Towards Cost-effective On-demand Continuous Media Service: A Peer-to-Peer Approach

Course Project Report for

CS590N: P2P Networks and Services (Fall 2002)

By

Yicheng Tu and Shan Lei

Project Supervisor:

Prof. Dongyan Xu

1. Introduction

The recent development of broadband networking technology has made deployment of continuous media (CM) service throughout the Internet possible. With CM service becoming feasible, attractive applications such as on-line entertainment video delivery, digital library, and telemedicine systems can be built on top.

Unlike traditional Internet file service, CM requires data of interest being delivered to clients in a streaming manner. That is, each part of the media has to be transmitted and arrive at the client side before a deadline. This brings tremendous challenges to the design of CM servers in a best-effort delivery networking environment, such as the Internet. Generally, the untimely transmission of media fragments becomes worse when more requests are received and the communication channel gets congested. The current solution to this problem is to replicate media data on different sites on the Internet to avoid any individual locations being swamped by requests. One popular practice for media replication is to deploy Content Distribution Networks (CDN) on the edge of the Internet. Each of these CDN servers holds a copy of all media files and is responsible for serving requests from its neighboring area.

However, the cost of constructing and maintaining a CDN is extremely high considering the massive CPU power, storage space and output bandwidth each CDN server has to possess. Among the computing resources in a CDN-based CM service, the output bandwidth was found to be the bottlenecking factor under a flash crowd situation [1]. For example, a server with a T3 line connected to it has a maximum bandwidth of 45Mbps. When the media streams being serviced are MPEG-1 videos with an average bitrate of 1.5Mbps, only 30 concurrent sessions can be supported. The peak number of sessions that need to be serviced simultaneously could be in the order of thousands or even higher. To make the whole system scalable in terms of number of concurrent requests it handles without swamping any server, a large number of such servers have to be deployed in the Internet. The primary concern of this research is to find an inexpensive way to alleviate the traffic load for media streaming on the servers in a CM service infrastructure.

Our approach to solve the above problem is motivated by the emerging new concept of peer-to-peer computing [2, 3, 4]. In a peer-to-peer system, there is no centralized entity controlling the behaviors of each peer. Instead, each peer contributes its share of resources and cooperates with other peers using some predefined rules for communication and synchronization.

In the context of media streaming, a well-organized community of clients can help relieve the service load of CM servers by taking over some of the streaming tasks that could only be accomplished by server machines in a CDN-based scheme. The basic idea is to let clients that obtained a media object act as streaming servers for following requests to that media object. One of the nice features of peer-to-peer community is that its total capacity grows when the content it manages becomes more popular [5]. And this is the most important difference between peer-to-peer and centralized strategy.

With the promising prospect of applying peer-to-peer in CM service, there are some problems we have to face, too. First of all, peers are heterogeneous in the storage capacity and out-bound bandwidth they can contribute. Any attempt to build peer-to-peer streaming infrastructure has to take this into account. A specific problem is how to avoid swamping a supplying peer. Meanwhile, smart replacement strategies have to be studied due to the limited storage a peer uses to cache video objects. Secondly, peers are heterogeneous in the duration of their commitment to the community. This come-and-go behavior makes a peer-to-peer system intrinsically dynamic. How to minimize the effects of this behavior is another research problem we need to address. On the other hand, the scenarios of CM service could also be complex. Besides the number of requests we have discussed above, the number of media objects, access patterns to the same object, Quality-of-Service (QoS) requirements on streaming are all concerns in the design of a CM service. Our proposed CM service infrastructure will address as many of these factors as possible. The idea of peer-to-peer streaming is not new. However, we propose some different methods that we believe are improvements to those in previous work.

To build up an Internet media streaming infrastructure, we have to face challenges in a number of research areas. These include: media coding, QoS control, media distribution, real-time system design for streaming servers, and streaming protocols [6]. Our proposed peer-to-peer streaming architecture brings up research topics that fall into the areas of media distribution and streaming protocols. The computing resources we consider in this study are CPU, storage, and bandwidth. We focus on the bandwidth throughout this research with some attention paid to CPU and storage in a couple of smaller problems.

The rest of this report is organized as the following: Section 2 is composed as a sketch of our proposed solution; In section 3, we will present the results of extensive simulation on system performance and streaming protocol properties. Section 4 discussed work by others that are related to this research; Section 5 concludes this report with conclusions and future works.

2. System architecture

A hybrid media-streaming architecture that combines CDN and a peer-to-peer community is proposed and analyzed in [7]. It is also shown in the paper that the CDN server's streaming load is alleviated by the group of clients that act as supplying peers. Our streaming architecture is based on this hybrid system.

2.1. Assumptions

We follow the basic assumptions in the design of our system as listed here:

- Peers are quite different in capability in terms of outbound bandwidth and disk cache space. We follow the rule of *limited contribution* in [7] to let each participating peer announce the bandwidth and storage it can provide. A little modification here is that we don't specify how many concurrent sessions a peer can accept. Peer should has so little connection as possible
- Video files are heterogeneous in length, bitrate variance, and quality; Peers may store multiple video files in its disk cache.
- Peers may join and leave any time

2.2. System components

We propose a continuous media streaming infrastructure (Figure 1) that is close to the hybrid model in [7]. There are three major components in the system: Directory Server, CDN servers, and the community of participating Peers.

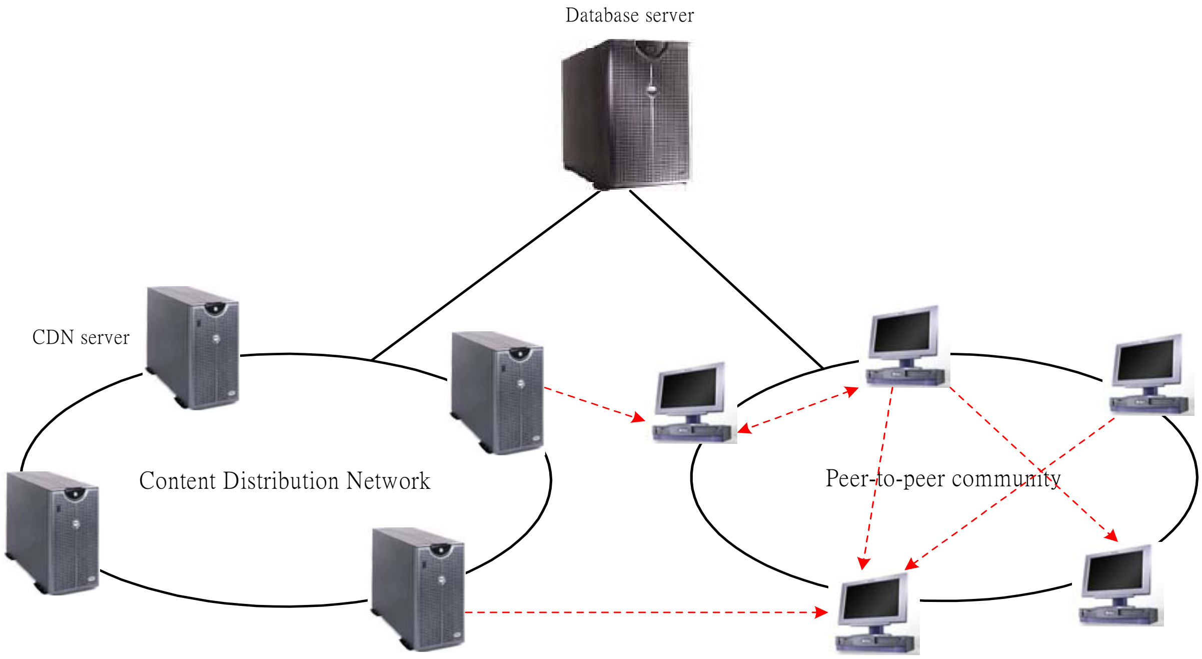


Figure 1. *System architecture of the hybrid CM service*

2.2.1. Directory server

The role of the directory server is to maintain peer/media information. Directory server maintain following information for each peer {list of files; total bandwidth (in unit of R/8); used bandwidth; total space; used space}. An Internet streaming system may have hundreds of thousands or even millions of users as well as a large number of media objects. . The first step for a user request is to locate media objects of interest. This typically involves keyword or content-based search that can be accomplished by the database server in a very efficient way. Keeping a directory of the availability and updated status of these users (peers) and media files that supports efficient update and search is not a trivial task. Although this is always a doable job, we are aiming at minimizing the computational complexity of the directory management. Specifically, our directory server has to keep the paired information of (*Peer*, *Object*), where *Peer* is a client that's willing to serve as a supplying peer and *Object* is the ID of a media file *Peer* holds. Basic operations on these paired data will be:

- Insert(*Peer p*, *Object o*): insert the pair (*p, o*) into the directory;
- Delete(*Peer p*, *Object o*): find and delete the pair (*p, o*) from the directory;
- Leave(*Peer p*): *p* leaves the community, delete all data points whose *Peer* field equals *p*;
- Join(*Peer p*): peer *p* joins the community;
- Find(*Object o*): Find all the peers that hold Object *o*.

We use 2-d tree as the data structure to store the paired directory entries. This is because both *Peer* and *Object* could be used as search keys. A 2-d tree provides efficient (*O*(*log*N) where

N is the total number of tree nodes) search on both dimensions and the range query operations such as Find(*Object o*) is also fast.

2.2.2. CDN Server:

It holds a copy of all media files and is responsible for streaming when the requested media cannot be serviced through the peer-to-peer network in any of following case:

1. Can not find the media file requested from peer community. This situation happens especially when the whole system start up.
2. Even if the file is found among peers. (possibly from multiple peers), the combining bandwidth is not enough.
3. Supplying peers drop from the network during streaming. In this case, the requesting peer will switch temporary to the CDN server to compensate the lost bandwidth until a new supply peer is founded
4. Supplying peers' bandwidth drop temporary due to network dynamics or congestion. The bandwidth drop is compensated by CDN server until network congestion is recovered.

2.3. System Operation Model

When a peer request a file, it follows the steps listed here:

1. Check in local cache, if it is there, then it is done, otherwise go to step 2
2. Send query to the directory server. The directory server search for peers that have the file and return those peers as possible supply peers to the requesting peer. Those peers are locked. If the available cache space in the request peer is not enough to hold the new file, the directory server will also suggest a list of files to replace based on files' access rate or other information. We have tried LFU and FIFO.
3. The requesting peer then probe all those possible supply peers parallel about the request file and get the delay and bandwidth of all those peers. Among those peers, choose the peer have good delay and bandwidth until the required bandwidth is reached. If no enough bandwidth is available, then check whether CDN server has enough spare bandwidth. If the total bandwidth available is enough, then start streaming. Otherwise the request is rejected.
4. If the request is successful, then send following information for each supply peer/CDN used (peer, bandwidth_requested) to directory service. Also send confirmation of file updates. The directory server then updates its directory and unlocks peers.

During streaming:

1. use TFRC in place of TCP for better usage of available bandwidth
2. If TFRC detect small bandwidth drop, request CDN to compensate it
3. If a peer leave the network or a big bandwidth drop, switch to CDN server temporary until a new supply peer is available (get it from the directory server).

After streaming:

Update the information in directory server

3. Peer-to-peer streaming protocol

In this section, we will describe a peer-to-peer video streaming protocol that realizes seamless media playback under variations of link status (e.g. bandwidth, delay, and loss rate). The failure or leave of a subset of supplying peers can be viewed as an extreme case of such variations. The main idea is to use receiver-driven control messages to synchronize and adjust the packet delivery from all the supplying peers. The coordination decisions are made exclusively by the data receiver based on the information received from all senders. Unlike other receiver-driven protocols such as RTP/RTCP, the streaming model our protocol targeting at is of a many-to-one style. Furthermore, we can only make very loose assumptions on the reliability and commitment duration of the data senders. In their recent works, Nguyen and Zakhor [19, 21] proposed a protocol for streaming from multiple reliable video servers. Although their system model is very different from ours, we share some of the ideas in the protocol design. Our proposed protocol is named *Redundant Multi-Channel Streaming Protocol* (RMCSP). An important assumption here is that spare channels can be found at most time in addition to the ones that contain sufficient capacity to satisfy a streaming session. This assumption is proved reasonable in Section 4. Taking advantage of this redundancy, RMCSP uses these spare channels to complement the loss and choose replacements for failed peers. The RMCSP consists of 3 components: a transport protocol, a dynamic connection migration algorithm, and a packet distribution algorithm. We will address the details on the remainder of this section.

3.1. Session initiation

By querying the index server, a client requesting a specific media object (requesting peer) gets a list *L* of all peers that are able to deliver the media. In addition, every client knows the set *C* of available CDN servers when they boot up the video-on-demand service. To initiate the streaming session, the requesting peer probes all peers in *L* and sorts them by increasing RTT of the probe message. The first *n* peers in the sorted list that satisfies the following condition will be used as supplying peers for the streaming:

$$\sum_{i=1}^{n} B(Li) \geq R$$

where *B(Li)* is the posted bandwidth contribution of peer *Li* and *R* is the playback rate of the video. If the sum of all *B(Li)* in *L* is smaller than *R*, we need to find available CDN servers from *C* to serve the video such that the total bandwidth is no less than *R*. If all the CDN servers are not available at this moment, the request is rejected.

When the initial set of supplying peers (and/or CDN servers) is decided, streaming begins. Meanwhile, the requesting peer sends a message containing the set of supplying peers (as an ordered list) to the index server, which, upon receiving the message, changes the bandwidth contribution of these peers in the index. The bandwidth contribution of all but the last peer/CDN server in the set now becomes 0. An observation here is that we always try to minimize the size of the set of supplying peers by taking as much bandwidth as we can from each peer. By this, a supplying peer only serves very few (1 or 2) requests and a requesting peer receives data from a small number of peers, as shown in section 5. The advantage of having a small set of peers in a streaming session is that the overhead of streaming control is low. On the other hand, the list *L* is optimized by latency rather than bandwidth of the links. The reason for this is: to avoid high link stress in the underlying IP network, a client should stream video from sites that are closest to it. This was the whole idea of CDNs where servers are responsible for requests from a relatively local area. Normally, we use packet round trip time (RTT) as a measure of network proximity.

After the initiation, the original peer set *L* is divided into 2 disjoint subsets: the set of the supplying peers (denoted as *LS*), and those that are not supplying peers (denoted as *LB*). During the streaming period, the requesting peer keeps a record of these sets as well as the CDN server set *C*. In case of channel failure or service degradation of any element in *LS*, replacement(s) will be chosen from the other 2 sets, among which *LB* has higher priority than *C*.

3.2. Transport protocol

In spite of its being the dominant transport protocol for a majority of the Internet activities, TCP is ill-fitted for real-time applications. To deal with the best-effort delivery of the IP Internet, TCP rapidly decreases the sending rate (by half) in response to individual packet loss, which results in dramatic changes in the latency of packets. On the other hand, the sending rate is doubled by TCP when the loss rate is found to be increasing. In contrast to the fuzzy back-off and speed-up mechanisms in TCP, the TCP-Friendly Rate Control (TFRC) algorithm proposed in [22] used a strategy called *Equation-based Congestion Control*, where the sending rate changes only as response to a group of loss events. To be specific, the loss rate in TFRC is estimated as the moving average of loss events detected over a time interval. Various algorithms for loss rate estimation were studied in [22]. One of the major concerns of the TFRC design is how to avoid unfair competition for bandwidth between TFRC and normal TCP connections. Therefore, the sending rate of TFRC is increased slowly when the loss rate is decreased. Thus, TFRC achieves smoother change of sending rates than TCP. However, the fluctuation of sending rates in TFRC may still be not smooth enough for streaming multimedia data due to the "slow-start" recovery mechanism. For our purpose of data streaming, TFRC is modified to minimize the change of sending rates by using the maximum available bandwidth. This requires, however, the sender to estimate the current bandwidth and adjust its sending rate. Our protocol will follow the idea of slow back-off and bandwidth estimation in TFRC to avoid dramatic fluctuations in the sending frequency.

For every probing cycle, the data receiver acknowledges the last data packet received from the data sender. Upon receiving the ACK packet, the estimated round-trip time *r* for the TRFC connection is measured. Meanwhile, the receiver computes the loss event rate *p*. The probing cycle is generally set to 2*r*. To smooth the fluctuations of *r* and *p*, both of them are calculated as moving averages of the measures obtained for each individual packet over a time window with size *m*. Methods to compute *r* and *p* can be found in [22]. An estimation of bandwidth availability of each peer-to-peer link can be achieved using *r* and *p*. One of the well-known equations [23] for bandwidth estimation is given as the following:

$$T = \frac{s}{r\sqrt{\frac{2p}{3}} + t_{RTO}\left(3\sqrt{\frac{3p}{8}}\right)p(1+32p^2)}$$

where *T* is the estimated bandwidth, t_{RTO} is the TCP timeout, and *s* is the packet size. This gives an upper bound of allocated sending rate. Therefore, in our model, the maximum sending rate of a specific sender *i* at any time point *t* is given as *S(i, t)* = min (*T(i, t)*, *B(i)*) considering the bandwidth contribution *B* of each participating peer should not be exceeded.

One can argue that our TFRC-like protocol may starve other TCP connections by backing off slower than TCP in case of congestion. This is hardly the case since the sending rate of RMCSP is bound by the bandwidth contribution *B*. The P2P streaming application can never aggressively use whatever bandwidth it finds available. On the contrary, the promise of providing *B* to the streaming task(s) may not be honored if the supplying peer abruptly starts more sessions than it can sustain. Under such circumstances, the bandwidth used for our protocol will be lowered at first and then recovers since TCP connections slow down rapidly. The challenge for RMCSP at this point is how to reduce and compensate for the data loss during the time when *B* is not fulfilled. As we can see later, RMCSP deals with this degradation of bandwidth availability by temporarily "borrowing" bandwidth from other peers. To make life of a requesting peer easier, we may consider deploying bandwidth reservation mechanisms (e.g. per-flow scheduling, differentiated services) in all the peers to isolate the RMCSP traffic from other use of bandwidth. The price for doing this is the overhead for maintaining these reservation mechanisms. This may violate the minimum contribution rule for participating peers therefore it is not a must-have component of our protocol.

3.3. Connection migration

As described in 3.2, the sending rate of each supplying peer is determined on the receiver side by calculating useable bandwidth using equation (2). When the measured bandwidth T of a link changes, RMCSP will respond by changing the sending rate according to the newly detected parameter. However, we don't need to modify the sending rate at any deviation of $T(i, t)$ from $S(i, t)$ since the estimation of bandwidth has some random errors introduced by rapidly changing network status. Instead of responding to any bandwidth change, we only record changes that are greater than a threshold value w, which can be a small fraction (5-10%) of $S(i, t)$. When the measured bandwidth is w or more lower than the current sending rate ($S(i, t)$ - $T(i, t) > w$) for consecutive γ times, the sending rate has to be changed to the $T(i, t)$. The same thing happens when RMCSP finds bandwidth that is w higher than the sending rate for γ times except that it doesn't allow the sending rate go beyond the publicly posted bandwidth contribution $B(i)$ (with exceptions).

From above discussions we know the sum of initial sending rates of all supplying peers are greater than the video playback rate *R*.. However, the decrease of bandwidth of one or more streaming links may end up with a total sending rate lower than R. Under this situation, we need to find more bandwidth from other peers to keep the streaming session sustainable. There are 2 ways to do this. If the lost bandwidth is small, we may increase the sending rate of some well-connected (to the receiver) supplying peers beyond their bandwidth contribution. This kind of 'stealing' is not very harmful to a peer as long as the stolen bandwidth if a small fraction of the bandwidth contribution. It is also reasonable in the real world considering the upload bandwidth is wasted most of the time in a desktop PC.

An alternative is to find more bandwidth from peers other than the ones already in service (e.g. those in set *LB* and set *C*). In the context of one-to-one data communication, connection migration mechanisms were proposed to handle the situation of service degradation. These involve locating new resources from the network and switching service to the newly identified server. Normally, per-connection state is kept in the server and disseminated among servers. Connection switching is done on the transport layer of the network to achieve smooth turnover. However, our problem in the context of many-to-one streaming cannot be solved by a single migration on the transport layer and bookkeeping in the supplying peers is not feasible.

In RMCSP, instead of putting all the workload on the server side, the requesting peer will find a new supplying peer, resynchronize the sending rates and initialize a UDP connection for data transferring. A basic algorithm (*spare channel replacement*) for service migration would contain the following steps:

1. Send a new request to the index server and obtain a list of potential supplying peers;
2. Probe every peer in the list and sort them by response time;
3. Choose peer(s) that are able to provide at least the bandwidth that is lost from the original session and initialize UDP sessions between them and the requesting peer;
4. Recalculate the sending rates and streaming recovers from lack of bandwidth.

Suppose the RTTs of messages passing in step 1 and step 2 are ρ_1 and ρ_2, and UDP initialization costs ρ_3, the recovery time for a connection migration is $\rho_1 + \rho_2 + \rho_3$ (the cost of step 4 is also ρ_2 but it is needed for streaming under healthy situation anyway) ignoring the CPU time spent on above steps. During this time, the streaming application only receives partial data thus the media playback suffers from degraded QoS.

As an improvement to the above algorithm, the requesting peer can keep the state information of redundant peers (what we call *spare channels*) when normal streaming is in progress. Recall the *LB* set defined in section 3.1., which is exactly a set of *spare channels*. When the streaming session is initiated, the requesting peer keeps probing not only all supplying peers (set *LS*) but k peers in *LB* as well. The number k is a small integer such that $k \leq |LB|$ and it is of high probability the k peers being probed can provide enough capacity to cover the bandwidth loss of current supplying peers. If *LB* is an empty set, at least one member in C (CDN servers) is probed instead. To reduce the overhead of probing spare channels, we can set the probing frequency to be lower than that of monitoring supplying peers. Intuitively, if the probing cycle for health monitoring is φ, then the probing cycle for spare channels is set to $k\varphi$. Now let's look at the recovery cost for this *improved spare channel replacement* algorithm. In case of bandwidth loss from current supplying peers, the improved algorithm can start directly from step 4 of the basic algorithm since it has all information needed. This leads to a recovery time of 0, which means the switch over to new data senders is achieved transparently to the users. From another angle, the improved algorithm trades higher bandwidth consumption for lower response time. In the basic algorithm, the probing of spare channels is only executed when the abnormality is detected while the improved algorithm aggressively updates the status so that the information comes handy when needed. We shall from our experiments that the overhead of spare channel monitoring is reasonable. One thing to point out is that the status in the initial *LB* set may be outdated. Therefore, the spare channel monitoring mechanism in the requesting peer may go over step 1 through step 3 to update *LB* while streaming is underway.

The departure of supplying peer(s) during a streaming session imposes no more difficulties than bandwidth loss. There are 2 types of departure: graceful leave and site failure. In a graceful leave, the peer explicitly sends out a *LEAVE* message to the index server as well as all peers that have any connections (streaming or spare channel probing) with it. Upon receiving the *LEAVE* message from a supplying peer, the requesting peer simply switches to other peer(s) using the improved spare channel replacement algorithm. If the message is from a spare channel, the requesting peer will stop probing it and try to acquire another peer as spare channel. Meanwhile, the index server will also update the status of the leaving peer. In case of peer failure, no *LEAVE* messages will be sent thus the failure has to be detected by the health monitoring function. The problem is how to determine a failure. Normally, for any peer *i* at time *t*, when the following condition holds true for consecutive γ times, we believe peer *i* has failed.

$$S(i, t) - T(i, t) > w'$$

where w' is a big fraction of $S(i, t)$. Notice when a real failure happens, it may take some time for the bandwidth estimation T to decrease to 0 since T is the moving average of measures over a time window. A remedy to this is that we declare a failure when we found the instant bandwidth estimation is decreased to some dramatic low level for γ times. After detecting a failure, the requesting peer does the same thing as if it was a graceful leave. Besides, it is the responsibility of the requesting peer to inform the index server of the failure.

To make the server migration algorithm more efficient and less complex, we may consider CDN servers as the only candidates for spare channels. When any bandwidth compensation is needed by a streaming session, the requesting peer can directly go to the CDN server(s). The regular bandwidth probing activities between the requesting peer and the backup peers can be spared assuming high availability of the CDN servers. If the CDN servers are fully loaded when it is needed as replacement in an ongoing session, the streaming task requesting bandwidth has to be terminated. The probability of such occurrences should be small. As we can see from the experimental results in section 4, the streaming load of the CDN servers is lower than their capacity for most of the time.

3.4. Packet distribution

The problem packet distribution trying to solve is how to allocate senders for the group of packets that should be delivered within a synchronization period knowing the sending rate *S* and latency *r* of each supplying peer. For RMCSP, we will use a solution that is very close to the packet partition algorithm proposed in [19]. The basic idea of this packet partition algorithm is: given the *S* and *r* values of all supplying peers as well as the set of media data packets within a synchronization period, each supplying peer locally determines the sender of each packet and sends those it is responsible for. The control message contains the *S* and *r* values of all peers calculated by the requesting peer and a synchronization serial number. Upon receiving the control message, each supplying peer computes the time difference *A* between a packet's estimated sending time and decoding time for every packet and every peer. The peer that maximizes *A* is chosen to be the sender of that packet. The calculation of A is done using the following equation:

$$A(j, p) = T_j - P_{j,p}\,\sigma(p) + 2r(p)$$

In the above formula, $A(j, p)$ is the time difference we are trying to calculate for packet *j* if sent by peer *p*, T_j is the time packet *j* should arrive at the client side for decoding, $P_{j,p}$ is the number of packets sent so far (within the same synchronization period) by *p*, $\sigma(p)$ is the sending interval of *p*, and $r(p)$ is the estimated delay between the receiver and *p*. As we can see, everything needed for calculating $A(j, p)$ is known to the supplying peers: $r(p)$ and $\sigma(p)$, which is the inverse of sending rate *S*, are explicitly included in the control packet; T_j can be obtained from the media stream itself and $P_{j,p}$ is generated on the fly. There is no ambiguity in calculating $A(j, p)$ so that for each packet, an unanimous decision can be made among all sending peers.

The above packet allocation algorithm has the drawback of consuming excessive CPU cycles at the supplying peers when the synchronization frequency becomes high. Some of the resources are wasted since each peer has to do the same floating point computations within every synchronization period. As an improvement to the above algorithm, we can move the burden of packet distribution to the receiver side. When the receiver computes the packet distribution plan locally, it packs in the control message to each supplying peer a list of sequence numbers of the packets needed to be sent from that peer. The same equation as that in the original algorithm can be used for calculating $A(j, p)$ for packet *j* at peer *p*. However, one problem has to be solved if we decide to compute $A(j, p)$ on the receiver side: for a non-CBR media stream, the deadline of each packet (value of T_j) is unknown to the receiver because copies of the media object are only kept in the data senders. One solution would be to download from the CDN server a digest of the media file containing a summary of the deadlines of all packets before streaming starts. The size of the digest file can be very small. For video streams, packets within a frame share the same deadline thus the digest size is only related to the frame rate of the media. Consider a typical MPEG-1 video with average bitrate of 800Kbps and frame rate of 30fps, the overhead for transmitting the digest file is only 30×24/800K = 0.088% assuming 24 bits are used to represent a packet deadline entry with a time instance, a starting packet number, and an ending packet number.

4. Experimental results

We now present results obtained from simulation studies on the aspects of system dynamics as well as performance of the proposed streaming protocol.

4.1 System performance

Simulation experiments have to be done to study the system dynamics. We want to get an idea of how the system behaves in the following aspects:

- Does our system show any benefit in contrast to a CDN-only solution?
- When does the system start to show the benefit?
- How do the different replacement policies (LFU, FIFO) affect the system performance?

We use request reject rate and the mean load of CDN servers as metrics of measuring the system performance. The "benefit" mentioned above is simply given by comparing the obtained metrics between our architecture and the CDN-only system.

4.1.1. Simulation setup

The parameters for simulation experiments of system performance are listed as the following:

- Files size: a random variable of normal distribution with mean of 100 minutes and standard deviation 80.
- File Access Rate: a random variable according to the Zipf Law. R=1/pow(I,0.8) , I is the order of access rate.
- Bandwidth requirement of files: R, currently we assume all files are CBR and same quality.
- Bandwidth of CDN: 30R
- Total File Number: 2000
- File request of the whole system is a Poisson process with a mean of 1 per minute. The request is generated randomly from all peers.
- Bandwidth of Peers:
  500 peers with bandwidth R/4
  200 peers with bandwidth 2R/4
  200 peers with bandwidth 3R/4
  100 peers with bandwidth R

Cache space of peers: a random variable of normal distribution with mean to hold 10 average size files. The peers with more bandwidth also get larger mean space.

4.1.2. Simulation results

In figure 2, we can get an idea of total bandwidth contribution of CDN server and peers. In the beginning, since there are almost no files in peers' cache, all requests must be served by the CDN server. As time passes, peers get files in their cache and begin to serve other peers, the bandwidth contribution of peers increase very quickly until it reaches 80 after about 3000 peer requests, after which the system becomes stable. In the stable state, CDN bandwidth usage is 20R, since its total bandwidth is 30R, so it still has bandwidth available that is enough for 10 more streaming tasks. Peer bandwidth contribution is about 80.

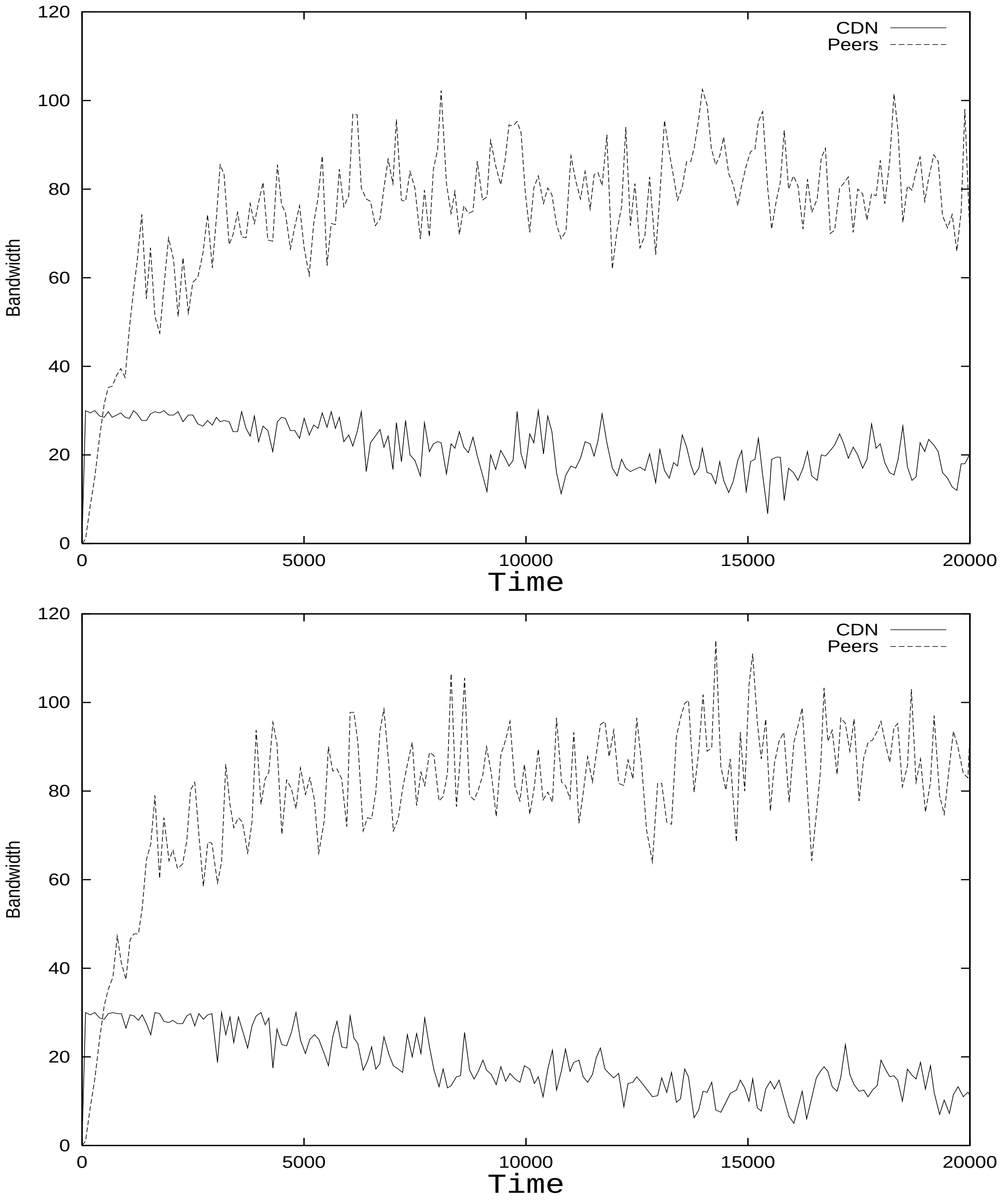


Figure 2. *Bandwidth usage of CDN servers and participating peers under different replacement policies. LRU (upper) and FIFO (lower).*

Rejection happens when there is no enough bandwidth to serve a peer request. An important benefit of our system in out system is to reduce rejection rate. When no supply peers is used, since CDN server has no enough bandwidth to serve all requests. The rejection rate will be very high as what is showed in the beginning of the graph. As time increase, the system get stable and peers can contribute more bandwidth, the rejection rate decrease dramatically, after the time 7000, the rejection rate is almost 0. We can also see from Figure 3 that the rejection rate for LFU case is lower than the rate for FIFO case.

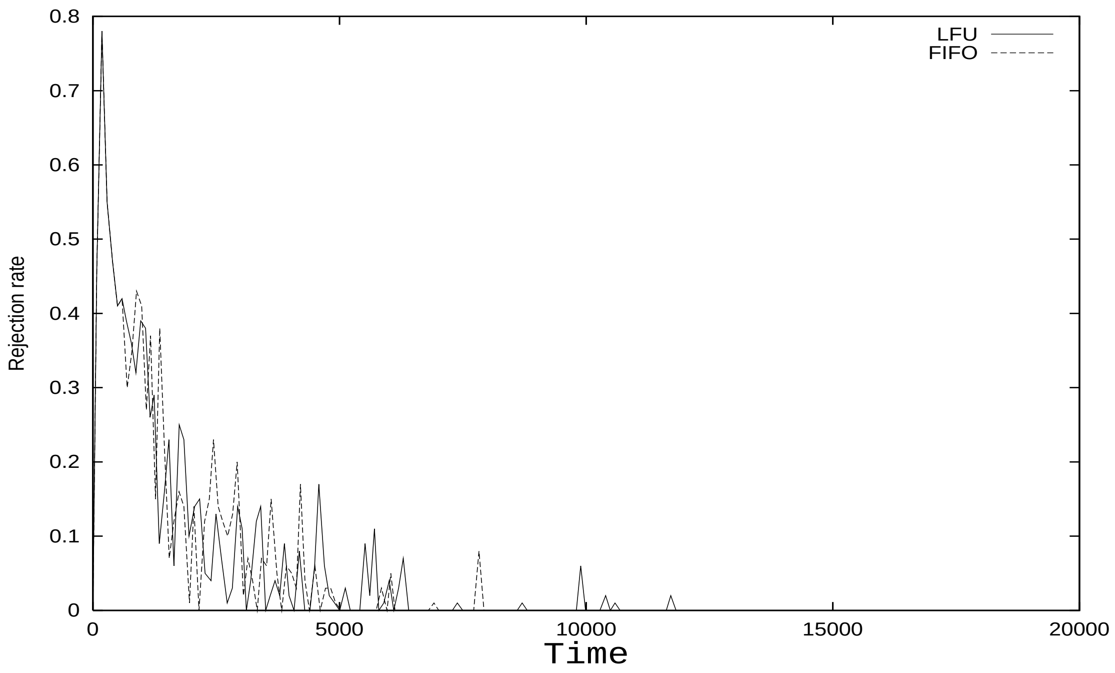


Figure 3. *Rejection rate of requests under different replacement polices.*

The following figure (Figure 4) shows the rate that a request can be satisfied from local cache. As time increase, the rate increase because of more files in cache. After peers cache space become full, the rate become stable at about 8%. LFU has higher local access rate than FIFO in most cases.

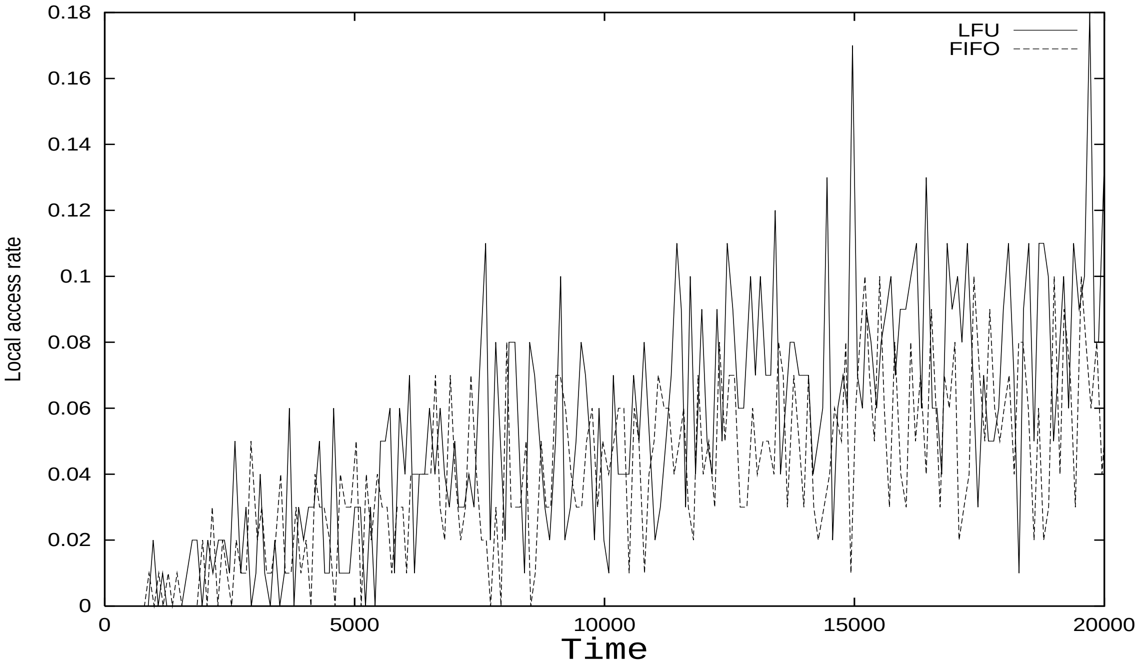


Figure 4. *The rate of requests being served locally under different replacement polices.*

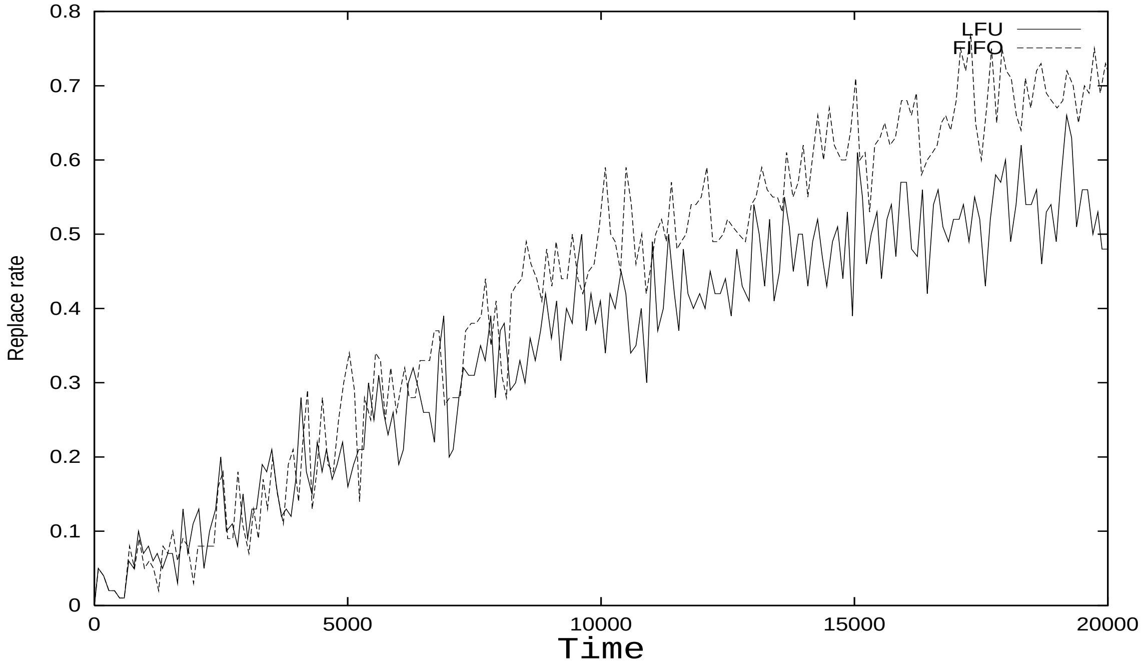


Figure 5. *The rate of local replacement under different replacement polices.*

Peers need to replace old files in its cache when cache becomes full. The above figure (Figure 5) shows that the replacement rate increase as time increase. That is because cache space will become more occupied as time increase. As time passes, the replacement rate for LFU will become stable around 0.5 vs. 0.7 for FIFO. In this sense, LFU is better.

4.2 Peer-to-peer streaming

The basics of the RMCSP were studied in the *NS2* network simulator. The major purpose of the simulation experiments is to test the feasibility of RMCSP, especially its robustness under peer failures. Unlike the experiments designed to examine system dynamics, the setup of the streaming simulation has less concern on the topology and dynamics of the underlying peer-to-peer network. Instead, we focused on investigating various parameters of individual links that are involved in one single peer-to-peer streaming session running RMCSP.

4.2.1. Simulation setup

The network configuration of the RMCSP session simulated is shown in Fig. &&&. In this simple topology, the requesting peer (Receiver) is connected to its local router R via a link with total bandwidth of 1.5Mbps, symbolizing a typical DSL user. Three supplying peers, Peer 1, 2, and 3 are shown to be linked to R by 3 different network connections that share no common congestion link among them. In the graph, these 3 connections are abstracted into virtual links with parameters different than the one between R and the receiver (Link 1). We picture all supplying peers are DSL users throughout the Internet so the bandwidth of the 3 virtual links are also set to 1.5Mbps each, assuming the outbound connection of a DSL subscriber is the throughput bottleneck of the virtual link. To simulate the distance between R and the supplying peers, the delays of Link 2, 3, and 4 are configured to be one order of magnitude higher than Link 1. The arrows in Figure && represent the direction of media data transmission. The control messages and ACK packets are always sent at the opposite direction. The RMCSP coordination unit in Figure && refers to the controlling module in RMCSP that does server migration and packet allocation.

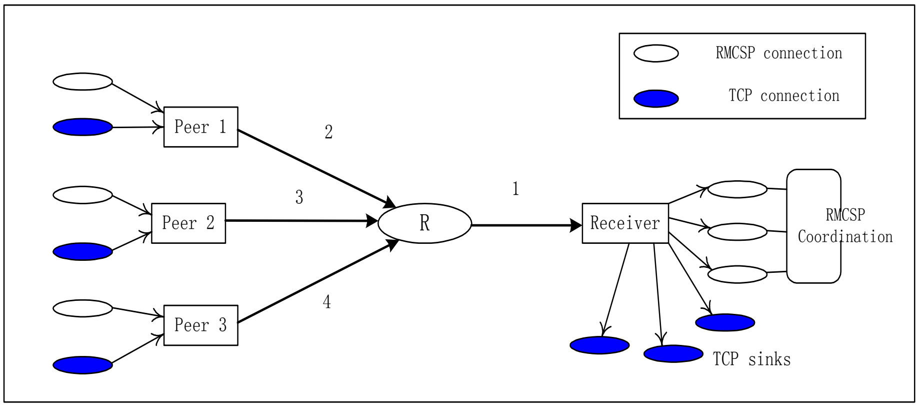


Figure 6. *Simulation setup for peer-to-peer streaming*

In the simulated streaming scenario, we consider a 300-second CBR video with bitrate 720Kbps. Peer 1 and Peer 2 are selected as the original supplying peers for the streaming of this video with Peer 1 contributing 480Kbps of bandwidth and Peer 2 providing 240Kbps. Peer 3 is a backup peer that is also capable of servicing the media stream to the receiver. When streaming begins, the sending rate for Peer 3's RMCSP is constrained to 8Kbps, reflecting the basic probing activities that send out messages in a 1-second interval. In addition to the RMCSP connections needed, arbitrary TCP connections (running FTP applications, shown as filled ovals in Figure 6) are also added to the same data communication paths for the streaming. Each RMCSP session is accompanied by one such TCP connection. From the simulation, we found the number of competing TCP connections has great impact on the performance of our streaming protocol. The TCP connections can easily put the link status into wild oscillations if their sending rates are not under any control. If we set an upper bound of bandwidth consumption on TCP, the link status can be friendly enough to data streaming using RMCSP. The reason for this is because RMCSP depends on the link stress, as we can see in section 4.2.3.

At time 0s, the FTP applications attached to the TCP agents started transferring data. At time 2s, the streaming of the video started with initial sending rates of both supplying peers being the bandwidth contribution posted (480Kbps and 240Kbps, respectively). These numbers are also the upper bound of sending rates for various peers. For the backup peer Peer 3, no data was transmitted through its RMCSP link at first except the packets for probing bandwidth availability. The requesting peer sends out control packets for synchronization every 500 milliseconds. The packet size for RMCSP was 1000 bytes. Algorithm used for computing loss rate is Weighted Average Loss Interval (WALI) method mentioned in [22]. Both Link 1 and Link 2 were applied a uniform random loss model with loss rate set to 0.01. Therefore, the loss pattern of the RMCSP connection between Peer 1 and Receiver is different from that between Peer2 and the Receiver, with the latter less prone to packet drop. Peer failure happens at time 270s, when Peer 2 was abruptly turned down. The video streaming ends at time 300s. As a control experiment, we also tested the same scenario using TFRC as the transport protocol. The results we are interested in obtaining from this simulation are the smoothness of the sending rate, estimated and real data losses, and comparison between TFRC and RMCSP.

4.2.2. Overall evaluation

Figure 7 demonstrated the measured sending rates of the above streaming task. For most of the streaming time, the sending rates of both supplying peers are on a smooth line that represents their predefined bandwidth contribution. Under such circumstances, streaming is going perfectly with the highest level of QoS guarantee. There are a few valleys for the plotted lines of both Peer 1 and Peer 2. Such decreases of sending rates are mostly likely caused by packet drops that came close to each other. In turn, the smaller loss interval was translated into bandwidth availability that is lower than the target sending frequency. Another observation is that the valleys in the curve for Peer 1 are generally deeper than those for Peer 2. This can be explained by higher link stress (480Kbps vs. 240Kbps) and loss frequency in Peer 1. According to Figure 7, our protocol recovers from dramatic changes of sending rates caused by degraded network conditions in the order of seconds. Most of the valleys are less than 5 seconds wide and none is wider than 10 seconds.

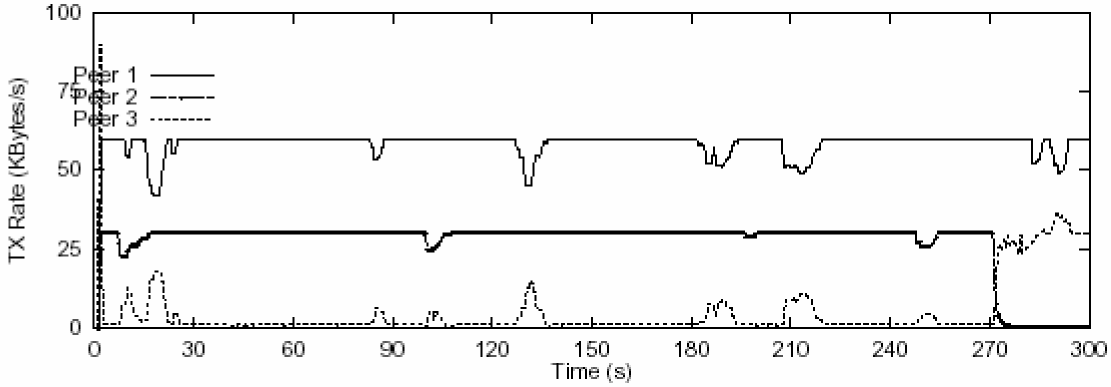


Figure 7. *Transmission rate of peers over time using RMCSP*

RMCSP was also found to be sensitive to bandwidth underflow and peer failure. Almost all the lost bandwidth due to lowered sending rates is compensated for by the contribution of the backup server, Peer 3. The latency for the bandwidth compensation is also low. Basically, the time between the detection and compensation of bandwidth underflow is directly related to the synchronization frequency. Actually, the latency for bandwidth compensation is exactly one synchronization cycle. We could resynchronize immediately upon detection of lack of bandwidth to make RMCSP more responsive. However, with a synchronization cycle of 500ms, our protocol is sensitive enough to handle most loss events. It may not worth the fuss to resynchronize every time abnormity is sensed. The extreme case of bandwidth loss, peer failure, was also very well handled by our protocol. In less than 3 seconds, the estimated bandwidth of the channel where failure (Peer 2) occurred goes down to near zero. Within the same time scale, Peer 3 started to take over the streaming load of Peer 2. Fluctuations can be observed on the transmission rate of Peer 3 when it first takes over.

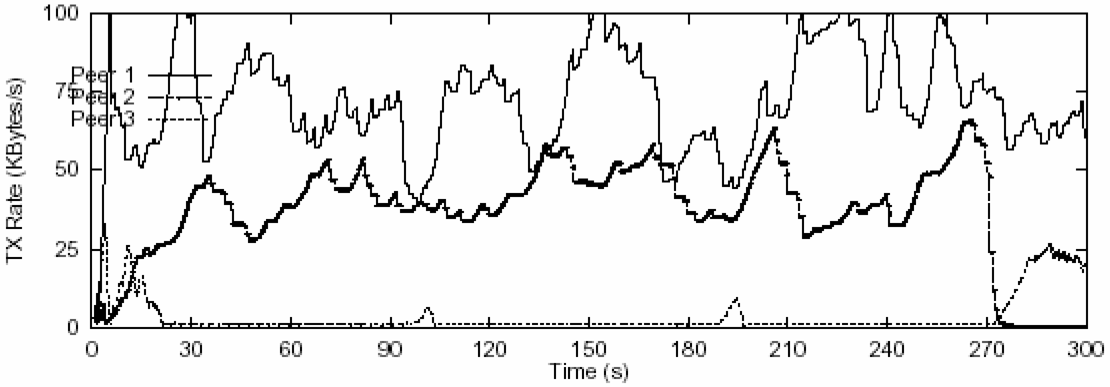


Figure 8. *Transmission rate of peers over time using TFRC*

The same simulation was run using TFRC as the underlying transport protocol. The measured sending rates (Figure 8) bear extremely high variance. This is undesirable for our streaming architecture in 2 aspects: first of all, TFRC frequently uses more bandwidth than the contribution value posted by supplying peers. For example, both Peer 1 and Peer are using more bandwidth than it is willing to contribute for a majority of the total time. As a result of this, the bandwidth consumed from Peer 3 is minimal. On the other hand, frequent change of bandwidth allocation among supplying peers increased the complexity of media synchronization on the application level. If we compare the estimated loss event rate of both protocols, we didn't see much difference (Figure 9). The only exception is at the early stage of the streaming, when TFRC obtained an excessively high loss rate value for Peer 2. This could be an unfortunate drop of a number of packets in a small time interval. The number of consecutive drops doesn't have to be big to make this happen because no historical data can be used to lower the averaged loss event rate at that moment. However, this abnormality recovers to a health level very quickly. The similarity of estimated loss rate between RMCSP and TFRC is as expected since RMCSP only modifies TFRC by capping the sending rate and getting rid of "slow start". For the same reason, the estimated bandwidth for both protocols are also similar to each other (data not shown). It is just RMCSP makes better use of bandwidth than TFRC. On the other hand, RMCSP's aggressive use of channel capacity is constrained within a boundary thus the competitive TCP connections are not starved. From this point of view, we may read RMCSP as a loose bandwidth reservation mechanism on the outbound link of the data senders (complete bandwidth reservation requires collaboration of intermediate routers). Of course, the assumption here is that the sender keeps no other connections that are more aggressive than RMCSP in consuming its outbound bandwidth, such as UDP connections. In a modern computer for general use, this assumption is realistic.

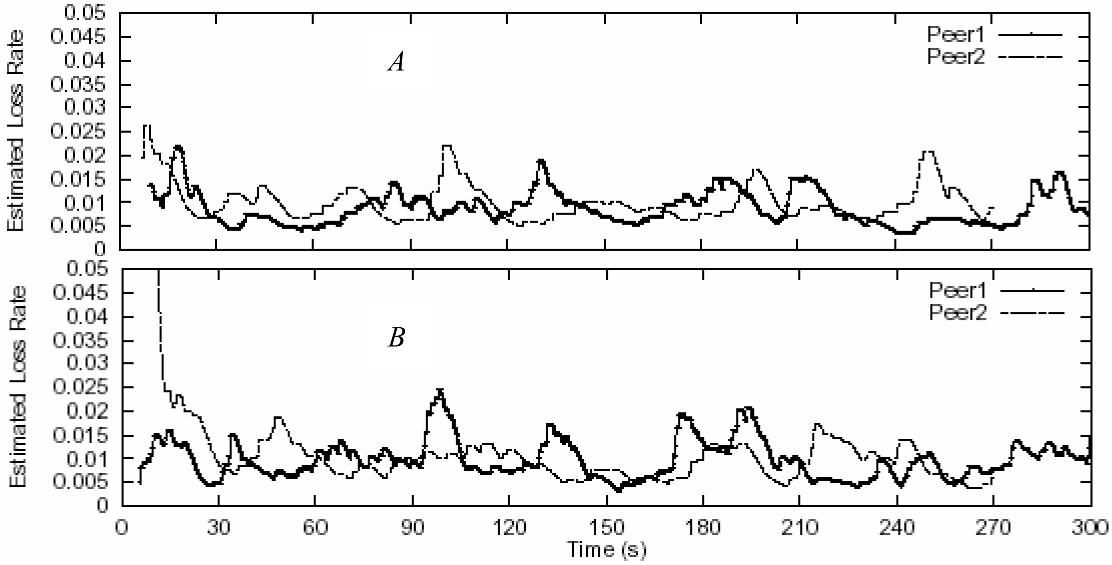
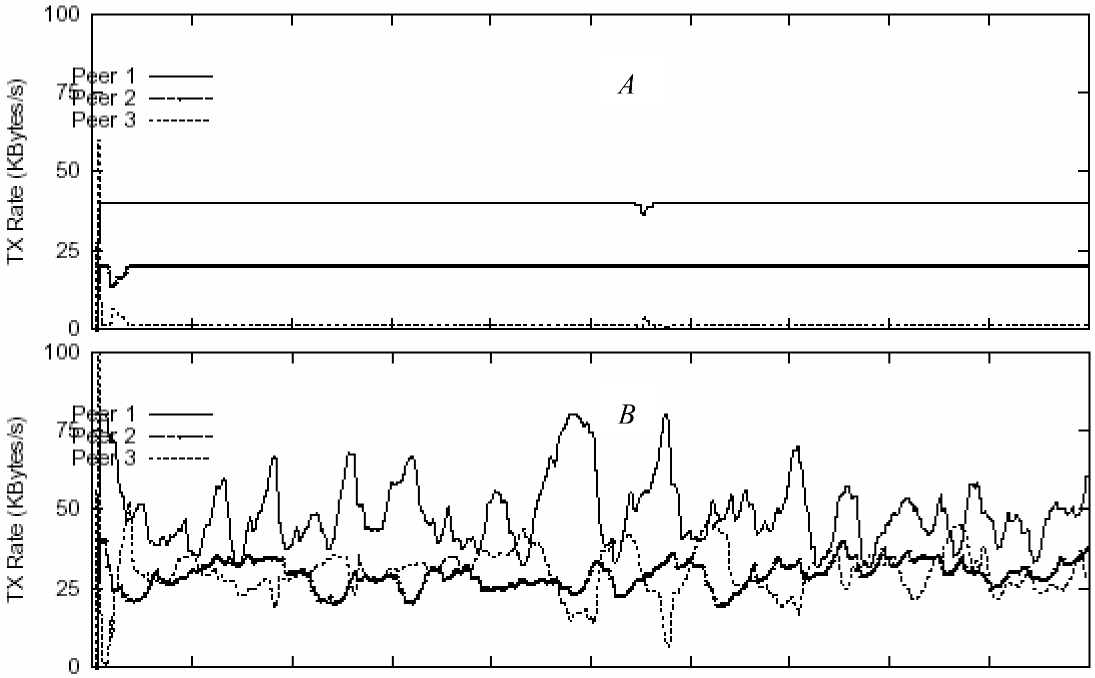


Figure 9. *Estimated loss rate for RMCSP (A) and TFRC (B)*

4.2.3. Effects of link load on streaming

Link stress also has significant effects on the performance of our streaming protocol. The sending rates for streaming videos with different bitrates are plotted in Figure 10. Figure 10A shows the results for a video with bitrate only 480Kbps with Peer 1 contributing 320Kbps and Peer 2 160Kbps (this might not be a realistic bitrate for a video). As we may see from the figure, the sending rates are almost straight lines for all 3 peers. This means there is always enough bandwidth for video streaming when the streaming job requires only less than 1/3 of the total bandwidth of a DSL connection. In this case, the link stress of streaming for Peer 1 is 320K/1.5M ≈ 20% and 160K/1.5M ≈ 10% for Peer 2. Figure ///B demonstrated another extreme scenario using a video of bitrate 960Kbps. Again, the contribution from Peer 1 is double the contribution from Peer 2 (640Kbps vs. 320Kbps). Under such circumstance, all the peers showed degraded performance in terms of smoothness of sending rate. Among them, Peer 1 has the worst curve since its link stress is the highest. The expected sending rate (640Kbps) actually becomes the peak value for Peer 1, meaning such high bandwidth contribution (640K/1.5M ≈ 43%) for participating peers can hardly be achieved in media streaming. The actual sending rate for Peer 1 was more or less oscillating around the 400Kbps (50KBps) line. For Peer 2, the base line for fluctuations is about 240kbps (30KBps). Without surprise, the backup peer (Peer 3) had to compensate for the insufficient bandwidth for most of the time. The level of contribution from Peer 3 was close to that from Peer 2. In Figure 7, the bandwidth contribution for Peer 1 was 480Kbps (60KBps), or 1/3 of the total bandwidth of the link, and the curve for Peer 1 even started to show some unpleasant fluctuations. Actually, 480Kbps is probably the upper bound for a meaningful bandwidth contribution in a 1.5Mbps DSL line. Even when the contribution goes up to 520Kbps (65KBps), dramatic changes of sending rates can be observed (data not shown here). Therefore, we regard 1/3 of total bandwidth as a rule of thumb for peer contribution.

In studying the link stress problem, we also found that it was the load of individual links that really counts. The total bandwidth of the streaming object is not the bottlenecking factor as long as it does not exceed the inbound bandwidth of the receiver. In Figure 10C, we showed the measurements of a video with 1.12Mbps bitrate. The difference from other scenarios is that 4 supplying peers are used (only 3 are plotted in the graph), each of them contributing the same amount of bandwidth (280Kbps). It is easy to see that the streaming performance is much better than the previous scenarios where there are certain links in heavy stress. We can conclude that to some extent, seeking bandwidth from more peers is better than aggressively using more bandwidth from fewer peers. How close can the media bitrate approach the 1.5Mbps theoretical limit? More simulations indicate that when the bitrate reaches 1.2Mbps the performance is inevitably inferior to lower bitrate streaming no matter how many supplying peers to use. Another factor to consider is the overhead of maintaining all concurrent RMCSP connections.



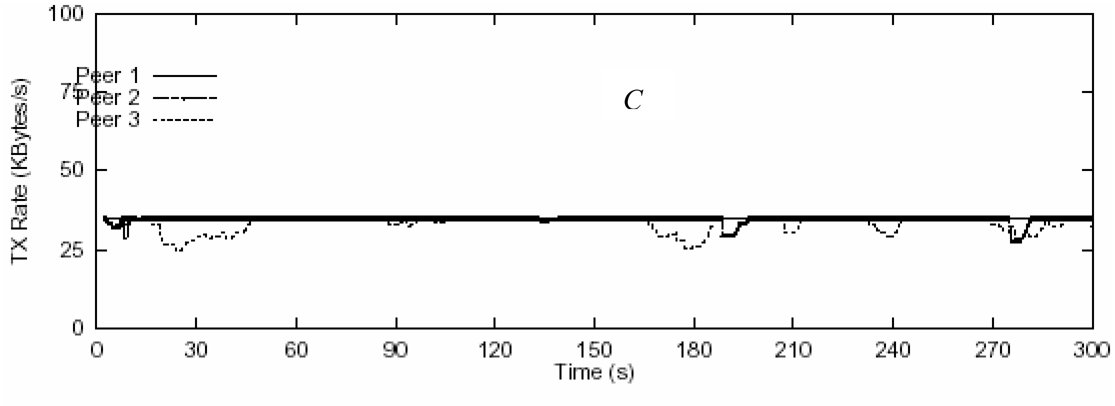
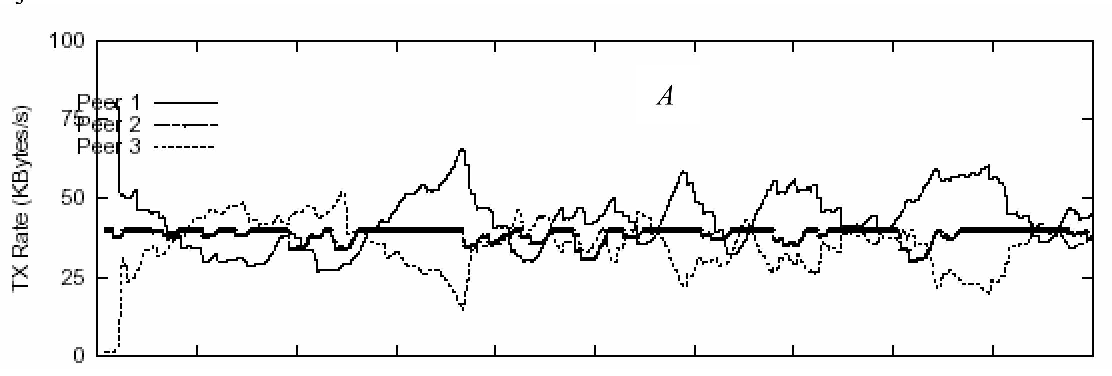


Figure 10. *Transmission rate under different total streaming bitrate. A.Total bitrate of 480Kbps with 320Kbps from Peer1and 160Kbps from Peer2; B. Total bitrate of 960Kbps with 640Kbps from Peer 1 and 320Kbps from Peer 2. C. Total bitrate of 1120Kbps with 4 supplying peers each contributing 280Kbps.*

In analyzing the impact of link stress on sending rates, we considered 2 possible factors: the competition with TCP and the inability of DMCSP to accurately estimate bandwidth over a lossy link. We wanted to know which factor gives more weight on the whole story. We studied this problem by simulating unrealistic situations where the above 2 factors can be isolated (Figure 11). We follow the same scenario of that studied in Figure 10B, where Peer 1 has very high link stress. The metric we used is again the sending rates of peers over time. In Figure 11A, we simulated the situation assuming no losses on all 4 links in the simulated topology. As we may see, the bandwidth usage for Peer 1 was still under heavy fluctuations and the baseline of sending rates for all 3 peers are very close. As compared to Peer 1, the sending rate change for Peer 2 is more stable. In another experiment whose results are shown in Figure 11B, all TCP connections are taken away when the streaming is undergoing. The streaming is accomplished over lossy links, however. Without competition of TCPs, the bandwidth usage of RMCSP is closer to the expected style: Peer 1 can reach its bandwidth contribution for at least half of the streaming time. And the valleys of Peer 1's sending rate curve are properly filled by Peer 3. We can clearly conclude, by comparing Figure 11A and figure 11B, that co-existing TCP connections is the controlling factor that inhibits bandwidth over-provisioning of supplying peers. Of course, this doesn't mean RMCSP is perfect in modeling the network status. Actually, as an ongoing work, we are rethinking the bandwidth estimation algorithm of RMCSP. But this is beyond the scope of this project.



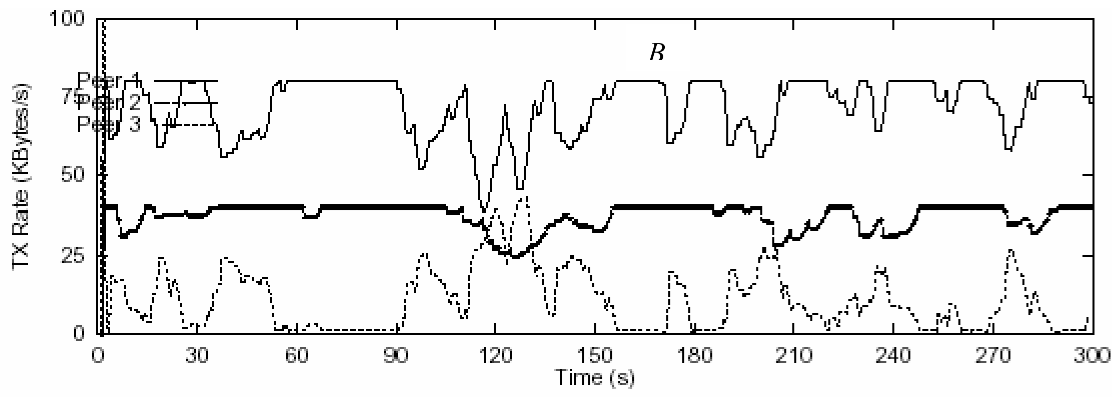


Figure 11. *Transmission rate under different situations. Same scenario as in Figure 10B with total bitrate of 960Kbps. A. Without link loss. B. Without competition from TCP connections.*

5. Related work

A good review of Internet video streaming can be found in [6]. The key areas of video streaming are identified in the review and thorough discussions are made on the major approaches and mechanisms within these areas. Future research directions are also pointed out. [2] and [3] introduced the general philosophy of peer-to-peer computing, current research efforts, and applications of peer-to-peer systems. The candidate systems that can be used as substrates of our peer-to-peer self-organization mechanism include Pastry [9], Chord [10], and CAN [11].

In the context of peer-to-peer streaming, both the CoopNet project of Microsoft [12] and the ZIGZAG prototype [13] of University of Central Florida explored the way to deliver media streams to many concurrent clients by application-level multicasting. In these 2 projects, attention was paid to efficiently maintain the multicast tree in an environment where user behavior is unpredictable. CoopNet utilized the method of MDC to deal with the in-session leave/failure of streaming peers. Details about MDC can be found in [14] and [15]. Systems such as SpreadIt [16], Allcast [17], and vTrails[18] are all peer-to-peer streaming systems that are close in spirit to CoopNet and ZIGZAG. Our system differs from these efforts in the sense that we are focusing on the delivery of on-demand media instead of live video.

Work on hybrid streaming architecture [7] that combines CDN and peer-to-peer is directly related to the system design in this research. In some sense, this proposal is directed to an extension of the model proposed in [7]. Mathematical analysis as well as simulation experiments showed that the peer-to-peer community significantly lowered the streaming load of the CDN servers. Based on their model, the contribution of our research would be to study the system behavior by putting more details into the model and provide solutions to problems discussed in Section 2 that are not addressed in [7].

An earlier work [5] to [7] developed an algorithm that assigns media segments to different supplying peers and a protocol for peer admission. Another research project by Nguyen and Zakhor [19] is more closely related to our research in the aspect of streaming protocol development. In this paper, they presented an RTP-like protocol that does rate control and packet synchronization. A packet partition mechanism used for the supplying peers to determine which packets to send is also assimilated into the design of their protocol. From [20] we can get access to open-source streaming software releases that are built upon RTP/RTSP.

6. Conclusions and future work

In this project, we studied the system dynamics of a hybrid peer-to-peer streaming system. The system can get great benefit from using supplying peers. As peers' cache become full, the bandwidth contribution of supply peers is almost 4 times of the CDN bandwidth. And the rejection rate becomes almost zero after system become stable. In the stable state, the CDN bandwidth usage is about 20, so it has enough space bandwidth to act as a backup server for all peers to deal with peer drop or network congestion. LFU is better than FIFO in terms of lower rejection rate and higher local access rate.

In addition to the system analysis, a peer-to-peer streaming protocol, RMCSP, was also presented. The design of RMCSP focused on the capability of the receiver to coordinate concurrent data transport links to keep smooth streaming, as well as a redundancy-aware bandwidth compensation mechanism to handle dynamic network status including peer failures. The idea of achieving smooth sending rates on the transport level of RMCSP was adapted from the TFRC protocol, which was modified to satisfy more strict requirements for data streaming. Simulation results showed that RMCSP performs fairly well under realistic network conditions. Various factors related to the performance of RMCSP were also analyzed and discussed. Based on the analysis, we also presented some guidelines on the selection of peer contributions and streaming organization for the purpose of achieving highest performance by RMCSP.

Future work involves more in-depth study of the dynamics of the RMCSP protocol, especially revisiting the estimation algorithms for loss event and bandwidth. The simulation for the streaming protocol made some assumptions that may not be realistic in a real-world system. For example, most of the popular video compression formats ended up with VBR video streams. The conversion of VBR into near-CBR streams via smoothing algorithms has to be considered an indispensable part of the research. Other issues on the video streaming application level such as video coding, error correction, and QoS control are also important. Furthermore, more efforts are needed to implement a peer-to-peer streaming prototype that can be deployed in a real network.

Reference

[1] V. N. Padmanabhan and K. Sripanidkulchai. The Case for Cooperative Networking. *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002

[2] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Rihard, S. Rollins, and Z. Xu. Peer-to-Peer Computing. *HP Labs Technical Report HPL-2002-57.*

[3] A. Rowstron and P. Druschel, Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. *ACM/IFIP Middleware 2001.*

[4] J. Crowcroft and I. Pratt. Peer to Peer: peering into the future. LNCS 2345, *Proceedings of the IFIP-TC6 Networks 2002 Conference*, Pisa, May 2002.

[5] D. Xu, M. Hefeeda, S. Hambrusch, and B. Bhargava. On Peer-to-Peer Media Streaming. *Proceedings of IEEE ICDCS 2002*, July 2002.

[6] D. Wu, Y. T. Hou, W. Zhu, Y-Q. Zhang, and J. M. Peha (2001). Streaming Video over the Internet: Approaches and Directions. *IEEE Transactions on Circuits and Systems for Video Technology.* Vol. 11, No. 1, February 2001.

[7] D. Xu, H-K. Chai, C. Rosenburg, and S. Kulkarni (2003). Analysis of a Hybrid Architecture for Cost-Effective Streaming Media Distribution. *Proc. of SPIE/ACM Conf. on Multimedia Computing and Networking (MMCN 2003)*, Santa Clara, CA, January 2003.

[8] V. N. Padmanabhan, H. J. Wang, P. A. Chou, and K. Sripanidkulchai. Distributing Streaming Media Content Using Cooperative Networking. *ACM NOSSDAV*, May 2002.

[9] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems. In *International Conference on Distributed Systems Platforms (Middleware)*, November 2001.

[10] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM '01*, San Diego, California, August 2001.

[11] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-

Addressable Network. In *Proceedings of the ACM SIGCOMM '01*, San Diego, California, August 2001.
[12] http://research.microsoft.com/~padmanab/projects/CoopNet
[13] D. A. Tran, K. A. Hua, and T. T. Do. ZIGZAG: An Efficient Peer-to-Peer Scheme for Media Streaming. *Technical report of University if Central Florida* (CS-UCF 2002).
[14] Y. Wang, M. T. Orchard, and A. R. Reibman. Optimal Pairwise Correlating transforms for Multiple Description Coding. In *Proceedings of International Conference of Image Processing*, Chicago, IL, October 1998. IEEE.
[15] J. Apostolopoulos, T. Wong, W. Tan, and S. Wee. On Multiple Descrption Streaming with Content Delivery Networks. In *Proceedings of IEEE INFOCOMM 2002*, June 2002.
[16] H. Deshpande, M. Bawa, and H. Garcia-Molina. Streaming Live Media over a Peer-to-Peer Network. Technical Report 2001-31, Stanford University, August 2001
[17] http://www.allcast.com
[18] http://www.vtrails.com
[19] T. Nguyen and A. Zakhor. Distributed Video Streaming over Internet. In *Proceedings of SPIE/ACM MMCN 2002*, January 2002.
[20] http://www.live.com
[21] T. Nguyen and A. Zakhor. Distributed Video Streaming with Forward Error Correction. *Packet Video Workshop 2002*, Pittsburgh PA, USA.
[22] Sally Floyd, Mark Handley, Jitendra Padhye, and Joerg Widmer. Equation-Based Congestion Control for Unicast Applications**.** *SIGCOMM 2000*. August 2000.
[23] J. Padhye, V. Firoiu, D. Towsley, J. Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. *Proc. ACM SIGCOMM'98*, Vancouver, CA, September 1998.