

PeerStreaming: design and implementation of an on-demand distributed streaming system with digital rights management capabilities

Jin Li · Yi Cui · Bin Chang

© Springer-Verlag 2006

Abstract We have developed *PeerStreaming*, an on-demand peer-to-peer (P2P) media streaming system. The behavior of a *PeerStreaming* client is very much like an ordinary media player: it streams and plays whatever media the user desires, starting at whatever point he/she desires. Under the hood, however, the media is delivered from multiple peers to the client. The serving peers can be the media server, but more commonly they are the clients that have received the media in a previous streaming session. Leveraging the network and storage resources of the P2P network and retrieving a major part of the media from nearby peers, *PeerStreaming* greatly improves streaming media quality, reduces server load, and eases traffic on the network. We describe the design and implementation of the *PeerStreaming* system in this paper, with a focus on a number of key components. They include: (1) peer discovery and content location, (2) scalable coded and randomly accessible media format, (3) receiver-driven streaming, (4) digital rights management, and (5) architecture for the media rendering, caching and serving system.

Keywords Media streaming · Peer-to-peer (P2P) · Peer discovery · Scalable coded media · Non-sequential

J. Li (✉)
Communication and Collaboration Systems,
Microsoft Research, Redmond, WA, USA
e-mail: jinl@microsoft.com

Y. Cui · B. Chang
Department of Electrical Engineering and Computer Science,
Vanderbilt University, Nashville, TN, USA
e-mail: yi.cui@vanderbilt.edu

B. Chang
e-mail: bin.chang@vanderbilt.edu

media access · Digital rights management (DRM) · Receiver-driven · On-demand streaming · P2P media streaming system

1 Introduction

According to the market research, more than half of the Internet users in the USA have accessed some form of streaming media in 2004. While streaming music is still the most popular activity of the users, the popularity of streaming video is growing rapidly. Compared to most web pages, a streaming media file is huge. An average web page is 60 KB (according to the 3rd State of the Web (SOWS II) survey conducted in 1999), while a 3-min movie trailer encoded at 2 megabits per second (Mbps) results in a 45 megabyte (MB) media file. Streaming media also carries a stringent demand in the timing of packet delivery. The large size of the streaming media as well as the delivery timing requirement causes a streaming media server to be expensive to set up and run. The current wholesale rate for the network bandwidth is about \$300 per Mbps per month, or \$4,500 per annum for a leased T1 line (1.5 Mbps) and \$160,000 per annum for a leased T3 line (45 Mbps). The network bandwidth cost is a significant component of running a media server. The high bandwidth cost causes the media server today to serve mostly low bitrate, poor quality media.

The emergence of peer-to-peer (P2P) networks revolutionize the way that the content is distributed. Popular P2P systems, such as KaZaA and BitTorrent, have attracted millions of users, and have efficiently distributed huge files, such as the Linux OS image. In such P2P networks, a peer node contributes resources, namely upload bandwidth and disk storage, in exchange for

services from the other peers to improve its own content retrieval experience. Despite various hurdles, such as the presence of leech nodes, copyright issues, virus and security concerns, the P2P networks and systems flourish because the users benefit from being able to retrieve content faster and cheaper than directly retrieving the file from the server.

In this paper, we have designed *PeerStreaming*, a system for on-demand P2P media streaming. Certain *PeerStreaming* components are designed by leveraging experience gained in prior P2P file sharing system design, e.g., the technology for the peer discovery and content location. Nevertheless, there are unique challenges that will lead to the development of new technologies in extending the P2P file sharing to P2P media streaming.

The first challenge is the streaming P2P content distribution mode. Most of the current P2P systems operate in the “file downloading” mode. The file can only be used after the entire file has been downloaded. Such “downloading” mode offers poor usability, especially because the media is so huge. In *PeerStreaming*, we develop a more preferred “streaming mode” for content distribution. Specifically, a receiver-driven streaming mode is employed, which is significantly different from the sender-driven streaming mode in the existing server–client media streaming solutions. We choose a receiver-driven streaming mode because it enables simple and light load serving peer design. The *PeerStreaming* peers need no collaboration among themselves. It is the receiver (the media playing client) that coordinates the streaming process with its best effort, retrieves the media from multiple peers, performs load balancing, handles the joining/leaving of the supplying peers, and decodes and renders the media, all in real time.

The second challenge lies in the special characteristics of the media. Unlike file distribution, where even a single bit of difference results in intolerable loss, distortion is tolerable in media compression and delivery. This is especially true for scalable media compression, where subsets of the master compressed bitstream may be extracted to form particular application bitstreams, which may exhibit a variety of compression ratios. When available, using the scalable coded media may significantly improve the capability of the *PeerStreaming* nodes to tolerate varying network bandwidth and conditions between the peers.

The third challenge is on-demand streaming of the media. Unlike watching TV, the user is accustomed to accessing the content in a non-sequential order on the Internet. The user can enjoy the capability to fast forward or rewind the media in the same way they operate their DVD or VCR player. In *PeerStreaming*, we have designed a new file structure that enables segments of

media to be read, written, accessed, and streamed in a non-sequential order.

The fourth challenge involves the content piracy issue. In many P2P networks, the content is distributed without proper authorization and compensation to the content owner, a problem that remains the major obstacle of deploying the P2P content distribution system. In this paper, we introduce a digital rights management (DRM) manager module that can be optionally invoked by the content owner. With the DRM protection turned on, the media is encrypted before it is distributed in the *PeerStreaming* network, and is decrypted prior to its use (play back). With the DRM protection, *PeerStreaming* achieves efficient content distribution without relinquishing control of the content. While a peer may still efficiently distribute and serve without seeking authorization from the owner, the client node must do so in order to view the content.

The rest of the paper is organized as follows. In Sect. 2, we present the overall architecture of *PeerStreaming* and identify its key components. In Sect. 3, we discuss in greater details the design of these components, namely, content lookup service, media format for storage and streaming, receiver-driven streaming protocol, and DRM manager. In Sect. 4, we introduce the system implementation of *PeerStreaming* and describe experimental results. In Sect. 5, we perform simulation study to evaluate the scalability of *PeerStreaming* in the large-scale Internet deployment. Section 6 reviews the related works. Finally, we conclude at Sect. 7.

2 PeerStreaming architecture

The P2P network that facilitates the *PeerStreaming* system is shown in Fig. 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 cached 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 client, we can consider it as a special serving peer that has a complete copy of the media, and may perform additional media administrative functionalities, e.g., maintaining a list of available serving peers, performing 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/instance to the other client.

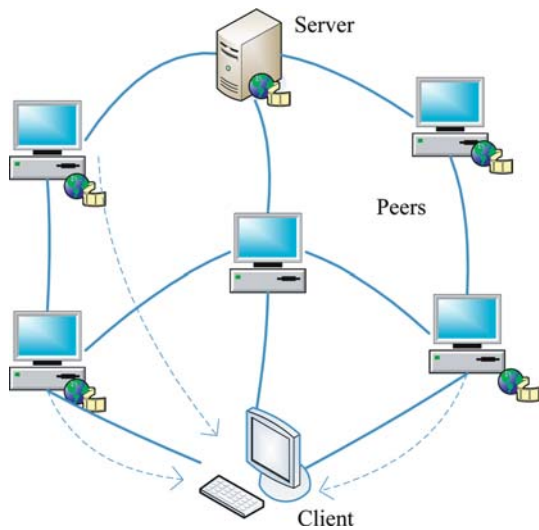


Fig. 1 The *PeerStreaming* framework

During the streaming session, the client locates a number of close-by peers that hold the requested media, and streams the media from the peers. By retrieving a majority part of the media from nearby peers, the *PeerStreaming* system reduces the burden of the server, eases the traffic on the network, and improves the streaming media quality. In a particular streaming session, the serving peers contribute network bandwidth and storage resources, and the client reaps the benefit. The peers do not directly benefit from serving the client. Nevertheless, if the P2P network has certain fairness mechanism built in, e.g., [9], it may expect better media quality next time it becomes the client and receives streaming media.

PeerStreaming incorporates a collection of innovative components as follows:

- *Content lookup service*: Also referred to as peer discovery service. Whenever a *PeerStreaming* client joins the network and requests a particular media, an important bootstrap step is to find out what serving peers are available to serve the interested media.
- *PeerStreaming media format*: The new format defined here extends traditional media format by allowing the user to watch the media in a non-sequential manner and writing the arriving non-sequential media packets into the file. Moreover, it also supports the serving peer to respond to the on-demand request by quickly locating and reading the media packets accessed.
- *DRM manager*: This entity grants *PeerStreaming* an efficient way for the content owner to distribute the media content without relinquishing control of the content. Since the content is encrypted by its owner,

a peer is free to distribute the media, but needs owner authorization to play it.

- *Receiver-driven streaming protocol*: *PeerStreaming* adopts a receiver-driven streaming mode, where the client coordinates the streaming process with its best effort, retrieves the media from multiple peers, performs load balancing, and handles the joining/leaving of the peers.

In the following section, we will describe the detailed design of the aforementioned components in *PeerStreaming* system.

3 PeerStreaming components

3.1 Content lookup, peer registration & discovery

In P2P systems, the peer discovery process is achieved in one of the two alternative ways: supernode assisted approach and/or the distributed hash table (DHT) approach; both of which have been implemented by *PeerStreaming*.

3.1.1 Supernode assisted approach

In the supernode assisted lookup process, one or a cluster of supernodes is put in charge of directing the newly arrived client node to the proper peers. The supernode may be a specific node setup for the entire *PeerStreaming* network. Alternatively, it can coincide with the media server. In such a case, the supernode is in charge of all media originated from the same server.

A content lookup table is established in each supernode, as shown in Fig. 2. Each media occupies one individual entry in the content lookup table. The media is identified by a unique ID, e.g., the SHA1 hash of the media, and a description, which allows the client to submit a text query to the supernode to find the media of interest. We build the content lookup table as a hash lookup table, so that entry lookup by media ID is especially fast, usually only takes a single lookup operation.

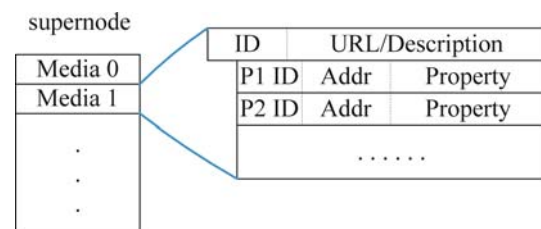


Fig. 2 Content lookup table in supernode/server

Each media entry further consists of a list of peer nodes, with each entry of the peer list containing the IP address of the peer and additional properties, such as the last registration time and a virtual landmark vector. By default, each *PeerStreaming* node sends periodic (every 15 s) registration packets to the control node. By examining the last time that the registration packet is received, the controller and the client can quickly weed out stale peer node. The virtual landmark vector consists of the round trip time (RTT) of the peer node to 7–9 popular landmarks [23]. By comparing the virtual landmark vector of the peer node versus that of the incoming client, the controller node is capable of directing the incoming client to a set of nearby peers, thereby reducing the potential traffic jam in the network.

The peer node in the list may have finished streaming the media in a previous session, and thus hold a complete copy of the media in its cache. Alternatively, it may just join in the streaming session and has nothing to serve yet. It is the responsibility of the client to find out how much portion of the media is held by the serving peer. This is achieved by retrieving availability vectors (described in Sect. 3.2.2) from the serving peer.

Note that in the supernode assisted lookup approach, we choose not to store the availability vector of each peer in the supernode. This is due to the service bandwidth and storage concern of the supernode and the time varying nature of the availability vector. If the peer holds a part of the media, the availability vector that describes what the peer exactly holds can be pretty large. Putting the entire availability vector in the supernode consumes storage space and service bandwidth to and from the supernode. Furthermore, for the node that is still receiving streaming media content, the availability vector may continuously change during the streaming session, rendering the availability vector stored at the supernode frequently obsolete. Therefore, a far more efficient approach which does not need constant update is to let the client retrieve the availability vector directly from the serving peer.

3.1.2 The DHT lookup

Another approach to accomplish the peer discovery process is via the distributed hash table (DHT). In this approach, all participating *PeerStreaming* nodes form a DHT. All information related to the media and the list of the peers that hold the media are stored in the DHT, and a newly arrived client discovers media serving peer by retrieving the information from the DHT. In DHT, all nodes perform the same action, and there is no special supernode. The task of peer registration and discovery is performed collectively.

There already have been several well-developed DHT solutions. In this work, we choose PeerNet [25], a DHT software development kit (SDK) that is based on the Pastry [21] algorithm. The PeerNet also supports advanced network functionalities, such as NAT traversal.

The basic service provided by PeerNet is the peer name resolution protocol (PNRP), a P2P routing protocol using the algorithm of Pastry [21]. In PNRP, every entity, whether a user, a machine or a content file, is represented by a unique PNRP ID. All nodes that are running the PeerNet and are connected to the Internet form a P2P network. Each node in this network manages a cache of PNRP IDs. Each entry in the cache contains the PNRP ID of an entity and the IP address of the entity. If the entity is a node, the IP address is just one of the node. If the ID is a user or a content file, the IP addresses are those of the node that holds the user or the content file. The entire set of PNRP IDs located on all nodes in the P2P network forms a DHT. If a requesting node, say *A*, wants to locate an entity *B* based on its PNRP ID, it first checks if the entry of *B* already exists in its local cache. If not, *A* finds from its cache a node, say *C*, whose PNRP ID is numerically closest to the ID of *B*, i.e., sharing the longest prefix with the ID of *B*. *A* then forwards the request to *C*. *C* resolves the request if the entry of *B* presents in its local cache. Otherwise, it finds another node *D*, whose ID is numerically even closer to the ID of *B*, and forwards the request. This procedure is repeated until an entry exactly matching the ID of *B* is found.

With PNRP as the basic routing service, PeerNet also provides the graphing service to group peer nodes into a graph. Essentially, graphing provides an application-layer multicast graph, where nodes within the graph can propagate data to each other. To create a graph, the creator node uses PNRP to construct a graph ID based on the graph name, and then registers itself with the graph ID via PNRP. Other nodes, in order to join this graph, also use PNRP to resolve the graph name, and then connect to the creator node. After connecting to the graph, the new peer nodes create connections to additional neighbors within the graph.

With the PNRP and graphing supports from PeerNet, we implement the DHT-based peer discovery service as follows. The media server creates a graph for each streaming media file. The graph name is the hash of the media file, e.g., via SHA1 hash function. The graph is then registered with the PNRP. Any peer that has received the streaming media becomes a member of the graph, and registers its own record, such as its IP address, listening port, and available upload bandwidth, with the PeerNet. A *PeerStreaming* client that

looks for serving peers may query the PeerNet graph to return the records of available serving peers in the graph.

Note that in the DHT-based approach, we also choose not to include the availability vector as part of the peer record circulated in the graph. The reason, as stated in Sect. 3.1.1, is to avoid unnecessary communication to keep the peer record up-to-date.

3.2 PeerStreaming media format

The traditional structure of a streaming media file, be it MPEG, AVI, Windows Media or Real, can be illustrated in Fig. 3. 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 of each channel, codecs used, author/copyright holder of the media. 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). The compressed bitstream of the packet is then attached after the packet header.

In *PeerStreaming*, we further extend the previous traditional media format. Our extension lies primarily in three areas: (1) to support scalable compressed media, (2) to support non-sequential, on-demand media reading and writing, and (3) to support simplified communication between the client and the serving peer.

3.2.1 Scalable coded media format

Most of the compressed media codecs today, such as MPEG1/2/4 audio/video, WMA/WMV, real audio/video,

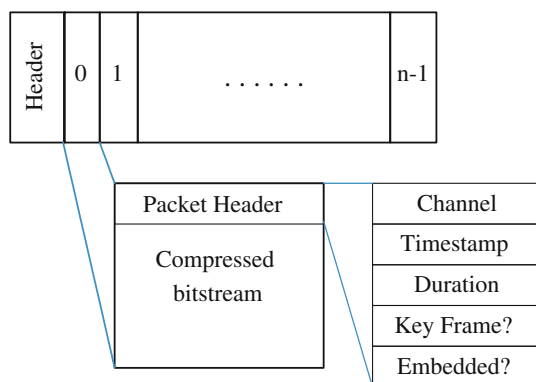


Fig. 3 Traditional file format of a streaming media

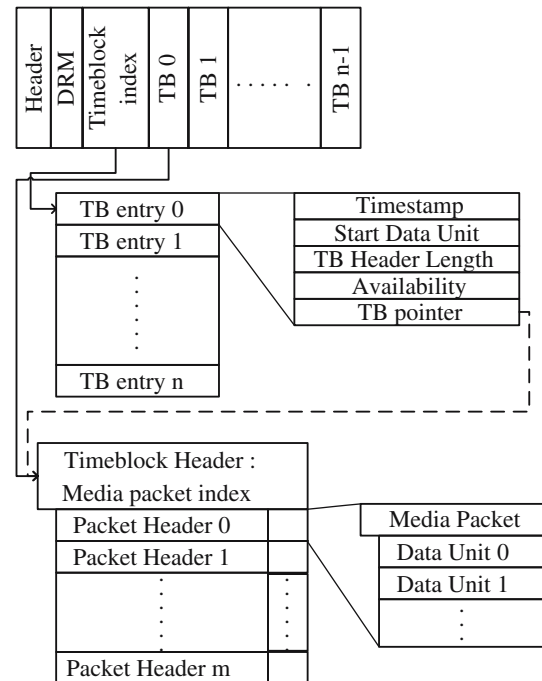


Fig. 4 The scalable coded media: droppable packet mode

generate non-scalable coded media packets. The size of these media packets cannot be changed. Moreover, losing one media packet in such bitstream either causes the media to be not decodable, or incurs significant penalty to the playback quality.¹ In addition to the support of this form of traditional compressed media, *PeerStreaming* also supports the scalable coded media. The scalability in the compressed media can usually be achieved in either or both of the following modes: (1) the droppable packet mode and, (2) the truncatable packet mode.

The media packet of the scalable coded media with droppable packet mode can be illustrated in Fig. 4. In this mode, certain enhancement layer packets of the coded media may be optionally dropped if the resources, e.g., storage capacity or the transmission bandwidth, become scarce. To identify which packet to drop first, we attach a priority value s_i , which is usually the rate-distortion slope at the end of packet coding, to each packet i . When the resource is tight, the packets with low-priority value are dropped first. An example of the scalable coded media in droppable packet mode is the resolution scalable MPEG-2 video bitstream.

An alternative, more flexible scalable coded media mode is the truncatable packet mode. In this mode, each media packet usually comprises a compressed bitstream generated by coding a block of audio/video transform

¹ Packets in non-scalable coded media may not be equal either. Certain packets, e.g., B frame packet in video coding, cause less penalty if lost. Nevertheless, very few media format marks the different packets.

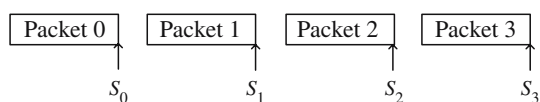


Fig. 5 The scalable coded media: truncatable packet mode

coefficients bitplane-by-bitplane, from the most significant bitplane (MSB) to the least significant bitplane (LSB). The resultant compressed bitstream (the media packet) may be truncated at arbitrary point with graceful rate-distortion tradeoff. With the truncated media packet, the several most significant bitplanes of all the coefficients are still decodable. Moreover, under such compression method, the truncated bitstream corresponds to exactly the compressed bitstream at a lower bitrate. We can consider the lower bitrate bitstream as embedded in the higher bitrate compressed bitstream. Hence, such scalable media codec is also called the embedded media codec. To identify the importance of the various segments of the media packet, we again assign a priority value $s_{i,j}$ to certain selected rate point $r_{i,j}$, as shown in Fig. 5, where we use j to index the rate point within each media packet i . Similar to the drop-pable packet mode, the priority value is commonly the rate-distortion slope at the rate truncation point. For convenience and matching with most of the embedded codecs, the priority value monotonically decreases with the increasing of rate point. The segment that corresponds to the first rate point $r_{i,0}$ is often called the base layer of the media packet. The last rate point $r_{i,n}$ always points to the end of the packet, with the priority value equaling that of the entire packet:

$$r_{i,n} = \text{length of packet } i \tag{1}$$

$$s_{i,n} = s_i \tag{2}$$

When the resource is tight, the media packet segments with low priority value are dropped first.

3.2.2 PeerStreaming format

The file format used in *PeerStreaming* can be shown in Fig. 6. Compared with the traditional media file shown in Fig. 3, we notice that *PeerStreaming* adds one level of hierarchy above (the *timeblock*) and below (the *data unit*) the media packet. A two-level index hierarchy, the timeblock index table and the media packet index table in the timeblock header, eases the locating, reading, and writing of non-sequential media packets. The *PeerStreaming* media may also have a DRM header, whose content will be described in Sect. 3.3.

The timeblock is an additional level of hierarchy added on top of the media packet. The media file is split

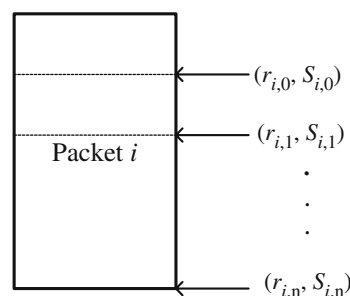


Fig. 6 *PeerStreaming* media file format

into a number of timeblocks, each of which consists of media packets of a short non-overlapping time span, say 1 s. Because most advanced video codecs use group of picture (GOP) structure, and usually a video can only be played from the beginning I frame of each GOP, ideally, the timeblock should be aligned with the boundaries of the video GOPs. The timeblock is the atom of writing media packets. The *PeerStreaming* client only writes out the timeblock after the media packets of the entire timeblock has been received.

An index table of all the timeblocks follows right after the media header and the DRM header. Each entry of the timeblock index table consists of the beginning timestamp of the timeblock, the beginning of the data unit index, the length of the timeblock header, an availability tag, and a timeblock data pointer. The timeblock index table helps the *PeerStreaming* system to quickly locate the timeblock and its media packets: the timestamp provides the capability to locate them through media playback time, the data unit index provides the capability of retrieval via data unit ID. The availability tag provides information on how complete the timeblock is: whether the timeblock is partial because the user starts or stops the streaming in the middle of the timeblock, or because the user only retrieves certain channels, or because the user retrieves only the high-priority media packets and/or the base layer of the truncatable media packets. The timeblock data pointer points out the location and the length of the timeblock data that is stored in the media file.² If a certain timeblock has not been received yet, the data pointer will be empty.

Each timeblock is led by a timeblock header, which consists of a table of index of the media packets. The table entry contains the media packet header (channel index, beginning timestamp, property flags, the original

² An alternative solution is to let each timeblock be an independent file. The pro is that defragmenting non-sequential timeblocks becomes the burden of the operation system. It is also easier to update the content of the timeblock. The con is that a long movie may consist of several thousand timeblocks, which lead to several thousand files. This may put unnecessary burden to the file system.

media packet length), and a pointer to the media packet data. For truncatable media packet, the current length of the media packet is also attached. If a particular media packet does not exist, the pointer will be empty. The bitstream of the compressed media packets then follows. The timeblock header enables the *PeerStreaming* system to quickly locate a media packet within the timeblock.

The timeblock index table and the media packet index table at each timeblock header form a two-level index table. Once retrieved, it enables the *PeerStreaming* client to gain a bird's eye view of the entire media, so that it can plan the receiver-driven P2P streaming intelligently. There are three categories of information in the timeblock index table and media packet index table. The first category is the media structure information. This includes the beginning timestamp, the beginning data unit index, the timeblock header length in the timeblock index table, and the media packet index table in the timeblock header. The media structure information is the same across all peers holding the media, even for a peer that holds only a partial or a scale-down copy of the media. As a result, it only needs to be retrieved once, and may be retrieved in a distributed fashion from multiple peers. The second category is the availability vector. It describes whether a particular timeblock/media packet is held by a certain peer. For scalable media with truncatable packet, it may further describe how much the packet is available from a certain peer. The availability vector is unique for each serving peer and needs to be retrieved individually. The third category is the data pointers, such as the timeblock pointer and the media packet pointer. The data pointers do not need to be sent over the network.

PeerStreaming breaks everything to be transmitted, including the media header, the DRM header, the timeblock index table, the timeblock header, and the body of the media packets into data units of maximum size L . There are three reasons for using fixed size data units. First, the *PeerStreaming* client and the serving peer may pre-allocate memory block of size L , thus avoiding the costly memory allocation operation during the receiving and serving process. Second, the data unit enables all media content to be mapped into a data unit ID space. Thus, the request to retrieve information, whether it is the header or the body of the media packet, becomes the request to retrieve the data unit with certain ID. This greatly simplifies the communication protocol between the peer and the client. A sample mapping between the media file and the data unit ID is provided in Table 1. Third, splitting all information into smaller data units allows the *PeerStreaming* client to retrieve the information in a distributed fashion. Each individual data unit may still be retrieved from a single serving peer. Never-

Table 1 Mapping between the content and the data unit

Data unit ID	Content
0x00000000	Length of the media header, the DRM header & the timeblock index table
0x00000100-0x000001ff	Media header
0x00000200-0x000002ff	DRM header
0x00000300-0x000003ff	Timeblock availability vector
0x00000400-0x00000fff	Timeblock index table
0x****0000-0x****ffff	Body of a certain timeblock ****
0x****0000-0x****00ff	Media packet availability vector in the timeblock
0x****0100-0x****01ff	Media packet header in the timeblock
0x****0200-0x**** *fff	Media packet body

theless, by retrieving the underlying data units of a media component from different serving peers, and by assigning more data units to the serving peer with high upload bandwidth, and fewer data units to the serving peer with low upload bandwidth, the *PeerStreaming* client can retrieve every information, including the media and the DRM header, the media structure, and the media packet body in a distributed fashion. Note that the data unit at the end of the media packet, DRM header, media header can be of sizes smaller than L . The actual size of the data unit can be calculated from the corresponding media packet size, DRM header size, and media header size. Therefore, there is no need to send the size of the data unit to the peer.

3.3 DRM (digital rights management) manager

The concept of DRM was first proposed to protect software copyright. Recently, it has been extended to prevent illegal consumption of digital media. As illustrated in Fig. 7, the media is encrypted, with the decryption key being maintained by a license server authorized by the content owner.

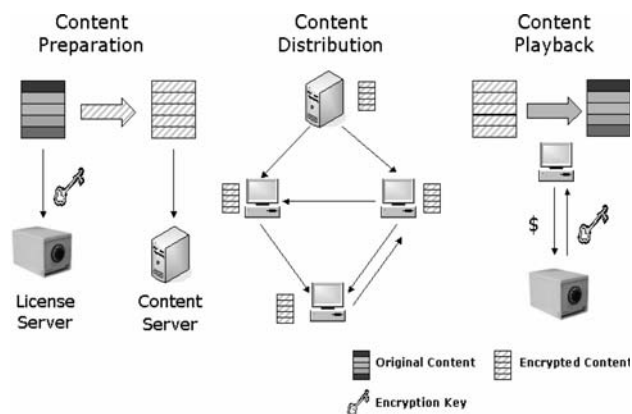


Fig. 7 Digital rights management in *PeerStreaming*

Windows Media DRM is a symmetric cipher based on the individualization of the critical components of each run-time client, rather than the individualization of the media. That is, if a Windows Media DRMed media is distributed to multiple clients, the piece of media delivered to each client is exactly the same. Windows Media DRM uses the same secret key to encrypt and decrypt a block of content. The protection mechanism focuses on the wrapping of key carefully so that the key is not exposed during the media delivery and media playback to any third-party users and programs. The reason for such implementation is to reduce the complexity of the media server, as individualization of media requires the server to encrypt the media with a different key, which greatly increase the computational load of the server and may limit the number of streams that a server can serve concurrently. At the time of playback, Windows media DRM individualizes the critical components of each client so that it binds the run-time client to the computer on which the client was initially installed. Every user is given a different executable file and different certified license keys. Every time the user plays the media, the *PeerStreaming* system checks if a license, which contains the secret key used for decryption and the terms of using the decryption key, is available that authorizes the use of the media. This license is issued by a license server and uniquely bound to a playback device, so that the license stored in one device cannot be transferred and used by another device for playback. This significantly reduces the danger of global breaks. If a specific DRM client becomes compromised, its acquired license cannot be used to playback the media in other device. To the authors' knowledge, most commercial DRM solutions rely on similar mechanism that use the same encryption key for all DRMed media, and uniquely identifying the playback device for protection. Though our implementation of *PeerStreaming* system is based on Windows Media DRM solution, the implementation method can be applied to peer-to-peer delivery of other DRMed media, e.g., Helix DRM solution as well.

In what follows, we describe in details the DRM implementation in *PeerStreaming*.

3.3.1 DRM media protection

Each DRM protected media has an additional DRM header, which is placed right after the media header. The DRM header contains an URL that points to the license server of the media, and a media ID that uniquely identifies the media.

The DRM protected media is encrypted prior to distribution. Only the compressed bitstream of the media packet is encrypted. The media structure information

and the availability vector are all in clear text. This enables all peers to parse the media, to efficiently distribute, and serve the media. However, when playing the media, the client has to retrieve from the license server a license that is uniquely bound to the playback device and contains both rights and the secret key that can be used to decrypt the media packet. To minimize the chance that the content may be cracked, a different packet key is individualized for each packet by combining the packet content and the original encryption key through a hash message authentication code (HMAC). Each packet is encrypted by this packet key, which is encrypted by the content key, and attached at the end of each packet. Since the recovering of the encryption key of a single packet does not automatically recover the keys of the rest of the media packets, we can afford to choose a relatively low-complexity algorithm to encrypt/decrypt the individual media packet. Moreover, since different media packets are encrypted with different keys, the master encryption key is less exposed, and is less likely to be cracked. We choose RC4, a widely used symmetric key algorithm with linear encryption/decryption complexity. The key size is selected to be 8 bytes. At the time of decrypting the packet, the packet key is first decrypted by the content key, which is then used to decrypt the media packet through RC4.

3.3.2 DRM license management and decryption

In order to playback the protected media, the user must contact the license server to obtain a license. A DRM manager module is implemented in our system to delegate the license acquisition task. During initialization of the player, the DRM manager generates a unique public-private key set for the player. This process makes each player unique. To purchase the key, the DRM manager send a request to the license server with the user's personal information (e.g., name, credit card number if payment is required) attached. After proper authentication and purchasing transaction, the license server sends a license back to the DRM manager. In order to defeat any interception effort, to discourage the user to peep into and modify the license file, and to individualize the license, the license server encrypts the license with the public key of the player. During the media playback, the DRM manager may decrypt the license via the private key of the player to obtain the content decryption key and examine the term of usage. Nevertheless, it does not store the decryption key in the hard disk.

Besides the license management task, the DRM manager is also responsible to decrypt the incoming media packet using the acquired decryption key. Once decrypted, the clear packet is fed into the audio/video

decoder. To deter piracy, the DRM manager also makes sure that the clear packet is fed into the decoder, rather than stored in a local file. After playback, the clear text media packet is immediately removed from the memory. The media packets cached by each peer are still the encrypted packets.

The license acquisition operation of the DRM manager is controlled by the end user through a specific user interface. Since playing the protected media may require compensation, the user must give explicit consent and/or payment in order to secure a license. If the user declines to acquire the license, the media playback function will be disabled. The *PeerStreaming* system can still receive, cache, and serve the media. But it cannot playback the media content. Nevertheless, if the user decides to acquire the license at a later time, he/she may be able to play locally cached media instantly.

3.4 Receiver-driven streaming protocol

PeerStreaming adopts a receiver-driven streaming mode. The peers do not collaborate during the service. Moreover, the peer may offer widely varying bandwidth, and may drop offline due to a variety of reasons, such as the demand of other local programs, the deterioration of the network link, the hardware and software instability, etc. It is the client that coordinates the streaming process with its best effort, retrieves the media from multiple peers, performs load balancing, and handles the joining/leaving of the peers.

Shown in Sect. 3.2.2, the *PeerStreaming* client requests every media components in the form of the data unit. The request is thus a 32-bit data unit ID. The reply is the content of the data unit. Depending on the length of the data unit ($L = 256$, $L = 512$ or $L = 1,024$), the size of the request is about 0.4–1.6% 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 received.

We explain the details of the receiver-driven streaming operation in the following section. In Sect. 3.4.1, we explain the reason why we choose TCP protocol to communicate between the client and the serving peer. In Sect. 3.4.2, we examine the request and staging queue of the *PeerStreaming* client. The streaming bitrate control of scalable coded media is examined in Sect. 3.4.3. The entire receiver-driven streaming operation is described in Sect. 3.4.4.

3.4.1 Network link: TCP connection

We choose TCP connections as the network links between the client and the serving peers. The choice

greatly simplifies our design, and results in minimum performance penalty compared with designing the network link with the UDP/RTP protocol.

In media delivery, there are three possible mechanisms to deal with the media packet loss: forward error correction (FEC), the selective retransmission, and automatic repeated request (ARQ, always retransmission). The first two mechanisms have to be implemented via the UDP/RTP protocol. The third mechanism can be conveniently implemented via the TCP protocol. Like any modern streaming media client, the *PeerStreaming* client has a streaming buffer many times larger than the round trip time (RTT) between the client and the peer; thus plenty of chances for 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). FEC is thus unfavorable compared with retransmission. Comparing ARQ with the selective retransmission, the latter has an edge only if many packets are selected not to be retransmitted. For non-scalable coded media, a lost packet usually leads to serious playback degradation. Therefore, the lost packet is almost always retransmitted. With the scalable 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. The ARQ scheme (implemented via TCP) can simply choose not to request these packets in the first place, thus achieve similar bandwidth usage and media playback quality with the selective transmission scheme. Yet, using TCP greatly simplifies the design of the media streaming system. Through TCP, there is no need to deal explicitly with flow control, congestion control and avoidance, keep alive, etc.. They are all handled by TCP.

3.4.2 Request and staging queues: throughput control, load balancing and request redirection

To coordinate the peers, the *PeerStreaming* client maintains a staging queue to hold the arrived data units. It also maintains a request queue for each peer to hold the unfulfilled request sent to the peer. The request and staging queues are shown in Fig. 8. The staging queue is also 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 sent back by the serving peer, and (3) to handle the disconnected peer.

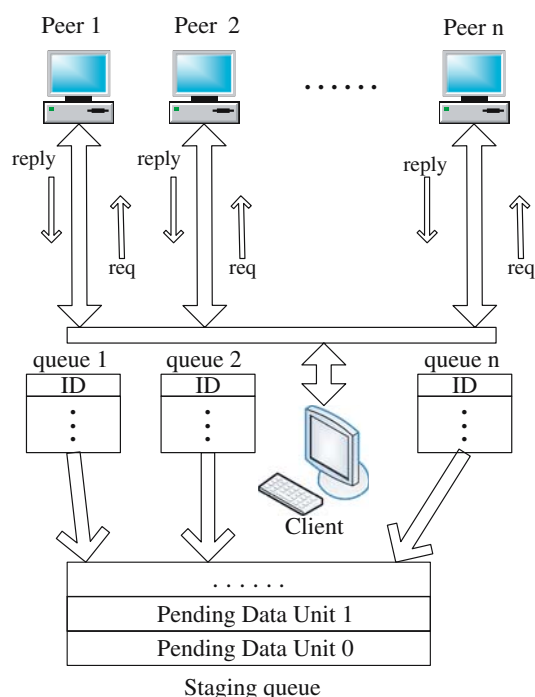


Fig. 8 The client maintains a staging queue for the arriving data unit, and one request queue for each serving peer

The first functionality of the request queue is to balance the load among the serving peers. During the streaming process, the *PeerStreaming* client retrieves the media structure, the availability vector, and the media packets continuously from the serving peers. After checking the received availability vector of the peer, and making sure that the peer has the requested media component, the client adjusts the number of data units requested from the serving peers to maintain a constant request fulfillment time (RFT) T_{rft} among the request queues of the peers. The RFT is defined as the time passed from the request is issued until its reply is received. The RFT consists of the following components: (1) the network delay from the client to the peer, (2) the bandwidth delay caused by sending the request to the peer, (3) the processing delay of the peer, (4) the network delay from the peer to the client, and (5) the bandwidth delay caused by sending the reply to the client. Normally, the client is continuously sending requests to the serving peer, and the peer processes the requests as soon as they arrive, and sends the replies back to the client until the sending buffer of the TCP socket from the peer to the client is full. Considering the fact that the request is much smaller in size than the reply, and the network delay is much longer than the processing delay, the jitter in the RFT components (1) to (4) may all be absorbed by the bandwidth delay of the reply from the serving peer to the client. The dominant factor in deter-

mining the expected RFT of a request and its reply is the serving bandwidth of the peer, and how much pending reply is still outstanding from the peer:

$$T'_{\text{rft}} = (B_{i,\text{outstanding}} + B_{\text{cur}})/Th_i, \tag{3}$$

where

T'_{rft} is the expected RFT of a new request,

Th_i is the serving bandwidth from the peer i to the client,

$B_{i,\text{outstanding}}$ is the length of unreceived replies from peer i before the current request,

B_{cur} is the length of the current reply.

With the request queues to all peers being the same length in RFT, the capacity of the request queue ($Th_i \cdot T_{\text{rft}}$) becomes proportional to its serving bandwidth. The *Peer Streaming* client thus balances the load among the peers. As an example, with T_{rft} be 1.0 s, a peer with serving bandwidth of 16 kbps allows 2 KB of unfulfilled request pending in its request queue, while a peer with serving bandwidth of 1 Mbps allows 128 KB of unfulfilled request pending. It is desirable to set the system-wise T_{rft} to be slightly larger than the maximum round trip time (RTT) between the client and the most remote peer plus the time it takes for the peer to process the request. Short T_{rft} leads to short request queue, which may not effectively absorb the network jitter between the client and the peer and the processing jitter of the peer. Large T_{rft} results in a long request queue, which may prevent the client from quickly adapting to the changes, e.g., the disconnection of a peer and the need to redirect request. Currently, we set T_{rft} to be 1.0 s.

The second functionality of the request queue is to identify the content sent back by the 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. We do not rely only on the TCP timeout function to detect the disconnection event of the serving peer. The reason is that the TCP protocol can be slow to detect disconnection, with the response time of the TCP protocol being as long as seconds. Instead, the *PeerStreaming* client relies on the fact that the peer is constantly sending data towards the client. The client monitors the incoming data flow from each serving peer. If no data arrives from a particular serving peer after a certain time span T_{idle} , say 0.5 s, the serving peer will be marked as inactive. The client

immediately reassigns all unfulfilled requests pending in the queue of the inactive 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. It may happen that a peer is falsely marked inactive, e.g., due to a long sequence of lost packets in the TCP channel. In this case, the client can simply reactivate the peer and put it back into service. If the peer is truly inactive and is disconnected, the disconnection event will be eventually picked up by the TCP protocol. The client then removes the peer from its peer list.

Whenever a reply arrives at the client, it is immediately pulled away from the TCP socket of the incoming peer. After pairing the arriving reply with the pending request, the fulfilled request is removed from the request queue. The identified data unit is then deposited into the staging queue. The content buffered in the staging queue is controlled to not exceed a certain size, which is measured by a duration T_{staging} . Whenever the staging queue becomes full, the *PeerStreaming* client stops issuing more requests to conserve the serving bandwidth of the peers.

3.4.3 Streaming bitrate control for scalable coded media

Non-scalable coded media is always streamed at the bitrate of the media. However, the bitrate of the scalable coded media may vary during the streaming session. The instantaneous streaming bitrate R_{recv} for scalable coded media may be calculated by

$$Th = \sum_i Th_i, \quad (4)$$

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

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

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

where

Th is the aggregated serving bandwidths from all peers,

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

T_{rft} is the desired RFT (default 1.0 s),

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

$B_{\text{outstanding}}$ is the length of outstanding replies from all peers to be received,

R_{raw} is the raw streaming bitrate,

R_{filter} is the filtered streaming bitrate,

R_{min} is the base layer bitrate,

α is a low pass control parameter.

Eqs. (4)–(7) control the streaming bitrate R_{recv} to follow the aggregated serving bandwidths Th and the staging and request queue statuses. Once the streaming bitrate is determined for the coming timeblock, the client sorts the media packets (or the media segments if the media packet is truncatable) according to their priority values. The media packets (and the media segments) with lower priority values are dropped until the remaining data fit the streaming bitrate R_{recv} .

3.4.4 PeerStreaming requests and replies

The life of a *PeerStreaming* request and its reply can be shown in Fig. 9. First, the client generates a request and sends it through the outbound TCP connection to a certain peer. After the request is delivered to the serving peer, it is stored in the TCP receiving buffer of the peer. The serving peer processes the request one at a time, in a single thread, blocking fashion. For each request, it deciphers the ID of the data unit requested according to Table 1, and determines whether the data unit belongs to the media header, the DRM header, the timeblock index table, the timeblock header, the media packet, or the availability vector of the timeblock and the media packets. The peer then fetches the associated data and sends it back to the client via the TCP link. If the peer cannot perform any of the aforementioned operations due to lack of resources, e.g., the data is pending hard disk access, the TCP sending buffer is full, the peer simply

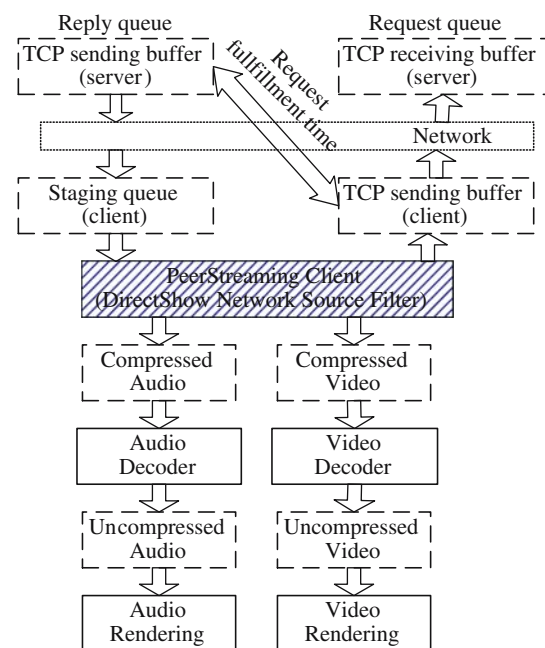


Fig. 9 The life of a *PeerStreaming* request and its reply: the buffers (in dashed box) and the processing modules (in solid box)

blocks and waits for the resources to become available. The peer is thus light loaded and simple to implement.

If the client requests a non-existing data unit, the serving peer may simply cut off the network link to the client. The client's responsibility is to fetch and check the availability vector beforehand, and makes sure that the underlying data corresponding to a certain request are available for retrieval.

Once the reply data unit arrives at the client, it is immediately moved to the staging queue. In the staging queue, the data units from multiple peers are combined into various components of the media, e.g., the media/DRM header, the timeblock index table, the timeblock header, and 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.

All the buffers in Fig. 9 constitute the *PeerStreaming* playing buffer and 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 $T_{\text{staging}} = 2.5$ s, set the desired RFT to $T_{\text{rft}} = 1.0$ s, and set the compressed audio/video buffer to 0.5 s. The total *PeerStreaming* buffer is thus 4 s.

4 Peerstreaming system evaluation

We develop the *PeerStreaming* system on the Windows operating system using Microsoft DirectShow framework [8]. In what follows, we first introduce the system implementation, then present testbed results obtained from the *PeerStreaming* prototype deployment.

4.1 System implementation

We start by describing the operation of the *PeerStreaming* serving peer. Whenever a client contacts the peer for service, the peer decides if the client should be admitted according to a certain criterion, e.g., whether it has sufficient serving bandwidth. Once admitted, the peer launches a separate serving thread to serve the client.

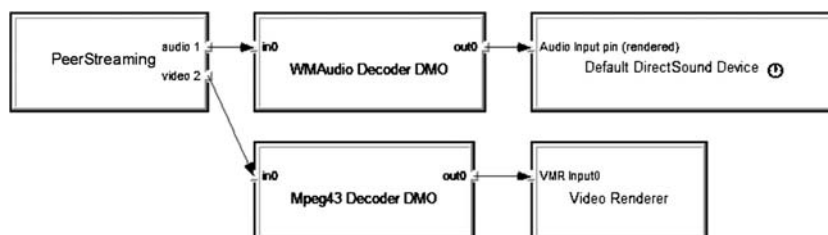
The majority of the operation is performed by the *PeerStreaming* client. Once launched, the client gets a

list of nearby peers that hold the requested streaming media, using either the supernode assisted lookup approach described in Sect. 3.1.1 or the distributed hash table (DHT) approach described in Sect. 3.1.2. It then retrieves the data unit 0x00000000 from any serving peer, which contains the super-structure information that includes the length of the media header, the DRM header, and the timeblock index table. The super-structure information enables the client to determine the constituent data unit IDs of the media header, the DRM header, and the timeblock index table, and retrieve these components in a distributed fashion from the peer cluster. Through the load balancing mechanism of Sect. 3.4.2, the peer with higher serving bandwidth is assigned with more requests, and the peer with lower serving bandwidth is assigned with fewer requests.

Once the media header arrives, the client analyzes the media header and sets up the audio/video decoder chain and the rendering devices. The *PeerStreaming* client is a DirectShow network source filter. The number of output pins of the source filter depends on the number of channels of the streaming media, i.e., the output of each channel bitstream is fed into the one 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 Fig. 10.

Via DirectShow, the *PeerStreaming* system is capable of streaming, decoding, and rendering media coded by any codecs that has a DirectShow decoder DMO component, such as MPEG 1/2/4, WMA/WMV, Indeo Video. *PeerStreaming* may also attach additional audio/video processing modules after decoding, such as resolution/color space conversion, de-interlacing, so that the decoded audio/video may match the capacity of the audio/video rendering device. The synchronization of the audio/video track is handled by DirectShow as well. Shown in Fig. 10, the audio renderer has a small clock attached to it. This indicates that the audio stream holds the reference clock of the entire media. When the video is played, 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

Fig. 10 The DirectShow filter graph of the *PeerStreaming* client



PC (or one with hyper-threading enabled), the computation load of various components of the *PeerStreaming* client, e.g., the receiver-driven streaming component, the audio/video decoder, 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.

In *PeerStreaming*, the user may access the media in a non-sequential fashion, and start at any point he/she likes. Right after retrieving the media header, the DRM header and the timeblock index table, the *PeerStreaming* client consults with the timeblock index table and finds the timeblocks to be retrieved given the timestamp of the access point. The client then retrieves a few timeblock headers from the peer cluster. If the timeblock availability vector of a particular peer indicates that the entire timeblock is not at all available from the peer, the client also retrieves the media packet availability vector from the peer. Once the timeblock headers and the media packet availability vector have been retrieved, the client then proceeds to retrieve the media packets sequentially. Using the timeblock header, the data unit IDs of the media packets are calculated, and the media packets are retrieved one by one from the peer cluster. The client again balances the load among the peers using the algorithm described in Sect. 3.4.2. For the scalable coded media, the streaming bitrate is dynamically adjusted based on the available serving bandwidths and the status of the client queues, as described in Sect. 3.4.3. Periodically, the client updates the serving peer list, and connects to potential new peers. The streaming operation continues until the entire streaming media is received, or the user stops the streaming.

The *PeerStreaming* client also stores the received media into a local file for the purposes of caching and serving other peers. The stored file format is shown in Sect. 3.2.2. As soon as the media packets of an entire

timeblock are received, they are written into the hard disk. The timeblock may then be used for serving.

4.2 Experimental results

We have built a working *PeerStreaming* system. A snapshot of the *PeerStreaming* client is shown in Fig. 11. Interested readers may download and try the prototype *PeerStreaming* demo from [16]. The *PeerStreaming* player consists of a seek bar at the bottom, a set of play/pause/stop buttons at the top, and a window at the right showing the current peers that serve the *PeerStreaming* player. The user interface of the *PeerStreaming* client is very much like an ordinary media player. The user may use the seek bar to jump to whatever point he/she desires, may start playing the media, stop the playing, and pause as he/she wishes. The serving peers and their contributing bandwidths are shown at the right window.

In this section, we evaluate the performance and efficiency of two components of the *PeerStreaming* system.

4.2.1 Peer discovery: supernode lookup versus DHT lookup

First, we evaluate the two peer discovery approaches: the DHT approach versus the supernode assisted lookup approach. We notice that it is not uncommon for the PeerNet to take more than 10s to locate the available peers. This may be due to the churn of the Pastry algorithm reported in [19]. Since the PeerNet is an experimental P2P network, many nodes are short lived. A test node may start, encounter an error, and simply stops within seconds. The frequent turnover of the member nodes in PeerNet may further aggravate the peer discovery performance.

In comparison, the supernode assisted lookup approach is much quicker to return the list of peer nodes.

Fig. 11 *PeerStreaming* system screenshot



The trade-off is that the supernode in the network needs to devote more network and computation resources to serve the rest of the nodes. To evaluate the burden of the supernode, we randomly seed the supernode with 10,000 simulated movie clips and 1,000,000 peers. We then generated random peer discovery requests to the supernode. In our supernode implementation (via C#), we notice that the previous movie and peer list consumes 280MB of memory. Since the number of peers is the dominant factor in determining the amount of the memory consumed, this translates to roughly 280 bytes of memory per peer. Each peer discovery request takes the supernode 0.16ms CPU processing time to serve (Pentium IV 3.06 GHz). Assuming each peer record is 30 B in length, and the peer discovery returns the 10 closest peers to the requesting client, the peer discovery request will consume 300 bytes upload bandwidth of the supernode. With simple calculation, a supernode behind a T1 line (1.5 Mbps) will be able to serve about 600 such requests per second. It seems that the supernode assisted lookup is more suited for peer discovery in small- to medium-scale P2P network, while the DHT approach is more suited for large scale network. We observe that in the supernode assisted lookup approach, the bandwidth of the supernode is the dominant factors in determining the number of peers that it is capable of serving.

4.2.2 Network utilization

Next, we evaluate the network utilization of the *PeerStreaming* system. We set up a *PeerStreaming* cluster, which consists of one client and three serving peers, with the serving bandwidth of the peer being 350 kbps (#1), 700 kbps (#2) and 1,400 kbps (#3), respectively. We then measure the actual bandwidth, i.e., the number of bytes received per second from each peer by the *PeerStreaming* client. The result is shown in Fig. 12. We plot the total receiving bandwidth from all peers with the solid line, and plot the actual bandwidth from peers #1,#2, and#3 with the dash-dotted, dashed, and dotted line. The media streamed is a 2-min movie clip with average bitrate around 2 Mbps. The file length is 28.1 MB. To test the capability of the *PeerStreaming* client to cope with the network anomalies, we stop the peer#3 at around 24 s, and restart the peer again at around 40 s.

From Fig. 12, we observe that the *PeerStreaming* client is able to distribute the serving load in proportion to the bandwidths of the peers. During the majority portion of the streaming sessions, the total serving bandwidth of the peer cluster is higher than the bitrate of the media. Therefore, the serving bandwidth of each peer is not fully utilized. The actual used bandwidth of each peer is roughly proportional to the serving bandwidth of the

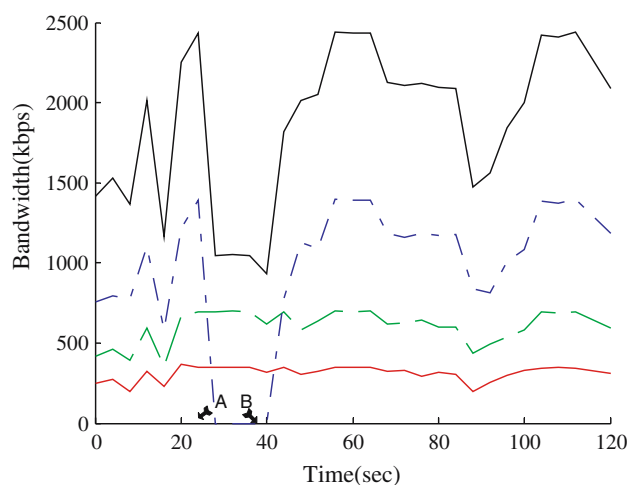


Fig. 12 The network utilization of the *PeerStreaming* system. The solid line is the total receiving bandwidth. The dash-dotted, dashed, dotted lines are the number of bytes received from the peers #1, #2, and #3

peer. At point A of Fig. 12, we shutdown the peer#3. The *PeerStreaming* client is able to detect this network disconnection event, and successfully transfer the requests originally directed toward the peer#3 to the surviving peers #1 and#2. The peers #1 and#2 also begin to serve at their full bandwidths at this point. Because the total serving bandwidths are below the media bitrate, the play of the movie clip becomes choppy. Due to insufficient serving bandwidths, the *PeerStreaming* client is unable to feed the media packets faster enough to the audio/video decoder. 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. At point B of the figure, the peer #3 is online again. After a few seconds, the *PeerStreaming* client detects the on-line event of the peer #3, and re-engages the peer into the serving loop. As the aggregated serving bandwidth exceeds the media bitrate (2 Mbps in this case), the clip plays smoothly again.

In another streaming session, we stream a scalable coded audio with truncatable packet mode via the *PeerStreaming* solution. The audio, compressed via a scalable audio codec of [12,13], is compressed to lossless. Each media packet of the compressed bitstream is independently truncatable at any point. 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. Coded to lossless, the compression bitrate of the audio is 700 kbps. Yet audio with perceptible quality may be streamed with serving bandwidths as low as 32 kbps.

5 Peerstreaming simulation study

Due to the scale limitation on our testbed, we conduct simulation study to evaluate the key performance metrics of *PeerStreaming* in Internet-based large-scale deployment.

5.1 Simulation setup

Using the BRITE topology generator [14], we create a 1,000-router topology, upon which 300 peers are attached to. The distance between two peers is measured by the hop count of their shortest path in the Internet topology. Our simulation also includes a media server, which could be seen as a permanent peer which possesses the entire movie content.

Each peer maintains limited disk space for the streaming movie. We assume that each peer's inbound is greater than or equal to the bitrate of the movie. Since the additional inbound bandwidth does not help with the *PeerStreaming* performance, our simulation sets the inbound bandwidth of all peers uniformly to the movie bitrate, for the sake of simplicity. However, the peers' outbound bandwidths are heterogeneous. We use average outbound/inbound ratio across all peers to measure the bandwidth availability in the *PeerStreaming* system.

We assume that each peer plays the movie randomly, i.e., the starting position of the playback is uniformly distributed from beginning to the end, and the duration of the playback is randomly distributed. Each peer caches

the received content to its disk in a FIFO fashion. Here, a circular buffer is maintained where the newly received content will flush out the oldest-cached data.

Since each peer streams to its children peers via TCP connection, and each peer has sufficient inbound bandwidth to receive the stream, the outbound bandwidth of the supplying peer becomes the bottleneck link. Combined with the fair-sharing property of competing TCP connections, it is reasonable to assume that the outbound bandwidth is equally shared to all children. Each peer conducts its peer selection as follows. It first chooses k of its nearest neighbors, then checks the availability vectors of its nearest neighbors to see whether the requested data is available. If so, the neighbor is chosen as its parent. Finally, if a peer cannot receive enough bandwidth from all its supplying peers, it will seek help from the server to retrieve the residual bandwidth.

In our simulation, the streaming movie has the constant bitrate of 100 Kbps. The cache size of each peer is measured by kilo-bits, or seconds in terms of time. Each simulation run lasts 1 h, in which the average request length is 100 s.

5.2 Simulation results

We choose the server load as the main performance metric to study the scalability of *PeerStreaming*. The result is shown as the average load along the simulation run. In what follows, we illustrate the impacts of peer request rate, cache size, and average outbound/inbound bandwidth ratio, to the server load.

Figure 13 shows the server load when the request rates are 0.77 and 2.2, respectively. In both figures, the server load drops by at most 50% when we increase the cache size from 1 to 5 s. The curves quickly level off as the cache size further increases, indicating that

Fig. 13 The server load under different cache sizes

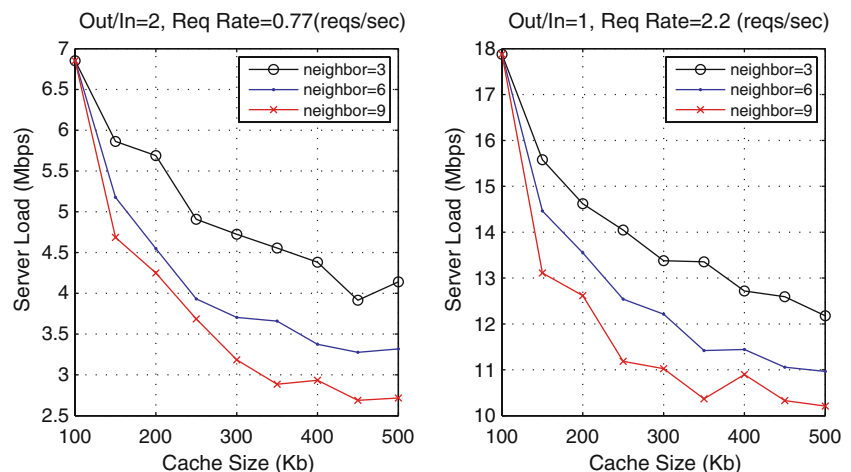


Fig. 14 The server load under different outbound/inbound bandwidth ratios

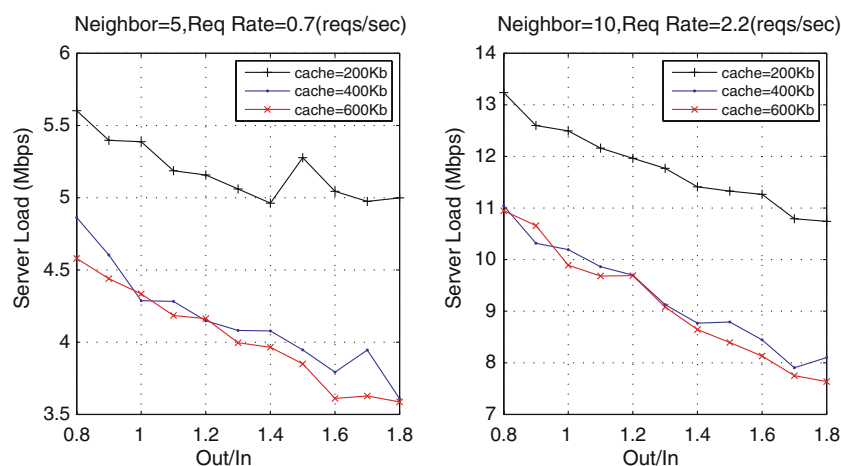
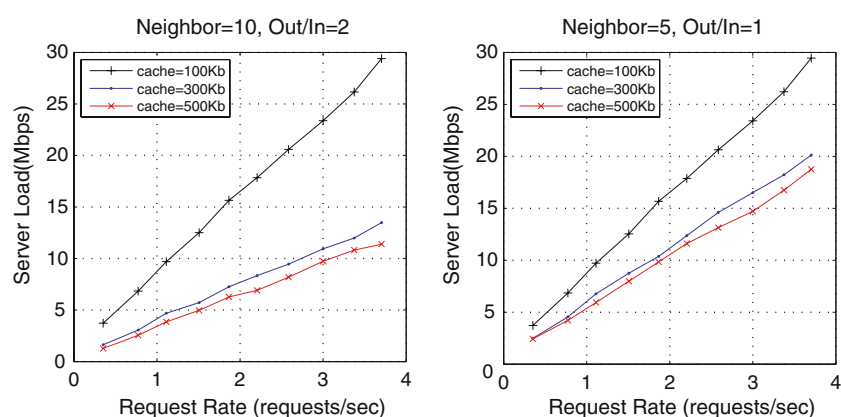


Fig. 15 The server load under different request rates



small cache size is sufficient to alleviate the server load to the limit of 50%. Note that the effect of cache size is tightly coupled with the request rate, i.e., only when the cache window is greater than the average request inter-arrival time can the peers be able to retrieve each other's content. Furthermore, when linearly increasing the number of neighbors k , the server load also drops by near constant speed.

In Fig. 14, increasing the outbound/inbound bandwidth ratio makes more outbound capacity available among peers, hence reduces the need to stream from server. However, we note that reliance on server will not vanish regardless the redundant outbound capacity. This is mainly due to the data availability issue, i.e., when a supplying peer terminates streaming, its children will be short of supply. A child peer could be orphaned if all its parents leave. Reissuing requests to find new supplying peers will increase the exchange overhead of availability vector and cause glitch to the user's viewing. Therefore, a more appropriate solution is to seek help from the server retrieve the needed bandwidth. Finally, the overlapped curves for cache size of 4 and 6 s reflects the observation made in Fig. 13, which shows the diminishing return effect of increasing cache size.

Finally, we show the server load as a function of request rate in Fig. 15. Under various settings (neighbor number, cache size, outbound/inbound ratio, etc.), the server shows linear dependency to the request rate. However, the speed of growth (slope) is radically different depending on the cache size. Compared to the outbound/inbound bandwidth ratio, cache size plays a more important role at improving the scalability of *PeerStreaming*. Greater cache size will enable a supplying peer to be the potential parents of more upcoming peers, which greatly facilitates the parent selection process, thus increases the utilization ratio of peer outbound bandwidth.

6 Related work

There have been a great number of works dedicated to the field of P2P network. Early works in this domain mainly focus on building general routing infrastructures, e.g., Chord [22], CAN [17], and Pastry [21]. All of them are distributed hash table (DHT) based approaches, which divide and distribute the routing lookup table to all participating nodes of the network. Systems for

the general-purpose of file storage and exchange have been built on top of these routing protocols, e.g., CFS [6] on Chord [22], and PAST [20] on Pastry [21]. However, to retrieve the content, these systems can only operate in a “download mode”. They cannot meet the stringent demand of media streaming.

Recently, P2P streaming applications have attracted considerable research attention. Based on the way the streaming data is distributed, existing systems can be categorized into two categories: single-sender approach and multi-sender approach.

In the single-sender approach, an application-level multicast (ALM) tree is constructed to cover all receiving peers. The root of the tree is the media server. The intermediate nodes in the tree are peers, which receive stream from the server or their parent peers, and relay it to their child peers. Each ALM-based system has its own protocol for building and maintaining the multicast tree. In Narada [3], a mesh is established to interconnect all peers. This approach suits best for small-scale streaming systems but does not scale well as the network size increases. NICE [1] and Zigzag [24] use hierarchical distribution trees and therefore can accommodate a large number of peers. PeerCast [7] constructs a distribution tree rooted at the sender for a live media streaming session. A new receiver joins by traversing the tree starting at the root till it reaches a node with sufficient remaining capacity. oStream [5] uses cache-and-relay approach so that the peer node may serve the client with previously distributed media from its cache.

A major drawback of the single-sender solution is that the upload bandwidths of many end peers are not utilized. Since the majority of the nodes in a multicast tree are leaf nodes, this represents a big waste of the network resource. A natural way to increase the bandwidth utilization is to construct multiple ALM trees or a distribution mesh. Each peer then receives different portions (stripes) of the stream from multiple ALM trees. CoopNet [15] employs multi-description coding and constructs multiple distribution trees (one tree for each description) spanning all participants. SplitStream [2] provides a cooperative infrastructure that can be used to distribute large files (e.g., software updates) as well as streaming media. Utilizing an application level P2P service called CollectCast, PROMISE [10] seeks to serve peers that are most likely to achieve the best streaming quality, and dynamically adapted to network fluctuation and peer failure. PALS [18] employs layered streaming technique to address network heterogeneity in P2P streaming.

PeerStreaming can be categorized as a multi-sender solution. We have investigated into many practical issues in the deployment of an efficient P2P streaming system.

In addition to the DHT-based routing and peer discovery approach, a supernode assisted lookup approach has been developed as well, which is demonstrated to be much more efficient in locating peers for small- to middle-scale P2P network. *PeerStreaming* adopts an on demand non-sequential media streaming mode, which is more flexible and user friendly than the broadcast mode employed by any prior P2P streaming systems. *PeerStreaming* may deliver scalable coded media, and thus has better tolerance to the network jitter and peer bandwidth fluctuation. It also incorporates the DRM technology to combat the piracy. This enables the *PeerStreaming* to be deployed with less legal controversy. The DRM issues have not been addressed in any prior P2P media streaming systems.

Even though there are many P2P streaming systems in the academic paper, few systems are actually built for real-time media streaming. A few real-time P2P media streaming systems built are: GnuStream [11] (built on top of Gnutella), SplitStream [2] (based on Pastry [21]) and CoolStreaming [4] (based on DONet, which forces data exchange during the media streaming process). All above systems deliver the media as a sequential file. In comparison, *PeerStreaming* adopts a much more sophisticated media format model, and couples more tightly with the media rendering system. This enables *PeerStreaming* to have a much better reaction time (small buffering delay), a smoother media playback, and better coordination of the serving peers. Moreover, SplitStream and CoolStreaming are all designed for the media broadcast. The users of the network have to watch the same media at the same time. In comparison, *PeerStreaming* is designed for on-demand streaming. Each user may start at whatever point he/she desires, and progress at whatever pace he/she pleases.

7 Conclusion

We have designed and implemented *PeerStreaming*, a working real-time P2P media streaming system. *PeerStreaming* adopts the user interface of a traditional media player, and allows the user to receive the media in an on-demand, non-sequential way. Under the hood, however, the media is delivered efficiently from multiple peers. *PeerStreaming* may form the peer cluster via either the supernode assisted lookup approach or the DHT approach. It may stream and serve scalable coded media in a P2P fashion. Via a receiver-driven P2P streaming protocol, *PeerStreaming* may retrieve content efficiently from multiple peers, balance the serving load of the peers, and redirect requests from inactive and/or disconnected peers to the active ones. Through the DRM module, the content owner can confidently

use the *PeerStreaming* system to distribute the media without losing control of it.

References

1. Banerjee, S., Bhattacharjee, B., Kommareddy, C., Varghese, G.: Scalable application layer multicast. In: Proceedings Of ACM SIGCOMM'02, pp. 205–220, Pittsburgh (2002)
2. Castro, M., Druschel, P., Kermarrec, A.-M., Nandi, A., Rowstron, A., Singh, A.: SplitStream: high-bandwidth content distribution in a cooperative environment. In: Proceedings of the International Workshop on Peer-to-Peer Systems, Berkeley (2003)
3. Chu, Y., Ganjam, A., Ng, T.S.E., Rao, S.G., Sripanidkulchai, K., Zhan, J., Zhang, H.: Early experience with an Internet broadcast system based on overlay multicast. Technical report CMU-CS-03-214, Carnegie Mellon University (2003)
4. CoolStreaming, <http://www.coolstreaming.org>
5. Cui, Y., Li, B., Nahrstedt, K.: oStream: asynchronous streaming multicast in application-layer overlay networks. *IEEE J. Select. Areas Comm.* **22**(1), 91–106 (2004)
6. Dabek, F., Kaashoek, M., Karger, D., Morris, D., Stoica, I.: Wide-area cooperative storage with CFS. In: Proceedings of ACM SOSP'01 (2001)
7. Deshpande, H., Bawa, M., Garcia-Molina, H.: Streaming live media over a peer-to-peer network. Stanford database group technical report (2001–20) (2001)
8. DirectX 9.0 Complete Software Development Kit (SDK)
9. Gelfand, B., Esfahanian, A.-H., Mutka, M.: An agent-based approach to enforcing fairness in peer-to-peer distributed file systems. In: Proceedings of the 9th International Conference on Parallel and Distributed Systems, Taiwan, China, pp. 157–162 (2002)
10. Hefeeda, M., Habib, A., Botev, B., Xu, D., Bhargava, B.: PROMISE: peer-to-peer media streaming using CollectCast. In: ACM multimedia 2003, Berkeley, pp. 45–54 (2003)
11. Jiang, X., Dong, Y., Xu, D., Bhargava, B.: GnuStream: a P2P media streaming system prototype. In: Proceedings of IEEE International Conference on Multimedia and Expo (ICME 2003), Baltimore (2003)
12. Li, J.: Embedded audio coding (EAC) with implicit psychoacoustic masking. ACM multimedia, Nice. 1–6 December 2001 (2002)
13. Li, J.: The efficient implementation of a high rate Reed-Solomon erasure resilient code. ICASSP'05 (in preparation)
14. Medina, A., Lakhina, A., Matta, I., Byers, J.: Brite: An approach to universal topology generation. In: Proceedings of IEEE MASCOTS (2001)
15. Padmanabhan, V., Sripanidkulchai, K.: The case for cooperative networking. In: Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS), Cambridge (2002)
16. PeerStreaming prototype, <http://www.research.microsoft.com/users/jinl/2004/P2Pstreaming/download.htm>
17. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. In: Proceedings of ACM SIGCOMM'01, San Diego (2001)
18. Rejaie, R., Ortega, A.: PALS: Peer-to-peer adaptive layered streaming. In: Proceedings Of ACM NOSSDAV'03, Monterey (2003)
19. Rhea, S., Geels, D., Roscoe, T., Kubiawicz, J.: Handling churn in a DHT. In: Proceedings of the 2004 Usenix Technical Conference, Boston, 28–30 June 2004
20. Rowstron, A., Druschel, P.: Storage management in past, a large-scale, persistent peer-to-peer storage utility. In: Proceedings Of ACM SOSP '01, Chateau Lake Louise (2001)
21. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Proceedings of International Conference on Distributed System Platforms (Middleware'03), Heidelberg (2001)
22. Stoica, I., Morris, R., Kaashoek, M., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for Internet applications. In: Proceedings of ACM SIGCOMM'01, San Diego (2001)
23. Tang, L., Crovella, M.: Virtual landmarks for the Internet. In: Proceedings of Internet Measurement Conference'03, Miami Beach, 27–29 October 2003
24. Tran, D., Hua, K., Zigzag, T.Do.: An efficient peer-to-peer scheme for media streaming. In: Proceedings of IEEE INFOCOM'03, San Francisco (2003)
25. Windows XP Peer-to-Peer Software Development Kit