Decentralized Media Streaming Infrastructure (DeMSI): A Peer-to-Peer Content Delivery Network

by

Alan Kin Wah Yim
B.Sc. in Computer Science (The University of Melbourne)

Under the Supervision of:

Dr. Rajkumar Buyya

A minor project thesis submitted in partial fulfillment
of the requirement for the degree of
Master of Information Technology

Grid Computing and Distributed Systems (GRIDS) Laboratory,
Department of Computer Science and Software Engineering,
The University of Melbourne, Australia

November 2004

Table of Contents

Decentralized Media Streaming Infrastructure (DeMSI): A Peer-to-Peer Content Delivery Network

Alan Kin Wah Yim
Grid Computing and Distributed Systems (GRIDS) Laboratory,
Department of Computer Science and Software Engineering,
The University of Melbourne, Australia
Email: a.yim3@pgrad.unimelb.edu.au

Abstract

Hosting an on-demand media content streaming service has been a challenging task mainly because of the outrageously enormous network and server bandwidth required to deliver large amount of content data to users simultaneously. We propose an infrastructure that helps online media content providers offload their server and network resources for media streaming. Using application level resource diversity together with the peer-to-peer resource-sharing model is a feasible approach to decentralize the content storage, server and network bandwidth. Each subscriber is responsible for only a small fraction of such resources. Most importantly, the cost of maintaining the service can also be shared amongst subscribers, especially when the subscriber base is large. As a result, subscribers can be benefit from lower subscription cost. There have been a few solutions out there that focused only on sharing the load of network bandwidth by division of a streaming task to be carried out by multiple sources. However, existing solutions require that the content to be replicated in full and stored in each source, which is impractical for a subscriber as the owner of the storage resource that is of consumer capacity. Our solution focuses on the division of responsibility on both the network bandwidth and content storage such that each subscriber is responsible for only a small portion of the content. We propose a light-weighted candidate peer selection strategy based on avoidance of network congestion and an adaptive re-scheduling algorithm in order to enhance smoothness of the aggregated streaming rate perceived at the consumer side. Experiments show that the performance of our peer-selection strategy outperforms the traditional strategy based on end-to-end streaming bandwidth.

1 Introduction

Hosting an on-demand streaming service of persistent media content, such as video-on-demand, has been a challenging task mainly because of the outrageously enormous network and server bandwidth required to deliver, in real-time, large amount of video data to users simultaneously. In order to deliver a near DVD quality video stream while using as low the streaming rate as possible, a video compression technology such as MPEG-4 [5] is typically used. Informal studies [6] show that the streaming rate required for a near-DVD reproduction is at least 500kbps. Maintaining a big network pipe enough to support the simultaneous video streams and a persistent 500kbps bandwidth per stream for the duration of a movie (ranging from 1-3 hours) is expensive. Therefore a pure single video server cluster to multiple consumers approach is quite a bad idea. The ability to scale is weak. A variant of that is to use multiple server clusters working like proxies in different regional locations or “edges” of the network to allow better scalability [8]. The consumer node is instructed to contact the proxy local to the consumer for streaming. Each proxy may act like the master that carries a replication of the contents, or caches a subset of contents most frequently requested by the local consumers. [7] Such distributed “edge architecture” helps reduce latency and number of hops before reaching the consumer, as the stream is scheduled to deliver from the proxy closest to the consumer. Hence the chance of encountering network congestion is lower. However, it does not mean that a local connection is free of congestion. As [9] suggests, packet loss (hence the congestion) in an end-to-end connection is usually caused by only a few hop-links in the path. Although there are more than one video server cluster to share the server and network loading, the system still suffers from single point of failure problem as the stream is still pushed from a single source over a single connection. Although the stream can be diverted through multiple paths of the network to avoid congestion [10][11][12], the technique is out of question as the routing is beyond the control of the content provider. In addition, since the cost of the servers and the network bandwidth for the streams belongs to the content provider, both the traditional single-server and the edge architecture suffer from under-utilized server and network resources problem during off-peak hours.

The existing problems of implementing a cost effective streaming service of persistent media content as mentioned above lead to the design of DeMSI – The Decentralized Media Streaming Infrastructure. The main objective of DeMSI is to ease the cost of content storage and workload of a video content distribution/delivery network (CDN), traditionally managed by the content provider, by offloading the streaming server, network and

storage resources to subscriber workstations and their upstream internet bandwidth, without sacrificing video quality. Subscribers are not only the consumer of the service, but also a member of the content server. The fundamental idea is to allow multiple subscriber peers to serve streams of the same video content simultaneously to a consuming peer rather than the traditional single-server-to-client streaming model, while allowing each peer to store only a small portion of the content. It is anticipated that a subscriber peer can be a PC workstation, or simply a set top box with a few gigabytes of disk space to spare. Each peer has a broadband connection of at least 1.5Mbps downstream and 256kbps upstream to the internet. DeMSI is designed to be independent of the type of the media content. It is anticipated to work with both CBR/VBR video of any formats and bit rate, and it is not limited to serve video content, but any other media types that are stream-able.

Like other peer-to-peer applications, DeMSI has to face with the reliability issues of peer resources. Since peer resources are pretty much beyond control by the service owner, the domain of the reliability problems that DeMSI has to overcome includes:

1. The unpredictability of dynamics in the condition of connection between the serving peers and the consuming peer. As a connection is made up of a path through hop-links, some links sharing traffic with other connections may be congested that result in delays and loss packets. Hence a varying end-to-end effective bandwidth;
2. A peer may be turned off at any time. Even worse, a peer can be shut down abnormally such that one cannot expect a peer to notify another party of its unavailability;
3. The integrity of contents is vulnerable as the content is stored at the peer end that is beyond control by the service owner.

The integrity of contents can be easily verified by employing a hash scheme, such as SHA-1, to the content data such that a tampered copy of the content can be detected upon deriving from the content a hash code different from the original. In DeMSI, the consumer may send the SHA-1 code of the content segment to the target peer along with the request for streaming. The peer then verify against the local copy and reply either by commencing the stream or a negative acknowledgement. This technique has been used in many P2P applications and we will not discuss this further here. Therefore, addressing problem 1 and 2 are the primary focus of this paper.

As the peers and their connections are unreliable, every P2P application have to deal with re-scheduling of streaming tasks and switching-over of peers when they become unavailable or the service level does not meet expectations. When a DeMSI consumer has a list of candidate subscriber peers discovered or previously contacted by others as consumers, it has to make a selection that achieves the following goals:

1. To maximize the utilization of the network and peers,
2. To minimize the number of peers to serve the content,
3. To minimize the frequency of re-scheduling or emergency switching-over to other candidates over the course of streaming.

The goals are attributed to two important facts. Firstly, the need for fewer peers at a time in streaming implies fewer transitions over the course of a streaming session, and fewer peers are required to be online at any point in time. Secondly, the goals promote stability in aggregated streaming rate from the active serving peers. As a result, less buffering is required for received content prior to a playback. At first glance, peers with largest historic streaming rate should be selected first in order to achieve such a goal. However, this may not be true. Since the internet is made up of hop-links where they can share the traffic from multiple connections, DeMSI should expect there exist two or more candidates that have to send packets through the same hop-link(s). If one of those hop-links has tight bandwidth or filled with cross-traffic, while the previous selection of any one of those peers allows the peer to give 100% of its offered streaming rate, the selection of two or more of those peers may result in congestion such that those peers may only serve at a fraction of their offered rate. As our performance evaluation shows, this results in not only adding fluctuations to the aggregated streaming rate, but also the need for more peers in subsequent scheduling of streaming tasks. Moreover, re-scheduling becomes more frequent. DeMSI deals with this problem from two major directions:

1. *Proactive scheduling*: Candidate peers with the largest historic end-to-end streaming bandwidth, smallest packet loss rate, and offer the largest portion of the content, while they share no or very few congested link(s) with the other actively serving peers, are selected first. The consumer constantly monitors and stores in its knowledgebase the above mentioned network metrics for each peer whenever it is actively serving. In addition, the consumer infers incrementally during the streaming session which peer connections are possibly sharing a congested link in the network, without contributing any additional overhead on the streams.

2. *Reactive scheduling*: The underlying network characteristics of the peer-consumer connections and the availability of the peer itself change over time. Re-scheduling of streaming tasks and emergency switching-over of actively serving peers is unavoidable despite of how good the selection algorithm is. We design a sophisticated divide-and-conquer based scheduling/re-scheduling algorithm that is highly adaptive, flexible, aware of deadlines, and promotes smooth transitions.

As the storage of media content under DeMSI's scenario is decentralized where no single peer contains the complete replication of the content, it is inherent that the consumer has to look for hundreds of peers, which means hundreds of transitions from one peer to another over the course of streaming. The scheduling and rescheduling algorithms have to be light-weighted and perform in a timely fashion.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 provides the design details of DeMSI. Section 4 analyzes the performance of our system in terms of the effectiveness of its peer selection strategy in the scheduling/re-scheduling processes, and the re-scheduling algorithm itself. Finally, the paper concludes with an outline of future work in section 5.

2 Related Work

In the past half a decade, there have been a number of P2P content delivery network models developed and deployed widely. The most popular P2P CDN model is the general file-sharing applications and infrastructures. Napster [28][29][34], Gnutella (Bearshare) [29][30][34], FastTrack (Kazaa) [31][34], eDonkey [32], and Bit Torrent [33] are the popular examples. Each application represents an iteration of improvement in the approaches on resource discovery, peer selection, and content delivery. Table 2-1 provides a summary of their approaches. However, none of today's file-sharing applications support real-time streaming of media content files. Acquisition of a file in those applications is basically accomplished by batched download, which has no notion of sequencing and timing constraint in the delivery timeline.

File-sharing Architecture	Type of Release	Resource Discovery Strategy	Peer Selection Strategy	Content Delivery Strategy
Napster	Application	The consuming peer contacts the centralized global directory server to locate where the file is. More than one peer claimed to have the requested file may be returned but no mechanism exists for verification of actual identity.	None (manual)	Point-to-point single-peer file transfer. Upon peer failure, the user has to manually select another peer that essentially restarts the file transfer.
Gnutella	Infrastructure focused on resource discovery – client examples: Bearshare, XoloX	The consuming peer contacts its neighbor seed peers to locate the file on its behalf. Each contacted neighbor peer in turn forwards the same request to its neighbor recursively if it does not have the file. More than one peer claimed to have the requested file may be returned but no mechanism exists for verification of actual identity.	None (client dependent)	None (client dependent). File transfer is usually performed in point-to-point single-peer fashion by clients released in early days. Most recently released clients support aggregated file transfer from multiple peers selected manually. However, a file has to be downloaded completely into a peer before it can be made available for sharing.
FastTrack	Infrastructure – client examples: Kazaa	The consuming peer contacts its local "supernode", which is another consuming or serving peer with the capability of maintaining a partial file directory of local peers. A supernode may query other supernodes for the requested file. Each peer informs its local supernode upon completion of a file download.	None (client dependent)	None (client dependent). Clients such as Kazaa schedule delivery of the requested file in different blocks, in no particular order, from multiple selected peers to be accomplished simultaneously. When an active peer fails, the user has to manually select another peer to fill in the gap. Recovery from a broken file transfer is inherent. A file has to be downloaded completely into a peer before it can be made available for sharing.
eDonkey	Application	Similar to FastTrack. In contrast, the eDonkey's directory peer also maintains a list of peers who are downloading the requested file as well.	Discovered peers are automatically selected for aggregated file transfer based on the time they are discovered and the available upload slots of the peer. The maximum number of peers allowed in the active set is predefined	eDonkey schedules delivery of the requested file in different blocks, in no particular order, from multiple selected peers to be accomplished simultaneously. When an active peer fails, it looks up another peer to fill in the gap automatically. Recovery from a broken file transfer is inherent. Whenever a fixed-sized chuck of the file is downloaded completely into a peer, it is made available for sharing.

			by user	
Bit Torrent	Application	The consuming peer locates the object by contacting a "Tracker" peer that keeps track of other peers who are currently downloading, and/or have bits and pieces of the same object. The object consists of a set of files predefined by the publisher. The location of the Tracker is found in a token file made available on the web by the publisher.	Random with preference to peers that carry the chucks of an object that are least commonly found in other peers	Bit Torrent schedules delivery of the requested file in fixed-sized chucks from multiple selected peers to be accomplished simultaneously. The chucks that are least commonly distributed are downloaded first. When an active peer fails, it looks up another peer to fill in the gap automatically. Recovery from a broken file transfer is inherent. Whenever a fixed-sized chuck of the file is downloaded completely into a peer, it is made available for sharing. The download rate of a peer is proportional to its upload rate in order to facilitate fairness.

Table 2-1: Summary of popular P2P file-sharing architectures

Another emerging P2P CDN model is commonly known as the *application level multicast* (ALM). As the term implies, the delivery of the content to multiple requested peers simultaneously is achieved on the application layer rather than the network layer, such that it can be used over a traditional unicast network. The motivation of ALM is due to the fact that multicast networks are still rare in today's internet. ALM is commonly accomplished by a one-to-many distribution tree of peers managed either in a centralized fashion at the content source peer such as CoopNet by Padmanabhan et el [26], or in a centralized-decentralized fashion at source and intermediate peers of the distribution tree such as PeerCast by Deshpande et el [27]. However, ALM is essentially a point-to-point content delivery model that relies on a single connection from one peer to each of its child peers. Failure of a parent peer or the path between two peers results in interrupted delivery when the re-orientation of the tree for switching-over to another parent takes place. The Padmanabhan group [26] addressed this problem by the use of multiple distribution trees where multiple sub-streams of the original stream are sent down each peer. The orientation of each node in the tree for one sub-stream is different from that for another. When the parent fails, there is still at least one of the sub-streams likely to reach each child peer. Padmanabhan employs the *multiple description coding* (MDC) on the media content in order to sustain uninterrupted playback of a content under interruption of some of the sub-streams. The MDC is an encoding technique for dividing a media content stream into *m* sub-streams, each of which can be delivered at a fraction of the rate required by the original stream. It also allows partial reproduction of the media content out of $p : p < m$ sub-streams being delivered simultaneously. If the content is a video, partial reproduction results in loss of video quality during playback, typically in terms of lower frame rate than the original.

The idea of allowing multiple peers to push sub-streams of the same media content simultaneously to a consuming peer, in order to share the network bandwidth that is originally required for a single media stream, is now commonly known as *aggregated streaming* or *multiple-sender path diversity* in the research community. This CDN model under P2P paradigm has received the least attention until recently. The concept was probably originated about 3 years ago as Calvert et el [13] outlined it in their Concast paper. The subject of aggregated streaming slowly came into research attention such as the works from Nguyen/Zakhor [14], CoopNet by Padmanabhan et el [26], and finally, Hefeeda et el [15] is probably among the first to integrate this concept with the peer-to-peer paradigm with the introduction of CollectCast (also known as PROMISE). The papers [14][15] and [26] brought out a number of important issues related to aggregated streaming with remarkable solutions. For example, the Nguyen group proposed the use of *forward error correction* (FEC) in their aggregated streaming architecture such that the receiver can recover the original stream by receiving any ξ of $\xi_{FEC} : \xi_{FEC} > \xi$ FEC encoded packets [14], as long as the number of lost packets during the transmission does not exceed $\xi_{FEC} - \xi$. The solution neatly avoids the need of lost packets re-transmission that imposes delay and control overhead. In [15], the Hefeeda group raised the importance of network topology awareness in the selection of candidate peers in the aggregated streaming scenario, in order to avoid having too many active serving peers that deliver the sub-streams through the same link of the network that causes congestion. The Padmanabhan group [26] attempted to support partial content storage in each active serving peer participating in the aggregated streaming in the CoopNet system, by employing MDC on the media content.

The keynotes of the papers mentioned above have become important inspiration in the design of DeMSI. However, the implementation of the MDC technique like the one being used by CoopNet is highly dependent on the type of media content. Moreover, in terms of storage, the content can only be split into *m* parts, where *m* is limited by the number of sub-streams required to be delivered simultaneously in order to achieve original

reproduction quality. In other words, *m* cannot be large. Therefore the partial content to be stored in each peer is still quite large in size. Hefeeda's CollectCast requires each serving peer to store the complete replication of the content rather than a small portion of it as employed in DeMSI's decentralized content storage model. The topology-aware peer selection technique being used in CollectCast is too costly for DeMSI's content storage model that requires visiting many serving peers over the course of a streaming session, typically in the order of hundreds for an hour-long video.

CollectCast requires prior knowledge on the inferred topology of the network being used by the connections between all candidate peers and the consumer before the streaming can commence. Its peer selection algorithm relies on the heavy-weighted *traceroute* utility to help obtain the network topology information before the selection can be made. The accuracy of the topology inference is high, and its granularity can be down to the hop-link level that is visible to traceroute. This enables CollectCast to precisely calculate, for each hop-link shared by a group of peers, which ones can be chosen to serve simultaneously. However, the use of traceroute introduces significant overhead on both time scale and network load because it requires cooperation with the routers. In addition, some routers may not even respond to traceroute requests [16]. Unlike CollectCast, DeMSI does not have any one peer that has a complete replication of the content available. Regular switching of actively serving peers set is required for the duration of a streaming session regardless of peer availability and network condition. In contrast, CollectCast heavily relies on a handful (an average of 4 as discussed in [15]) of peers. Switching of active peers is inherently less frequent in CollectCast's scenario as it only occurs when the peer becomes unavailable or network condition becomes inferior. The number of candidate peers in DeMSI's scenario is way larger than that of CollectCast. The overhead required to infer the topology of the network being used by all candidate peers before the streaming commences is completely out of question for DeMSI. It is intuitive to visualize that the use of a topology-aware peer selection method will not be as effective in DeMSI as in CollectCast. The need to visit a large number of peers makes DeMSI less likely to pick the peers that have to push sub-streams through the same tight hop-link before reaching the consumer for the duration of the streaming session, except in the situation where the total unused bandwidth of all such hop-links in the network is approaching the aggregated streaming rate required to serve a consumer.

The goal of peer selection is to maximize the utilization of the network while minimize the number of active peers at a time to serve the content, and the frequency of re-scheduling or switching-over to other candidates over the course of streaming. DeMSI also requires it to be timely. For that reason, we design alternative solutions for inference of network characteristics and peer selection that sacrifice granularity for efficiency. As the performance evaluation shows, our solution outperforms the selection strategy purely based on end-to-end bandwidth in terms of achieving the goal.

The inference of internal network characteristics using end-to-end measurements is one of the popular areas of research. The idea is commonly referred to as "network tomography". There are two major research directions in this area that we are interested in: 1) Inference of network topology [16][21]; 2) Inference of shared congestion points of the network [17][1][2]. Within each, there are two main focuses on the sender-receiver relationship: Namely the single-sender-multiple receiver (sometimes known as the inverted Y-topology), and the multiple-sender-single-receiver (sometimes known as the Y-topology). To summarize quickly, the topology based inference techniques generally exhibit high approximation granularity, slow convergence (in order of minutes) and overwhelming algorithm complexity. On the other hand, the congestion based inference techniques generally offers lower approximation granularity, converge quickly (in order of seconds) and are light-weighted. Therefore, DeMSI's inference solution is based on inference of shared congestion points, or "congestion based" in short. Its design is inspired by Flowmate [2] - a tool for partitioning flows into clusters each of which represents a congested link in the network. Flowmate uses the packet delay correlation test algorithm proposed by Rubenstein et el in [1] to periodically determine whether the two flows traverse through the same congested link when they come from or go to the same partner. It requires in-band or out-of-band poisson probe traffic to be injected from the sender node to work properly. The two end-nodes may either be receivers or senders, while the partner may either be a sender or a receiver respectively. The idea is that when the flows share a congested link, their probes reach the congested link at time that is a poisson random variable, but they are queued up and serviced at a deterministic rate. As a result, the spacing between packets of different flows after the bottleneck is smaller than the spacing between packets within the same flow. Rubenstein suggests the comparison of correlation coefficients on delay samples from the two flows to detect such phenomenon. As the correlation test algorithm only addresses two flows at a time, Flowmate is built on top of it to support the clustering of multiple flows in an efficient way. Unfortunately, it works with inverted Y-topologies, whereas in DeMSI, connections and flows between serving peers and the consuming peer follow a Y-topology formation. Rubenstein et el also proposed an alternative correlation test algorithm based on packet loss of the two flows. However, experiments show that it converges slower than delay correlation. Another remarkable attempt of inferring shared congestion

points of the network is by Katabi et el [17] who uses an entropy function of packet spacing to determine whether the flows traverse through the same congested link. The idea relies on the fact that packets from various senders are sent at different rates and times. Since the packets from various flows are queued up and serviced at a fixed deterministic rate at the congested link, their inter-packet spacing measured after the bottleneck should be least varied regardless of where the packet is from. Therefore, Katabi's approach does not require extra probe traffic and it is capable of partitioning multiple flows into clusters each represents a congested link. However, it takes more packet samples (hence more time) than Rubenstein's algorithm to converge, especially when the congested link is filled with cross-traffic. Another serious drawback is that it requires prior knowledge on the number of congested links to be identified amongst the flows.

3 Architecture of DeMSI

This section presents our system initially in terms of an overview of its functional components. At the time of writing this report, not all components are fully implemented due to the scope defined in this paper. Each component is described in different level of detail based on the degree of completion in the implementation.

3.1 Overview of Functional Components

DeMSI is the P2P media streaming service middleware that bridges between the content player system at the subscriber end and the other end made up of the CDN itself and other online subscribers. Its key objective is to promote decentralized media streaming from a selection of multiple subscriber peers, and decentralized storage of media contents divided and distributed amongst subscriber peers. At this stage, selection of peers is primarily based on past history of their streaming performance, and congestion avoidance by the analysis of correlation with the sub-stream flows from other selected peers. Figure 3.1-1 shows the block diagram of the components in DeMSI and their relationship in terms of their interactions. Here is an overview of the main workflow of DeMSI: When the user requests a video to be played via the user interface of the Player, it informs DeMSI through the DeMSI-Player API, which in turn kicks off the Scheduler. The Scheduler is in-charge of the initial selection of candidate peers discovered by the Peer Hunter as per Scheduler's request through the DeMSI-Peer Hunter API, and schedules each selected peer to serve the segment(s) of the content, one segment at a time. The peers to which the streaming task is scheduled become active serving peers. In each active serving peer, the segment(s) of the content are then retrieved from the file system locally via the Storage Manager and delivered from the Segment Sender. The sub-stream is received by the Segment Receiver at the consumer side. It stores the sub-stream segment by segment on-the-fly in the Segment Cache, and collects network statistics of a sub-stream flow from the originating peer of the received packet. Concurrently, the Player plays the content by pulling the received segments from the Segment Cache in order, via the DeMSI-Player API. On the other hand, the Peer Monitor performs the following periodically: 1) Checking the health of each active serving peer and determine whether the peer needs more help from another redundant peer candidate; 2) Inferring points of network congestion shared by sub-stream flows if there are any. The Peer Monitor informs the Re-scheduler if there is a need to schedule another peer to assist one of the current active serving peers found to be "unhealthy", such as when a peer goes offline, or the actual streaming rate is below expectation.

Implementation of systems like DeMSI is challenging. Most interactions amongst the components and their activities are actually occurring concurrently. Therefore, in figure 3.1-1, each component represented as a rounded rectangular block is a separate thread executing on its own. In other words, DeMSI is designed and implemented as a team of autonomous agents. When two components are connected by a fat arrow, it means that their interactions are purely one-way asynchronous requests. A mixture of a fat and a thin arrow pointing at the opposite direction of the fat one denotes request-response type interactions. They originate from the starting end of the fat arrow. The fat arrow denotes a request and the thin one denotes a response in this case. The normal rectangular block and the cylinder denote a package of methods to be executed under the caller's thread. However, the cylinder also denotes a repository of data objects: local segment, remote segment, peer, and point of congestion. Most data objects are persistent except remote segment, which stays in the Segment Cache between the time it is received and the time it is consumed by the Player.

As there are many existing works in the research community on resource discovery or lookup substrate over P2P networks, we make DeMSI independent of the substrate as the Peer Hunter agent as long as its implements the DeMSI-Peer Hunter API. As this is not our primary focus at this stage, we do not discuss this further except an outline of what DeMSI requires the Peer Hunter agent to perform, in section 3.4.

Each DeMSI peer uses one TCP port for incoming control flows from other consumers and a UDP port for content sub-stream flows from active serving peers.

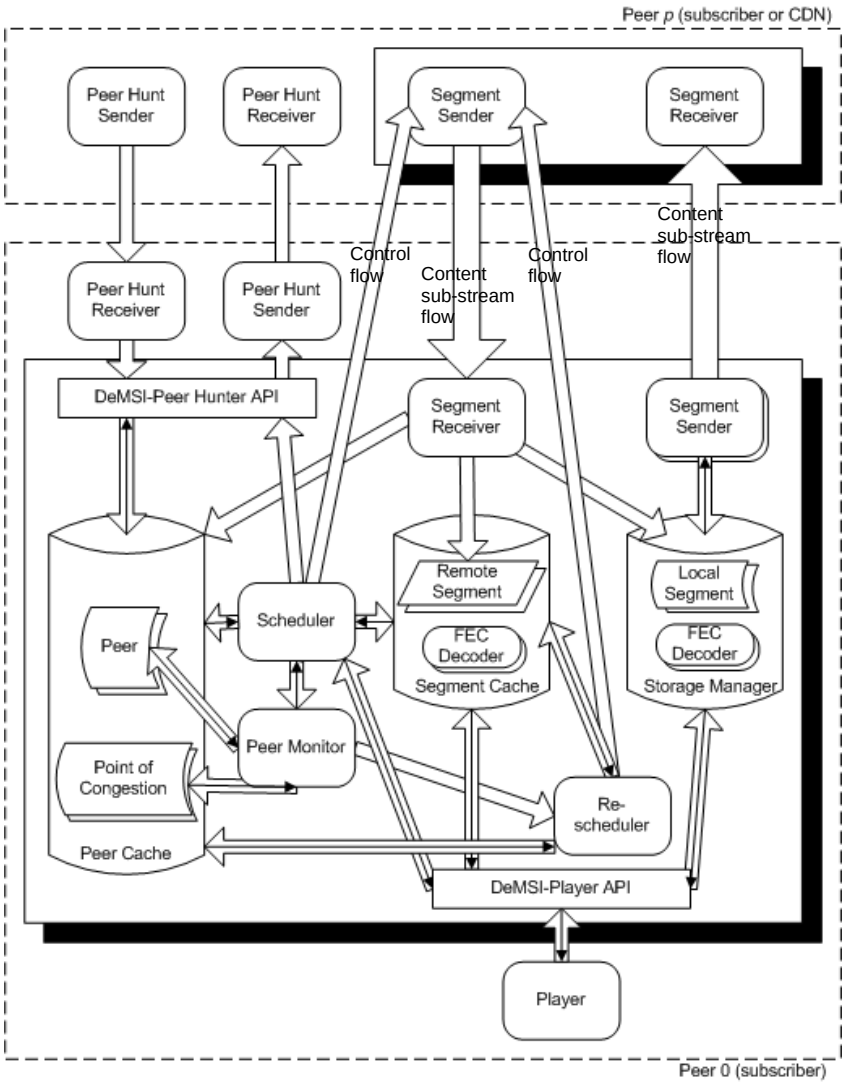

Figure 3.1-1: Components of DeMSI - a team of agents

3.2 Storage Strategy

The storage of media content in DeMSI employs a decentralized with division of responsibility approach. No single subscriber peer stores a complete replication of the content, but a small part of it. As the peers and their connections are unreliable, the aggregated streaming may need to partially rely on reliable resources from the CDN when there are not enough peers available. For that reason, DeMSI supports a special type of peers called the *dedicated server* that offer a complete archive of the contents. Therefore, while it handles unreliable subscriber peers as serving peers, it assumes the existence of some of the more reliable dedicated servers. It may be thought of such peers as being owned by the CDN and/or content publishers. DeMSI assumes that the dedicated servers are online all the time, although it still assumes that the connectivity may still be unreliable. Dedicated servers are treated differently from other peers in terms of peer selection, which is described in more detail in section 3.5 and 3.6.

The division of responsibility strategy leads to the need of dividing a media content into segments before distribution to subscriber peers. A media content M_v is divided into *n* equal-sized segments S_i where $0 \le i \le n-1$. The stream of a media content is now represented by

$$S_0 S_1 S_2 ... S_{n-1}$$

A segment is the smallest unit of data block to be stored in the subscriber's workstation. Each subscriber keeps at least one segment of the same ID for each M_v. Each peer may store *k* consecutive segments of each M_v in such order:

$$S_s, S_{s+1}, S_{s+2}, ..., S_{s+k-1}$$

There are several advantages of assigning a *k* segments consecutively as opposed to scattered. Firstly, it ensures the segment offerings information of each peer to be represented in the most compact manner. Secondly, scaling the segment offerings up or down is simple to manage. Thirdly, it helps ease the subsequent scheduling effort once the consumer finds a peer that offers, for example, the next *k* segments it needs. Most importantly, it helps increase the accuracy of correlating the sub-stream of one peer against others.

While a peer has segments stored locally, other segments that are streamed from other peers for the purpose of consumption are transient. In the current implementation of DeMSI, both types of segments are managed by the Segment Cache. The future version will include the Storage Manager agent that manages the inter-peer re-distribution of the new segments received from other peers through their re-distribution process.

3.2.1 Forward Error Correction and Segment Structure

In order to avoid re-transmission of lost packets that may occur in the streaming, each segment is encoded using a Forward Error Correction (FEC) algorithm before sending to the consumer. The FEC encoding process is associated a parameter known as tolerance level $l_{FEC} : 0 < l_{FEC} < 1$, which indicates the maximum packet loss rate that the FEC can tolerate. DeMSI employs a fixed tolerance level approach such that each segment is stored pre-encoded with FEC at the peer. FEC deals with data in blocks, or in other words, packets. A segment has to be split up further into small blocks, which we call fragments, such that it can be transmitted in series of packets. The fragment size is defined such that each fragment can be fitted into a packet of the sub-stream. It makes perfect sense to use the same segment structure for the FEC algorithm to encode.

Let as define ς as the size of a segment and ρ as the size of a fragment in bytes. Then each segment consists of $\xi = \varsigma / \rho$ fragments. When a segment is encoded with FEC, the size of each segment stored in a subscriber peer becomes $\varsigma /(1 - l_{FEC})$. Hence an encoded segment consists of $\xi_{FEC} = \varsigma / \rho(1 - l_{FEC}) = \xi /(1 - l_{FEC})$ encoded fragments. At the consumer side, a segment S_i is decoded on-the-fly using a separate thread after receiving any ξ of the ξ_{FEC} fragments that belong to S_i.

We decide to employ a Reed-Solomon based FEC algorithm in DeMSI because it guarantees the tolerance level, regardless of order and which ξ of the ξ_{FEC} fragments are received. Moreover, it has existing Java code available [4]. However, the downside of the algorithm is slow, although this does not introduce much of a problem during the evaluation when it is executed in a Pentium 4 class PC. Another FEC implementation known as the Tornado Codes [20] should be more desirable. Unfortunately an existing piece of working code is yet to be found. Although Tornado Codes uses a probabilistic approach, where it does not guarantee a 100% QoS in terms of tolerance level, it claims to be a lot more efficient than the Reed-Solomon's approach.

3.3 The Knowledgebase of Discovered Peers

DeMSI has to maintain the Peer Cache – a semi-persistent knowledgebase of discovered peers for the purpose of monitoring and selection of candidates to be active serving peers. The name "semi-persistent" comes from the fact that the Peer Cache does not maintain a global collection of peers, although the knowledge is stored in the file system for subsequent streaming sessions. Rather, the Peer Cache maintains a limited number of candidates. When DeMSI acquires knowledge of a new candidate peer and the Peer Cache reaches the limit, the least recently contacted candidate is removed. The knowledge of a new candidate peer comes from one of the two sources: either from the response of a Peer Hunter's hunting request, or the hunting request from another peer.

DeMSI maintains a number of service level metrics for each peer in order to aid the peer selection during the scheduling/rescheduling processes, and the decision by the Peer Monitor on whether a particular flow has to be rescheduled. There are 2 groups of metrics: dynamic and static. The dynamic metrics change over time, whereas the static ones remain constant at least for the period of a streaming session. Let P_j be a peer of ID *j*. The static metrics are as follows:

- First segment ID offered: $s_j, s_j \geq 1$
- Number of segments offered: $k_j, k_j \geq 1$

The dynamic metrics are as follows. In particular, the first three metrics are obtained based on the methodologies discussed in [3].

- Average net receive rate of content sub-stream: R_j - This is the actual receive rate of content data detected by the consumer. Let $R_{recv,j}$ be the historic gross receive rate of peer *j*, and *U* be the packet data utilization. The average net receive rate is calculated as
$$R_j = R_{recv,j}(1-l_j)U$$
- Average loss rate of sub-stream packets: $l_j : 0 < l_j \leq 1$ - The percentage of packets lost over a number of packets supposed to be received from P_j.
- Average round-trip time: T_j - This is the time taken for a packet to take a consumer- P_j -consumer round-trip.
- Average response time to a hunting request: t_j - The time taken between the sending of a hunting request and the receipt of the corresponding response from P_j.
- Inferred point of congestion: G_j - DeMSI detects whether the sub-stream flows from the two of the active serving peers share a congested link. Each peer P_j from which the flows are inferred to share the same congested link are put into a group G_j. Please refer to section 3.7 for details.
- Congestion index: $C_j : 0 \leq C_j \leq 1$ - If there exist a G_j for a peer P_j It indicates how congested the shared link, that this peer is believed to be using, is currently. The lower the value the less congested. A C_j of zero indicates that the flows from P_j are believed not to share any congested link with flows from other active serving peers. The value of C_j changes as the set of active serving peers changes:
$$C_j = \sum_{j \in \wp} R_j / (R_{up\max} U)$$ where $\wp$ is the set of active serving peers with the same G_j

3.4 Peer Hunting

It is inherently necessary for DeMSI to look for peers that carry the segment(s) of the content it needs in a decentralized way. It is indeed a challenge to look for hundreds of candidate peers at once. Fortunately, this is not necessary since the Player consumes the content one segment at a time over a period, in ascending order. Peer hunting can be performed for at least 2 segments at a time incrementally. DeMSI works independently from the resource discovery algorithms in order to promote reuse, as there are many such technologies available in the field.

The Scheduler and Re-scheduler agents rely on the Peer Hunter agent to look for at least *c* candidate peers for each segment S_i where

$$\sum_{a=1}^{c} R_{cand[a]} \geq R_{content}, \forall cand[a]$$ are dedicated servers or not dedicated servers,

$R_{content}$ is the minimum required aggregated content consumption rate, $cand[a]$ denotes the peer ID of the *a*-th candidate peer in the candidate list, and $R_{cand[a]}$ denotes the net receive rate of content sub-stream from peer ID: $cand[a]$. At the beginning of a streaming session, DeMSI refreshes and enriches the Peer Cache by asking the Peer Hunter agent to find peers that carry one or more of the *k* segments required by the requested content. Whenever DeMSI is running short of candidate peers that supply a particular segment, such that it has to contact the publisher's dedicated servers for delivery whenever anyone of the serving peers fails to satisfy its estimated net content receive rate and loss rate, DeMSI will ask the peer hunter agent again to find more peers that carry one or more of the next *h* segments including the current segment being delivered. This is known as a repeated peer-hunting request. The number *h* must be at least 2, and is determined such that it is enough to fill up the segment cache of at least $\xi_{\min}$ fragments – which is also a threshold cache level for DeMSI to determine whether it should involve dedicated servers right away, without trying other candidate peers, for delivery. Even though the implementation of this algorithm is beyond the scope of this project, the substrate must satisfy the following requirements:

- As each peer stores the same set of segments for every movie, the resource that the algorithm needs to look for is segment.
- It is essential for the algorithm to find more than *c* candidate peers for each segment requested, at least one of which must be a publisher's dedicated server. If it manages to find only one candidate peer, it must be a dedicated server.
- It shall confine the scope of peer-hunting down to the consumer's local communities. The scope of hunting may only be expanded upon a repeated hunting request.
- It is preferred that the substrate to be capable of estimating the candidate peer's upstream bandwidth in return. One way to achieve that is to employ a fast packet-dispersion based estimation method, such as SProbe [19] at the candidate peer side. The estimation involves overhead of only a few packets and a couple of round-trips of several tens of miniseconds. For candidate peers of which the upstream bandwidth cannot be estimated and are new to the consumer, the requested streaming rate R_{req} when the peer is selected, is initially $R_{up\min}$ - the minimum gross upstream rate of the peer.

Existing resource discovery substrates such as Pastry [18] can be a good candidate to be the Peer Hunter agent, as it has the notion of locality in the search. However, further enhancement on the substrate is unavoidable in order to satisfy the requirements stated above and be compatible with the DeMSI-Peer Hunter API.

3.5 Scheduler

The Scheduler is an agent that coordinates peer hunting and dispatches various streaming and peer monitoring tasks to be carried out during the streaming session upon request from the Player agent. The media content is served in terms of an aggregation of *p* sub-stream flows from *p* active serving peers at a time where $p > 0$. Let *actv(a)* denotes a function that returns the peer ID of *a*-th active serving peer. *p* is determined according to the historic average net receive rate of content sub-stream $R_{actv(a)}, 1 \le a \le p$ of each selected peer. As figure 3.5-1 shows, each active serving peer is assigned a fraction of the segment to be delivered to the consumer. The number of fragments to be delivered is proportioned by

$$\min(R_{actv(a)}\alpha / R_{content}, 1)$$

where $\alpha : 0 < \alpha < 1, \alpha \in \Re$ is called the rescheduling threshold. The use of α prevents the decision to reschedule from being too sensitive to noise from the network and the statistical oscillations in calculation of $R_{actv(a)}$. The peers serve the assigned range of fragments in parallel until the consumer instructs them to stop. The total number of fragments $\xi_{total,actv(a)}$ of segment S_i to be delivered from all the active serving peers is based on the smallest of the tolerance level, and the 2 times the highest loss rate ($l_{\max}$) of active serving peers such that

$$\xi_{total,actv(a)} = \varsigma / (\rho(1 - \min(l_{FEC,} 2l_{\max})))$$

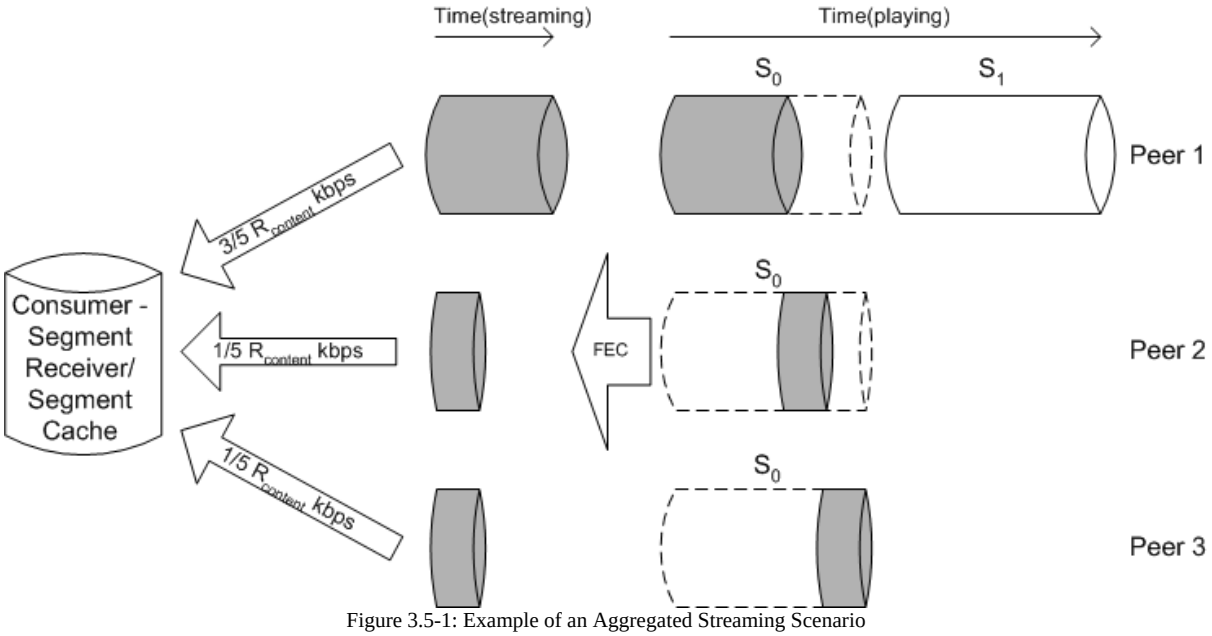

Figure 3.5-1: Example of an Aggregated Streaming Scenario

The segment next to the most recently delivered one, or the first segment to be scheduled for delivery in a streaming session is called the *urgent segment*. There is only one urgent segment at any time of a streaming session. The urgent segment is given priority in scheduling and re-scheduling processes. The selection of candidate peers and scheduling of streaming tasks for each segment is described in the following pseudo code:

1. $f_b = 0; f_e = 0; l_{\max} = 0; \xi_{total,i+1} = \xi_{FEC};$ // where f_b, f_e denote first fragment ID, last fragment ID to be scheduled for delivery, respectively
2. For each segment $S_i, 0 \le i < numSegments(M_v)$
3. Get list of discovered peer candidates that carry segment S_i (excluding the ones tried in previous round) sorted by subscriber first, C ascending, R descending, online first, *s* descending, *k* descending, l ascending, T ascending, t ascending;
4. For each peer candidate P_j from the list until all $\xi_{total,i}$ fragments have been scheduled or end of list
5. If S_i is not urgent & (P_j is a dedicated server or P_j does not carry the urgent segment as well or C_j >0), continue with next candidate;
6. If S_i is urgent & no. of fragments decoded $\le \xi_{\min}$ & P_j is not dedicated server, continue with next candidate;
7. If no. of fragments decoded > $\xi_{\min}$ & P_j is dedicated server, *PeerHunter.findPeers*(S_i, S_{i+1});
8. If P_j can be connected, wait until $R_{recv} - \min(R_{up\max}, \psi R_j) < R_{down\max}$; Else continue with next candidate; // R_{recv} is the aggregated gross receive rate; $R_{up\max}$ is the maximum gross upstream rate from a peer; ψ is the growth factor allowed for R_j ; $R_{down\max}$ is the max allowed aggregated gross downstream rate at the consumer;
9. $f_e = f_b + \xi_{FEC} \min(R_j \alpha / R_{content}, 1);$ // $0 < \alpha < 1, \alpha \in \Re$ is the rescheduling threshold
10. If *requestDelivery*(P_j, f_b, f_e, R_j) is successful, { $f_b = f_e + 1$; $R_{e,i,j} = R_j$; $\xi_{total,i} = \varsigma / (\rho(1 - \min(l_{FEC,}, 2l_{\max})))$; Repeat from 3} else $f_e = f_b$; // where $R_{e,i,j}$ is the estimated content receive rate for the delivery request
11. End For;
12. If S_i is not urgent, wait until S_i is urgent;
13. If there are still fragments remained to be scheduled, repeat from 3;
14. $l_{\max} = 0; \xi_{total,i+1} = \xi_{FEC};$
15. End For;

For newly discovered peers, the consumer has only the static service level information about the candidate serving peers discovered. The dynamic service level information is mostly unknown except t_j. Peers offering the same segment are initially selected in ascending order of t_j, and if the candidate list is big enough, the selection process avoids picking peers that have the same first 24 bits of the IP addresses except the first one in the sorted candidate list. The intuition is that the longer the t_j, the more probable that the candidate is further from the consumer, the more probable that the packet path encounters a congested link. As it is common to allocate the last 8 bits of the IP addresses to the same ISP, or in many cases, to the same LAN of an enterprise, peers that have the same first 24 bits of the IP address has quite a high probability of sharing the same backbone that may be of limited capacity. A special case of that is when there are peers that have the same IP address, that probably suggests they are behind the same firewall/NAT that may introduce bottleneck. All selected candidate peers that are new are initially allocated the request streaming rate R_{req} equals to $R_{up\min}$, except in the case where the Peer Hunter agent supports rate estimation as discussed in section 3.4. This is also the basis for the estimated net content receive rate $R_j = R_{up\min}(1 - l_j)U$. As the sub-stream flow arrives the consumer node, the rest of the dynamic service level metrics can be collected.

3.6 Segment Receiver

While the Scheduler agent schedule the streaming of content segment by segment, the Segment Receiver agent listens to the UDP port for streams of fragments from the active serving peers. It parses each packet received and updates any dynamic service level information: R_j, l_j, T_j of the source peer data object P_j whenever applicable. The timestamps from both the origin and the receiving end are stored if the packet contains either a round-trip-time reply or a probe for inference of shared congestion points. Please refer to section 3.7 for details.

In an event of changing packet loss rate and/or round-trip time for the sub-stream flow from P_j, the Segment Receiver adjusts the upstream rate $R_{req,j}$ according to the renewed calculation of the *TCP friendly rate* [3] based on round-trip time and packet loss rate. This congestion control mechanism ensures that both the round-trip time and the loss rate can be under control. The peer P_j is informed of such change only regularly by the Peer Monitor as discussed in section 3.7. We fine-tune the TCP friendly rate equation in order to allow a slightly more aggressive streaming rate allocation in the expense of a slightly higher delay to reach its equilibrium state:

$$R_{req,j} = \frac{\rho}{U(T_j\sqrt{\frac{2000l_j}{3}} + 12T_j\sqrt{\frac{3000l_j}{8}}l_j(1+32l_j^2))}$$

We anticipate that the future version of the Segment Receiver will also handle the reception of segments of a new content re-distributed from other peers, and co-ordination of the archival process with the Storage Manager.

3.7 Peer Monitor

The Peer Monitor agent invokes itself regularly by a fixed interval. It performs the following tasks at each execution for each active serving peer:

1. The sending of a request for measurement of round-trip-time between the consumer and the active serving peer. The request is sent once per second except at the first two seconds of a session with a particular peer, the evaluation frequency is at 4 at the first followed by 2 at the second in order to reduce the extra delay in response occurred in the initialization stage at the peer side.
2. As the loss rate of a peer is usually well below the FEC tolerance level, it is a waste of network resource to have all the redundant fragments delivered in order to support the tolerance level. Therefore, the Scheduler usually does not schedule all the redundant fragments to be delivered to the consumer. However, when the loss rate of an active peer goes beyond the rate estimated at the time of scheduling, the Peer Monitor will assign an instance of the Re-scheduler agent to schedule another group of candidate peers to deliver the remaining redundant fragments of the segment. This task is performed once a second.
3. Examination of the dynamic service level information at an interval of one per second. The Peer Monitor informs the Re-scheduler to re-schedule the delivery of a range of fragments $[f_b, f_e]$ upon encountering one of the following events from an active peer P_j:
   - When the renewed TCP friendly rate $R_{req,j} < R_{up\min}$
   - When the current net content receive rate R_j is smaller than the estimated content receive rate of the current delivery request: $R_{e,i,j}\alpha$
   - When an active serving peer goes offline suddenly
   - When $l_j > l_{FEC}$
4. Inference of network congestion points possibly shared by the sub-stream flows, as discussed below.

The peer selection technique being used by DeMSI employs a network congestion avoidance strategy. In order to achieve the goal, DeMSI has to have knowledge about the path of the sub-stream flow from each peer such that the hop-links that shared by two or more sub-stream flows can be identified. Recent works [15][16][21] indicate that the inference of fine-grain knowledge such as network topology takes too much time to converge even for a small network consisted of a few peers. DeMSI has to visit a diverse selection of candidate peers over the course of the streaming due to the decentralized storage of content segments. The adverse effect of having a particular group of peers streaming through the same congested link of the network becomes less significant as each sub-stream flow is likely to be short-lived. The inference of coarse-grain knowledge such as shared congestion points [17][1][2] of the network is enough for the purpose. The short period of convergence is what

DeMSI requires, as a sub-stream flow from each peer is often short-lived. A life span ranging from 1 to 5 seconds out of a segment of 10-second playing time is typical. It can be worked around by forcing a peer to deliver at least 2 segments consecutively, but this significantly reduces the flexibility to scale the storage offering from a subscriber. Another challenge for DeMSI to implement a congestion based inference algorithm is the need to have packets flow at the same time, from the peers to be correlated, for the period of correlation. As it is impossible to correlate a large number of candidate peers (in the order of hundreds) before the selection process can even start, the knowledge is accumulated incrementally during a streaming session.

Our inference algorithm extends Rubenstein's method of determining whether two flows share a congested link by correlation test on packet delay samples [1]. The Peer Monitor regularly performs pair-wise correlation of sub-stream flows from the set of active serving peers once every second to determine whether there are any two peers share a congested link. The mappings between the peers and the congested links are kept across streaming sessions. This is made possible by measuring the correlation of time spacing between adjacent probe packets, spaced apart by time $\omega_x > 0$ from two sub-streams (in terms of a cross-correlation coefficient M_x) and the correlation of time spacing between successive probe packets, spaced apart by time $\omega_a > \omega_x$ from one of the two sub-streams (in terms of an auto-correlation coefficient M_a). When $M_x > M_a$, the sub-streams share at least one congested link. Otherwise they do not. The idea is that, there are two sub-streams of packet flow where the time spacing between successive probe packets within a flow is a poisson random variable of mean λ. When they flow through a pipe with a service rate larger than their aggregated rate, the time spacing between packets should remain pretty much the same as they do not queue up. Therefore the time spacing remains poisson – hence uncorrelated. In contrast, when the probe packets of the two sub-streams travel through a congested link, the time spacing between adjacent probe packets from two sub-streams is shorter than that between successive probe packets of one sub-stream. The spacing between the probe packets no longer follows the poisson distribution due to the fact that they now follow the same independent-identically-distributed general distribution as that of the congested link's service rate, which introduces correlation in the spacing between packets of the sub-stream flows. The delay of each probe packet is calculated using the timestamps from both the origin and the receiving end. As Rubenstein's correlation test algorithm assumes no network layer path diversity in the topology used by the flows, the same assumption applies to our inference algorithm.

Here is how the active serving peers are grouped together by point of congestions identified incrementally during a streaming session. At the very beginning, the Peer Monitor assumes no peers share any congested links. When the sub-stream flows from active peer P_1, P_2 are found to share a congested link, a group g_1 that represents a point of congestion is created. P_1, P_2 are then inserted into that group. Later in the streaming session, P_1 no longer delivers but P_3 starts delivery. The Peer Monitor takes the duration of time *d*, or d / λ probes in each sub-stream to find out that P_2, P_3 share a congested link. Knowing that the flow paths from the peers converge as they approach the consumer, and the paths usually remain unchanged for at least a day [9], it is quite safe for our algorithm to adopt a transitive induction approach to relate a new inference to existing ones inferred minutes before. Therefore, P_3 joins g_1 as a result because P_2 belongs to g_1.

Now let us assume there is another group g_2 formed with members P_7, P_8, P_9 in another streaming session. P_2 is no longer an active peer but P_1, P_8 , and they are found to share a congested link. Since P_1 belongs to g_1, P_8 belongs to g_2, and g_2 has more members than g_1. g_1 is deleted and the members of g_1 are moved to g_2 as a result.

The inference algorithm of the Peer Monitor employs a conservative approach in determining whether the two sub-stream flows are routed through a congested link. In other words, the inference algorithm would rather return false negative (determine the sub-stream flows do not share a congested link but they actually share one) than false positive. Both types of error have an adverse effect on the inference accuracy. False negatives lead to increasing probability of selecting peers that share a congested link into the set of active serving peers. However, false positives lead to lower utilization of peers that do not shared a congested link as well as the adverse effect of false negatives. First let us denote $M_{x-1,2}$ as the cross-correlation coefficient resulted from the calculation of delay samples from P_1 against those from P_2. M_{a-1} is the auto-correlation coefficient resulted from the

calculation of delay samples from P_1. When two sub-stream flows from active peer P_1, P_2 are found to be correlated (share a congested link) initially by testing whether $M_{x-1,2} > M_{a-1}$, the sub-streams is then re-tested on whether $M_{x-2,1} > M_{a-2}$. Second, our experiments indicate that false positives often result when M_a is small, which implies that the flow itself is unlikely to be congested by any hop-links it traverses though. Therefore, our algorithm considers P_1, P_2 to be correlated only if they pass both tests on correlation coefficients, and if $\min(M_{a-1}, M_{a-2}) \geq \delta$. If they pass the two-sided correlation tests but $\min(M_{a-1}, M_{a-2}) < \delta$, no conclusion is made and the outcome of the comparison is ignored. Otherwise, they are considered uncorrelated. Experiments showed that the combined use of the two-sided correlation tests and the avoidance of small M_a had significantly reduced the chance of getting false positives. The downside is that it also slightly increases the chance of getting false negatives. Another case is if two active peers are found to be uncorrelated but they have been allocated in the same group, they will be both removed from that group according to the philosophy of the conservative approach.

3.8 Re-scheduler

Network conditions in terms of dynamic service level metrics and the peer availability change over time. Although the trend on time-series usually follows a pattern [22][23], when it comes to very short and immediate terms, the changes occur by random quantities at random time possibly within a range bounded by the trend. It is crucial for DeMSI to be reactive of random adverse changes in a timely fashion, by assigning additional peers to rectify the lagging aggregated streaming rate and time-to-play deadlines. This is where the Re-scheduler agent comes into play. There can be multiple instances of Re-scheduler agent each of which takes care of a re-scheduling task concurrently for various ranges of fragments to be received.

Assuming that there is an active serving peer P_{actv} which is delivering fragments up to $f_{curr,0}$ of segment $S_{dr(0)}$, where *dr(0)* denotes the segment ID of the delivery request *r* = 0 currently being served. It has been scheduled to deliver up to fragment $f_{e,0}$ but the Peer Monitor has detected an event that requires re-scheduling. The role of the Re-scheduler is to find and schedule another candidate peer that is suitable for assisting P_{actv} to deliver the range of outstanding fragments. The algorithm for the Re-scheduler takes a highly adaptive divide-and-conquer approach. Firstly, as P_{actv} is still delivering the fragments at a slower than expected rate, the range of outstanding fragments is re-scheduled to be delivered by another peer P_j in a reversed direction of the current sub-stream by P_{actv} in order to avoid repeated delivery of the same fragments. Secondly, as it cannot assume that P_j can assist P_{actv} within the newly estimated time frame, the re-scheduling algorithm simply treats this new schedule as another smaller delivery request r_j which is assisting the original one r_{actv} scheduled to P_{actv}. In other words, the algorithm may locate another peer to assist r_j. We call this the "*spiral*", or recursive divide-and-conquer re-scheduling strategy. Like the Scheduler, the re-scheduling algorithm has a notion of the "urgent segment", which is the segment next to the most recently delivered one. The key implication of the urgent segment in the perspective of the Re-scheduler is that any active serving peers will be called upon if they have a copy of the urgent segment, unless they are serving some other fragments of the urgent segment. In other words, even though the peer is delivering a non-urgent segment, the peer will be preempted to serve the urgent segment first as instructed by the Re-scheduler.

The pseudo code for re-scheduling is as follows. Figure 3.8-1 illustrates an example on how a delivery request is re-scheduled, in a spiral fashion, to be carried out by another peer.

1. *r* = 0; This is round 1 of the workflow;
2. For each $S_{dr(r)}$ in delivery request *r* scheduled for the active serving peer P_{actv}
3. If this is round 1, Get list of discovered peer candidates that carry segment $S_{dr(r)}$ (excluding P_{actv} & others tried in previous round for the same delivery request) sorted by subscriber first, online first, C ascending, l ascending, R descending, *s* descending, *k* descending, T ascending, t ascending;

4. If this is round 2, Get list of discovered peer candidates that carry segment $S_{dr(r)}$ (excluding P_{actv} & others tried in previous round for the same delivery request) sorted by subscriber first, offline first, C ascending, l ascending, R ascending, *s* descending, *k* descending, T ascending, t ascending
5. For each P_j from the list until end of list
6. If this is round 1 & P_j is offline, {re-sort the list of discovered candidates by subscriber first, offline first, C ascending, l ascending, R ascending, *s* descending, *k* descending, T ascending, t ascending; The workflow is now in round 2; go to 5 };
7. If this is round 2 & P_j is online
8. If *r* is 0 & $S_{dr(r)}$ is not an urgent segment,
9. Wait until $S_{dr(r)}$ becomes urgent; The workflow is now back to round 1;
10. Repeat from 3;
11. Else go to 33; // It means that the Re-scheduler has run out of candidates. P_{actv} has to be on it own!
12. End If;
13. End If;
14. If *r* is 0 & $S_{dr(r)}$ is not an urgent segment & (P_j is a dedicated server or P_j does not carry the urgent segment as well or C_j >0), continue with next candidate;
15. If size of segment cache < $\xi_{\min}$ & P_j is not a dedicated server, continue with next candidate;
16. If P_j can be connected, wait until $R_{recv} - \min(R_{up\max}, \psi R_j) < R_{down\max}$; Else continue with next candidate;
17. $\Delta = \left| f_{e,r} - f_{curr,r} \right|$; // Δ is the number of fragments left to be delivered - 1
18. $\eta = \left\{ \begin{array}{c} 0, if \quad \Delta == 0 \\ -1\dfrac{f_{e,r} - f_{curr,r}}{\left| f_{e,r} - f_{curr,r} \right|}, otherwise \end{array} \right\}$ // $\eta \in \{-1,0,1\}$ is the unit-direction multiplier to indicate the direction of streaming for the new schedule. That is, the opposite of the direction for the current schedule.
19. $R_{res} = \min(R_{content}, (\Delta + 1)\rho / \tau_{left})$; // R_{res} is the new content stream rate required from the candidate; $\tau_{left} = \max(1, (\varsigma / R_{content}) - \tau_{elasped} - 1)$ is the time left for fulfilling the delivery of this range of fragments; $\tau_{elasped}$ is the time already spent on the delivery of the current fragments range
20. $f_{b-new} = f_{e,r}$;
21. $f_{e-new} = f_{b-new} + \eta\Delta \min(R_{j-cand}\alpha / R_{res}, 1)$;
22. If *requestDelivery*(P_j, f_{b-new}, f_{e-new}, R_j) is successful
23. If current active peer is still online
24. Inform the current active peer to deliver up to fragment $f_{e-new} + \eta$;
25. Renew the estimated net content receive rate of the current active peer $R_{e-actp} = \min(R_{content}, (\left| f_{e-new} + \eta - f_{curr,r} \right| + 1)\rho / (\tau_{left}\alpha))$;
26. Go to 34;
27. Else
28. If $\eta f_{e-new} < \eta f_{curr,r}$, { $f_{e,r} = f_{e-new} + \eta$; go to 3 }
29. End If;
30. Go to 34;
31. End If;
32. End For;

33. If *r* is 0 & tried all P_j & $S_{dr(r)}$ is not urgent, { Wait until $S_{dr(r)}$ becomes urgent; The workflow is now in round 1; go to 3 };
34. If P_j is still online, quit;
35. End For;

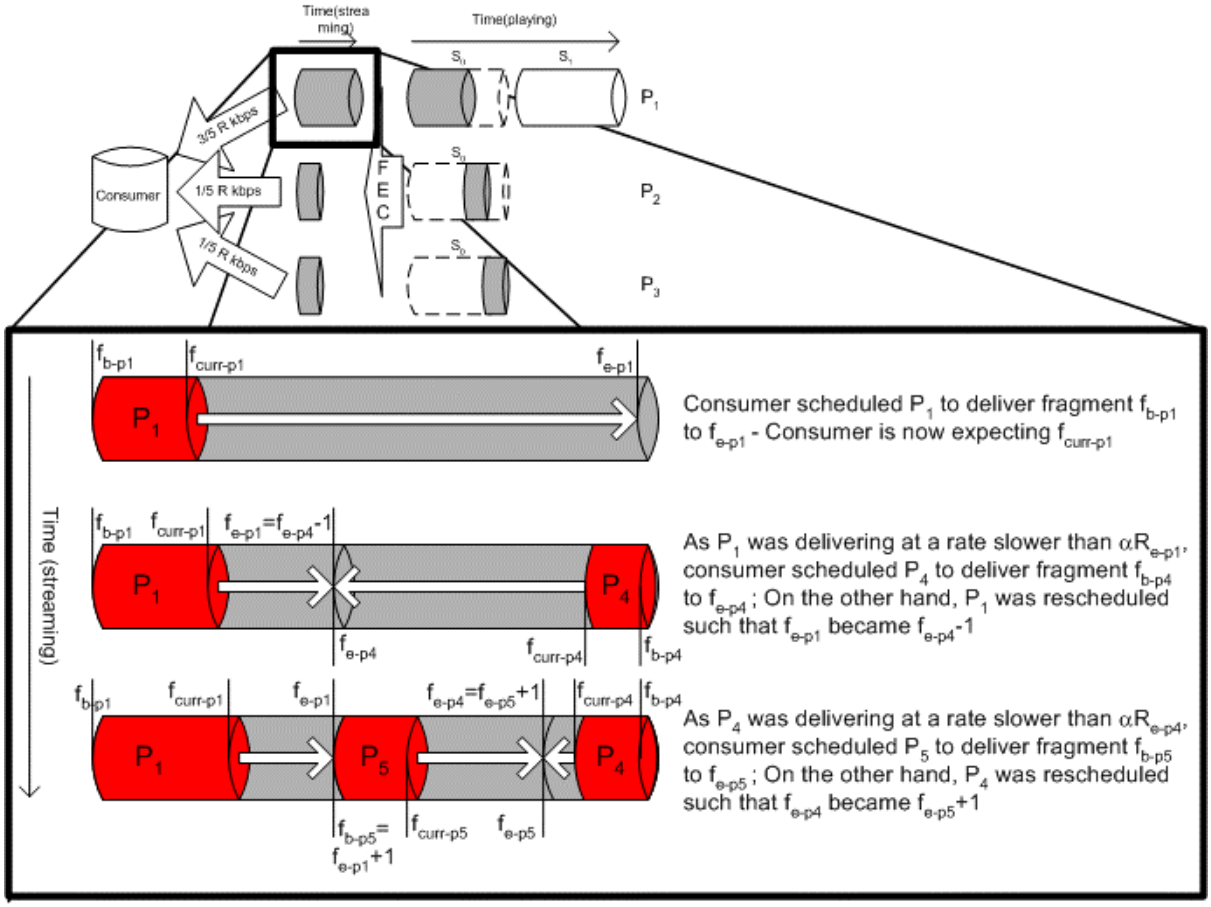

Figure 3.8-1: An example to show the Re-scheduler at work – a delivery request is rescheduled in a spiraling divide-and-conquer fashion

A more aggressive extension for scheduling/re-scheduling algorithm is to maintain an idle connection with a redundant peer after each segment is scheduled for delivery by the Scheduler, and after an outstanding delivery request is re-scheduled by the Re-scheduler. This strategy moves the time-consuming socket connection process to an earlier time before the failure event occurs. This ensures a smooth transition in the case of an emergency switch-over such as when an active serving peer goes offline while the streaming is in progress. One way to implement this is to have the Scheduler/Re-scheduler spawn a separate thread, which tries to establish a TCP connection with the next candidate peer in the sorted list until one of them is connected. This peer only serves as a stand-by when there is no re-scheduling activity. Otherwise, the Re-scheduler agent spawned at a later time may communicate with the redundant peer right away without the need to make a prior TCP connection. When the redundant peer is consumed, the Scheduler/Re-scheduler has to locate another one immediately in case of subsequent use. In the case where the candidate list is exhausted or left with only dedicated servers, the thread approaches the peer hunter to discover more peers that carry the segment it needs before the trial connection process can continue.

3.9 Segment Sender

The Segment Sender agent is responsible for the delivery of segment in part or in whole, in terms of a sub-stream of fragments as per delivery request from the consumer. Fragments can be streamed in either forward or backward sequence in order to be compatible with the re-scheduling algorithm. The streaming in progress may be preempted by a subsequent delivery request from the Re-scheduler, if it is requesting a segment of which the ID is smaller than the current one in delivery. The Segment Sender also handles round-trip-time request token, and the generation of probes as required by the Peer Monitor for the inference of congestion points. The probes are generated such that they are spaced apart by x where x is a poisson random variable with mean λ. The round-trip-time reply and the probe are piggybacked onto the sub-stream packet. As every sub-stream packet

contains timestamps at the origin and receiving end, the probe does not introduce any additional overhead. It is distinguished from a normal sub-stream packet by simply flipping the packet ID field to a negative value.

We anticipate that the future version of the Segment Sender will also participate in the new content re-distribution process. Its role will be to deliver the whole segment to other peers.

4 Performance Evaluation

We evaluate DeMSI under a simplex (one-way) network as shown in figure 4-1 emulated by the NS-2 network simulator [25]. The network is made up of eight hop-links. Each cloud represents a combination of 3 Pareto traffic sources as cross-traffic. In particular, each of the clouds c1, c4, c5, c8 also contains 3 CBR traffic sources. Pareto sources are good approximation of the web traffic that is self-similar, whereas CBR sources are to approximate other long-lived streaming traffic. To simulate the asymmetric upstream/downstream bandwidth offered by mainstream ADSL modems of today, every peer is offered a 32kB/s connection to the network. On the other hand, the consumer has a 192kB/s connection to the network. The maximum gross upstream rate $R_{up\max}$ offered by each peer is also set to 32kB/s. In other words, assuming the cross-traffic arrives at its maximum rate allowed at its link, each of the four tight hop-links: *r1-r9, r4-r9, r5-r9, r8-r9* allows at most one peer streaming at maximum rate, while another is streaming at marginally less than the maximum rate simultaneously. Although every link has the same propagation delay of 1ms, the bandwidth allocated to each link, the average rate of each cross-traffic source, and the shape parameter of each Pareto traffic source is different in order to promote heterogeneity. As the flow of the control packets is of low volume and the control packets are small in size, the impact of control flow delay and its difference between the consumer and each peer is insignificant relative to the difference in delays of the sub-stream flows. Therefore we focus on emulating the downstream paths (towards the consumer) of the network.

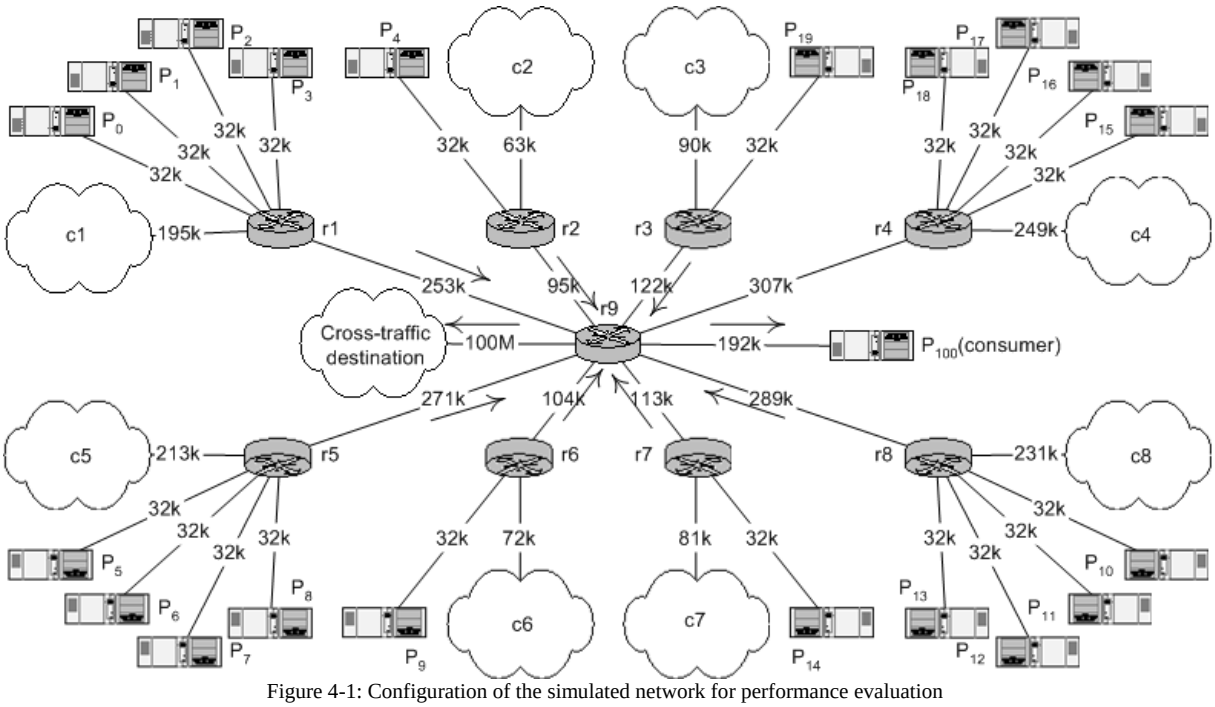

Figure 4-1: Configuration of the simulated network for performance evaluation

Figure 4-2 illustrates how the system is set up for the experiments to be carried out for evaluation. We split the peers into 2 groups of 10. One group consists of peers with odd peer ID numbers, while another group consists of peers with even numbered peer IDs. Each group is assigned to be executed on a Pentium 4 2Mhz class workstation. The consumer peer is executed on one of the two workstations. Since we emulate a network in real-time using NS-2, we assign the third workstation for the NS-2 exclusively. NS-2 has to be executed in real-time mode under Windows XP such that it can catch up with the events occurring in real-time. Normally, during the scheduling or re-scheduling process, the consumer tries to establish a TCP connection with the selected candidate peer before the control flow, consists of delivery requests and round-trip-time requests, commences. The candidate peer becomes an active serving peer by pushing directly to the consumer a sub-stream flow of content fragments on UDP packets. Under the NS-2 scenario, the way to establish TCP connections remain as normal. However, UDP flows are emulated. The UDP packets from an active serving peer are sent to the NS-2

workstation as if it is the consumer. NS-2 eventually forwards most UDP packets to the real consumer at emulated rates and with emulated delay. Some packets are not forwarded due to emulated packet loss occurred in the middle of the network.

We have implemented a prototype of DeMSI which includes a Player with a progress monitor user interface as shown in figure 4-3. Although a DeMSI peer is both a consumer and a content server, we have implemented a prototype that supports an optional serving-peer-only execution. With this option, the consumer related components including the Player and its user interface, Segment Receiver, Scheduler, Re-scheduler and the Peer Monitor are turned off. The process under this execution option is compact enough to allow multiple instances of it to be executed on the same workstation for evaluation purpose. The prototype is implemented in Java 1.4.2 with Java Media Framework 2.1.1. The perceived dynamic service level statistics: R_j, l_j, T_j, and number of active serving peers are collected and written into a file once a second for further analysis. On the other hand, each claim and peer-point of congestion mapping update resulted from a pair-wise flow correlation test is written into a file whenever it becomes available.

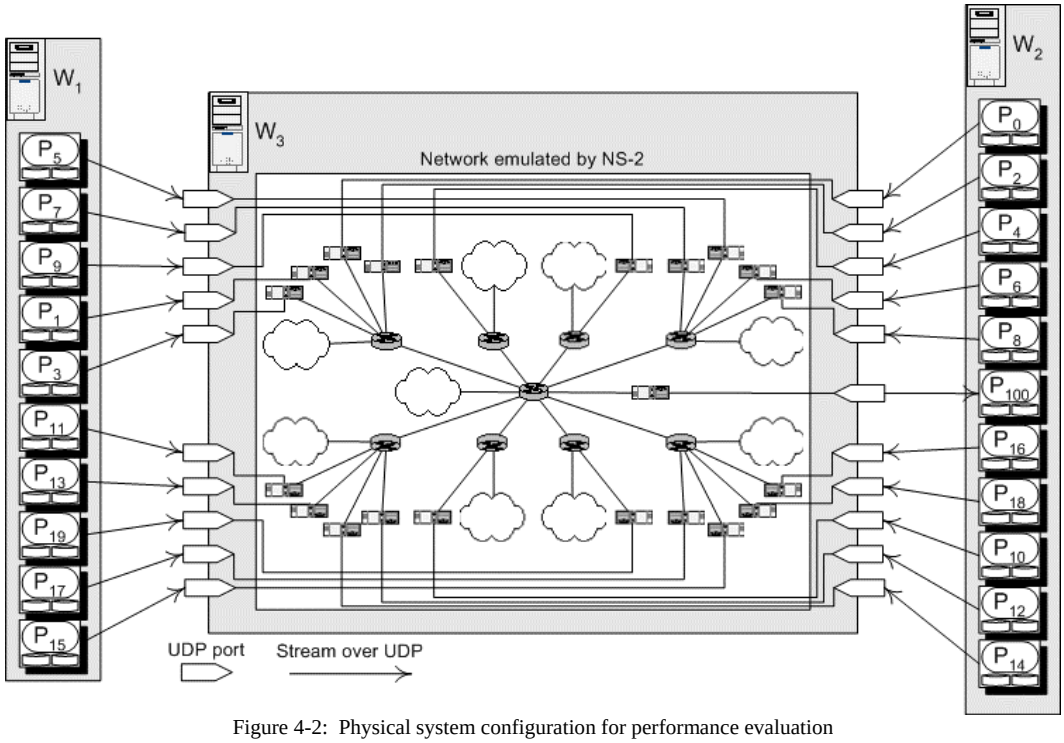

Figure 4-2: Physical system configuration for performance evaluation

We encode a small portion of a video clip using MPEG-1 with a constant consumption rate $R_{content}$ of 100kB/s for evaluations. We use a rather legacy MPEG-1 format simply due to the constraint of the Java Media Framework that we have leveraged on a quick implementation of the primitive Player agent. The clip consists of 25 10.24-second segments. Each segment that is ready to play contains 1024 fragments. The size ρ of each fragment is 1kB. The data utilizes 96% of a stream packet on average. We use a tolerance level l_{FEC} of 0.2 for the FEC such that each segment encoded with FEC contains 1280 fragments. The Peer Hunter agent has been implemented as a stub that simply reads from an XML formatted file a pre-defined list of candidate peers as if they are discovered as per hunting request. In order to ensure the congestion occurs in the experiments, each candidate *j* is assigned the following every time when DeMSI is started:

$$R_j = R_{up\max}(1-l_j)U,\ l_j = 0.001,\ T_j = 1ms,\ t_j \text{ is assigned a random value}$$

When DeMSI is started, it has no knowledge of congestion information. Therefore, the initial selection of peers is essentially by random. We use the α of 0.84 for all experiments such that if peers deliver at $R_{up\max}$, the Scheduler will schedule 4 peers to stream. Segments are distributed to peers evenly. Each segment $S_i, 5 \le i \le 19$ is distributed to 8 peers. Each segment $S_i, 0 \le i \le 4, 20 \le i \le 24$ is distributed to 4 peers. 4 peers are dedicated servers. Table 4-1 provides the details of the assignment.

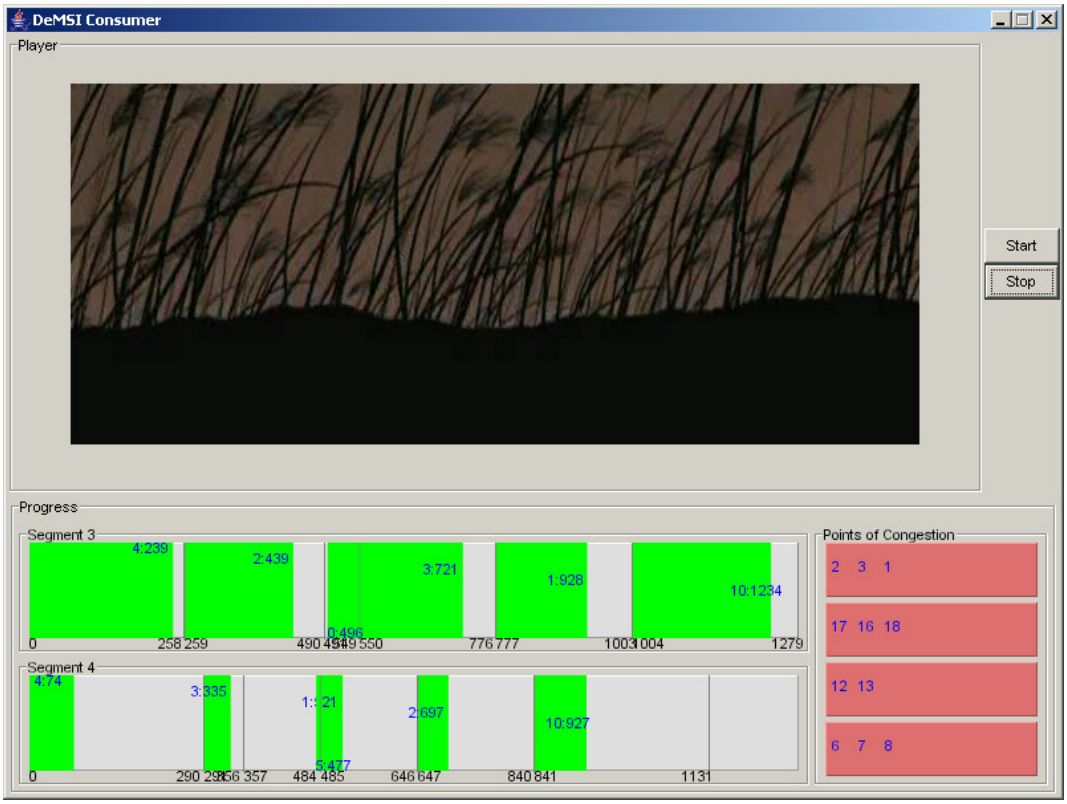

Figure 4-3: DeMSI player UI showing the progress bars (shown in green) of each sub-stream flow and the inferred POC on the lower-right (shown in grayish red). Each blue number shown within a POC block represents the ID of a peer believed to have pushed sub-stream through that POC. A blue number tuple separated by a colon represents <*ID of the peer from which the sub-stream is delivered*>:<*ID of the fragment to be received*>. The black numbers represent the start and end points of a sub-stream expressed in fragment ID.

Peers	**Segments Assignment**
P_0, P_5, P_{10}, P_{15}	$S_0 ... S_{24}$ (dedicated server)
$P_1 ... P_4$	$S_0 ... S_9$
$P_6 ... P_9$	$S_5 ... S_{14}$
$P_{11} ... P_{14}$	$S_{10} ... S_{19}$
$P_{16} ... P_{19}$	$S_{15} ... S_{24}$

Table 4-1: Distribution of segments to peers

4.1 Finding the Optimal Parameters for Correlation Tests

First, we survey a range of parameter value pairs: poisson probe rate and correlation time, in order to find out the optimal combination for the point-of-congestion inference algorithm under DeMSI's aggregated streaming scenario. For each parameter pair, we start DeMSI and play the video two times in a row. Then we restart DeMSI and play the video two times again. Each positive claim (where two flows share a point of congestion) from the pair-wise comparison of flows is verified against the actual network topology. Each playback typically generates tens of positive claims and the number of positive claims decreases in subsequent playback without quitting DeMSI. The reason is that as DeMSI accumulates knowledge of where the congestion points are, it avoids visiting more than one peer in each partially identified group. Hence the chance of getting positive claims decreases. Our experience is that the number of positive claims generated out of the third playback in the same DeMSI session is of little statistical value. This survey is also helpful for us to determine an optimal δ value to use. We have tested a range of correlation time between 2 to 8 seconds. The results basically exhibit a trade-off between accuracy of inference and number of positive claims during a playback. Accuracy improves as the correlation time increases, but the rate of increase is very small when the correlation time is more than 5 seconds. On the other hand, the number of claims decreases at a converging rate as the correlation time increases. This is expected because the sub-stream flow from a peer is short-lived. The probability of having two peers stream together for as long as the correlation time decreases as the correlation time increases. Therefore, we have narrowed down the survey to correlation time between 3 and 5 seconds. The δ of 0.2 is determined. We first

try a few variety of the mean probing rates with fixed correlation time of 4 seconds. This survey is conducted under the network topology as shown in figure 4-1 but without cross-traffic. The congestion in the hop-links *r1-r9, r4-r9, r5-r9, r8-r9* is made possible by a reduction of bandwidth to 64kB/s instead. The result suggests an obvious increase in accuracy as the probing rate increases, perhaps except that the accuracy of claims for the probing rate of 10 is slightly less than that for the probing rate of 8, without δ filtering. This is possibly due to statistical artifact resulted from a small number of samples in the survey. Table 4.1-1a shows the result of the survey.

Probing Rate (/s)	Interval b/w Correlation Tests (no. of probes)	Total no. of Correct Positive Claims	No. of Correct Positive Claims Survived after δ Filtering	Total no. of Positive Claims incl False Positives	Accuracy, Accuracy with δ Filtering (col 3/col 5, col 4/col 5)
5	20	45	26	119	0.378, 0.218
8	32	44	27	74	0.595, 0.365
10	40	52	41	91	0.571, 0.451

Table 4.1-1a: Implications of increasing probing rate and the use of δ using the correlation time of 4 seconds – The network topology without cross-traffic is used

Correlation Time (s)	Interval b/w Correlation Tests (no. of probes)	Total no. of Correct Positive Claims	No. of Correct Positive Claims Survived after δ Filtering	Total no. of Positive Claims incl False Positives	Accuracy, Accuracy with δ Filtering (col 3/col 5, col 4/col 5)
3	30	125	108	162	0.772, 0.667
4	40	100	93	115	0.870, 0.809
5	50	52	47	60	0.867, 0.783

Table 4.1-1b: Implications of increasing correlation time and the use of δ using the probing rate of 10/s – The same network topology with cross-traffic is used

Despite of the fact that an increase of probing rate increases the accuracy, we stop at the probing rate of 10 per second. Since the probes is sent in-band with the sub-stream flow, the probing rate is directly proportional to the minimum gross upstream rate $R_{up\min}$ such that

$$R_{up\min} = \frac{\rho}{U\lambda}$$

A probing rate of 10 translates to 10.4kB/s according to our configuration. A further increase of $R_{up\min}$ reduces the coverage of low-end broadband community where the upstream bandwidth of each connection can be as low as 16kB/s.

This accuracy figures as shown in table 4.1-1a are particularly discouraging. However, when the congestion is partly due to cross-traffic, the accuracy improves significantly as shown in table 4.1-1b. We change the focus on surveying a variety of correlation times but fix the probing rate at 10 per second. Fortunately, the network with cross-traffic resembles the internet more closely than the network without cross-traffic.

4.2 Efficiency of Scheduling and Re-Scheduling Processes

This section provides more insights about the performance of the streaming task scheduling and re-scheduling algorithms. The objectives of the evaluation are as follows:

1. To show that the concept of the proactive peer selection algorithm based on congestion avoidance is useful under DeMSI's decentralized storage scenario.
2. To show how our reactive re-scheduling algorithm enhances the performance of any proactive scheduling strategies.

In order to achieve the first objective, the algorithm has to be independent of its underlying inference algorithm. That is, the pair-wise flow correlation test algorithm by Rubenstein [1]. The experiment has to assume that the inference algorithm is 100% accurate on the point-of-congestion inference such that it can show how well the concept works when it is compared against the peer selection based on end-to-end bandwidth measurement (or "best-bandwidth-first" as we refer to in the remaining of this paper) We achieve such independence by injecting the correct peer-point of congestion mappings into the data structure of the Peer Cache and Peer Monitor agents, before the streaming session starts. We override the correlation test algorithm completely in this experiment. We then repeat the experiment and present the comparison using the correlation test algorithm.

In addition, we turn off most Re-scheduling functionalities except the handling of active serving peers going offline, and the handling of loss rate exceeding l_{FEC}. In other words, the streaming sessions in this experiment relies almost completely on proactive scheduling of streaming tasks except in the event that requires emergency switch-over. As a reminder, the Scheduler agent schedules streaming tasks mainly at the beginning of a segment delivery. The partitioning of a segment is revised only when it proceeds with the next segment. We run the experiment under the network topology with cross-traffic as shown in figure 4-1. The experiment involves running and quitting the DeMSI Player for 5 times. Each time the Player plays the video for 3 repetitions without quitting DeMSI. We repeat the experiment for each of the following configuration:

1. Peer selection based on end-to-end bandwidth
2. Peer selection based on congestion avoidance with ideal inference simulation
3. Peer selection based on congestion avoidance with correlation test algorithm

We also work on the second objective in this experiment by repeating for each of the above configuration with the Re-scheduler fully enabled.

The dynamic service level statistics of each active peer is aggregated and extracted once a second during the playback. The statistics from the 5 runs are aligned by the repetition number and the elapsed time of the playback. Each record of statistics from the 5 runs over the same elapsed timeline and repetition number are averaged.

Figure 4.2-1a, 4.2-2a, 4.2-2c, 4.2-3a, and 4.2-3c show how far the peer selection based on congestion avoidance can go ideally. Under the congestion avoidance selection strategy, the average number of active serving peers (hence number of sub-streams) scheduled by the consumer at almost any time of the playback is lower than those scheduled by the consumer using selection based on end-to-end bandwidth. The average utilization of each active serving peer is also higher than that its bandwidth-based counterpart at almost any time of the playback. Likewise, the consumer using selection based on congestion avoidance yields lower average round-trip-times between the consumer and peers, than the consumer using selection based on end-to-end bandwidth. As expected, the lower average round-trip-times lead to lower average loss rates than the counterpart, as shown in figure 4.2-4a. There is one characteristic in common. That is, the difference in performance between the two selection strategies, in terms of any type of statistics, converges towards the end of the playback. This is because there are only 4 peers available to deliver the last 5 segments: $S_{20}...S_{24}$, and 3 peers out of 4 share the same hop-link. As the Segment Cache has accumulated a considerable amount of fragments towards the end of the streaming session, the Scheduler does not need to contact the dedicated servers for help. As a result, the same set of peers is selected for the delivery of the last 5 segments regardless of the selection strategy. Hence the difference in performance converges towards the end.

However, when we compare the average aggregated net content receive rates between the two selection strategies, as shown in figure 4.2-5a and 4.2-5d, the figures achieved by the peer selection based on congestion avoidance are lower than those achieved by the selection based on end-to-end bandwidth. It is indeed the case that the consumer employing the congestion avoidance selection strategy takes longer than the one employing the best-bandwidth-first strategy to finish streaming. This is largely due to the phenomenon of diversity on peer revisit. The selection of peers by best-bandwidth-first promotes diverse selections on subsequent revisit of previously used peers that have encountered congestion before. This can be illustrated by an example. When peers P_0, P_1, P_2, P_3 sharing a congested link are selected and scheduled to stream, their end-to-end bandwidths perceived by the consumer decrease considerably. When the next segment delivery is due to be scheduled, the selection of P_0, P_1, P_2, P_3 alone is no longer enough. Thus the selection algorithm adds two more peers: P_6, P_7 which apparently have similarly slow end-to-end receive rate perceived by the consumer during the previous playback, due to a previous selection of P_5, P_6, P_7, P_8 which share another congested link. Now since only P_6, P_7 are selected, the actual receive rate increases considerably from the original estimate. The outcome is an increase of aggregated receive rate from the selection: $P_0, P_1, P_2, P_3, P_6, P_7$. The larger the set of active peers selected, the higher the chance of encountering such a phenomenon. In contrast, the congestion avoidance selection tends to avoid fluctuations in perceived end-to-end receive rate. Unless the peer has encountered independent congestion before during the streaming, the headroom for the previously perceived receive rate to grow is limited.

Fortunately, the aggregated net content receive rates can be boosted by the Re-scheduler as shown in figure 4.2-5b, 4.2-5c, and 4.2-5e. The boost is regardless of the selection strategy being used for scheduling and re-scheduling. Figure 4.2-6 shows that the Re-scheduler participates on easing the fluctuations of the aggregated receive rates as well. As the Re-scheduler acts upon slower-than-expected sub-stream flows in a defensive manner by adding a redundant peer to assist the streaming, it slightly increases the average number of active serving peers at almost any second of the playback regardless of the selection strategy. Figure 4.2-1e, 4.2-1f, and 4.2-1g illustrate this. This is perhaps the cost of smoothing down the aggregated receive rate with a minor boost. However, as figure 4.2.1c and 4.2.1d show, this cost is small relative to the significant performance improvement of the congestion avoidance selection strategy over the best-bandwidth-first counterpart. Other statistics show no evident or dominating differences after enabling the fully functional Re-scheduler in the experiments. As the congestion avoidance selection promotes smoother end-to-end receive rate when it is compared to the best-bandwidth-first selection, it reduces the frequency of re-scheduling as figure 4.2-7 shows.

Another crucial feature of the Re-scheduler is to ensure smooth transition in the event of peer failure, and to reduce the impact of such events on the aggregated streaming. We examine the impact of a single-peer failure on aggregated receive rates during a playback. We shut down a peer when it becomes active and is pushing a sub-stream of fragments to the consumer. Then the aggregated receive rates and the number of active serving peers obtained for the 10 seconds before and after the failure event are captured. This is repeated 5 times on each peer-selection strategy. Out of 10 trials, 9 of them exhibit no sudden drop in aggregated receive rate. 6 of the 9 cases exhibit a varying degree of burst in the next 2 to 5 seconds after the failure event. During the burst period, the number of active serving peers often increases by 1. It implies that in most cases, there are 2 peers being re-scheduled to finish the outstanding streaming task. The remaining 3 of them exhibit no obvious changes. It is observed that the trials using the best-bandwidth-first selection strategy exhibit less evident burst in aggregated receive rate than those using the congestion avoidance counterpart. This is expected because the size of the active peer set resulted from the best-bandwidth-first selection strategy is often larger than that resulted from the congestion avoidance strategy. In addition to the fact that the utilization of each active serving peer, under the best-bandwidth-first strategy, is lower than that under the congestion avoidance strategy, the contribution of each active peer under the best-bandwidth-first strategy is relatively less significant than that under the congestion avoidance counterpart. This applies to the impact of peer failure as well. Figure 4.2-8 illustrates an example of a short burst due to re-scheduling upon peer failure from one of the playback trials using the ideal congestion avoidance selection strategy.

Finally, we repeat the experiments using the congestion avoidance selection with correlation test as the underlying inference algorithm. As shown in figure 4.2-1a versus 4.2-1b, and 4.2-2a versus 4.2-2b, the difference in performance between the correlation test version and the selection based on end-to-end bandwidth is less evident than that using the ideal version. This is expected as the correlation tests cannot give the full picture of the peer-congestion points relationships, although false positives in the groupings are rare. At its best out of all rounds of experiments, the algorithm successfully identifies all 4 points of congestion with 3 peers in each. Although the selection algorithm avoids picking more than one peer from each group when there are enough candidates, there are still occasions where more than one peer from the same group is selected as active peers at the same time. False negatives, which disintegrate the groupings, may be introduced when those peers in the same group are tested for correlation, while there are not enough active peers in that group to produce congestion. Moreover, as the point of congestions are inferred incrementally during the streaming session, the performance statistics obtained from the first playback of each DeMSI Player session have adversely affected the average values over all runs by some degree. We deliberately include the statistics from the first playback in the overall averages because, in reality, each streaming session should probably encounter a significant population of discovered candidates that have not been contacted before. In addition, the network path between the consumer and a previously contacted peer will change [9]. Some of the previously inferred knowledge may become invalid. The opportunity of new exploration is always there for DeMSI.

Notations	
AC-ideal-nores	Peer selection by ideal congestion avoidance without Re-scheduler
AC-ideal	Peer selection by ideal congestion avoidance with Re-scheduler
AC-nores	Peer selection by congestion avoidance using correlation tests without Re-scheduler
AC	Peer selection by congestion avoidance using correlation tests with Re-scheduler
BW-nores	Peer selection by best-bandwidth-first without Re-scheduler
BW	Peer selection by best-Bandwidth-first with Re-scheduler

Table 4.2-1: Notations to be used in figure 4.2-1 to 4.2-8

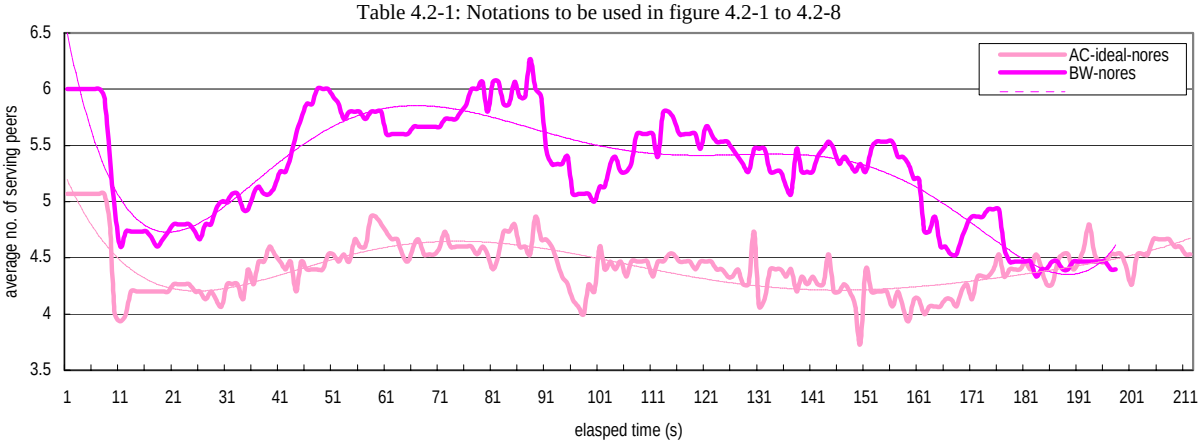

Figure 4.2-1a: Average number of active serving peers (sub-streams) – aim for less

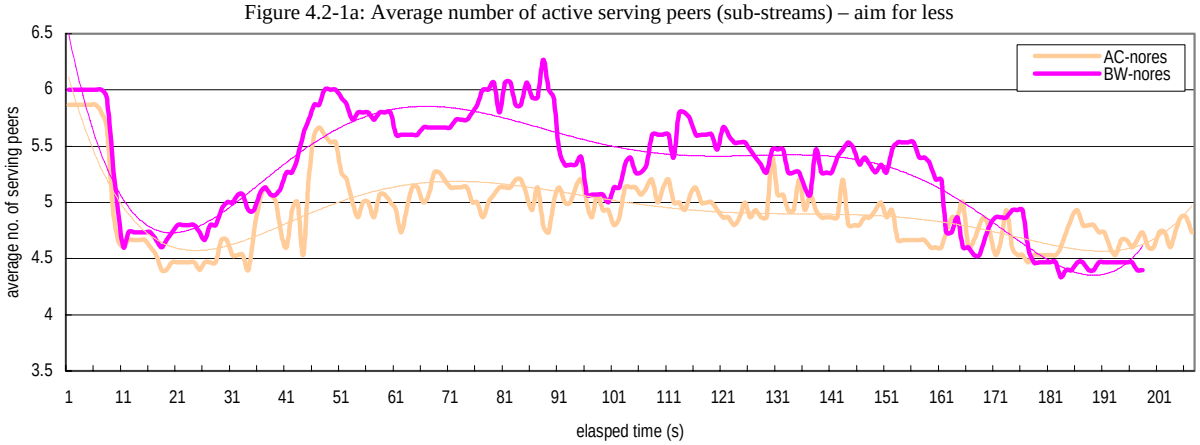

Figure 4.2-1b: Average number of active serving peers (sub-streams) – aim for less

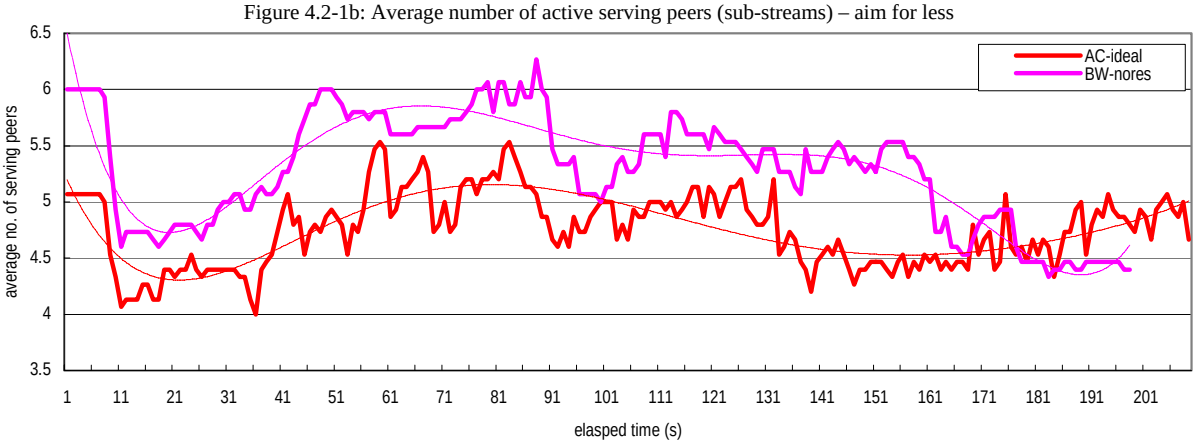

Figure 4.2-1c: Average number of active serving peers (sub-streams) – aim for less

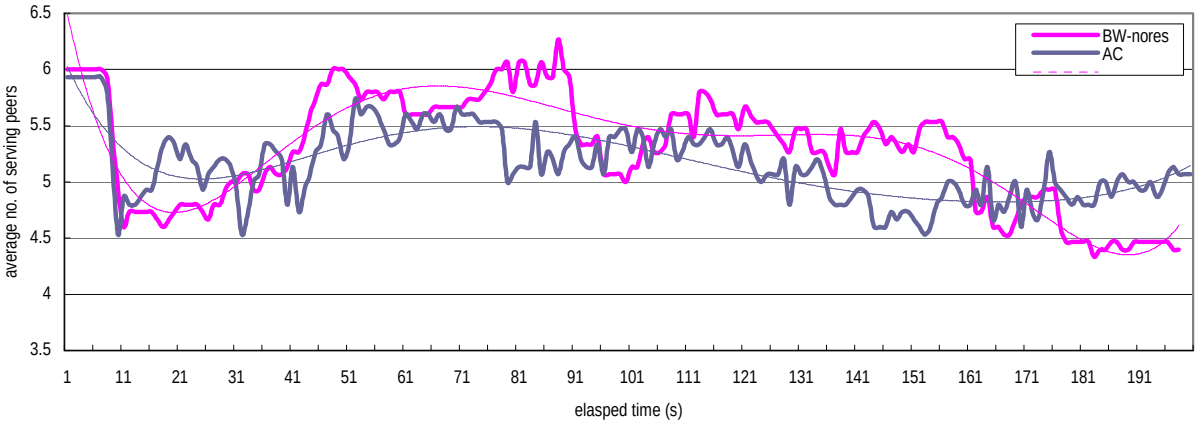

Figure 4.2-1d: Average number of active serving peers (sub-streams) – aim for less

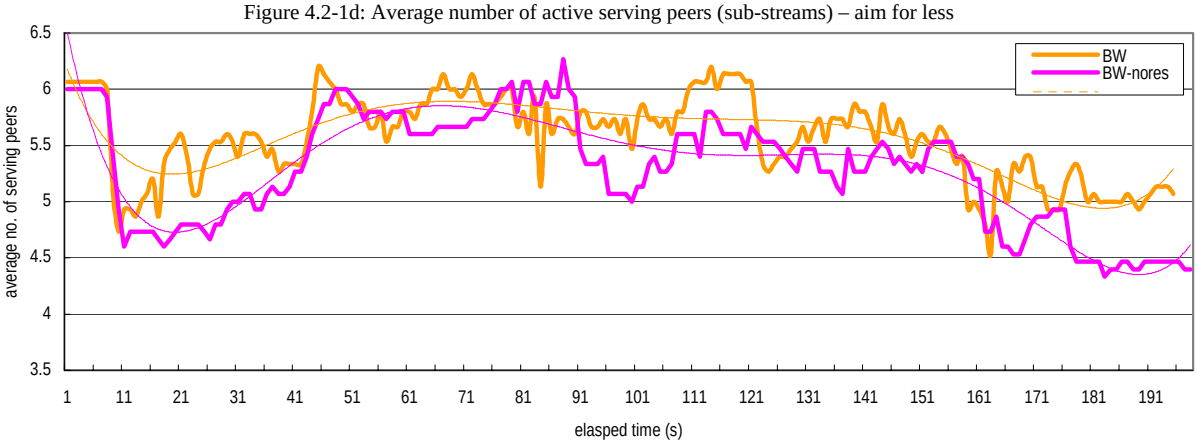

Figure 4.2-1e: Average number of active serving peers (sub-streams) – aim for less

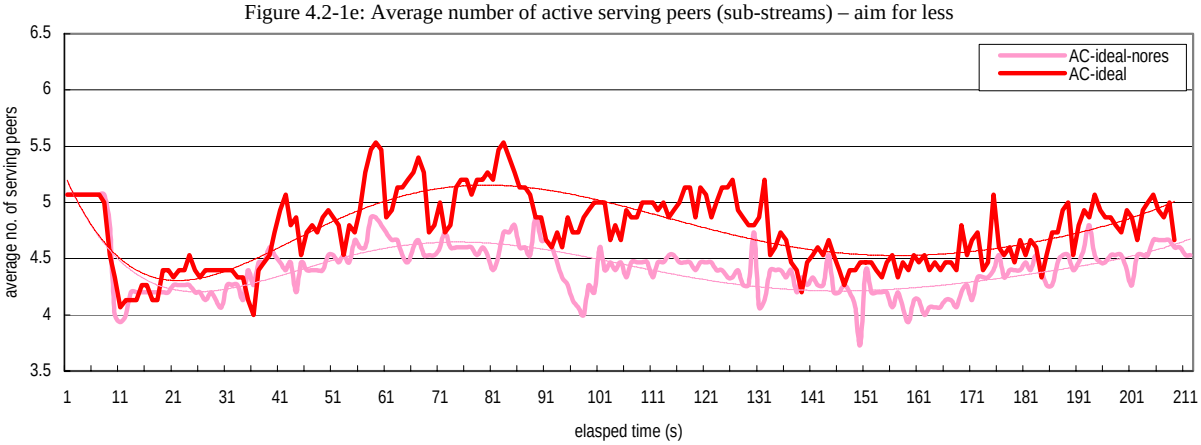

Figure 4.2-1f: Average number of active serving peers (sub-streams) – aim for less

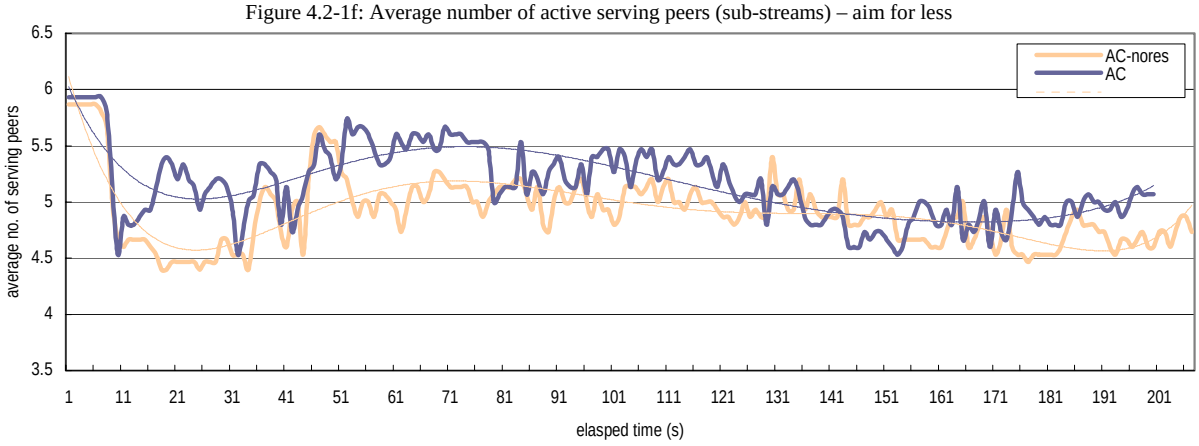

Figure 4.2-1g: Average number of active serving peers (sub-streams) – aim for less

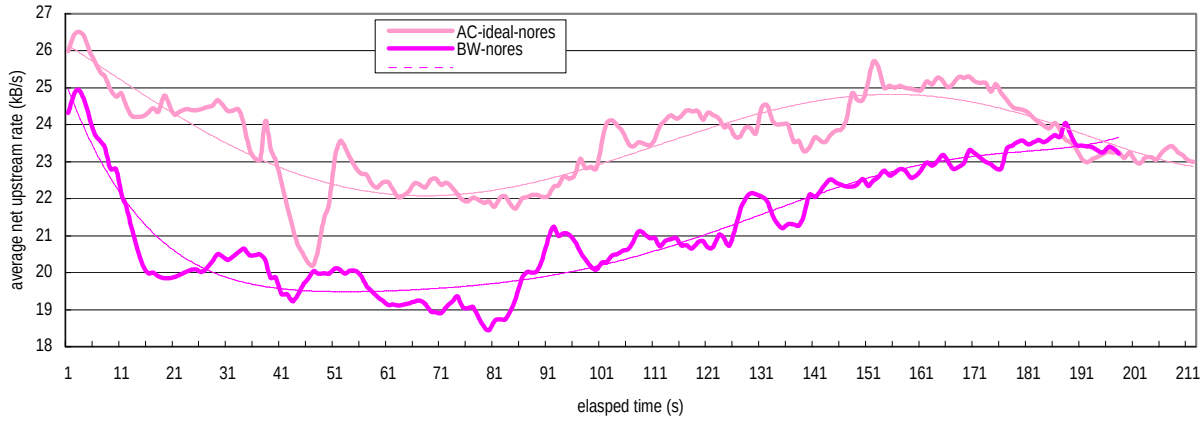

Figure 4.2-2a: Average net content upstream rate of active serving peers – aim for more

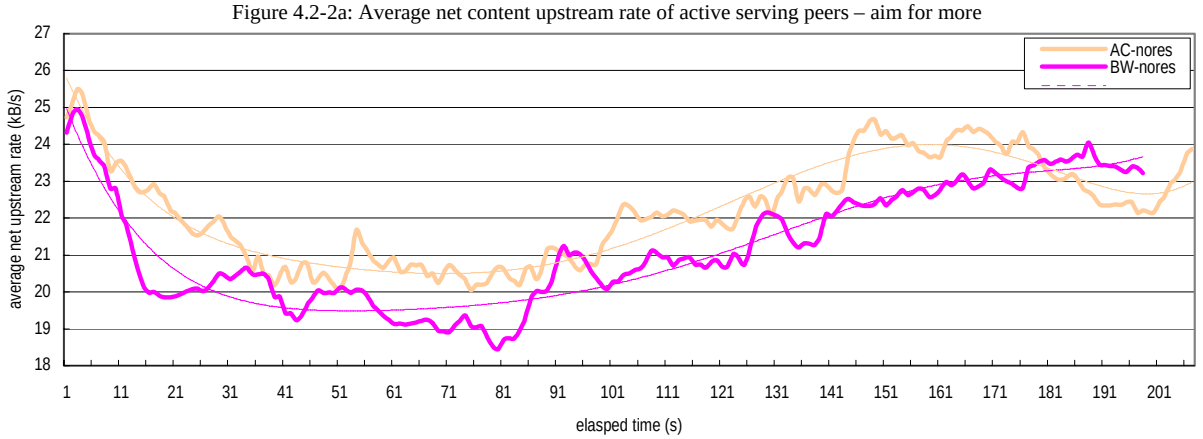

Figure 4.2-2b: Average net content upstream rate of active serving peers – aim for more

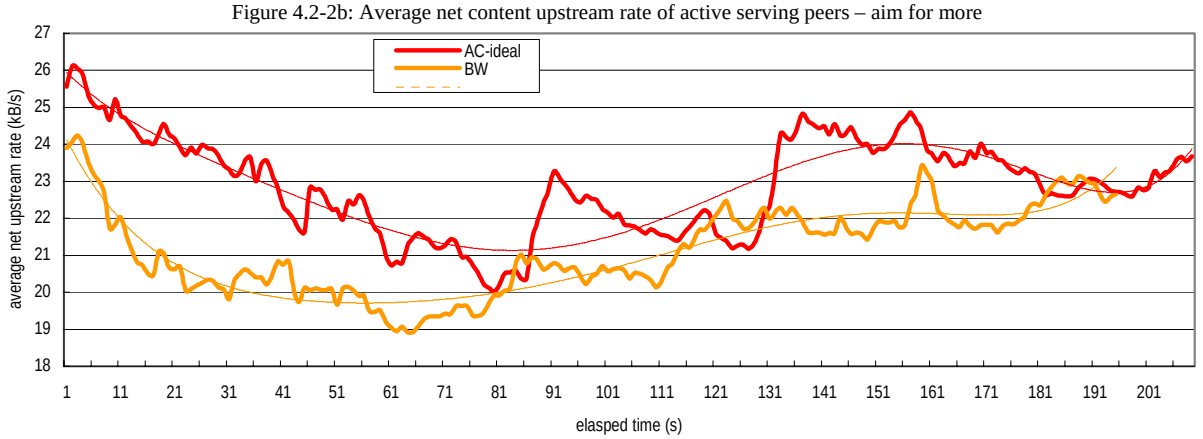

Figure 4.2-2c: Average net content upstream rate of active serving peers – aim for more

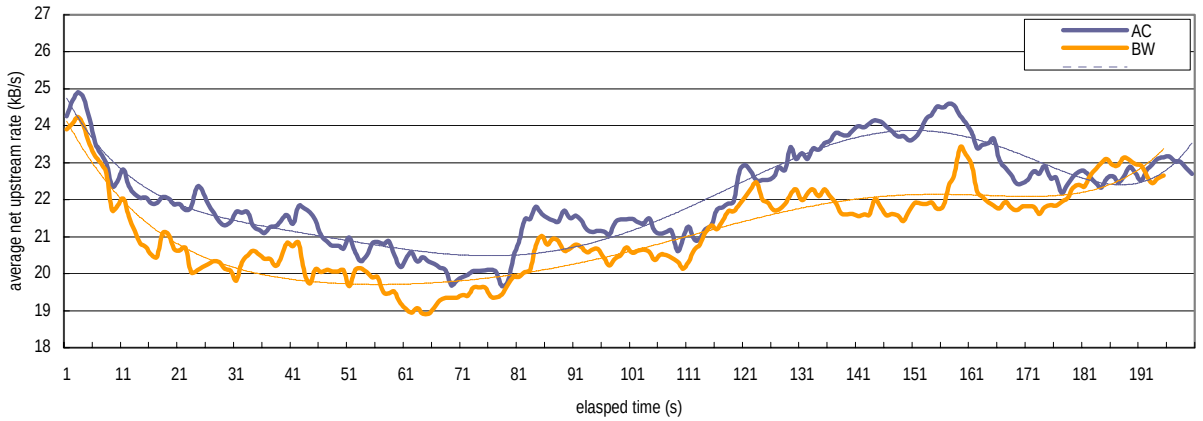

Figure 4.2-2d: Average net content upstream rate of active serving peers – aim for more

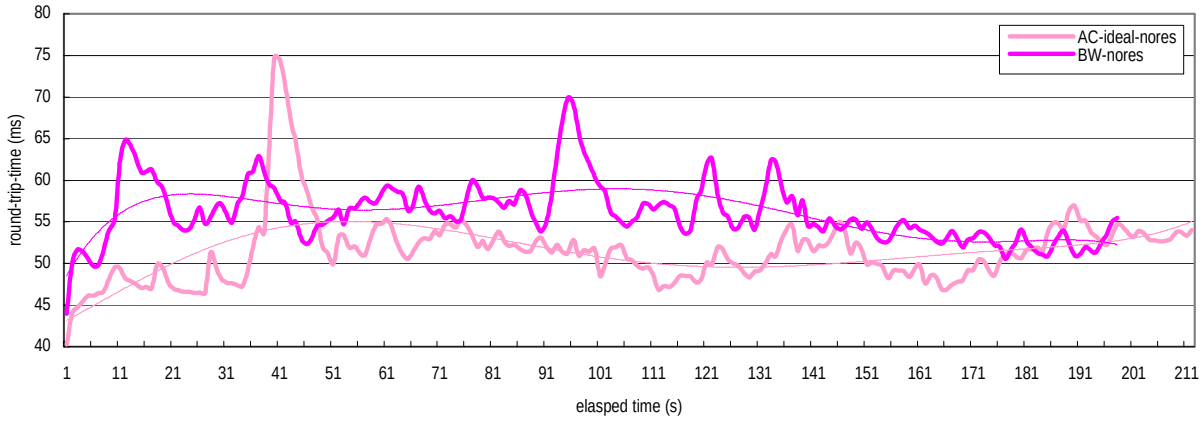

Figure 4.2-3a: Average round-trip-time between the consumer and the active serving peers – aim for less

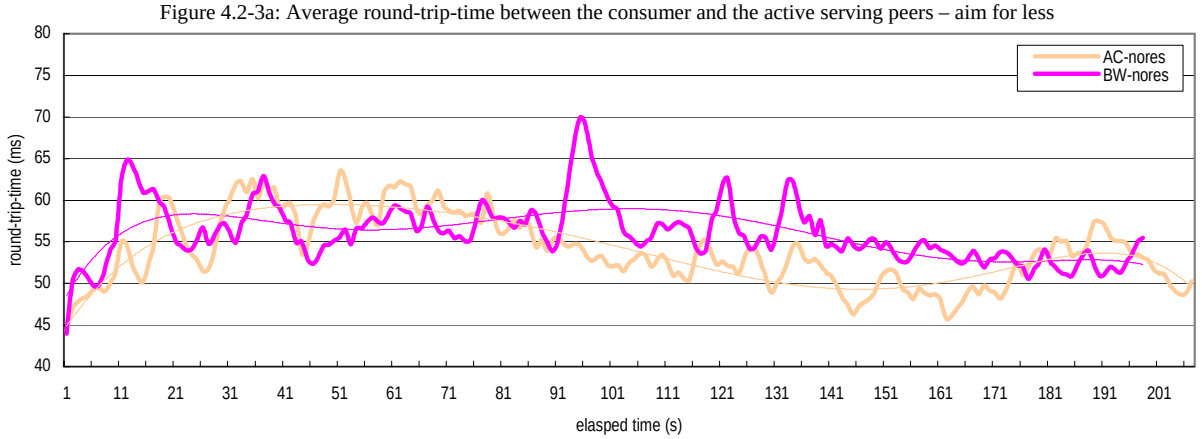

Figure 4.2-3b: Average round-trip-time between the consumer and the active serving peers – aim for less

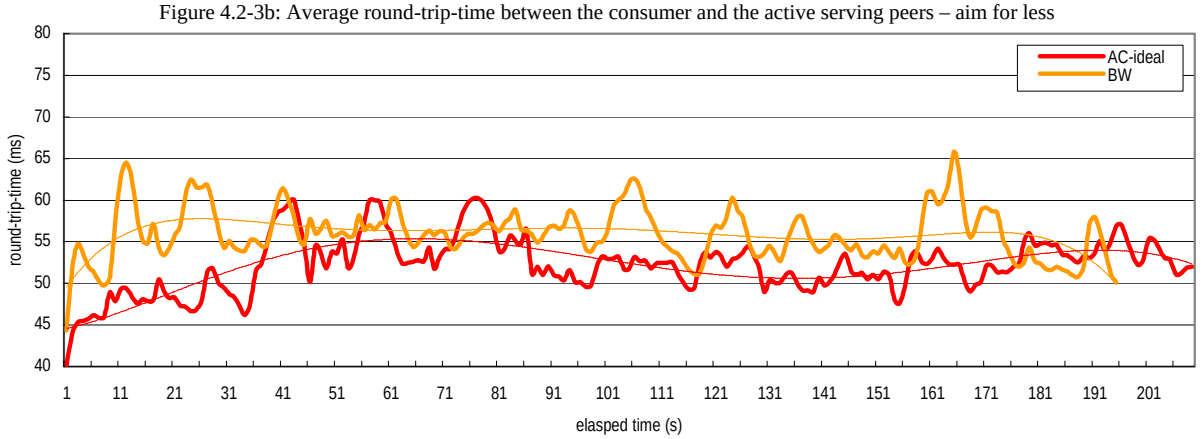

Figure 4.2-3c: Average round-trip-time between the consumer and the active serving peers – aim for less

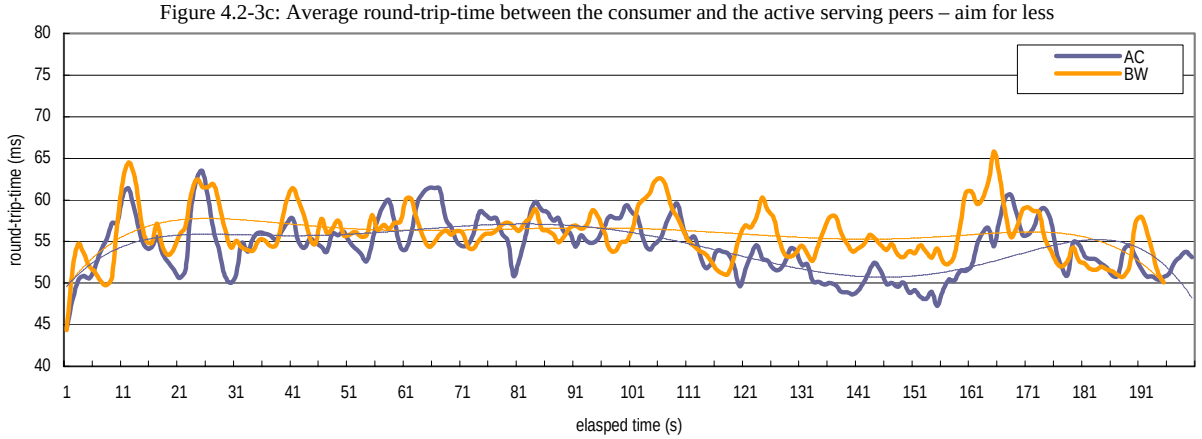

Figure 4.2-3d: Average round-trip-time between the consumer and the active serving peers – aim for less

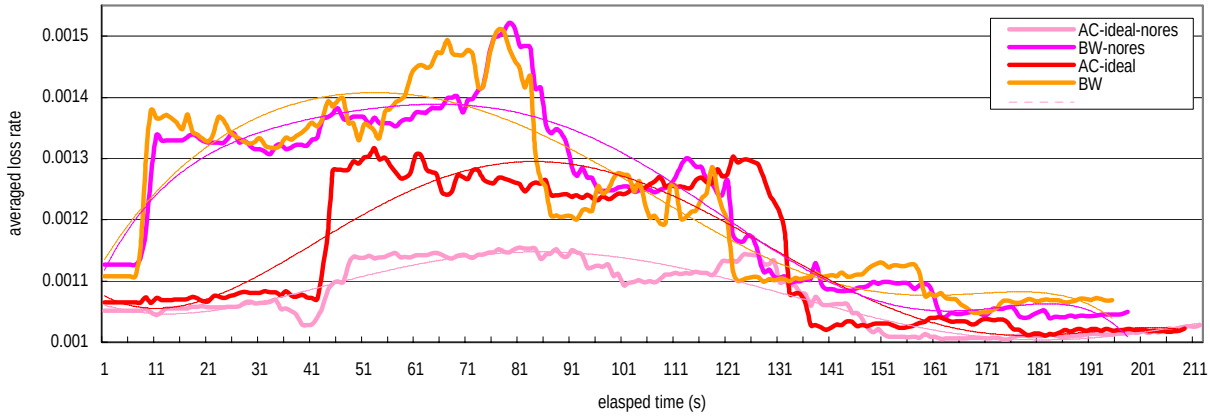

Figure 4.2-4a: Average packet loss rate of sub-stream flows from active serving peers – aim for less

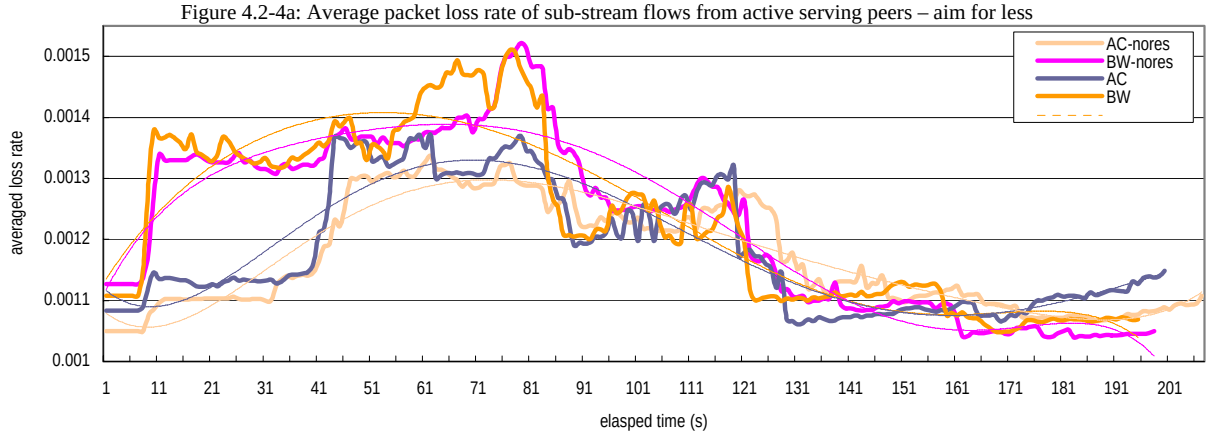

Figure 4.2-4b: Average packet loss rate of sub-stream flows from active serving peers – aim for less

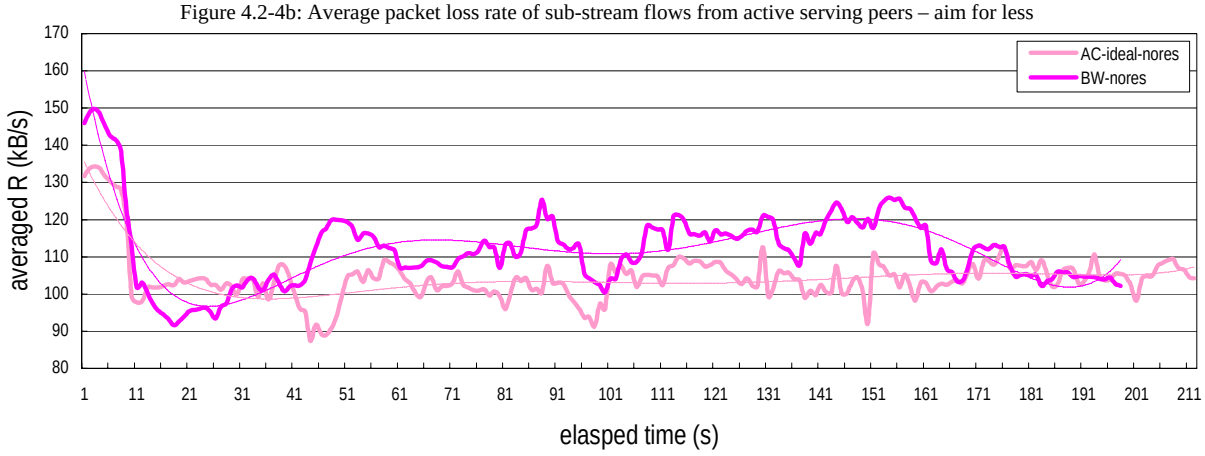

Figure 4.2-5a: Average aggregated net content receive rate perceived by the consumer – aim for smoothness and at least $R_{content}$

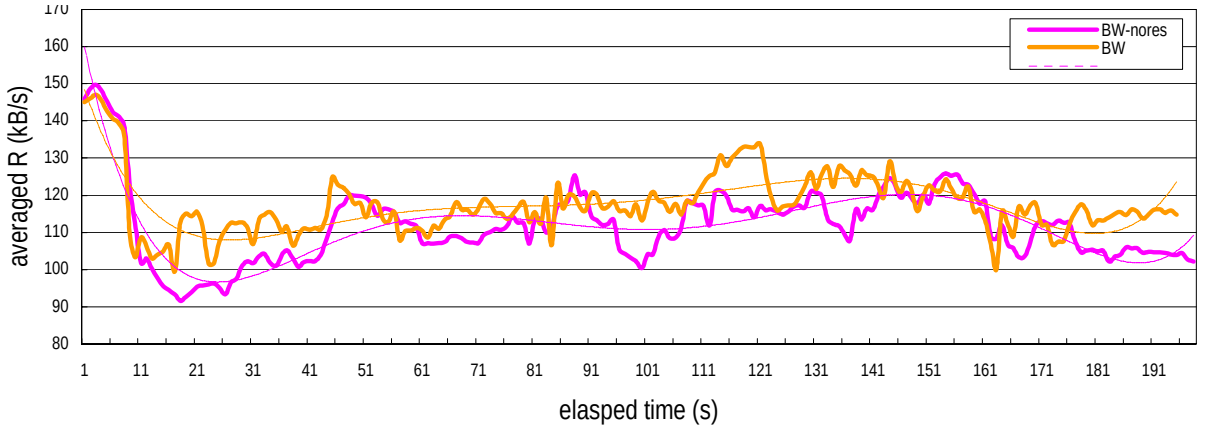

Figure 4.2-5b: Average aggregated net content receive rate perceived by the consumer – aim for smoothness and at least $R_{content}$

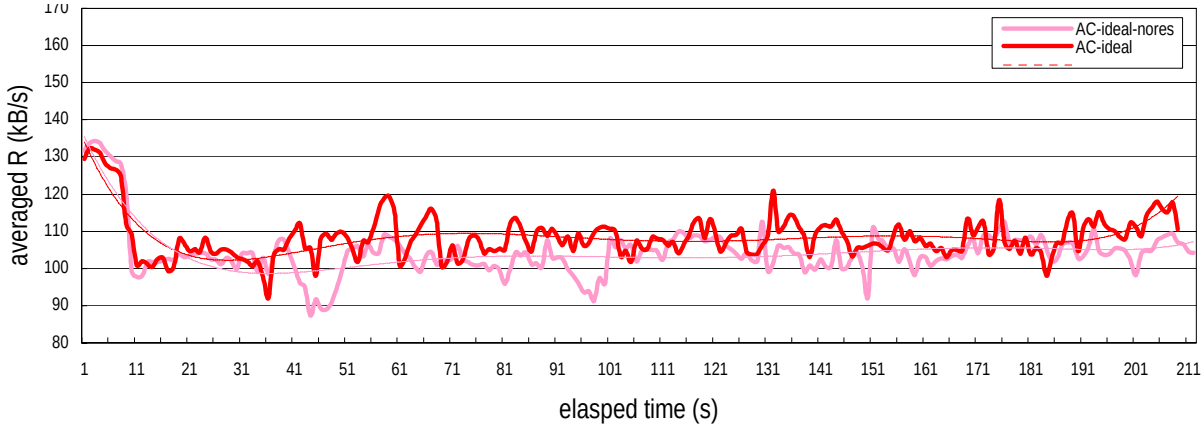

Figure 4.2-5c: Average aggregated net content receive rate perceived by the consumer – aim for smoothness and at least $R_{content}$

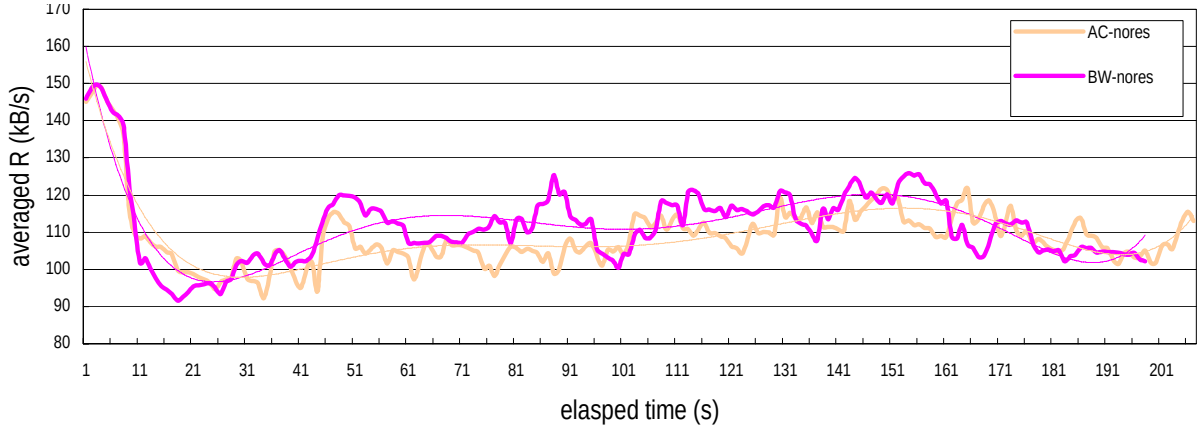

Figure 4.2-5d: Average aggregated net content receive rate perceived by the consumer – aim for smoothness and at least $R_{content}$

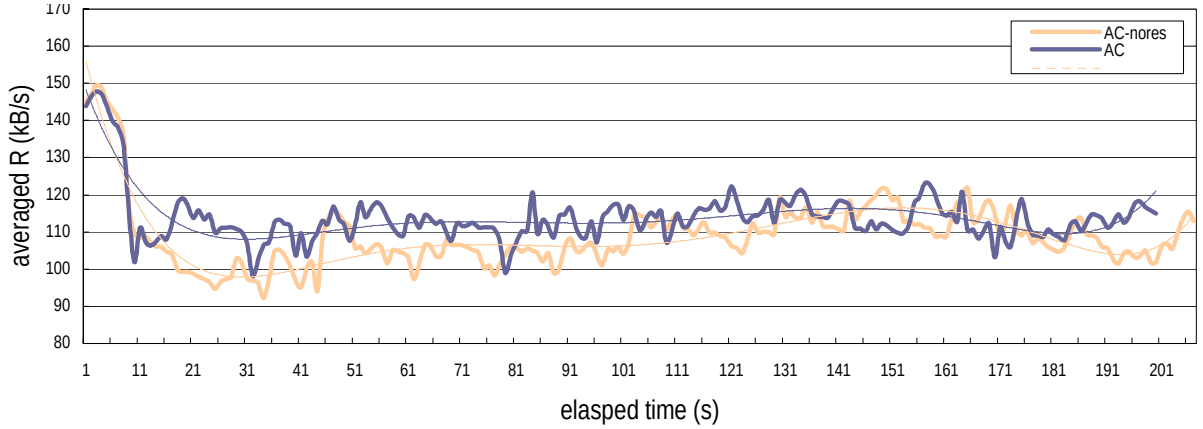

Figure 4.2-5e: Average aggregated net content receive rate perceived by the consumer – aim for smoothness and at least $R_{content}$

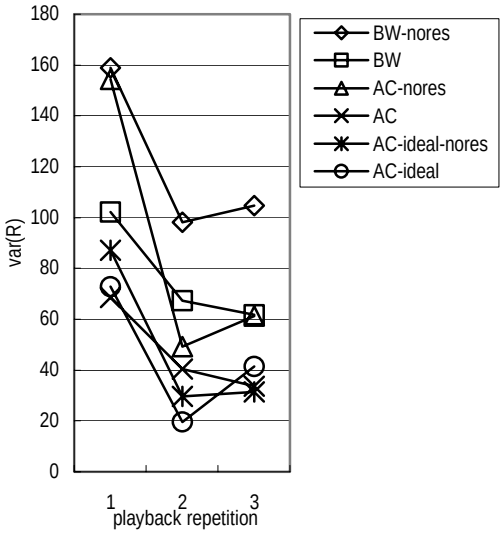

Figure 4.2-6: Variance of net content receive rates obtained from each playback. The rates are averaged over 5 runs. The lower the variance, the more stable (smooth) the receive rates perceived over the course of the playback.

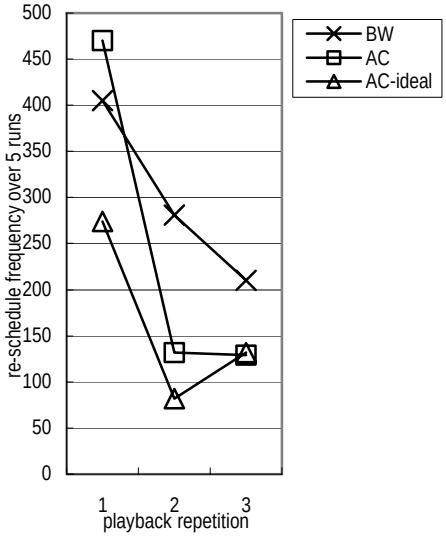

Figure 4.2-7: Total number of re-schedules during each playback over 5 runs. Note that as the congestion avoidance algorithm using correlation tests takes time to infer the peer-point of congestion mappings. The performance of the first play is similar to that when the best-bandwidth-first peer-selection is used.

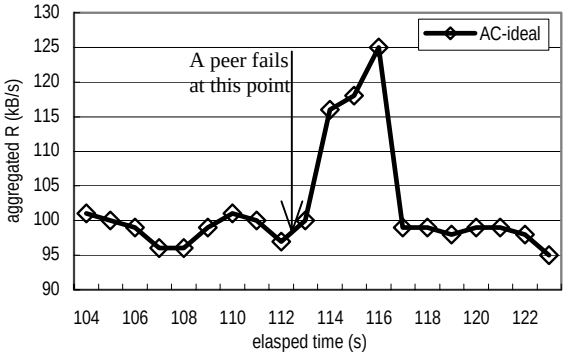

Figure 4.2-8a: A typical impact of a single-peer failure on aggregated net content receive rate from an example playback

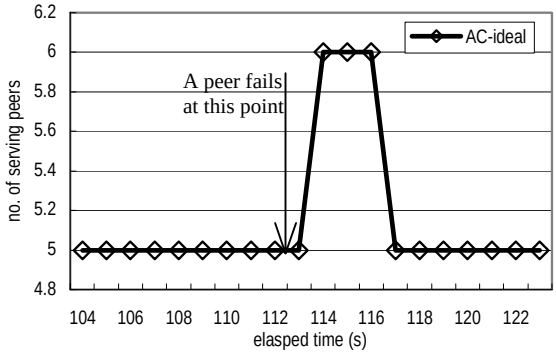

Figure 4.2-8b: A typical impact of a single-peer failure on number of active serving peers from an example playback

5 Conclusion, Discussion and Future Work

This paper presents an infrastructural solution to address aggregated media streaming from a decentralized collection of unreliable subscriber resources, under the scenario where the media content is collectively stored at the subscriber ends. Unlike other P2P resource sharing solutions, each subscriber is responsible for only a small portion of the content rather than a complete replication of it. Our simulations demonstrate the effectiveness of the peer selection algorithm that employs a proactive congestion avoidance strategy, which only requires coarse-grain point-of-congestion inference and clustering of peers, under DeMSI's scenario. It can be concluded that the use of congestion avoidance strategy in peer selection outperforms the use of best-bandwidth-first strategy in terms of the following goals set out in section 1:

1. To maximize the utilization of the network and peers,
2. To minimize the number of peers to serve the content,
3. To minimize the frequency of re-scheduling or emergency switching-over to other candidates over the course of streaming.

We also demonstrate the power of our novel approach to promote smooth reactive re-scheduling of aggregated streaming tasks. It has been shown to improve the performance of aggregated streaming, in particular on the streaming rate and its smoothness regardless of which proactive peer-selection strategy has been used in scheduling and re-scheduling. The combined use of the proactive peer-selection and the re-scheduling algorithm simply brings the best of both worlds together.

It is anticipated that the smoothening of aggregated receive rate by using reactive re-scheduling, in events of fluctuating perceived receive rate of a single peer, can also be achieved solely by scheduling as the segment size decreases. As the segment size decreases, the frequency of scheduling increases. In that case, the scheduling

process has more up-to-date data on dynamic service level metrics. Therefore, its adaptability in changing network conditions increases. However, it is expected that the decrease of segment size reduces the frequency of claims produced by the correlation test algorithm being used for the point-of-congestion inference. Hence the longer it takes to infer. One way to work around this problem is to have more consecutive segments distributed to each peer such that the continuity of the sub-stream flow from a peer can be maintained across schedules, in order to ensure enough time for a correlation test against another flow. However, smaller segment size also implies more loading on the network caused by more frequent use of control packets by the Scheduler for sending delivery requests to the active peers. In contrast, the Re-Scheduler sends additional delivery requests to other peers only when there is a need.

As we have discovered from the experiments, our inference algorithm is particularly vulnerable to false positives from the correlation tests of the sub-stream flows. The existing peer-point of congestion mappings can be easily disintegrated by false positives. It is due to the fact that the introduction of false positives into the group leads to subsequent correlation tests of an existing peer that is correctly identified against the one that is not. Hence an increase in the probability of removing correctly identified peers out of the group together with the incorrect ones. Nevertheless, our conservative approaches applied to correlation tests have significantly reduced the rate of false positives in the results. Our experiments also confirm that the correlation tests yield more accurate inference under asymmetric network with shared links congested by heavy cross-traffic, than under the same asymmetric network with shared links congested by tight bandwidth assignment. A possible explanation is that the shared links with cross-traffic promote varying differences in each other. The outcome is a network that is more asymmetric than that without cross-traffic. The phenomenon is in line with the findings discussed in [2] that the correlation test performs better under an asymmetric network than under a symmetric one.

The subject of P2P aggregated media streaming is large and involves a diverse collection of disciplines such as security, networking, agent-oriented design and development, artificial intelligence, and statistics. The future research directions of DeMSI are also diverse. We outline the most important ones in descending order of priority:

5.1 Intelligent Pattern Learning for Enhanced Proactiveness in Peer-Selection

Peer-selection approaches based on past history of network characteristics are proved effective in aggregated streaming scenario. However, the approach discussed in this paper does not proactively predict whether the candidate peer is available at the time of selection, and the probability that the peer will become unavailable during the delivery. In that sense, DeMSI is completely reactive when it comes to the dynamics of peer availability. Selection based on past history and even prediction of peer availability as well as the network characteristics should be an interesting field of research. Inspired by the fact that users of peer-to-peer file-sharing systems generally have a regular usage pattern over time [22][23], the availability of peers and their underlying network characteristics over time should also have a pattern. Such properties can be exploited by the peer selection algorithm such that only the peers that are believed to be most probably available at the time of selection, are selected. Likewise, it is anticipated that the peer selection can also be based on the prediction of the streaming rate of the candidate peer, and even the prediction on peer-point of congestion mappings at the time of selection. Hefeeda et el in [15] have briefly proposed a pure statistical method of estimating current availability of a peer upon request by the consumer. The estimation process is situated at the peer end. However, the architecture does not allow prediction of future availability due to the fact that the size of the data sample for estimation is probably too large to be maintained collectively on the consumer side in order to promote prediction. For example, the consumer has no way to predict whether the candidate peer selected to be contacted is actually available at all. Moreover, the estimation algorithm assumes that the usage pattern repeats every 24 hours, which probably can only cover a narrow range of users.

Let us narrow down the focus to the peer availability prediction for now. There are two main approaches on the architecture for pattern learning. The first approach is to have the peer collect the usage statistics and send a summary of it to the consumer regularly. The regularity here is possibly an interval of at least a day. The consumer then analyses the summary and infer the future availability of a peer incrementally. In this approach, the summary has to be as compact as possible and the interval of summary generation cannot be too frequent in order to minimize overhead to the network. On the other hand, the second approach is to have the consumer infer the future availability of a peer based on past experience of connection attempts to that peer. This approach does not require any actions on the peer side.

It is anticipated that the architecture may employ some of the existing *incremental learning* algorithms on time-series data such as [24]. In traditional neural networks such as the back-propagation neural networks, the

network has to be trained with a stream of data samples for a number of iterations in order to predict what the next data sample in the stream is. When new data samples come in, the network has to be re-trained with the original set of data samples plus the new data samples in order to ensure accurate predictions. In contrast, the incremental learning algorithm allows the network to be trained incrementally using the new data samples together with a fixed-sized metadata or "hypothesis". The outcome of the training is a renewed hypothesis and it can be used for the next training. This model can be applied to the first approach as mentioned above: The summary to be sent from the peers regularly is the hypothesis resulted from incremental training with availability and usage data obtained since the last training at the peer side. The past experience "hypothesis" or metadata of each candidate peer is to be stored persistently at the consumer side across multiple streaming sessions. However, there must be a limit on the number of peers with which the past experience can be stored. The size of the hypothesis, its update interval and the prediction accuracy are open issues. On the other hand, the second approach is even more challenging as it has to deal with availability data resulted from polls (trial connection attempts) occurred irregularly over the time series. Although the perceived data can be grouped and expressed in terms of some interpolated and accumulated statistics as a function of poll rate over a time period, the accuracy of the statistics itself is difficult to be consistent along the time line. It is impossible for DeMSI to maintain a consistent poll rate over a time period as there are too many candidates to be polled. In addition, the consumer may bring the DeMSI offline at any time. Therefore, the second approach is unlikely to be of consideration.

5.2 Publishing of New Contents to Peers

We have discussed the storage strategy of DeMSI in this paper. However, it cannot be considered complete without the content publishing and re-distribution processes. It can be very costly if a new content is published to the peers from a single source such as the content provider itself. A more scalable and cost effective solution is to employ a power-law approach: The content provider first publishes the content in blocks of segments to an initial set of peers. Then those peers are scheduled to do the re-distribution work on behalf of the content provider. Each peer that receives the re-distribution is scheduled to re-distribute the new segments again in different combinations to its local peers subsequently. Such a decentralized approach has to face with the challenge of making sure every single peer that comes online at a later time can be synchronized with the new content. Another challenge is to ensure evenness of the re-distributions such that the peers in a local community are not biased to offer a particular range of segments of the content. The re-distribution strategy must ensure some degree of redundancy or overlap in the range of segments to be offered by a local collection of peers.

5.3 Incentive Model

Since the purpose of the DeMSI is to ease the workload of a traditional single point (or client-server based) CDN by offloading it to the subscriber peers, it is inherent to hope that the longer and the more peers stay online the more workload can be offloaded from the provider. However, who cares if the provider does not offer any incentive for those who stay online? The incentive can be calculated based on accumulated online time and the amount of content data delivered to other consuming peers. In other words, the system must be able to record the above usage statistics reliably and accurately. Since the delivery of content is decentralized, the accounting service has to rely on the peers to report usage statistics. It is anticipated that such a decentralized usage accounting model is subject to higher risk of fraud attacks from malicious users, than the conventional centralized model that is pretty much under the content provider's control.

6 References

[1] D. Rubenstein, J. Kurose, D. Towsley, "Detecting Shared Congestion of Flows Via End-to-End Measurement", IEEE/ACM Transactions On Networking, Vol. 10, No. 3, June 2002

[2] O. Younis, S. Fahmy, "On Efficient On-line Grouping of Flows with Shared Bottlenecks at Loaded Servers", Technical Report CSD-02-018, Purdue University, Aug 2002

[3] M. Handley, S. Floyd, J. Padhye, J. Widmer, "TCP Friendly Rate Control (TFRC) Protocol Specification – RFC 3448", Jan 2003

[4] Onion Networks Inc. , Java FEC Library v1.0.3, http://www.onionnetworks.com/developers/

[5] "MPEG-4 Industry Forum FAQ", http://www.m4if.org/resources/mpeg4userfaq.php

[6] Dixon, "Streaming Media: Trends and Formats", Manifest Technology - 2003

[7] Bouras, Kapoulas, Konidaris, Sevasti , "A Dynamic Distributed Video on Demand Service", 20th IEEE International Conference on Distributed Computing Systems-ICDCS 2000, Taipei, Taiwan, April 10-13 2000, pp. 496-503

[8] Akamai Technologies Inc., http://www.akamai.com

[9] V. N. Padmanabhan, L. Qiu, H. J. Wang, "Server-based Inference of Internet Link Lossiness", Infocom 2003, IEEE, 2003

[10] R. Teixeira, K. Marzullo, S. Savage, G. M. Voelker, “In Search of Path Diversity in ISP Networks”, IMC 03, ACM, Oct 2003

[11] T. Nguyen, A. Zakhor, “Path Diversity With Forward Error Correction (PDF) System For Packet Switched Networks”, Infocom 2003, IEEE, 2003

[12] J. G. Apostolopoulos, M. D. Trott, “Path Diversity For Enhanced Media Streaming”, IEEE Communications Magazine, IEEE, Aug 2004

[13] K. Calvert, J. Griffioen, B. Mullins, A. Sehgal, S. Wen, “Concast: Design and Implementation of an Active Network Service”, IEEE Journal on Selected Area in Communications, 19(3):426-427, Mar 2001

[14] T. Nguyen, A. ZakHor, “Distributed Video Streaming with Forward Error Correction”, Packet Video Workshop 2002, Pittsburgh PA, USA, Apr 2002

[15] M. Hefeeda, A. Habib, D. Xu, B. Bhargava, B. Botev, “CollectCast: A Peer-to-Peer Service for Media Streaming”, ACM Multimedia 2003, Berkeley CA, USA, Nov 2003

[16] M. Coates, R. Hero, A. Nowak, and B. Yu, “Internet Tomography”, IEEE Signal Processing Magazine, 19(3), 2002

[17] D. Katabi, C. Blake, “Inferring Congestion Sharing and Path Characteristics for Packet Interarrival Times”, MIT-LCS-TR-828, Dec 2001

[18] A. Rowstron and P. Druschel, “Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems”, In Proc. of 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), Heidelberg, Germany, November 2001

[19] S. Saroiu, P. K. Gummadi, S. D. Gribble, “SProbe: A Fast Technique for Measuring Bottleneck Bandwidth in Uncooperative Environments”, Infocom 2002, IEEE, 2002

[20] B. Byers, M. Luby, M. Mitzenmacher, A. Rege, “A Digital Fountain Approach to Reliable Distribution of Bulk Data”, In Proc. ACM SIGCOMM 98, pages 56-67, Vancouver, British Columbia, Aug 1998

[21] A. Bestavros, J. Byers, K. Harfoush, “Inference and Labeling of Metric-Induced Network Topologies”, Computer Science Department, Boston University, Boston, MA, USA, Tech. Rep., BUCS-2001-010, Jun 2001

[22] S. Saroiu, P. Krishna Gummadi, S. D. Gribble, “Measuring and Analyzing the Characteristics of Napster and Gnutella Hosts”, Multimedia Systems Journal, Volume 8, Issue 5, November 2002

[23] S. Sen, J. Wang, “Analyzing Peer-To-Peer Traffic Across Large Networks”, IEEE/ACM TRANSACTIONS ON NETWORKING, VOL. 12, NO. 2, APRIL 2004

[24] K. Okamoto, S. Ozawa, S. Abe, “A Fast Incremental Learning Algorithm of RBF Networks with Long-Term Memory”, Proc. of Int. Conf. on Neural Networks 2003 (IJCNN2003-Portland)

[25] UCB/LBNL/VINT Groups, “Network Simulator NS-2”, http://www.isi.edu/nsnam/ns

[26] V. Padmanabhan, H. Wang, P. Chou, K. Sripanidkulchai, “Distributing Streaming Media Content Using Cooperative Networking”, In Proc. of ACM International Workshop on Networking and Operating Systems Support for Digital Audio and Video (NOSSDAV’02), Miami Beach, FL, USA, May 2002

[27] H. Deshpande, M. Bawa, H. Garcia-Molina, “Streaming Live Media Over a Peer-to-Peer Network”, Technical report, Stanford University, Aug 2001

[28] Marshall Brain, Howstuffworks “How File Sharing Works”, http://computer.howstuffworks.com/file-sharing1.htm

[29] S. M. Lui, S. H. Kwok, “Interoperability of Peer-to-Peer File Sharing Protocols”, ACM SIGecom Exchanges, Pages 25-33, Vol. 3, Issue 3, 2002

[30] J. E. Berkes, “Decentralized Peer-to-Peer Network Architecture: Gnutella and Freenet”, University of Manitoba, Winnipeg, Manitoba, Canada, April 2003

[31] “Peer-to-Peer (P2P) and How Kazaa Works”, http://www.kazaa.com/us/help/glossary/p2p.htm

[32] K. Tutschku, “A Measurement-Based Traffic Profile of the eDonkey Filesharing Service”, Passive and Active Network Measurement, 5th International Workshop, PAM 2004, Antibes Juan-les-Pins, France Apiil 19-20, 2004. Proceedings, LNCS, Vol. 3015/2004.

[33] B. Cohen, “Incentives Build Robustness in BitTorrent”, May 2003, http://bittorrent.com/bittorrentecon.pdf

[34] C. H. Ding, S. Nutanong, R. Buyya, “Peer-to-Peer Networks for Content Sharing”, Technical Report, GRIDS-TR-2003-7, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, December 2003