 Flash crowd arrival pattern

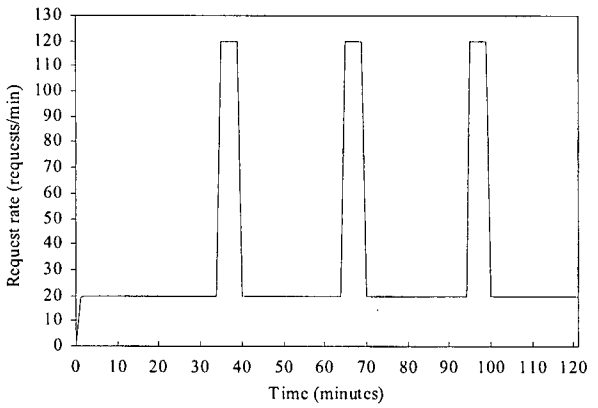


Figure 5-3 Periodic flash crowd arrival pattern

shows this arrival pattern.

The popularity of videos follow Zipf-like distribution, where the popularity of the i^{th} most popular video is proportional to $1/i^{\alpha}$. The authors of [4] reported that in a long

period (in months), the video file access frequencies in HP corporate media solutions server and HPLabs media server follow Zipf-like distribution, with α ranging from 1.4 to 1.6. Therefore, in our simulation, we set α to 1.4.

The final parameter setting is the length of the video watching sessions. Again, based on the reports from [4], we let 50% of the sessions last 5 ~ 10 minutes (watching 1 ~ 2 segments), 30% of the sessions last 10 ~ 30 minutes (watching 2 ~ 6 segments), and 20% of the sessions last 30 ~ 60 minutes (watching 6 ~ 12 segments).

5.2 Simulation Results

This section presents the results of our simulations. We first evaluate the effectiveness of our media segments distributing (MSD) algorithm and seed re-distributing (SRD) mechanism. Then, we evaluate our multiple suppliers scheduling (MSS) algorithm, showing that it can result in a small *initial buffering time*. Finally, we study the behaviour of our proposed architecture under different parameter settings and conditions.

5.2.1 System Streaming Capacity Amplification

In this set of experiments, we show that our media segments distributing (MSD) algorithm plus seed re-distributing (SRD) mechanism can result in fast system streaming capacity amplification. We define the system streaming capacity as the number of video watching sessions that can be served concurrently, and use the simple random segments distributing algorithm as the comparison base. In our simulation, each video watching session may have a different length, in another word, each session may contain a different number of segment requests. Thus, we use the segment requests rejection ratio as our

measurement metric. A lower segment requests rejection ratio means that more segment requests can be accepted at a specific time, which results in higher system streaming capacity.

Figure 5-4 shows the simulation result for the constant video requests arrival pattern; Figure 5-5 shows the result for the flash crowd arrival pattern; and Figure 5-6 shows the result for the periodic flash crowd arrival pattern. We ran 10 round simulations on each of the 10 topologies, thus in total 100 rounds of simulations were performed. Each simulation lasts for 2 hours, and every minute, we compute the average segment requests rejection ratio. The results presented in the figures are the average results of the 100 simulation rounds. From all of these three figures, we can see that compared to the

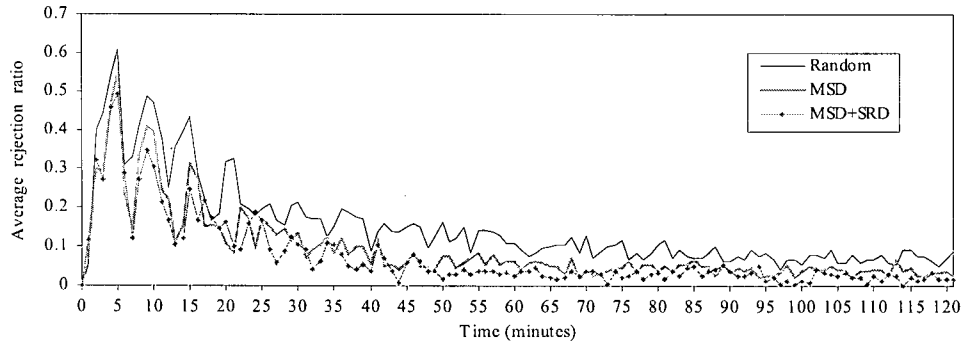


Figure 5-4 Average rejection ratio for constant arrival pattern

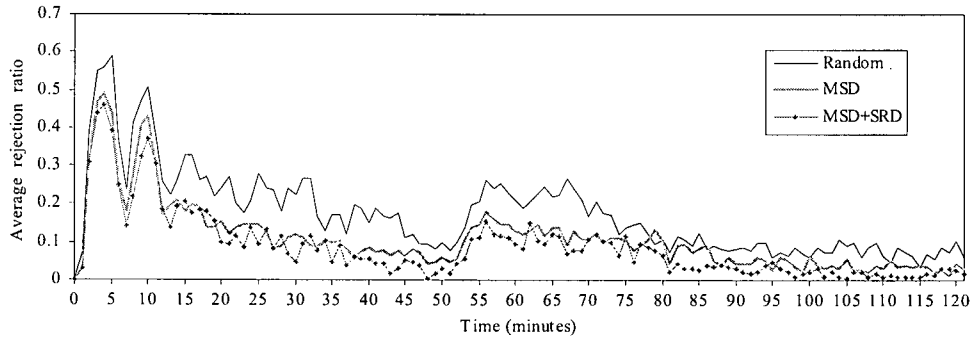


Figure 5-5 Average rejection ratio for flash crowd arrival pattern

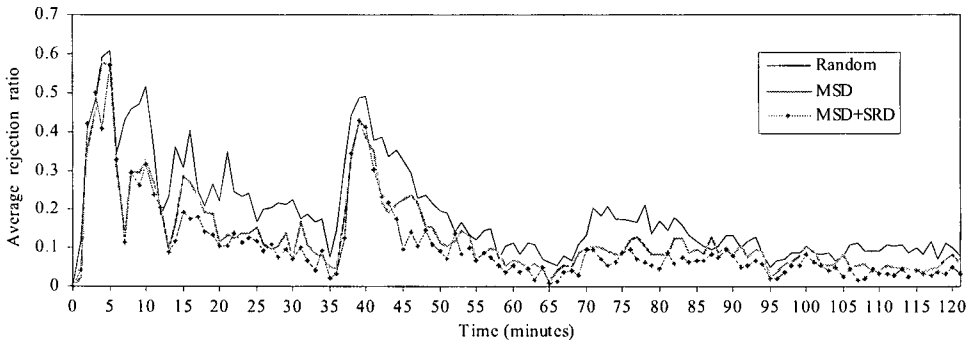


Figure 5-6 Average rejection ratio for periodic flash crowd arrival pattern

random segments distributing, our MSD algorithm quickly decreases the rejection ratio, and if the seed re-distributing (SRD) mechanism is applied, the decrease is faster.

5.2.2 Seed Peers Load

Our next set of experiments evaluates the load on seed peers. As in the previous experiments, we ran 10 round simulations on each of the 10 topologies. Each simulation lasts for 2 hours, and every minute, we compute the average load on seed peers. The results presented in the following figures are the average results of the 100 simulation rounds. Figure 5-7 shows the simulation result for the constant video requests arrival pattern; Figure 5-8 shows the result for the flash crowd arrival pattern; and Figure 5-9 shows the result for the periodic flash crowd arrival pattern. As all of the figures show, the average seeds load with MSD algorithm is less than the load with random segments distributing. And if the seed re-distributing (SRD) mechanism is applied, the seeds load will decrease further. The reason for this result is that MSD tries to distribute segments to the peers that are more stable and have more available outbound bandwidth, thus decreasing the demand for seed peers. If SRD is applied, the segments that cannot be served by regular peers will be distributed to peers by seeds. Thus next time, requests for

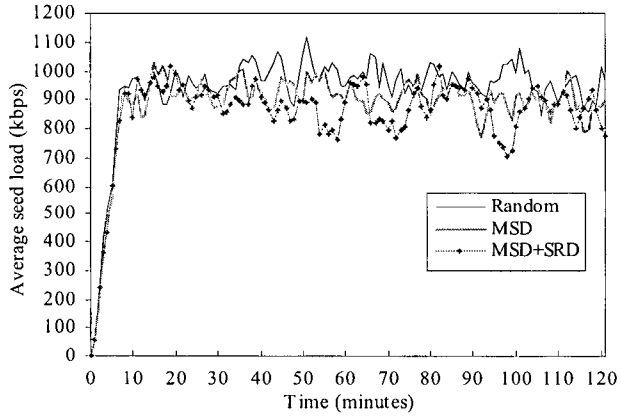


Figure 5-7 Average seeds load for constant arrival pattern

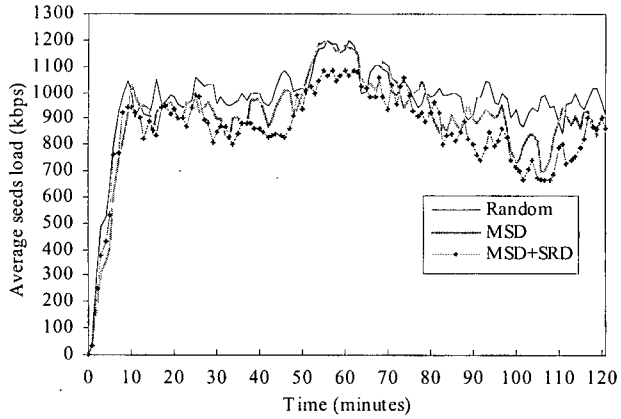


Figure 5-8 Average seeds load for flash crowd arrival pattern

these segments can be served by regular peers, therefore further reducing the seed peers' load.

Reducing the load on seed peers is an important feature of the proposed architecture. Because it means that the seed peers need not be powerful machines with high outbound bandwidth; they just need to be stable and have large storage capacity, which is very cheap nowadays.

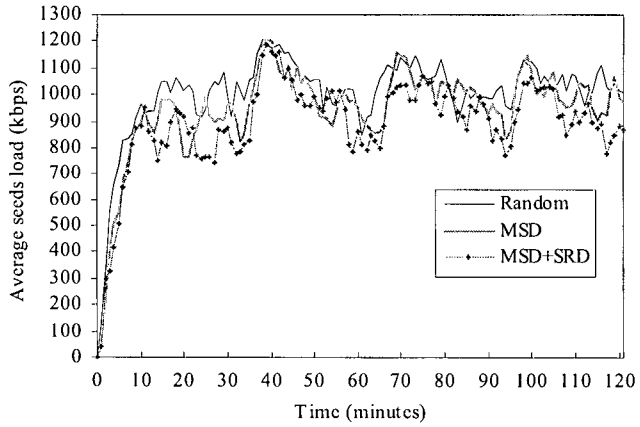


Figure 5-9 Average seeds load for periodic flash crowd arrival pattern

5.2.3 Receiver Initial Buffering

In this set of experiments, we evaluate the effects of choosing different settings for *initial buffering*. Unless otherwise specified, all of the simulation settings are same as Section 5.2.1. In the simulation, the packet size is set to 32KB, thus a block needs 2 packets to send. We vary $S_{initBuff}$ from 0 to 16 blocks. For each simulated *initial buffering*, we measure the average number of times the receiver experiences buffer underflow. Every buffer underflow causes a pause in the playback until sufficient data packets arrive. These pauses are mainly due to *supplier switching*, which happens if the incoming rate from a supplier decreases due to peer leave/failure or network congestion. When *supplier switching* happens, we delay sending packets from the replacement supplier(s) by a random time between 0 and 1 second. This is called *switching time*, during which the degraded peer is detected and a replacement is notified.

To simulate playback of the video, an independent playback process is scheduled at regular times. The first call of this process is after receiving the initial $S_{initBuff}$ blocks. Then,

it is called every 1 second (the block length in this experiment). When the playback process is invoked to play block *i*, it checks the *ring buffer* for all packets belonging to block *i*. These packets are identified through their sequence numbers. If all packets are available, the playback of block *i* is successful and the playback process is scheduled for block *i*+1 after 1 second from the current simulation time. If any packet is missing, a pause is encountered. The playback process is scheduled for the same block after the pause time, which is 1 second in this experiment.

We simulate the following scenario. A value for $S_{initBuff}$ is set. A receiver is chosen at random. Then the receiver initiates a video watching request and the streaming starts. *Supplier switching* happens at random times and replacement suppliers are chosen from the *standby* set. On average, 16 switching events occur during each video watching session. After the *initial buffering time*, the first invocation of the playback process is scheduled. We count the number of pauses encountered throughout the session. After the streaming session is over, the experiment is repeated for another randomly chosen receiver. We simulate 10 different sessions. We compute the minimum, maximum, average number of pauses over these 10 sessions. Then another value for $S_{initBuff}$ is set and

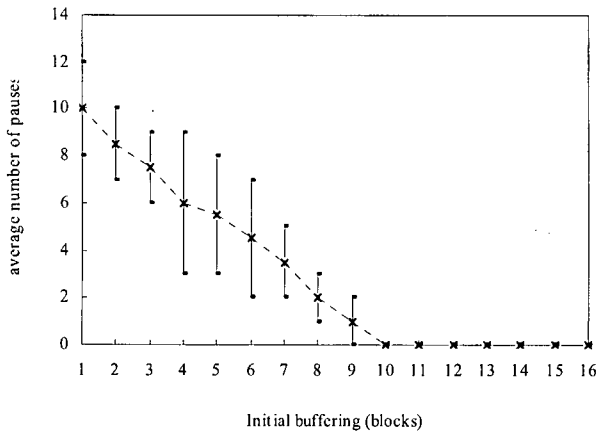


Figure 5-10 Effects of different initial buffering settings

the whole scenario is repeated again.

Figure 5-10 shows the simulation result. Note that in the figure, mid-point is the average number of pauses, while the bottom, and the top points are the minimum and maximum number of pauses, respectively. For a small initial buffering of 2 blocks, we expect an average of 8.5 pauses and a maximum of 10 pauses. A buffer size of 10 blocks or more will absorb all transient effects during the supplier switching.

5.2.4 Initial Buffering Time

The next set of experiments evaluates the multiple suppliers scheduling (MSS) algorithm. We compare MSS algorithm with Round Robin (RR), showing that MSS can achieve a smaller *initial buffering time*. We generate 5,000 video requests, in constant arrival pattern, flash crowd pattern, and periodic flash crowd pattern respectively. We then record the *initial buffering times* for each accepted request using RR and MSS respectively. In our simulation, the initial buffering length is set to 8 blocks ($S_{initBuff} = 8$)

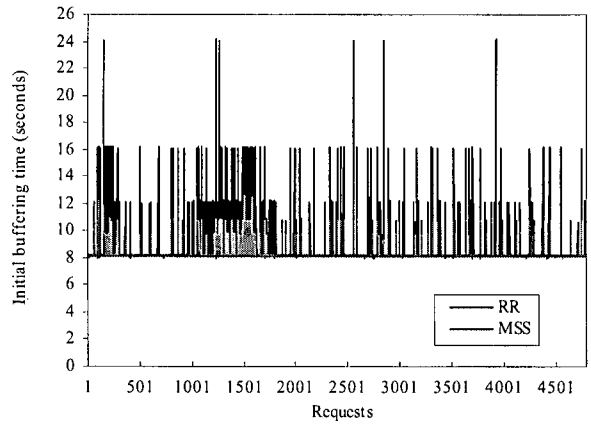


Figure 5-11 Initial buffering time using different scheduling algorithm

and the bandwidth reservation unit is set to 64kbps (Bw'_u = 64kbps). Figure 5-11 shows the simulation results. The results presented in the figures are the average results of the experiments on the three request arrival patterns. It is clear that MSS always achieves an equal or smaller *initial buffering time* compared to RR. Figure 5-12 shows the *initial buffering time* gain using MSS, where gain is defined as follows:

$$InitialBufferingTimeGain = InitialBufferingTime_{RR} - InitialBufferingTime_{MSS} \quad (5\text{-}1)$$

As the figure shows, the gain is always bigger than or equal to 0, and can be as large as more than 14 seconds.

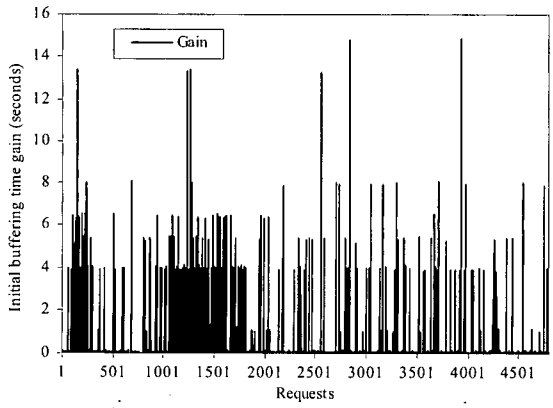


Figure 5-12 Initial buffering time gain using MSS scheduling algorithm

5.2.5 Varying Network Size

In this set of experiments, we evaluate the performance of our proposed architecture in different sized networks. We measure the average segment requests rejection ratio for 3 different sized peer-to-peer networks: (1) 3000 peers network

(consists of 2050 routers and 3000 peers), (2) 6000 peers network (consists of 2050 routers and 6000 peers), and (3) 9000 peers network (consists of 2050 routers and 9000 peers). The video requests arrive in flash crowd pattern. Other simulation settings are same as 5.2.1.

Figure 5-13 shows the simulation results. From the figure, we can see that since the network size increases, more segment requests will be issued by peers at the same time, thus at the beginning, the rejection ratio for 6000 peers network is bigger than the one for 3000 peers network and the rejection ratio for 9000 peers network is bigger than the one for 6000 peers network. However, the rejection ratios decrease fast. After about 25 minutes, the rejection ratio for 6000 peers network is almost same as the one for 3000 peers network. And after about 35 minutes, the rejection ratio for 9000 peers network is almost same as the one for 3000 peers network. The simulation results verified our hypothesis that as more peers participating in the system, more segment requests can be supported at the same time, since more resources are contributed to the system by peers. The simulation results also imply that our proposed architecture is scalable, as long as the participating peers contribute some of their resources to the system.

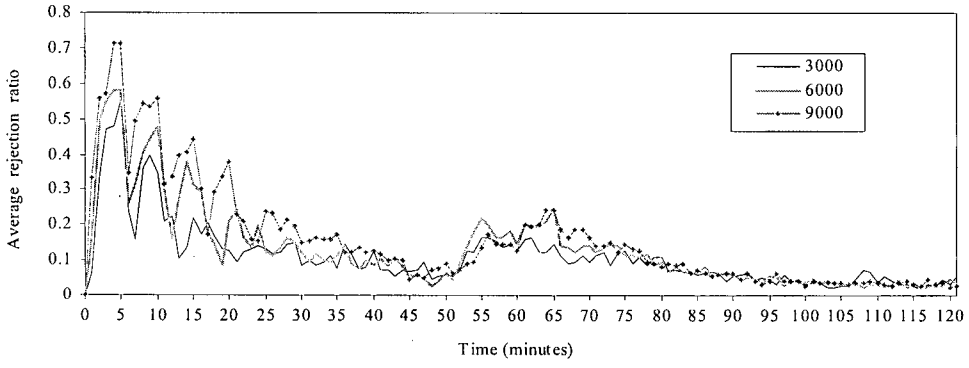


Figure 5-13 Average rejection ratio for various sized network

5.2.6 Varying Peers' Cooperation Level

Our final set of experiments evaluates system performance under different peers' cooperation level. We measure the average segment requests rejection ratio for 3 different peers' cooperation level: (1) Low cooperation level, where peers contribute their bandwidths ranging from 64kbps to 512kbps and storage ranging from 1 segment (19MB) to 3 segments (57MB); (2) Medium cooperation level, where peers contribute their bandwidths ranging from 128kbps to 1Mbps and storage ranging from 3 segments (57MB) to 6 segments (114MB); (3) High cooperation level, where peers contribute their bandwidths ranging from 1Mbps to 2Mbps and storage ranging from 5 segment (95MB) to 10 segments (190MB). The video requests arrive in flash crowd pattern. Other simulation settings are the same as 5.2.1.

Figure 5-14 shows the simulation results. From the figure, we can see that as the peers' cooperation level increases, the segment requests rejection ratio decreases faster, which means that the system streaming capacity is amplified faster. The reason for this is that if peers contribute more resources to the system, there will be more storage to cache segments and more bandwidth to support streaming requests; thus, the system streaming capacity increases faster.

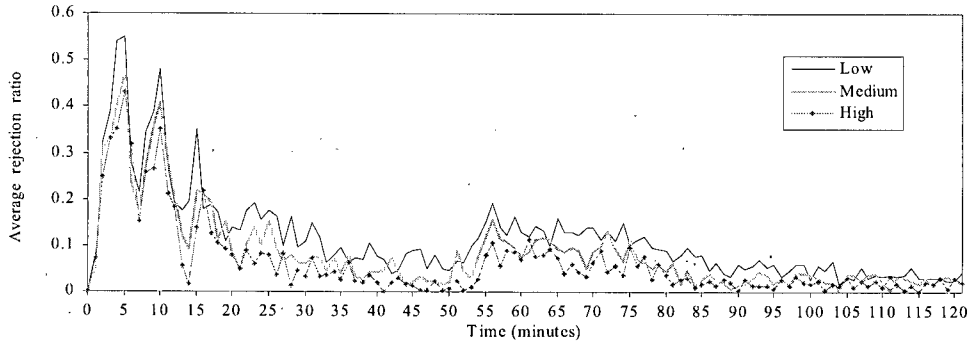


Figure 5-14 Average rejection ratio for various peers cooperation level

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we propose BitVampire, a novel low-cost on-demand media streaming architecture for heterogeneous peer-to-peer networks. In this architecture, published videos are split into segments and distributed to peers; thus outbound bandwidths form multiple peers can be aggregated to serve a single video streaming request. Instead of relying on powerful servers/proxies, our architecture exploits the often underutilized peers' resources, which makes it cost-effective. To deploy this architecture in the dynamic heterogeneous peer-to-peer networks, three key techniques are used: (1) A media segments distributing (MSD) algorithm, a seed re-distributing (SRD) mechanism, and a caching scheme are proposed to distribute and cache media segments. (2) An application-level overlay, called Category Overlay is chosen as the underlying search infrastructure to efficiently find the desired media segments. (3) To parallel download

streaming content from multiple supplying peers in real-time mode, we further divide segments into blocks and propose a multiple suppliers scheduling (MSS) algorithm to assign blocks to different supplying peers to send.

Based on our proposed architecture, a prototype has been implemented using Java and JMF. We designed a general purpose P2P application framework, RTG, to facilitate the implementation procedure. To evaluate the performance of our proposed architecture, we conducted extensive simulation experiments on large, Internet-like topologies. The simulation results show that the proposed architecture can support large-scale on-demand media streaming service in the dynamic heterogeneous peer-to-peer networks. It also demonstrates that our media segments distributing (MSD) algorithm can achieve fast system streaming capacity amplification, and our multiple suppliers scheduling (MSS) algorithm can achieve a small *initial buffering time.*

6.2 Future Work

There are five major directions call for further investigation. First, our simulator models the bandwidth limit and propagation delay on the physical links, but it does not model queuing delay and packet losses because modelling these would prevent large-scale network simulations. Therefore, to learn more about BitVampire and its behaviour in various real network conditions, a set of experiments on the Internet is necessary. We plan to improve our prototype and deploy it on PlanetLab [32] to evaluate its performance in the near future.

Second, currently we use LRU as the cache replacement algorithm to find the victim segment if a peer does not have enough available storage to hold the new cached

segment. However, LRU could reduce the system streaming capacity under some special cases. For instance, if most of the streaming sessions last for a long period, many of the segments at the beginning portion of the video would be replaced, which would decrease the system streaming capacity. Therefore, a more intelligent cache scheme could be part of our future work.

Third, our scheduling algorithm can aggregate bandwidths from multiple supplying peers and achieve a small *initial buffering time*. However, a more aggressive scheduling algorithm could be proposed, in which the blocks are assigned to suppliers in a roughly round robin manner and also in proportion to their contributed bandwidths. Furthermore, the blocks with small sequence numbers should be assigned to the suppliers that contribute more bandwidths, since these blocks are more time constrained and these suppliers are more powerful. For example, in Figure 3-6, a more aggressive scheduling algorithm would assign block 5 to P_1 and block 7 to P_3, instead of assigning block 7 to P_1 and block 5 to P_3.

Fourth, in our current architecture, we deliver the video in full quality. If full quality media delivering could not be achieved due to peer leaving/failure or network congestion, we simply pause the playback until all of the desired streaming packets arrive. However, adaptive streaming could be used to improve the quality of service. One possible approach is to use layered coding, in which a video is encoded into multiple layers. The receiver decides how many layers it can receive based on the current bandwidths from the supplying peers. If network congestion happens, the receiver can ask supplying peers to send fewer layers, which results in a smooth quality adaptation. Another approach is to use the priority drop technique [22], in which the supplying peers can drop some less important packets if they detect network congestion.

Finally, to encourage peers to contribute their storage and outbound bandwidths, an incentive mechanism should be proposed. With the incentive mechanism, the peers that contribute more resources should obtain a better service, while the ones that contribute little resources will encounter poorer service quality if others compete for the resources with them. Since our architecture relies on peers' resources to support on-demand media streaming, as more and more resources contributed to the system, better services could be achieved for all of the participating peers.

Bibliography

[1] S. Acharya and B. Smith, "MiddleMan: A Video Caching Proxy Server", in *Proceedings* of *NOSSDAV '00*, 2000.

[2] K. Aberer, M. Hauswirth, "Peer-to-Peer Information Systems: Concepts and Models, State-of-the-Art and Future Systems", in *Proceedings of ICDE '02*, 2002.

[3] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable Application Layer Multicast", in *Proceedings of ACM SIGCOMM'02*, Pittsburgh, PA, 2002.

[4] L. Cherkasova, M. Gupta, "Charactering Locality, Evolution, and Life Span of Accesses in Enterprise Media Server Workloads", in *Proceedings of NOSSDAV'02*, 2002.

[5] Y. Cai, K. Hua and K. Vu, "Optimzed Patching Performance", in *Proceedings of ACM/SPIE Multimedia Computing and Networking (MMCN '99)*, 1999.

[6] Y. Chae, K. Guo, M. Buddhikot, S. Suri and E. Zegura, "Silo, Tokens, and Rain-bow: Schemes for Fault Tolerant Stream Caching", *Special Issue of IEEE JSAC on Internet Proxy Services*, 2002.

[7] Y. H. Chu, S. G. Rao, S. Seshan, and H. Zhang, "Enabling Conferencing Applications on the Internet Using An Overlay Multicast Architecture", in *Proceedings of ACM SIGCOMM'01*, San Diego, CA, August 2001.

[8] Y. H. Chu, S. G. Rao, and H. Zhang, "A Case for End System Multicast", in *Proceedings of ACM SIGMETRICS'00*, 2000, pp. 1–12.

[9] H. Deshpande, M. Bawa, and H. Garcia-Molina, "Streaming Live Media over A Peer-to-Peer Network", in Submitted for publication, 2002.

[10] D. Eager, M. Vernon and J. Zahorjan, "Minimizing Bandwidth Requirements for On-Demand Data Delivery", *IEEE Transactions on Knowledge and Data Engineering* 13(5), 2001.

[11] Gnutella. Website: http://www.gnutella.com

[12] P. Ganesan, Q. Sun, H. Molina, "YAPPERS: A Peer-to-Peer Lookup Service Over Arbitrary Topology", in *Proceedings. of IEEE INFOCOM '03*, 2003.

[13] M. M. Hefeeda, B. K. Bhargava, D. K. Y. Yau, "A Hybrid Architecture for Cost-Effective On-Demand Media Streaming", *Computer Networks 44 (2004)*.

[14] K.A. Hua, Y. Cai and S. Sheu, "Patching: A Multicast Technique for True On-Demand Services", in *Proceedings of ACM Multimedia '98*, 1998.

[15] K.A. Hua and S. Sheu, "Skyscraper Broadcasting: A new Broadcasting Scheme for Metropolitan VOD systems", in *Proceedings of ACM SIGCOMM '97*, 1997.

[16] S. Jain, R. Mahajan, D. Wetherall, and G. Borriello, "Scalable Selforganizing Overlays", *Technical Report*, University of Washington, 2000.

[17] Java Media Framework (JMF). Website: http://java.sun.com/products/java-media/jmf/

[18] JXTA Technical Document. *Project JXTA: An Open, Innovative Collaboration.*

[19] JXTA v.2.3.x: *Java™ Programmer's Guide*, April, 2005.

[20] JXTA. Website: http://www.jxta.org

[21] KaZaa. Website: http://www.kazaa.com/us/index.htm

[22] C. Krasic, J.Walpole, W.C. Feng, "Quality-Adaptive Media Streaming by Priority Drop", in *Proceedings of NOSSDAV'03*, Monterey, CA, June 2003.

[23] B. T. Loo, R. Huebsch, I. Stoica, J. M. Hellerstein, "The Case for a Hybrid P2P Search Infrastructure", in *Proceedings. of IPTPS '04*, 2004.

[24] X. Liu, S. T. Vuong, "Supporting Low-Cost Video-on-Demand in Heterogeneous Peer-to-Peer Networks", to appear in *Proceedings of Seventh IEEE International Symposium on Multimedia (ISM'05)*, Irvine, CA, December 2005.

[25] X. Liu, J. Wang, S. T. Vuong, "A Peer-to-Peer Framework for Cost-Effective On-Demand Media Streaming", to appear in *Proceedings of 3rd IEEE Consumer Communications and Networking Conference (CCNC'06)*, Las Vegas, NV, January 2006.

[26] X. Liu, J. Wang and S. T. Vuong, "A Category Overlay Infrastructure for Peer-to-Peer Content Search", in *Proceedings of APDCM'05* (in conjunction with *IPDPS'05*), Denver, Colorado, USA, 2005.

[27] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, Z. Xu, "Peer-to-Peer computing", *Technical Report HPL-2002-57*, HP Laboratories, Palo Alto.

[28] Napster. Website: http://www.napster.com

[29] Padhye, J. Kurose, D. Towsley, and R. Koodli, "A Model-based TCP-friendly Rate Control Protocol", in *Proceedings of NOSSDAV'99*, 1999.

[30] V. N. Padmanabhan, H. J. Wang, P. A. Chou, and K. Sripanidkulchai, "Distributing Streaming Media Content using Cooperative Networking", in *Proceedings of ACM/IEEE NOSSDAV'02*, Miami, FL, USA, May 12-14 2002.

[31] Peer-to-Peer Working Group Website: http://www.P2Pwg.org

[32] PlanteLab. Website: http://www.planet-lab.org/

[33] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, "A Scalable Content-Addressable Network", in *Proceedings. of ACM SIGCOMM '03*, 2003.

[34] R. Rejaie, M. Handley, and D. Estrin, "RAP: An End-to-End Rate-based Congestion Control Mechanism for Realtime Streams in the Internet", in *Proceedings of IEEE INFOCOM'99*, 1999.

[35] S. Ramesh, I. Rhee and K. Guo, "Multicast with Cache(mCache): An Adaptive Zero-delay Video-on-Demand Service", in *Proceedings of INFOCOM '01*, 2001.

[36] S. Saroiu, P. Gummadi, S. Gribble, "A Measurement Study of Peer-to-Peer File Sharing System", in *Proceedings of Multimedia Computing and Networking (MMCN'02)*, San Jose, CA, USA, January 2002.

[37] K. Sripanidkulchai, A. Ganjam, B. Maggs, and H. Zhang, "The Feasibility of Supporting Large-Scale Live Streaming Applications with Dynamic Application End-Points", in *Proceedings of SIGCOMM'04*, 2004.

[38] S. Sheu, K. A. Hua, and W. Tavanapong, "Chaining: A generalized batching technique for video-on-demand", in *Proceedings* of *the IEEE Int'l Conf. On Multimedia Computing and System*, Ottawa, Ontario, Canada, June 1997, pp. 110–117.

[39] I. Stoica, R. Morris, D. Karger, M. Kaashoek, H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications", in *Proceedings. of ACM SIGCOMM '01*, 2001.

[40] S. Sen, J. Rexford, and D. Towsley, "Proxy Prefix Caching for Multimedia Streams", in *Proceedings of IEEE INFOCOM'99*, 1999.

[41] D. A. Tran, K. A. Hua, T. Do, "ZIGZAG: An Efficient Peer-to-Peer Scheme for Media Streaming", in *Proceedings of INFOCOM'03*, 2003.

[42] S. Viswanathan and T. Imielinski, "Metropolitan Area Video-on-Demand Service using Pyramid Broadcasting", *Multimedia Systems* 4, 1996.

[43] J. Wang, "Efficient Content Locating in Dynamic Peer-to-Peer Networks", Master Thesis, Department of Computer Science, The University of British Columbia, 2005.

[44] G. O. Young, C.C. Aggarwal, J.L. Wolf and P.S. Yu, "On Optimal Batching Policies for Video-on-Demand Storage Servers", in *Proceedings of ICMCS '96*, 1996.

[45] E. Zegura, K. Calvert, and S. Bhattacharjee, "How to Model An Internetwork", in *Proceedings of IEEE INFOCOM'96*, 1996.

[46] B. Zhao, J. Kubiatowicz, Jeseph, Anthony, "Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing", *U.C. Berkeley Technical Report UCB/CDS-01-1141*, April, 2001.

[47] X. Zhang, J. Liu, B. Li, and T. P. Yum, "CoolStreaming/DONet: A Data-driven Overlay Network for Peer-to-Peer Live Media Streaming", in *Proceedings of INFOCOM'05*, 2005.