Department of
Information Engineering
and Computer Science
DISI

UNIVERSITY
OF TRENTO - Italy

DISI - Via Sommarive, 14 - 38123 POVO, Trento - Italy
http://disi.unitn.it

Implementing Streamers with GRAPES: Initial Experience and Results

Renato Lo Cigno, Csaba Kiraly, Luca Abeni, Alessandro Russo and Marco Biazzini

June 2010

Technical Report # DISI-10-039
Version 0.1

This work is supported by the European Commission through the NAPA-WINE Project (Network-Aware P2P-TV Application over Wise Network – www.napa-wine.eu), ICT Call 1 FP7-ICT-2007-1, 1.5 Networked Media, grant No. 214412

Abstract

Streaming on peer-to-peer systems is gaining a lot of attention from both academia and industry. Several P2P streaming systems have been designed and developed in the last few years. However, most of such systems relies on exstensive simulative studies done within protected environment, which is completetly different from real world Internet which includes bandwidth fluctuation, router, peers firewalled, and many other constraints which cannot be simulated.

In this paper we present three epidemic style P2P streamers for the real world. Such streamers are written on top of software libraries for P2P Streaming called GRAPES. We show DumbStreamer, a very simple streamer in both logic and code: a good starting point. We have improved such a streamer, adding a few features obtaining the RockStreamer which has signaling messages. Finally, we present OfferStreamer, which is smarter than the others, because includes singaling and negotiation before chunks exchange.

We present a set of results obtained running our streamers on a network of one hundred nodes, distributed in the Europe. We compare all the streamer, showing how OfferStreamer outperforms the others, and then we continue our analysis focusing on OfferStreamer.
Keywords – Peer-to-Peer; Live Streaming; Chunk-Based Distribution; Distributed Applications; Application-Layer Multicast.

1 Introduction

Live streaming systems are attracting a large population of customers, becoming the next killer application of the Internet after file sharing P2P systems. Such multicast systems require to distribute a noteworthy amount of data to many participants within a given deadline, otherwise data received after the corresponding deadline become useless. P2P involve all peers to support the system with their resources, providing a scalable system to deal with such multicast application.

The literature proposes many P2P systems [9, 5] to analyze different scheduling policies and overlay management algorithms. Such studies are always presented through extensive simulative studies, only a few of them become applications. Although simulation studies allow to reproduce different scenario (e.g., network dynamism, bandwidth fluctuation) in a simple and

controlled environment with low effort, obtaining useful feedback on the overall system, however, they are unable to reproduce real-world scenario, where peers are real devices, geographically distributed, where the live streaming application in each peer is not the only one that uses the resources at the peer (e.g., bandwidth, memory). Moreover, those protocols which become applications are tested in controlled environment only on a few nodes

In this work we present our generic P2P PeerStream. Such application offers several well-known policies for both piece and peer selections, and on overlay management as well. It is highly customizable, allowing users to write and plug their own policies into the PeerStreamwith low effort. Moreover, it collects a wide set of measures that can be eventually extended, which may be also used in run-time for dynamic reactive applications, and obviously at the end to compute the corresponding performance metrics. The PeerStreamis completely written in C and uses the GRAPES (Generic Resource-Aware P2P Environment for Streaming) software libraries which are developed under the NAPA-WINE[1] project.

We focus on explaining how the PeerStreamworks, providing three different streaming applications based on it: DumbStreamer, RockStreamer and OfferStreamer. We describe the architecture of the PeerStream, the different modules (e.g, scheduler, overlay), which lead to a highly customizable applications which can be easily extended to users' needs, and the streaming process: from either a video file or a DVB-T stream to real distributed streaming system.

Finally, we show our results based on different streaming applications derived from the streamer, which uses different scheduling policies and overlay algorithm, using the NAPA-WINE test-bed where real peers geographically distributed in Europe exchange chunks emerged by one source is able to streams DVB-T channels.

The rest of this paper is organized as follows. Section 2 extensively describes the PeerStream, its architecture and mechanism. In section 3 we compare different streamers and policies, showing results on the Internet, where peers are real node with real constraints (e.g., bandwidth fluctuation, RTT variation, heterogeneous bandwidth). Finally, in section 4 we draw our conclusions.

[1] www.napa-wine.eu

2 PeerStream

In this section we present the PeerStream, its architecture and the main components. We describe how it works, from a raw video file to a live streaming application. Moreover, we show three different streaming applications that obtained from the PeerStream, just writing the desired peers and chunks policies, the negotiation phase, and the neighborhood management.

2.1 Background

Current P2P streaming system may be classified under several aspects: overlay network, information exchange, content delivery. The overlay network provides the logical connections among peers, resulting in the P2P network. P2P networks can be classified as unstructured and structured, based on criteria adopted to built the overlay network. In structured systems [3, 7, 8], peers join the network following a well defined set of rules, resulting in a tree structure rooted at the source, where peers establish a parent/child relation, and each parent feeds their child with the chunks received by its parent. In unstructured systems[5, 4, 6], each node is connected to several nodes which define its neighborhood, and it will communicate with its neighbors.

The delivery strategies are either sender or receiver oriented. The Push approach is typical sender driven delivery strategy. In this case the sender takes the decision about which chunks to which peers, and pushes to the receivers which passively receive. On the other side, the Pull approach is receiver driven. In particular, the received selects the chunks it wishes to retrieve and the peers that may satisfy its requests, issuing pull messages to that peers that eventually will satisfy such requests.

Peers exchange different information among them about their state (e.g., chunks owned or missed, their neighbors, and other information) to increase their local knowledge. The reason is very simple: higher the information collected, higher is the possibility to satisfy its needed. Thus, they exchange information to know which peers have the missed chunks, or to which peers provide chunks, avoiding useless blind behavior. Note that such information exchange is not in structured systems for retrieving chunks, but is used as control messages to maintain the tree-structure.

2.2 PeerStream Architecture

In this section we describe the PeerStream architecture and the work flow from the input stream to chunks, how they are diffused in the network, and how the overlay network is built.

The PeerStream follow the epidemic live streaming paradigm where the content is split in pieces spread among peers in a fully distributed fashion. Gossip protocols keep the peers connected in sampled overlay on which negotiation and chunk transmission are performed. All communication are performed using UDP connections.

The PeerStream provides many options for tuning the application. For instance, the user may define chunks buffer size which represents the window on which peers may negotiate chunks among them.

The source node takes a video stream in input which is split in several chunks using ffmpeg[2] libraries. The source at the PeerStream performs the input stream conversion. The source takes in input a video stream (e.g., video file or DVB-T stream) and split the content in chunks, filling the frame header and the video packet. The *Chunkiser* provides such a service. The chunks obtained from this operation are pushed in the *Chunk Buffer*, becoming available to the *Streaming Protocol* to be redistributed in the neighbors. The stream-to-chunk procedure is pictorially represented in figure 1.

The PeerStream supports several codes (e.g., ffmpeg1, ffmpeg2, theora, H261, and many other codec formats). Once the chunks are generated from the video stream, they are are timestamped from the source, and then distributed to neighbors. PeerStream actually provide a stream-to-chunk algorithm which provides one chunk for each frame. However, if the user wishes to use another rule (e.g., on either frame or time basis), he will write his favorite rule to plug into the PeerStream.

The user define the source node which in its simpler form takes its IP address and Port to listen for neighbors, and the stream to diffuse. Whenever the source is contacted by one peer, it adds the peer to its neighborhood which is used for chunks negotiation and exchange. The negotiation and the exchange policies are discussed later.

[2]http://ffmpeg.org

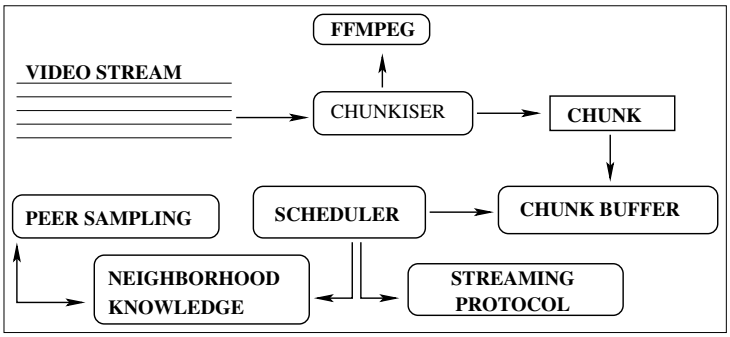

Figure 1: PeerStream architecture at the source: from video stream to chunks.

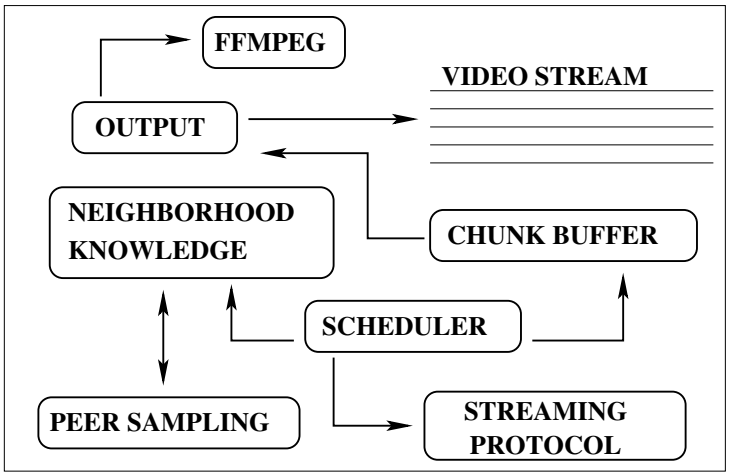

Figure 2: PeerStream architecture at the peer: from chunks to video stream.

2.3 Peer Sampling

Peers establish a set of connections with several nodes, building their neighborhood list. Each peer has a local view on its neighbors, which is updated through control messages between them (e.g., buffer maps).

There are several ways to keep the neighborhood and perform peers sampling. The PeerStream uses a gossip-based peer sampling protocol. It offers a random sample of the entire network, allowing peer to maintain a set of peers in its cache which represents the local view on the network. Peers exchange topology messages among them to update their neighborhood cache, eventually adding new peers.

The source node is the entry point of the network, because it is a well known peer. Peers get in touch with other peers through the source. In particular, the source exchanges a subset of its neighbors with new peers. Once they receives these information from the source, they send their buffer map to them, notifying their existence to such peers, populating their neighborhood, and increasing and updating buddy list of the other peers as a consequence. During the streaming session, newscast sends such topology message, advertising a subset of its neighborhood to its neighbors. The neighbor size as well as the number of peers to exchange are parameters to configure in the application.

2.4 Scheduling policies

The scheduler takes the decision about chunk and piece selection, in accordance to its local view of the network (i.e., information on its neighbors). In this work we show three examples of push-based applications, where the scheduler has to select the chunks to diffuse and which peer to push, namely chunk and peer selection algorithm, an the order of their execution as well. The order in which such algorithm are executed in the peer changes completely the distribution system. If a peer performs the chunk selection before the peer selection, it obviously selects the peers in function of the chunks selected. Such an approach is "content-oriented" because the priority is given to select which content has to be diffused, and the target peers are selected on the basis of selected chunks (e.g., priority to fresh content). On the other side, whenever the target peers are chosen first, the chunks will be limited to those needed by the selected peers. In this case the system is more "neighbor-oriented", where the peer tries to increase the quality of the neighborhood (e.g., selecting the most deprived peers, feeding old missed chunks), before the continuity of the content. Such analysis is out of the scope of this work, however, our applications may be used to investigate on such issues.

The literature proposes several scheduling algorithm for both peer and chunk selection [2, 1, 9]. Some of them are very simple (e.g., random blind) while other are smarter [], and require additional information on neighbors (e.g., bandwidth, degree, RTT).

In this paper we show P2P live streaming applications which adopt different peer and chunk selection policies. We recall that users may easily write the policies they wish, and plug them into the PeerStream. We show applications which use the following peer and chunk selection schemes:

- Random Blind peer (RBp): the target peer is picked uniformly at random among the neighborhood of the peer.
- Random Useful peer (RUp): the target peer is chosen uniformly at random among those neighbors that need some chunks owned by the peer.
- Random Blind chunk (RBc): the chunk to push is picked uniformly at random among those owned in the Chunk Buffer at the peer.
- Random Useful chunk (RUc): the chunk to push is chosen uniformly at random among those chunks owned in the Chunk Buffer at the peer and missed at some neighbors.
- Latest Useful chunk (LUc): the chunk to push is the more recent chunk among those owned in the Chunk Buffer at the peer and missed at some neighbors.

Algorithm 1 Peer Main Loop

```
1: at every peer P_i
2: δ_wait                                   ▷ define the waiting timeout
3: loop
4:     wait δ_wait
5:     if (Message from P_j) then
6:         Passive Mode
7:         Parse(message,P_j)
8:     else
9:         Active Mode
10:        Send Message to neighbors
11:    end if
12: end loop
```

2.5 Streaming Protocol

The Streaming protocol is the main intelligent part of the application. It takes decisions based on the local state of the peer and the local knowledge on its neighborhood, performing chunks negotiation and exchange with its neighbors.

The objective of the streaming protocol is to fill the chunk buffer at the peer, spreading the content among peers, and eventually retrieving the missed chunks. The GRAPES libraries provides the primitives to perform either push or pull strategies, information exchange and neighborhood sampling.

The streaming protocol performs the chunks negotiation and exchange with other peers. In particular it combines the information collected on its neighbors, with its local state, taking the decision that the user define in the application.

In each peer we identify two main threads: active and passive. The *active* thread is actively executed by the peer, which performs a set of action defined by the user in the main protocol such as performing chunk and peer selection for sending a chunk. The *passive* thread defines the peer behavior when receives a message from another peer, for instance when the peer receives a topology message it may update its neighborhood sampling. The combination of active and passive results in the peer protocol which defines the actions taken at the peer during the whole streaming session, namely the streaming protocol.

The PeerStream is characterized by a loop cycle where the peer, before the loop initializes its the libraries modules, while in the loop it listen for receiving data, and eventually handle the message received. The streaming protocol in managed by this entity which, triggers a given set of action in accordance to the message received, or, in general, to the behavior defined by the user. The code reported in algorithm 1 show both passive and active peer behavior. Passive means the peer receive one message from another peer and behaves in accordance to the message received and the corresponding policies defined. Active means that the node sends control messages or chunks to its neighbors

We present three different live streaming P2P applications obtained by our PeerStream by simply defining the scheduling policies and both active an passive behavior. Such applications are based on push, however, the PeerStream architecture allows users to define the peers behavior by writing and plugging the pull rules, or even both push and pull, that is also what we plan to do in the next future. ciao

DumbStreamer The DumbStreamer does not perform any kind of negotiation (i.e., no signaling messages), it provides only blind exchange: the application is briefly shown in algorithm 2. The streaming application waits for

receiving messages from other peers. When it receives a message, the application executes the passive part of the loop, parsing the message and updating its internal structures (e.g., topology message updates its local view). Otherwise, the node executes its active thread performing RBc and RBp, pushing the chunk to the target peer. This application is dumb and inefficient, policies are completely blind, and peers may eventually push also chunks to the source, or push the same chunks to the same peers.

Algorithm 2 DumbStreamer Main Loop

```
1: at every peer P_i
2: δ_wait
3: loop
4:     wait δ_wait
5:     if (Message from P_j) then                    ▷ Passive Behavior
6:         if (Topology) then
7:             Update Peers←P_j
8:             Reply to P_j
9:         else if (Chunk c_j Received) then
10:            ChunkBuffer ← c_j
11:        else
12:            Skip P_j Message
13:        end if
14:    else                                          ▷ Active Behavior
15:        c_t←RandomChunk(H)                        ▷ RBc
16:        P_t←RandomPeer(P)                         ▷ RBp
17:        Push c_t to P_t
18:    end if
19: end loop
```

RockStreamer In RockStreamer whenever a peer sends one chunk to one target peer, it sends also its buffer map, increasing the local view at the target peer. There is negotiation phase before chunks exchange, each peer uses the information collected through buffer map exchanges. The resulting application is reported in algorithm 3. The application waits in the main loop for receiving messages from other peers, thus executing either its active or passive thread. The peer may receive either a topology message to update its neighborhood, or a chunk to add in its chunk buffer. Otherwise,

the active thread is executed, and the peers uses the RUc and RUp, then it pushes the chunk to the target peer as well as its buffer map. RockStreamer is smarter than DumbStreamer: buffer map exchange allows peers to know which chunks are missed in which peers, and RUc and RUp reduce the probability of chunk duplication and wasting bandwidth.

Algorithm 3 RockStreamer Main Loop

```
1: at every peer P_i
2: δ_wait                                          ▷ define the waiting timeout
3: loop
4:     wait δ_wait
5:     if (Message from P_j) then                  ▷ Passive Behavior
6:         if (Topology) then
7:             Update Peers←P_j
8:             Reply to P_j
9:         else if (Buffer Map) then
10:            Update P_j Buffer Map
11:            Reply to P_j
12:        else if (Chunk c_j Received) then
13:            ChunkBuffer ← c_j
14:            P_j BufferMap ← c_j
15:            Send P_i Buffer Map to P_j
16:        else
17:            Skip P_j Message
18:        end if
19:    else                                        ▷ Active Behavior
20:        c_t=RandomUsefulChunk(H)                ▷ RUc
21:        P_t=RandomUsefulPeer(P)                 ▷ RUp
22:        SendBufferMap to P_t                    ▷ Update P_t Local View
23:        Push c_t to P_t
24:    end if
25: end loop
```

OfferStreamer In OfferStreamer whenever a peer sends one chunk to one target peer, it sends also its buffer map, increasing the local view at the target peer. There is negotiation phase before chunks exchange, each peer uses the information collected through buffer map exchanges. The resulting

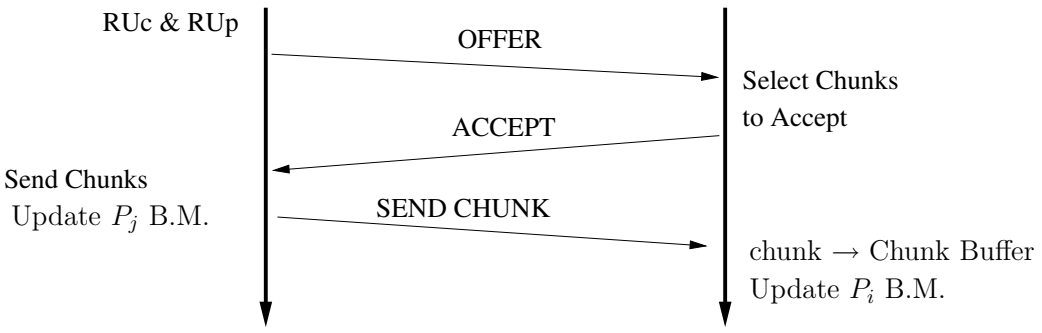

Figure 3: Offer-Accept protocol from P_i to P_j.

application is reported in algorithm 3. The application waits in the main loop for receiving messages from other peers, thus executing either its active or passive thread. The peer may receive either a topology message to update its neighborhood, or a chunk to add in its chunk buffer. Otherwise, the active thread is executed, and the peers uses the RUc and RUp, then it pushes the chunk to the target peer as well as its buffer map. RockStreamer is smarter than DumbStreamer: buffer map exchange allows peers to know which chunks are missed in which peers, and RUc and RUp reduce the probability of chunk duplication and wasting bandwidth.

3 Results

In this section we show real Internet results obtained by comparing three PeerStream.

We are interested in measuring following metrics: playout jitter, chunk loss ratio, number of missed chunks in the playout window, and the number of consecutive frames that are present in the output buffer. The playout jitter ..... The chunk loss ratio is the ratio between the number of chunks lost over the number of chunks which compose the streaming session. Such measure shows how chunks are spread into the system, thus how the application diffuses chunks in the network. The number of missed chunks in the playout window, gives a good idea on the number of holes in the output buffer, and thus on the quality perceived by user: lower are the missed chunks, higher is the quality perceived. Finally, the number of consecutive frames in the output buffer represents the number of frames that can be played before frame

Algorithm 4 OfferStreamer Main Loop

```
1: at every peer P_i
2: δ_wait                                          ▷ define the waiting timeout
3: loop
4:     wait δ_wait
5:     if (Message from P_j) then                  ▷ Passive Behavior
6:         if (Topology) then
7:             Update Peers←P_j
8:             Reply to P_j
9:         else if (OfferReceived) then
10:            Update P_j Buffer Map
11:            Send Accept to P_j
12:        else if (AcceptReceived) then
13:            Send Chunks to P_j
14:            Update P_j Buffer Map
15:        else if (Chunk c_j Received) then
16:            ChunkBuffer ← c_j
17:            P_j BufferMap ← c_j
18:            Send P_i Buffer Map to P_j
19:        else
20:            Skip P_j Message
21:        end if
22:    else                                        ▷ Active Behavior
23:        SendOffer RUp and LUc
24:    end if
25: end loop
```

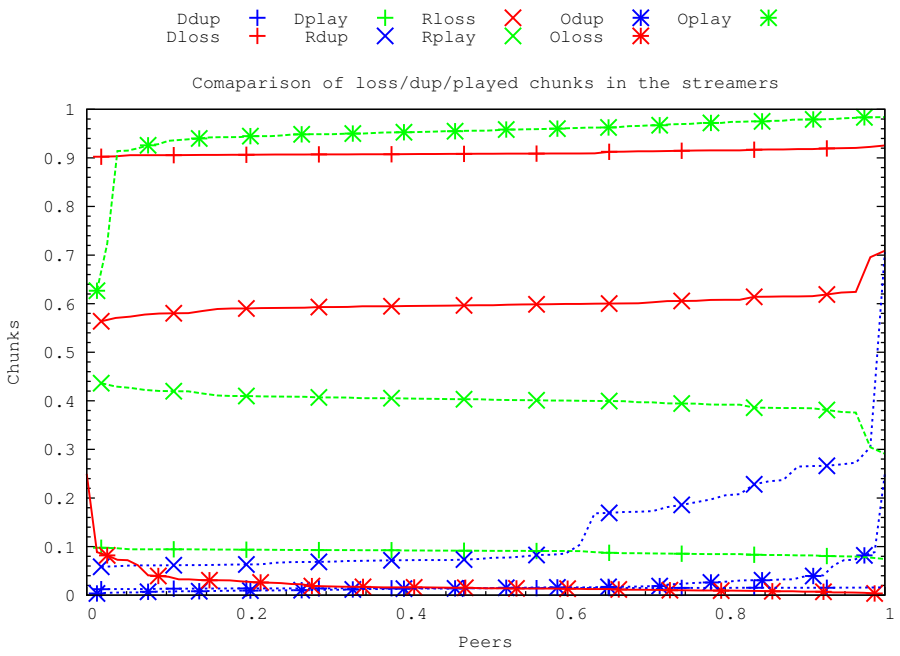

Figure 4: Comparing chunks loss, duplicated and played among DumbStreamer, RockStreamer, and OfferStreamer.

loss, if no more frames will arrive.

In our experiments, there are two nodes equipped with DVB-T[3] cards that are the sources which streams the digital channel. The stream is encoded using ffmpeg libraries, to define the codec, the bitrate, the image size and many other options. and then given to the PeerStream as standard input. Note that the source may also take video file to stream. There are 60 nodes geographically distributed in Europe, among NAPA-WINE partners. Each node runs only one instance of the streamer.

In figure 4 we compare the three streamers on chunks loss, duplicated and played. The number of push performed per second is denoted with **C**, while the size of the chunk buffer on which the trading is done is by peers is denoted with **B**, and the size of the output buffer used to play the chunks and eventually count the losses is identified with **O**.

In this configuration the streamers performs 25 push per seconds (i.e., C=25), the size of the chunk buffer is 100 (i.e., B=100), while the size of the

[3]Digital Video Broadcasting - Terrestrial

output buffer is 100 (i.e., O=100). The DumbStreamer experiences many losses due to blind "dumb" diffusion schema, without any control message for chunks negotiation, therefore the played chunks are below 0.1% of the chunks composing the streaming session, while the chunks duplicated are low for the same reason of chunks played out: many losses, and those received are played out and a few of them are duplicated. The RockStreamer reaches higher playout rate than the DumbStreamer, around 0.4% while the losses are still high, around 0.6%, and chunks duplicated are also higher due to RUc and RUp without signaling messages: the target node passively receives misses chunks many times from its neighbors. The effect of the signaling messages is clear with OfferStreamer. In this case each node proposes a set of chunks to its neighbors using LUc and RUp, pushing chunks only to those nodes which accept the offer. The number of chunks played out, apart few nodes, is over 0.9%, keeping the number of chunks loss lower than 0.1% for the whole session, limiting also the number of chunks duplicated below 0.09%. The OfferStreamer outperforms both DumbStreamer and RockStreamer, it uses signaling messages to limit chunks duplications and losses.

4 Conclusions

In this paper we present three real streamers that live in the real world, dealing with constraints of real network. This streamers may be used to simulate P2P streaming protocols in the real Internet.

We start from DumbStreamer, a very dumb and simple streamer showing how it works, then we present RockStreamer, which uses a few signaling messages to exchange buffer map and reduce the useless transmission. Finally, we show the OfferStreamer which is smarter than the others, it uses more signaling achieving better results, increasing the number of useful chunks consumed in the playout, reducing the loss chunks and also the duplicated chunks.

References

[1] L. Abeni, C. Kiraly, and R. Lo Cigno. On the optimal scheduling of streaming applications in unstructured meshes. In *IFIP Networking*, 2009.

[2] T. Bonald, L. Massoulié, F. Mathieu, D. Perino, and A. Twigg. Epidemic Live Streaming: Optimal Performance Trade-offs. *SIGMETRICS Perform. Eval. Rev.*, 36(1):325–336, 2008.

[3] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-Bandwidth Multicast in Cooperative Environments. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 298–313, New York, NY, USA, 2003. ACM.

[4] B. Li, Y. Qu, Y. Keung, S. Xie, C. Lin, J. Liu, and X. Zhang. Inside the new coolstreaming: Principles, measurements and performance implications. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, 2008.

[5] N. Magharei and R. Rejaie. Prime: Peer-to-peer receiver-driven mesh-based streaming. *Networking, IEEE/ACM Transactions on*, 17(4):1052 –1065, aug. 2009.

[6] L. Massoulie, A. Twigg, C. Gkantsidis, and P. Rodriguez. Randomized decentralized broadcasting algorithms. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 1073 –1081, 6-12 2007.

[7] V. N. Padmanabhan, H. J. Wang, P. A. Chou, and K. Sripanidkulchai. Distributing Streaming Media Content Using Cooperative Networking. In *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 177–186, New York, NY, USA, 2002. ACM.

[8] D. Tran, K. Hua, and T. Do. ZIGZAG: An Efficient Peer-to-Peer Scheme for Media Streaming. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, volume 2, pages 1283–1292vol.2, 30 March-3 April 2003.

[9] M. Zhang, Q. Zhang, L. Sun, and S. Yang. Understanding the power of pull-based streaming protocol: Can we do better? *Selected Areas in Communications, IEEE Journal on*, 25(9):1678 –1694, december 2007.