PeerStreaming: A Practical Receiver-Driven Peer-to-Peer Media Streaming System

Jin Li
One Microsoft Way, Bld. 113, Redmond, WA 98052.
Email: jinl@microsoft.com

ABSTRACT

We have developed PeerStreaming, a receiver-driven peer-to-peer (P2P) media streaming system. Recognizing the fact that the peer is performing a favor for the client and the server during the streaming session, the design philosophy of PeerStreaming is to ensure that the peer is lightweight and the P2P network is loosely coupled. The peer performs simple operations, and may elect to cache only part of the streaming media. It does not collaborate with other peers, may be unreliable and may drop offline or come online during the streaming session. The client coordinates the peers, streams the media from multiple peers, performs load balancing, handles the online/offline of peers, decoding and rendering the media, all in real-time. Through the high rate erasure resilient code, the serving peers may hold partial media without conflict, and the client simply retrieves a fixed number of erasure coded blocks regardless of where and what specific blocks are retrieved. PeerStreaming can stream the embedded coded media, and vary the streaming bitrate according to the serving bandwidths and the client queue status. Via the Microsoft DirectShow framework, PeerStreaming is capable of live P2P streaming, decoding and rendering a number of media format, such as MPEG1/2/4, WMA/WMV, and the embedded media of [10].

Categories and Subject Descriptors

C.2.4 [**Distributed system**]: Distributed applications
H.5.1 [**Multimedia Information Systems**]: Audio, Video

General Terms

Algorithms, Design, Performance

Keywords

Media streaming, peer-to-peer (P2P), receiver-driven, on-demand, high rate erasure resilient code, scalable media, embedded coded media, loosely coupled P2P network, live streaming, practical streaming system.

1. INTRODUCTION

According to market research [1], over half of the Internet users in the United States have accessed some form of streaming media in 2004. Streaming music is still the most popular activities of the users, but the popularity of streaming video is growing rapidly. Unlike web pages, a streaming media file is huge. A 3 minute movie trailer encoded at 2 megabits per second (Mbps) results in a 45 megabyte (MB) media file. Streaming media also carries stringent demand in the timing of packet delivery. The large size of the streaming media as well as its delivery timing requirement causes a streaming media server to be expensive to set up and run. Currently, the going rate for the streaming media server is \$10 per 1GB of serving traffic. This may not seem much, until you realize that the server bandwidth alone costs \$0.45 per movie trailer distributed. Apparently, more efficient way of distributing the streaming media needs to be developed.

Recently, there is great interest in using the peer-to-peer (P2P) network in media streaming. The idea is to let the peer node assist the media server in distributing the streaming media. A great number of P2P media streaming systems have been developed. The end system multicast (ESM) [2] and PeerCast [3] were two systems using the application-level multicast (ALM) for media streaming. In ESM and PeerCast, the peer nodes self organized into an overlay tree over the existing IP network and the streaming data were distributed along the overlay tree. The cost of providing bandwidth was shared amongst the peer nodes, reducing the burden of the media server. In ESM and PeerCast, the leaf nodes of the distribution tree only received streaming media, and did not contribute to content distribution. CoopNet [5] and SplitStream [6] built multiple distribution trees that spanned the source and the peer nodes. Each tree in CoopNet and SplitStream might transmit a separate piece of streaming media. As a result, all peer nodes were involved in content distribution. OStream [8] used cache-and-relay approach so that the peer node might serve the client with previously distributed media from its cache. GnuStream [6] was a receiver driven P2P media streaming system built on top of Gnutella. Utilizing an application level P2P service called CollectCast, PROMISE [7] sought for serving peers that were most likely to achieve the best streaming quality, and dynamically adapted to network fluctuations and peer failure. These were only a few examples of the recent schemes of the P2P media streaming solutions.

In this work, we propose and implement PeerStreaming, which is a receiver-driven P2P media streaming system. Compared with the existing works [2]-[8], PeerStreaming has a number of unique features. The PeerStreaming serving peers are designed to be lightweight. They may only hold a portion of the streaming media, and perform simple operations during the service. The P2P network of PeerStreaming is loosely coupled. The peers do not collaborate with other peers, may be unreliable, and may drop offline or come online during the streaming session. It is the PeerStreaming client that drives the P2P streaming process, connects to peer that just comes online, redirects the requests dropped by offline peers, and balances the load among the peers. PeerStreaming supports the embedded coded media. It also uses the high rate erasure resilient code to allow each peer to hold partial media without conflict, and simplifies the operation of the client as it retrieves the partial media from the multiple peers. The PeerStreaming client is built on top of the Microsoft DirectShow framework. It is capable of live P2P media streaming, decoding and rendering, and supports a number of coded media format, such as MPEG 1/2/4 audio/video codec, WMA/WMV, as well as an embedded audio codec developed by the author [10].

2. PEER-TO-PEER NETWORK

The P2P network that facilitates the PeerStreaming system is shown in Figure 1. For a particular streaming session, let the *server* be a node that originates the streaming media. Let the *client* be a node that currently requests the streaming media. Let the *serving peer* be a node that serves the client with a complete or partial copy of the streaming media. In the PeerStreaming system, the server, the client and the serving peers are all end-user nodes connected to the Internet. Because the server is always capable of serving the streaming media, the server node is always a serving peer. The server node may perform media administrative functionalities that cannot be performed by a serving peer, e.g., main-

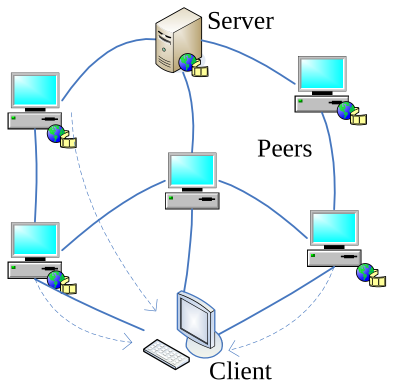

Figure 1 The peer-to-peer media streaming framework.

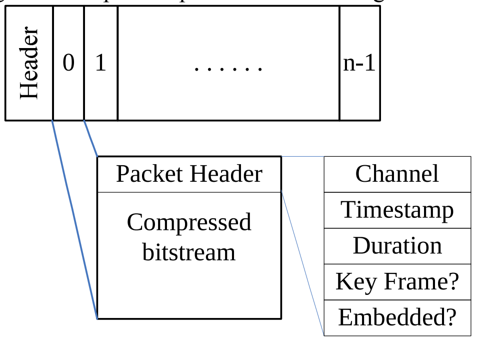

Figure 2 The file format of a streaming media.

taining a list of available serving peers, performing digital right management (DRM) functionality. The role of the node in the P2P network may change as well. A certain node may act as the client in one particular streaming session. In another session, it may act as the serving peer, and serve the received media of the last session to the other client.

During the streaming session, the client locates a number of close-by peers that hold the requested media, and streams the media from the multiple peers (which may include the server). Apparently, the serving peer is doing a favor for the client and the server. By using its upload bandwidth and storage, the serving peer reduces the burden of the server. The client may also receive much better media quality as there are more serving bandwidths available. The peer does not directly benefit from serving the client. Nevertheless, if the P2P network has certain fairness mechanism, e.g., [9], it may expect better media quality next time it becomes the client. PeerStreaming can be considered as an add-on component to an existing media player component. When only a few PeerStreaming nodes are deployed, the benefit may not be much. However, with more and more PeerStreaming nodes deployed, every PeerStreaming node as well as the media server benefit, because the media server will become less costly to run, and the PeerStreaming node will be able to receive much better media quality during the streaming session.

Recognizing the fact that the serving peer is performing a favor for the client and the server during the streaming session, a good design philosophy is to ensure that the serving peer is lightweight and the P2P network is loosely coupled. The serving peer should only need to perform very simple operations with low CPU load. It may also elect to cache only part of the media to save its storage space used for caching the media. It should not be required to collaborate with other peers. Other programs running on the serving peer may also have a higher priority in claiming the CPU and network resource. As a result, the serving peer may be unreliable, with fluctuation of serving bandwidth and may drop offline and become online anytime during the streaming session.

On the contrary, it is fair to let the PeerStreaming client devote resources in the streaming session. The client needs to receive the streaming media from multiple peers, so it is connected to the peers already. It is motivated to do a good job to coordinate the peers, which can improve its own streaming experience. We therefore design the PeerStreaming with a receiver-driven solution, with light weight serving peer and loosely coupled P2P network.

The rest of the paper is organized as follows. In section 3, we examine the media model. We also introduce the high rate erasure resilient code that let the serving peer hold partial media without conflict. The operation of PeerStreaming is discussed in Section 4. Experimental results are shown in 5.

3. MEDIA MODEL, PARTIAL CACHING AND ERASURE RESILIENT CODE

3.1 Streaming Media Model

A streaming media consists of a stream of packets that are decoded and rendered as they arrive (hence the name streaming). Without streaming, the entire media has to be downloaded in one big chunk before it can be used. The general structure of a streaming media file is illustrated in Figure 2. The media is led by a header, which contains global information of the media, e.g., the number of channels in the media, the property and characteristic (audio sampling rate, video resolution/frame rate) of each channel, codecs used, author/copyright holder of the media, etc. The media header is usually downloaded before the start of the streaming session, so that the client may set up the necessary tools to decode and render the following packets. A streaming media may consist of several channels, each of which is a media component that can be independently selected and decoded, e.g., an English audio track, a Spanish audio track, a 4:3 video, a 16:9 video. The header is followed by a sequence of media packets, each of which contains the compressed bitstream of a certain channel spanning across a short time period. Each media packet is led by a packet header, which contains information such as the channel index, the beginning timestamp of the packet, the duration of the packet, as well as a number of flags, e.g., whether the packet is a key frame (a MPEG I frame), whether the packet is an embedded coded packet (with truncatable bitstream), etc.. The compressed bitstream of the packet then follows.

In PeerStreaming system, a media packet can be embedded coded (scalable) or non-embedded coded (non-scalable). Most of the compressed media codecs today, such as MPEG1/2/4 audio/video, WMA/WMV, Real Audio/Video, generate non-embedded coded media packets. The size of these media packet can not be changed. Moreover, the lost of one media packet in such bitstream either causes the media to be not decodable, or incurs significant penalty to the playback quality. In addition to the support of this form of the traditional compressed media, PeerStreaming supports the embedded coded media. With the embedded coded media, each media packet is encoded in such a way that it can be independently truncated afterwards. This is generally achieved by coding a block of audio/video transform coefficients bitplane-by-bitplane, from the most significant bitplane (MSB) to the least significant bitplane (LSB). If the bitstream is truncated after encoding, the information is retained for the several most significant bitplanes of all the coefficients. Moreover, the truncated bitstream corresponds to a lower bitrate compressed bitstream, which can be considered as embedded in the higher bitrate compressed bitstream, hence the name embedded coding. As a result, the media packet generated by the embedded coder can be truncated, with graceful rate-distortion trade-off[1]. In PeerStreaming, we support the embedded audio codec of [10].

3.2 Media Structure

To operate in a receiver-driven mode, the PeerStreaming client needs the structure of the to-be-requested media packets, so that it may know what packets and what portion of each packet to request from each peer. The media structure also provides the PeerStreaming client with a bird's eye view of the entire media, so that it can plan the P2P streaming intelligently, and make sure that the media packets are arrived in time for decoding and rendering.

The media structure of a set of packets is simply the packet headers plus the packet bitstream lengths. Before the media packet can be retrieved in a receiver-driven P2P fashion, its media structure has to be retrieved first. The media structure is pretty compact. On five test movie clips of 31-49 megabytes (MB), the media structures of the entire clips range from 37KB-53KB. Therefore, the media structure is typically 0.10-0.15% of the media body. The retrieval of the media structure does not cost additional bandwidth. It just shuffles the information, and requires certain information of the media packets to be retrieved in advance. In current PeerStreaming, the media structure of the entire media is retrieved in the streaming setup stage. This causes an additional small delay in the startup of streaming[2]. An alternative implementation is to generate a media structure for each media segment (say 10 seconds), and only retrieve the media structure before the corresponding media segment is to be streamed in the near future.

3.3 Data Units

PeerStreaming breaks the media packet, the media header and the media structure into fixed size data units of length L. The reason of using fixed size data units is that the PeerStreaming client and the serving peer may pre-allocate memory block of size L, thus avoid the costly memory allocation operation during the streaming process. Splitting the media packets (potentially very large) into small fixed size data units also allows the PeerStreaming client to distribute the serving load to the peers with small granularity, thus achieves better load balancing. A length P packet (can be the media packet, the media header and the media structure) is split into $\lceil P/L \rceil$ data units, where $\lceil x \rceil$ is the ceiling function that returns the smallest integer that is larger than or equal to x. All data units are of fixed length L, except the last data unit of the packet, which is of length P mod L.

The non-embedded coded media packet generates data units that cannot be dropped during the network transmission. They are designated as the essential data units. On the other hand, when an embedded coded media packet is split into data units, only the base layer data unit must be delivered, the rest data units may be optionally dropped if the serving bandwidths are insufficient. We designate such optional data units as the non-essential data units.

[1] There is an additional form of embedded coded media – layer coded media. In the layer coded media, the media content is compressed into a base layer and multiple enhancement layers, each of which occupies a separate channel. Due to space limitation, we will not further discuss the layer coded media here.

[2] With the exception of the full length movie, the delay is pretty small. Assuming that the serving bandwidths are greater than or equal to the media bitrate, and the media structure is 0.15% of the media body, downloading the media structure of a 10 minute clip causes an additional delay of less than 0.9s.

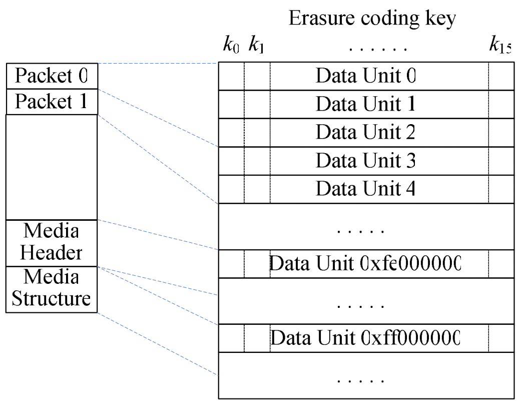

Figure 3 The data units of PeerStreaming.

Let an embedded coded media packet last T seconds. Assuming the media packet is split into a number of data units. To serve the data unit at layer i, all data units below layer i must be served as well. As a result, the serving bandwidth required to serve the data unit at layer i is:

$$R_i = (i+1)L/T, \qquad (1)$$

We call (1) the bitrate of the data unit. In PeerStreaming, the client adjusts to changing serving bandwidths by dropping non-essential data units with bitrate above the serving bandwidths.

All data units of a particular media, including the data units of the media packet, the media header and the media structure, are mapped into a unique ID space. We index the data units of the media packets from 0x00000000 to 0xfdffffff, the data units of the media header from 0xfe000000-0xfeffffff, and the data units of the media structure from 0xff000000-0xffffffff. The data units of PeerStreaming are illustrated in Figure 3. To obtain the data unit IDs of the media header and the media structure, we need the lengths of the media header and the media structure, which can be considered as their mega-structure. To obtain the data unit IDs of the media packet, we need the lengths of the media packet bitstream, which is included in the media structure.

3.4 Partial Caching of Media

An effective way to decrease the amount of storage resource required by the serving peer is to allow it to hold only a portion of the media. Note that for serving purpose, the serving peer only needs to hold the portion of the media in proportional to its serving bandwidth, which may be substantially less than its download bandwidth that dictates the highest streaming bitrate that the node may receive. The end-user node on the Internet tends to have an imbalance between its upload bandwidth and its download bandwidth. For the node on the ADSL/cable modem network, it is not uncommon for the download bandwidth to be an order of magnitude higher than its upload bandwidth. Even for the node on the campus/corporate network, the node may cap its serving bandwidth so that its participation in the P2P activity may not affect other mission-critical functions. Thus, holding portion of streaming media may not interfere with its serving functionality.

Let the bitrate of the non-embedded coded media be R, let the maximum serving bandwidth provided by the peer in a streaming session be B, the peer node only needs to keep p portion of the streaming media in its cache, where the value p is:

$$p = \max(1.0, B/R). \qquad (2)$$

As an example, let us assume that the media bitrate is twice the serving bandwidth: R=2B. The serving peer only needs to keep half of the streaming media in its storage. The reason is that the peer alone can not serve the client at the full streaming bitrate. It

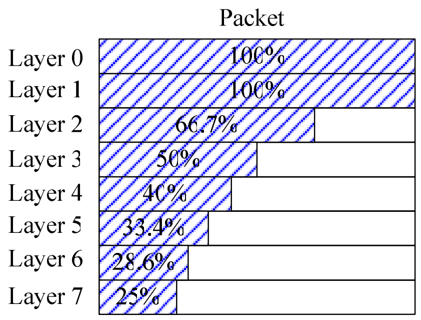

Figure 4 Partial caching of an embedded coded media packet (split into 8 data units, with L/T=0.5B).

can at most supply half the media, and thus only need to keep half in cache. The rest must be supplied by the other serving nodes.

Combing equations (1) and (2), we can determine the amount of media to keep for the embedded coded media as well. Recall from Section 3.3, the media packet of the embedded coded media is split into a number of data units with different bitrate. With R being the bitrate of the data unit at a certain layer, the equation (2) now gives the portion of the media to be kept for that data unit. Shown in Figure 4, an embedded media packet is split into 8 data units. The amount of media that needs to be cached for each data unit (with L/T=0.5B) is shown in Figure 4.

When the storage resource of the serving peer is abundant, the serving peer may elect to cache a larger portion of the media by using a higher potential serving bandwidth B' in equation (2). The extra portion of the media cached enables the media to be served in a choppy, yet high quality fashion. Assuming that all serving peers elect to use a potential serving bandwidth B' twice of its actual serving bandwidth: $B'=2B$, the resultant amount of media in the P2P network will be enough for the client to retrieve the media at half the streaming rate. That is, assuming that the aggregated serving bandwidths of all the available peers are larger than $R/2$, the client should be able to first download half the media, then continuously stream and playback the rest half. It can also elect to download a $T_s/2$ segment of the media (with time T_s), continuously stream another $T_s/2$ segment and playback the segment, then download and stream another segment. The streaming media may thus be play back at rate R, albeit in a choppy fashion.

3.5 High Rate Erasure Coding

How should the peer keep p portion of the media? Or, since the media is ultimately split into the data units, how should we keep p portion of the data unit? A simple strategy is to separate each data unit into k blocks. The peer keeping p portion of the media may hold random $\lceil k \cdot p \rceil$ blocks, with $\lceil x \rceil$ being the ceiling function again. There is a weakness of such scheme. Even if there are much more than k blocks available in the peer cluster, it is possible that the cluster may lack a particular block j, thus renders the entire data unit irretrievable. Besides, the client needs to locate each and every distinct block from the peers, which complicates the design of the protocol between the client and the peers.

To resolve the issues, we introduce the high rate erasure resilient code. An erasure resilient code is a block error correction code with parameter (n, k), where k is the number of original messages, and n is the number of coded messages. High rate erasure resilient code satisfies the property that n is much larger than k, thus the k original messages are expanded into a much larger coded message space of n messages.

As a block error correction code, the operation of the high rate erasure resilient code can be described through a matrix multiplication over the Galois Field GF(p):

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ \vdots \\ c_{n-1} \end{bmatrix} = \mathbf{G} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{k-1} \end{bmatrix}, \qquad (3)$$

where p is the order of the Galois Field, $\{x_0, x_1, \cdots, x_{k-1}\}$ are the original messages, $\{c_0, c_1, \cdots, c_{n-1}\}$ are the coded messages, $\mathbf{G}$ is the generator matrix. We do not use equation (3) to generate all the coded messages at once. Rather, the generator matrix $\mathbf{G}$ defines a coded message space. When the client receives k coded messages $\{c'_0, c'_1, \cdots, c'_{k-1}\}$, they can be represented as:

$$\begin{bmatrix} c'_0 \\ c'_1 \\ \vdots \\ c'_{k-1} \end{bmatrix} = \mathbf{G}_k \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{k-1} \end{bmatrix}, \qquad (4)$$

where $\mathbf{G_k}$ is a sub-generator matrix formed by the k rows of the generator matrix $\mathbf{G}$ that correspond to the coded messages. If the sub-generator matrix $\mathbf{G_k}$ has full rank k, the matrix $\mathbf{G_k}$ can be inversed, and thus the original messages can be decoded.

In PeerStreaming, the high rate erasure resilient code used is a modified Reed-Solomon code [11] on the Galois Field GF(2^{16}). The number of the original messages k is 16. The size of the coded message space n is 2^{16}=65536. Reed-Solomon code is a maximum distance separable (MDS) code. Thus, any 16 rows of the generate matrix $\mathbf{G}$ form a sub-generator matrix with full rank 16, which is equivalent to say that the original messages can be recovered from any 16 coded messages. The Reed-Solomon code of [11] can be encoded and decoded efficiently. It achieves an encoding/decoding throughput of 80Mbps on a 2.2 GHz Pentium computer. This is equivalent to say that the erasure decoding of a 2Mbps media stream only consumes 2.5% of the CPU on that machine.

With a high rate (n, k) erasure resilient code, we may assign each peer node k keys in the coded message space of n, with each key being the row index of the generator matrix $\mathbf{G}$. The key assignment may be carried out by the server. If the number of peers caching the media is smaller than n/k, it is possible to assign each peer with a unique set of keys. As a result, we can guarantee that each peer holds distinctive coded messages. The strategy requires a central coordination node. An alternative strategy that does not need a central coordination node is to let each peer choose k random keys. If the number of peer nodes is greater than n/k or the key is assigned with no central coordination node, certain peer nodes may hold the same keys. Nevertheless, in any PeerStreaming session where the client is connected to m peers, m is usually much smaller than n/k. As a result, the probability that two serving peers happen to hold the same key, and thus one key of one of the peers is not useful, is small. Even if there is conflict, the client can easily identify such conflict when it connects to the peer, and invalidates one of the duplicated keys. The client thus does not need to deal with the key conflict during the streaming process.

With (65536,16) Reed-Solomon code, each data unit is dissected into 16 blocks. Using a set of pre-assigned keys, the peer chooses to cache $\lceil 16p \rceil$ erasure encoded blocks, where p is a parameter calculated from (1) and (2). The keys assigned to the peer as well as its maximum serving bandwidth B constitute the *availability vector* of the peer, as the client can determine how many and what erasure coded blocks are held by the peer using the information. The client resolves the key conflict, if there is any, when the peer is connected. During the streaming session, the client can retrieve any *k* coded messages from any serving peer nodes, and decode the associated data unit.

4. PEERSTREAMING OPERATIONS

We describe the PeerStreaming operations in details in this section.

4.1 Locating Serving Peers

The first task that the PeerStreaming client performs is to obtain the IP addresses and the listening ports of a list of neighbor serving peers that hold a complete or partial copy of the serving media. This list is also updated during the media streaming session. There are in general three approaches that this list can be obtained: 1) from the media server, 2) from one known serving peer, 3) using a distributed hash table (DHT) approach.

Currently, we assume that each serving peer keeps a list of the servers and the peers that hold the streaming media, and the PeerStreaming client is able to connect to at least one server/peer that holds such list. In the future, we plan to use a DHT approach, such as the Microsoft Peer-to-Peer SDK, which may retrieve the initial peer lists with neither the media server nor a known serving peer online.

4.2 Decoding/Rendering Setup

After securing the serving peer list, the PeerStreaming client attempts to connect to each of the serving peer. Once connected, the client retrieves the availability vector of the peer, and resolves the key conflict, if there is any. Then, the client retrieves the lengths of the media header and the media structure from one of the peers. After both lengths are retrieved, the IDs of the data units of the media header and media structure are constructed. The media header and the media structure can then be retrieved in a P2P fashion as shown in Section 4.6.

Once the media header is retrieved, the client constructs a DirectShow filter graph [13]. The network component of the PeerStreaming client is a DirectShow network source filter, whose output is feed into the proper audio/video decoder DirectX media object (DMO), which is further connected to the appropriate audio/video rendering device. A sample DirectShow filter graph of a PeerStreaming session is shown in Figure 5. In this example, the streamed media is non-embedded coded. The audio bitstream is compressed by WMA, and the video bitstream is compressed by MPEG-4.

We implement the PeerStreaming client via the DirectShow framework so that it may use the huge library of existing audio/video encoders/decoders developed under DirectShow. With DirectShow, the PeerStreaming system is capable of streaming, decoding and rendering media coded by a variety of codecs, such as MPEG 1/2/4, WMA/WMV, Indeo Video (in fact, any codec that has a DirectShow decoder DMO component). DirectShow also provides additional audio/video processing modules, such as resolution/color space conversion, de-interlacing, so that the decoded audio/video may match the capacity of the audio/video rendering device. DirectShow automatically handles the synchronization of the audio/video track. Shown in Figure 5, the audio renderer has a small clock attached to it. This indicates that the audio stream holds the reference clock of the entire stream. When playing the video, DirectShow makes sure that the system timing clock of the video stream is doing its best to stay near the audio stream. The lip sync is thus achieved. Finally, a DirectShow application is inherently multithreaded. On a multiprocessor PC (or one with Hyper-Threading enabled), the computation load of various components of the PeerStreaming client, e.g., the network component, the audio decoder, the video decoder, and the audio/video rendering engine, can be distributed onto the multiple processors. This greatly speeds up the execution of the client, and allows more complex audio/video decoders to be used.

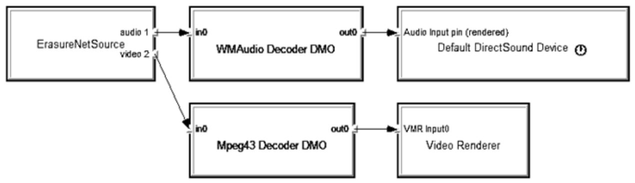

Figure 5 The DirectShow filter graph of the PeerStreaming Client.

4.3 Network Link: the TCP Connection.

Most media streaming clients, such as the windows media player or RealPlayer, use the real time transport protocol (RTP), which is carried on top of UDP. The UDP/RTP protocol is chosen for media streaming applications because: 1) the UDP protocol supports IP multicast, which can be efficient in sending media to a set of nodes on an IP multicast enabled network; 2) the UDP protocol does not have any re-transmission or data-rate management functionality. As a result, the streaming server and client may implement advanced packet delivery functionality, e.g., forward error correction (FEC), to ensure the timely delivery of media packets.

However, in PeerStreaming, we choose TCP connections as the network links between the client and the serving peers. The reason of our choice is as follows. First, IP multicast is not widely deployed in the real world because of issues such as inter-domain routing protocols, ISP business models (charging models), congestion control along the distribution tree and so forth. Second, like many commercial media players, the PeerStreaming client incorporates a streaming media buffer (of 4*s*) to combat the network anomalies such as jitter and congestion. With the presence of the streaming media buffer many times larger than the round trip time (RTT) between the client and the serving peer, we claim that the TCP ARQ (automated repeated request) mechanism is good enough for the delivery of the media packet.

There are three possible mechanisms to deal with the media packet loss: FEC, the selective retransmission, and ARQ (always retransmission). For the Internet channel, which can be considered as an erasure channel with changing characteristics and unknown packet loss ratio, a fixed FEC scheme either wastes bandwidth (with too much protection) or fails to recover the lost packets (with too little protection). It thus does not efficiently utilize the bandwidth resource between the client and the peer. With a streaming buffer many times larger than the RTT, thus plenty of chances for retransmission, retransmission based error protection is preferable over FEC. Our choices are now down to ARQ and the selective retransmission, which will have an edge over ARQ only if many packets are not selected to be retransmitted. For non-embedded coded media, a lost packet usually leads to serious playback degradation. Therefore, the lost packet is almost always retransmitted. With the embedded coded media, a lost packet may not prevent the media from playing back. However, the lost of a random packet still causes a number of derivative packets to be not useable. As a result, only the top most enhancement layer packets may select not to be retransmitted. Compare with the selective retransmission, ARQ always retransmits the packets once they are requested; even they belong to the top most enhancement layer. Nevertheless, the ARQ scheme can choose not to request the top most enhancement layer packets of the following media packets, thus achieve the same bandwidth usage and perceived media playback quality with the selective transmission scheme. Unless the network condition varies very quickly, the ARQ

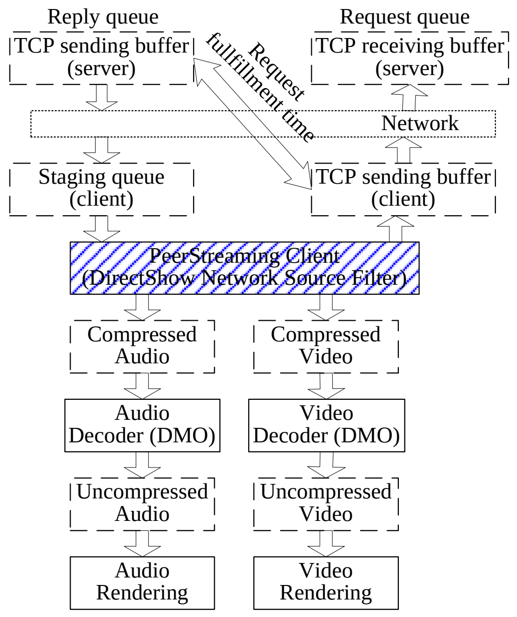

Figure 6 The life of a PeerStreaming request and its reply: the buffers (in dashed box) and the processing modules (in solid box).

mechanism employed by the TCP protocol is sufficient to handle the packet loss in media streaming.

Using TCP as the network links also brings a number of side benefits into the development of PeerStreaming. There is no need to deal explicitly with flow control, throughput estimation, congestion control and avoidance, keep alive, etc.. They are all handled by TCP. TCP also detects the peer going offline, and gracefully handles the shutdown of the connection link.

4.4 Streaming Bitrate Control

Non-embedded coded media is always streamed at the bitrate of the media. However, the streaming bitrate of the embedded coded media may vary during the streaming session. We set a streaming bitrate R_{recv} for each media packet, which is calculated by:

$$R_{raw}= Th\cdot(1+T_{rft}-T_{staging})+B_{staging}-B_{outstanding}, \quad (5)$$

$$R_{filter}=(1-\alpha)R_{filter}+\alpha R_{raw} \quad (6)$$

$$R_{recv}=\min(R_{min}, R_{inst}), \quad (7)$$

where Th is the aggregated serving bandwidths,

$T_{staging}$ is the target staging buffer size (default 2.5*s*),

T_{rft} is the desired request fulfillment time (RFT, default 1.0*s*),

$B_{staging}$ is the length of the received packets in the staging queue,

$B_{outstanding}$ is the length of outstanding replies to be received,

R_{min} is the base layer bitrate (with only essential data units),

α is a low pass control parameter.

The equations (5)-(7) control the streaming bitrate R_{recv} to follow the serving bandwidths Th and the staging and request queue statuses (to be described in Section 4.6). Once the streaming bitrate is determined, the client only issue requests for the data units with bitrate below the streaming bitrate R_{recv}.

A more advanced strategy is to control the bitrate R_{recv} by considering the distortion contribution of the data units as well. However, this requires that the client gains access to the distortion (or the rate-distortion slope) of the data units, which must be included in the media structure and sent to the client. Unlike existing information in the media structure, the distortion of the data units is not needed in decoding and is thus an overhead. It is thus a trade-off between the amount of overhead to be sent to the client versus the rate-control accuracy. In current PeerStreaming system, we use the rate only bitrate control strategy of (5).

4.5 PeerStreaming Requests and Replies

Let us examine the life of a PeerStreaming request and its reply. Shown in Figure 6, the client generates the request and sends it through the outbound TCP connection to a certain serving peer. In network delivery, TCP may bundle the request with prior requests issued to the same peer. If a prior request is lost in transmission, TCP handles the retransmission of the request as well. After the request packet is delivered, it is stored in the TCP receiving buffer of the serving peer. The peer processes the request one at a time. For each request, it reads the requested erasure coded blocks from its disk storage, and sends the requested content back to the client. In case the TCP socket from the serving peer to the client is blocked, i.e., no more bandwidth is available, the serving peer will block until the TCP connection opens up. The time interval between the request is issued by the client till its reply is received by the client is defined as the request fulfillment time (RFT). Because the request is usually much smaller than its reply, and the operations involved in processing the request, e.g., disk read, are trivial compared with the network delivery time used to send the content back, we may calculate the RFT of the request T'_{rft} by:

$$T'_{rft}= (B_{i,outstanding}+B_{cur})/Th_i, \quad (8)$$

where Th_i is the serving bandwidth of peer *i*,

$B_{i,outstanding}$ is the length of unreceived replies before the request,

B_{cur} is the length of the content requested.

RFT is determined by the serving bandwidth of the peer, size of the request and size of the unreceived content from the peer.

Once the content packet arrives at the client, it is immediately moved to a staging queue. In the staging queue, the erasure coded blocks from multiple peers are combined and decoded into the data units, which are further combined into the media packet. Periodically, the PeerStreaming client removes the delivered media packets from the staging queue, and pushes them into the corresponding audio/video decoder DMO. After the media packets are decompressed by the decoder, the uncompressed audio/video data streams are sent to the audio/video rendering unit.

Except the uncompressed audio/video buffers, which are under the control of the DirectShow filter graph and are not programmable, all the rest of the buffers in Figure 6 may be shrunk to combat network anomalies such as the packet loss and jitter. In current implementation, we set the size of the staging buffer to $T_{staging}$=2.5*s*, set the desired request fulfillment time to T_{rft}=1.0*s*, and set the compressed audio/video buffer to 0.5*s*. The total PeerStreaming buffer is thus around 4*s*.

Each request in PeerStreaming is formulated as the request of a group of erasure coded blocks of a certain data unit. The erasure coded block group is identifiable with the start block index and the number of blocks requested. The data unit is identifiable through a 32 bit ID. The request is thus in the form of:

data unit ID [32], *start_index* [4], *number_of_blocks*[4], (9)

where the number in the bracket is the number of bits of each component. Shown in (9), each request is 5 byte long. On the other hand, the content requested measures from 128 to 2048 bytes (data unit length *L*=2048, *k*=16). As a result, the size of the request is about 0.24%-3.91% of the reply. The amount of the upload bandwidth spent by the PeerStreaming client to send the request is thus very small compared to the content requested.

4.6 Request and Staging Queues: Throughput Control, Load Balancing and Request Redirection

In PeerStreaming, the client maintains a staging queue to hold the arrived erasure coded blocks, and to assemble them into the data units and the media packets. It also maintains a request queue for each of the serving peer to hold the unfulfilled request sent to

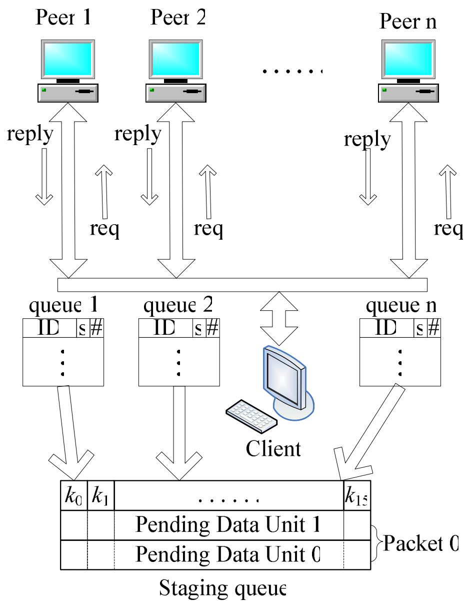

Figure 7 The client maintains a staging queue for the arriving data unit, and one request queue for each serving peer.

the peer. The request and staging queues are shown in Figure 7. The staging queue is the main streaming buffer of the PeerStreaming client. All received contents are first deposited into the staging queue. The request queues serve three purposes: 1) to perform throughput control and load balancing, 2) to identify the reply send back by the serving peer, and 3) to handle the disconnected peer.

The first functionality of the request queue is to balance the load among the serving peers. The request of the data unit is broken into the requests of multiple groups of erasure coded blocks, with each group directed to one peer. The requests are generated through the following operations. Upon requesting a data unit, the client first checks the availability vector of the peers, and calculates the number of erasure coded blocks (a_i) held by each peer for the data unit. If the total number of blocks held by all peers online is less than k, the data unit is irretrievable. If the irretrievable data unit is non-essential, the client simply skips the data unit. If the irretrievable data unit is essential, i.e., belongs to a non-embedded coded media packet or the base layer of an embedded coded media packet, the client cannot proceed. It will wait for more peers to come online to supply the missing blocks[3]. After ensuring that the data unit is retrievable, i.e.,

$$\sum_i a_i \geq k, \tag{10}$$

we check the space available in the request queue of each peer. It is desirable to maintain the RFT of each peer to be around a system constant T_{rft}, say 1.0s. Too short request queue may not effectively combat the network anomalies from the client to the peer. In case that the request packet is lost or delayed, the serving peer may be left with nothing to send, which wastes its serving bandwidth. Too long request queue may prevent the client from quickly adapting to the changes, e.g., the disconnection of a peer.

[3] An alternative strategy is to skip the entire media packet, and marked it as missing to the following audio/video decoder DMO. Nevertheless, if one essential data unit is irretrievable from the peer cluster, it is very likely that more following essential data units will be irretrievable too. Therefore, it is better to let the client wait.

With the request queues to all peers being the same length in RFT, the capacity of the request queue becomes proportional to its serving bandwidth: $Th_i \cdot T_{rft}$. As an example, with T_{rft} be 1.0s, a peer with serving bandwidth 16$kbps$ allows 2KB of unfulfilled request pending in its request queue, while a peer with serving bandwidth 1$Mbps$ allows 128KB of unfulfilled request pending. The number of erasure coded blocks that can be requested from a particular peer is thus capped by the space left in its request queue:

$$e_i = \min(\ a_i\ ,\ (Th_i \cdot T_{rft} - B_{i,outstanding})/\mathrm{bk}\ ), \tag{11}$$

where e_i is the number of erasure coded blocks that can be requested from the peer i, and bk is the size of the erasure coded block. The equation (11) guarantees that the client never sends out a request that has an expected RFT greater than T_{rft}. If the client cannot find enough current available erasure coded blocks, i.e.,

$$\sum_i e_i < k, \tag{12}$$

it will wait until the request queue of the serving peer clears up. Only when $\sum_i e_i \geq k$, the data unit requests are formed and sent to the peers. The actual number of blocks (b_i) requested from a certain peer is calculated by:

$$\begin{cases} \sum_i b_i = k, \\ b_i = \min(e_i, c \cdot Th_i), \end{cases} \tag{13}$$

where c is a constant that satisfies $\sum_i b_i = k$. In general, the procedure outlined above allocates the serving load to each peer in proportional to its serving bandwidth Th_i (equation (13)). It also makes sure that the client does not request more blocks than what are possessed by the serving peer, and the RFT of the request does not exceed T_{rft} (equation (11)).

The second functionality of the request queue is to identify the content send back by the serving peer. The PeerStreaming client and the peer communicate through TCP, which preserves the order of data transmission, and guarantees packet delivery. Furthermore, the peer processes incoming requests in sequence. As a result, there is no need to identify the content sent back. It must be for the first request pending in the request queue.

Finally, the request queue is also used to redirect the requests of the disconnected peers. Whenever a certain serving peer is disconnected from the client, the disconnection event is picked up by TCP and is reported to the client. The client reassigns all unfulfilled requests pending in the queue of the disconnected peer to the other peers. The procedure for reassigning the request is very similar to the procedure of assigning the request in the first place. The only exception is that the number of blocks already requested from the peer must be considered in the request reassignment.

Whenever the erasure coded blocks arrive at the client, they are immediately pulled away from the TCP socket. After pairing the arriving content with the pending request, the fulfilled request is removed from the request queue. The identified erasure coded blocks are deposited into the staging queue. The size of the staging queue increases as a result. If the staging queue reaches a predetermined size $T_{staging}$, no further requests of the media packets/data units are sent. Once all erasure coded blocks of a certain data unit has been received, the data unit is erasure decoded, and is marked as ready. A media packet becomes ready if all its requested data units are ready. Periodically, the audio/video decoder removes the "ready" media packet from the staging queue. This reduces the size of the staging queue, and may trigger the generation of new media packet requests.

4.7 PeerStreaming Operation: Complete Overview

As a summary, the PeerStreaming operation is as follows. First, the client gets a list of nearby peers that hold the requested streaming media. It connects to each serving peer, obtains its availability vector. It then retrieves the lengths of the media header and the media structure from one of the peers. After both lengths are retrieved, the client calculates the data unit IDs of the media header and media structure, and retrieve them from the peer cluster. Once the media header arrives, the client analyzes the media header, and setups the audio/video decoder DMOs and the rendering devices in the form of a DirectShow filter graph. It then proceeds to the on going stage of media streaming. Using the media structure, the data unit IDs of the media packets are calculated, and the media packets are retrieved one by one. For the embedded coded media packets, the streaming bitrate is dynamically adjusted based on the available serving bandwidths and the status of the client queues. Periodically, the client updates the serving peer list, and connects to potential new serving peers. Currently, this is achieved by issuing a TCP connect() function call every 4*s* for each potential serving peer. After the client establishes the connection to a new serving peer, it first retrieves the availability vector. The new peer may then join the other active peers. The client coordinates the peers, balances the serving load of the peers according to their serving bandwidths and content availability, and redirects unfulfilled requests of the disconnected peer to the other active peers. The received erasure coded blocks are deposited into the staging queue of the client, where the media packet is assembled. The completely assembled media packets are sent downstream to the audio/video decoder DMOs for decoding and rendering. By controlling the length of the staging queue, the request queue, and the compressed audio/video buffer, the client maintains a streaming buffer of around 4*s*. The combined buffer is used to combat network packet loss and jitter. The streaming operation continues until the entire streaming media is received, or the streaming operation is stopped by the user.

5. EXPERIMENTAL RESULTS

A working PeerStreaming system has been built. The PeerStreaming system currently supports a number of media format, such as MPEG1/2/4, WMA/WMV, and the embedded audio coder of [10]. It is currently capable of live P2P media streaming, decoding and rendering in real-time.

In the first experiment, we test the capability of the PeerStreaming system in streaming non-embedded coded media. The test clips are five movie trailers compressed at 2Mbps. The sizes of the movie clips range from 31-49MB. They last around 2-3 minutes. We set up a PeerStreaming cluster, which consists of one client and seven serving peers. One serving peer is behind an ADSL link of 128kbps upload bandwidth. The rest peers are on the Internet, with an upload cap that limits its serving bandwidth to 16kbps – 1Mbps. The PeerStreaming client then retrieves the media from the PeerStreaming serving cluster. During the streaming session, we randomly shut down and restart the serving peers and observe the behavior of the PeerStreaming client.

The PeerStreaming client is able to automatically detect the peers that drop offline and the peers that come online. We observe that as long as the aggregated serving bandwidths provided by the serving peers exceed 2Mbps, which is the media bitrate of the test clip, the clip plays smoothly. Whenever the serving bandwidths drop below 2Mbps, the play of the movie clip becomes choppy, though the client still strives to play the clip. With insufficient serving bandwidths, the PeerStreaming client is unable to feed the media packets faster enough to the audio/video decoder DMOs. As a result, the audio/video rendering devices do not get uncompressed audio/video data stream in a timely fashion. This leads to the choppy playback. We notice that the choppy audio is much more noticeable and annoying than the choppy video. As soon as the new peers are online, and the serving bandwidths exceed 2Mbps, in a short time the movie playback becomes smooth again.

In the second experiment, we stream an embedded coded audio via the PeerStreaming solution. The audio, compressed via an embedded audio codec of [10], is compressed all the way to lossless. Every media packet of the audio bitstream is independently truncatable at any point during the storage and transmission stage. During the streaming session, the PeerStreaming client selects the streaming bitrate depending on the serving bandwidths and the queue status. We observe that the audio playback is always smooth, no matter what the serving bandwidths are. It is the audio playback quality that changes with the serving bandwidths. When coded to lossless, the compression bitrate of the audio is 700kbps. Yet it may be streamed with serving bandwidths as low as 16kbps. When the serving bandwidths vary between 16 to 128 kbps, there is a noticeable change in audio playback quality with the increase of the serving bandwidths. After the serving bandwidths are above 128kbps, the change in the perceived audio playback quality becomes minimum.

6. REFERENCES

[1] Nua Internet Surveys, http://www.nua.com/surveys/index.cgi

[2] Y. Chu, A. Ganjam, T. S. E. Ng, S. G. Rao, K. Sripanidkulchai, J. Zhan, and H. Zhang, "Early Experience with an Internet Broadcast System Based on Overlay Multicast", *Technical Report CMU-CS-03-214, Carnegie Mellon University, December, 2003.*

[3] H. Deshpande, M. Bawa and H. Garcia-Molina, “Streaming Live Media over a Peer-to-Peer Network”, *Stanford database group technical report (2001-20), Aug. 2001.*

[4] V. Padmanabhan and K. Sripanidkulchai, “The Case for Cooperative Networking”, In *Proc. of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, USA, March 2002.

[5] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron and A. Singh, "SplitStream: High-bandwidth content distribution in a cooperative environment", In *Proc. of the International Workshop on Peer-to-Peer Systems*, Berkeley, CA, February, 2003.

[6] X. Jiang, Y. Dong, D. Xu, B. Bhargava, "GnuStream: a P2P Media streaming system prototype", In *Proc of IEEE Intern. Conf. on Multimedia and Expo(ICME 2003)*, Baltimore,MD, June 2003.

[7] M. Hefeeda, A. Habib, B. Botev, D. Xu, B. Bhargave, “PROMISE: peer-to-peer media streaming using CollectCast”, in *ACM Multimedia 2003*, Berkeley, CA, 2003, pp.45-54.

[8] Y. Cui, B. Li, K. Nahrstedt, “oStream: asynchronous streaming multicast in application-layer overlay networks,” *IEEE Journal of Selected Areas in Comm.*, vol. 22, no. 1, pp. 91-106, Jan. 2004.

[9] B. Gelfand, A.-H. Esfahanian, and M. Mutka, “An agent-based approach to enforcing fairness in peer-to-peer distributed file systems”, In *Proc. 9th Int. Conf. on Parallel and Distributed Systems*, pp. 157-162, Dec. 2002, Taiwan, China.

[10] J. Li, "Embedded audio coding (EAC) with implicit psychoacoustic masking", *ACM Multimedia 2002*, Nice, France, Dec.1-6, 2001.

[11] J. Li, “The efficient implementation of a high rate Reed-Solomon erasure resilient code”, in preparation for ICASSP’05.

[12] Windows XP Peer-to-Peer Software Development Kit.

[13] DirectX 9.0 Complete Software Development Kit (SDK).