Chapter 20
PPPC—Peer-2-Peer Streaming and Algorithm for Creating Spanning Trees for Peer-2-Peer Networks

Amir Averbuch, Yehuda Roditi, and Nezer Jacob Zaidenberg

Abstract We describe a system that builds peer-2-peer multicast trees. The proposed system has a unique algorithm that incorporates real-time and priority-based scheduler into a graph theory with robust implementation that supports multiple platforms. Special consideration was given to conditional access and trust computing. We also describe the system design as well as the computational aspects of processing the graphs used by the system.

20.1 Introduction

The bandwidth cost of live streaming prevents cost-effective broadcasting of rich multimedia content to Internet users.

For Internet streaming, the old-fashioned client-server model puts a considerable cost burden on the broadcaster. In the client-server model, a client sends a request to a server and the server sends a reply back to the client. This type of communication is at the heart of the IP [11] and TCP [12] protocol, and most of UDP [10] traffic as well. In fact, almost all upper layers of communication such as HTTP [5], FTP [1], SMTP [13], etc., implement the client-server models. The client-server communication model is known as unicast where a one-to-one connection exists between the client and the server. If ten clients ask for the same data at the same time, then ten exact replicas of the same replies will come from the server to each of the clients (as

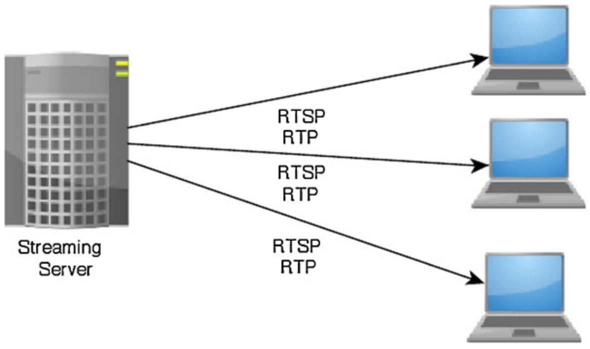
A. Averbuch (✉)
School of Computer Science, Tel Aviv University, P.O. Box 39040, Tel Aviv 69978, Israel
e-mail: amir@math.tau.ac.il

A. Averbuch · N.J. Zaidenberg
Department of Mathematical Information Technology, University of Jyväskylä, P.O. Box 35 (Agora), 40014 Jyväskylä, Finland

N.J. Zaidenberg
e-mail: nezer.j.zaidenberg@jyu.fi

Y. Roditi
Academic College of Tel-Aviv-Yaffo, P.O. Box 8401, 61083 Tel Aviv, Israel
e-mail: jr@mta.ac.il

Fig. 20.1 Unicast streaming



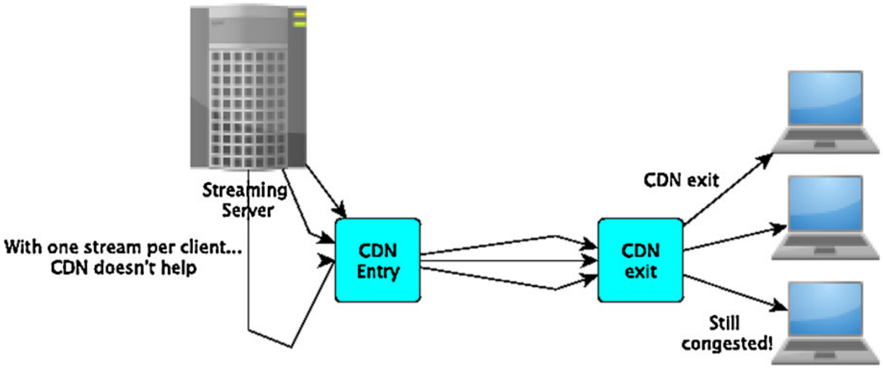


Fig. 20.2 CDN currently does not provide the solution

demonstrated in Fig. 20.1). This model remains the same regardless of the number of concurrent requests from the same number of unique clients, placing additional stress on the server with each additional user.

Furthermore, the problem exists to a much greater extent in live streaming scenarios with large crowds of listeners such as sport events, etc., as caching techniques such as proxies do not work with live streaming.

These problems also arise even when Content Delivery Networks (CDNs) are used for replicating static content to other servers at the edge of the Internet. Even when CDNs are used, every client is still served by one stream from a server, resulting in the consumption of a great amount of bandwidth (see Fig. 20.2). These infrastructure and cost challenges place a significant hurdle in front of existing and potential Web casters. While the media industry is seeking to bring streaming content with TV-like quality to the Internet, the bandwidth challenges often restrict a feasible, profitable business model.

In order to reduce the dependence on costly bandwidth, a new method of Internet communication called "multicast" was invented. Rather than using the one-to-one model of unicast, multicast is a "one-to-selected-many" communication method. However, multicast is not available on the current Internet infrastructure IPv4 and

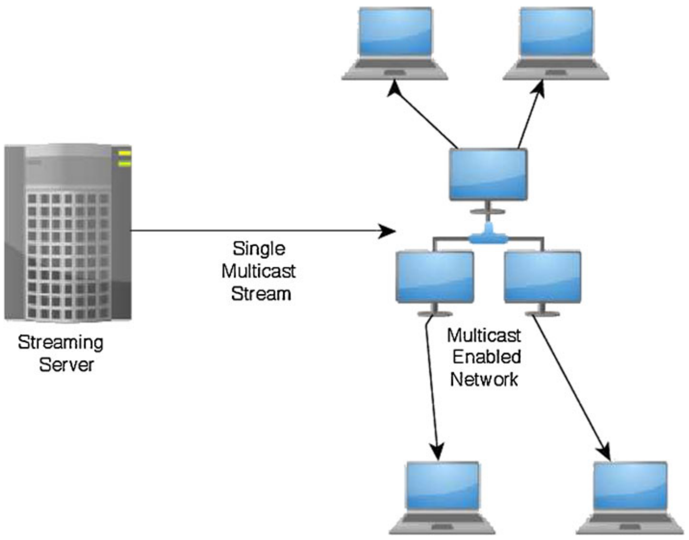


Fig. 20.3 Multicast streaming could provide a solution

may never be available outside private networks. An example of what multicast streaming looks like is demonstrated in Fig. 20.3.

A solution commonly proposed is to deploy Internet users as "broadcasters" using peer-2-peer [4, 7, 15] connections as a type of CDN.[1]

In this paper we describe our system for peer-2-peer streaming and our algorithm for handling network events.

20.2 System Design

The software industry has already anticipated the need for cost-effective, high-quality streaming and has developed applications that support multicast. Our peer-2-peer streaming system, called PPPC (Peer-2-Peer Packet Cascading) bypasses the lack of multicast in IPv4 Internet by providing multicast-like capabilities via peer-2-peer, and allows the use of the already available multicast software.

The concept of peer-2-peer streaming is a distributed architecture concept designed to use the resource of a client's (desktop computer) upstream in order to alleviate congestion in the broadcaster streaming server. (Using the client upstream does not affect its ability to surf or download files. The upstream resource is usually idle for most clients not involved in peer-2-peer activity (such as bittorrent [3]).)

In a peer-2-peer streaming system, the server only serves a fraction of selected simultaneous clients requesting the same stream and turns them into relay stations.

[1]Content delivery network.

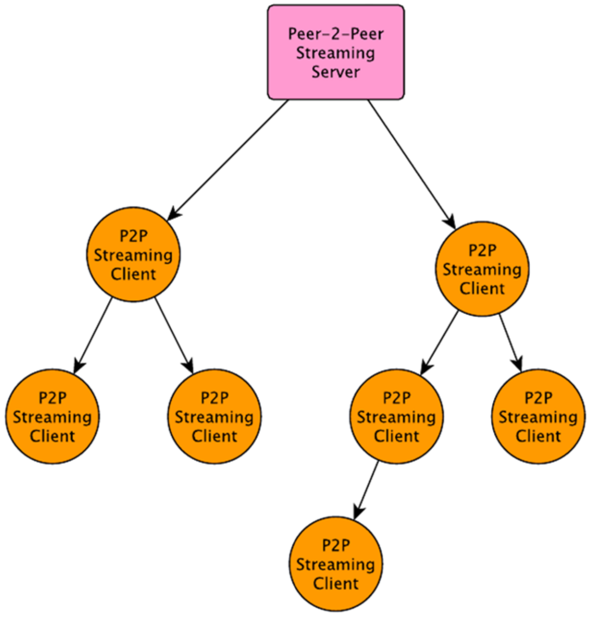


Fig. 20.4 Only peer-2-peer streaming solves the streaming problem on the Internet

Hereafter, the other clients who are requesting the same data will be served from one of the clients who received the stream first.

The clients shall only maintain a control connection to the server for receiving control input and reporting information. Also, we shall use every client as a sensor, to detect stream rate drops, to report the problem, and to complement the missing packet from either the PPPC router or another client. It is vital to detect any streaming issues in advance before the media player has started buffering or the viewer has noticed anything. Therefore, by following the peer-2-peer streaming concept and serving a fraction of the users, the server can serve a lot more users with the same bandwidth available. This is shown in Fig. 20.4.

Peer-2-peer packet cascading, or PPPC, is an implementation of the peer-2-peer concept to the streaming industry. PPPC provides a reliable multicasting protocol working on and above the standard IP layer. A PPPC system consists of the PPPC router and the PPPC protocol driver. The PPPC router stands between a generic media server, such as an MS Media server, a Real Media server or a QuickTime server, and the Viewers (see Fig. 20.5). The PPPC driver is in charge of the distribution of data and the coordination of the clients.

In a PPPC live stream, the PPPC router will receive a single stream from the media server and will route it directly to several "root clients". These clients will then forward the information to other clients and so on and so forth. Users connecting to each other will be relatively close network-wise. In this method, the data is cascaded down the tree to all the users while the PPPC router only serves directly (and pays bandwidth costs) for the root clients. As users that join and leave, the trees are dynamically updated. Moreover, the more users join the event, the more the PPPC

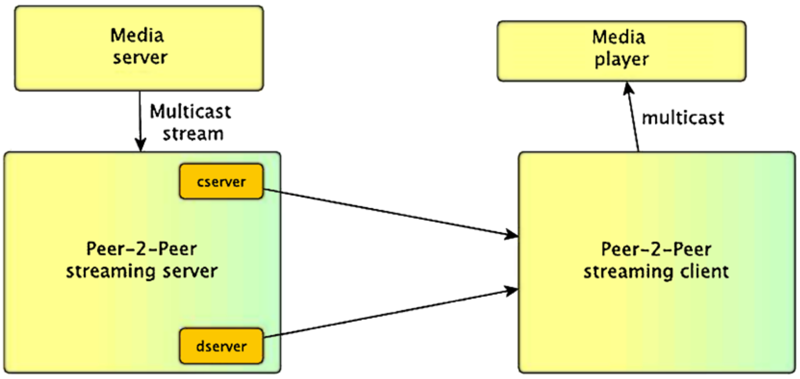


Fig. 20.5 PPPC data flow

router can build better trees saving even more, eliminating the financially undesirable linear connection between the cost of streaming and the number of users.

20.3 PPPC System Overview

Peer-2-peer packet cascading is a system designed to provide audio and video streaming clients with the capability to receive data from other clients and relay them to clients. The PPPC system is divided into a *PPPC router* and a *PPPC driver*. The PPPC router contains two logical components: the *Coordinating Server* (also called CServer) and the *Distributing Server* (also called DServer).

The PPPC driver installed on a client workstation (any standard PC) consists of thin client software that handles the reception and relay of the packets, and also "feeds" them to any media player. The client does not interact with a media player, it only delivers packets to the media player.

The coordinating server (CServer) is a command and control system in charge of all PPPC drivers listening to a single stream. The CServer is responsible for all the decisions in the system: For example, for a given client, from which client should it receive data and to which client should it transfer data, how should the tree be rebuilt after a new client arrives, what to do if a client in the middle of the tree is disconnected, and what happens when any given client reports on problems with receiving stream from his parent.

The distributing server (DServer) is a data replication and relay system. The DServer receives a multicast (data-only) stream and encapsulates the data in a PPPC format (recognized by PPPC driver). The DServer delivers the encapsulated packets to the roots of PPPC clients' trees (root clients). The CServer decides who the root clients are.

20.3.1 Data Flow in the PPPC System

In a PPPC system, a PPPC router must receive a data-only stream (i.e. no meta-data) from a generic media server and is responsible for delivering the stream to clients. In some way, a PPPC router acts very much like a multicast router. (A data-only stream is required because a stream with metadata will require the decoding of the stream and the right metadata to be sent to each of the clients thus missing the system's goal of generality.)

Most standard media servers can provide data-only stream, either directly or via a "multicast" option. The DServer in our system will receive the multicast stream or other data-only stream and pass it forward to the root clients. The PPPC drivers running on root clients' work stations pass the stream to other drivers on other clients. Therefore, each client acts as a small server, reusing the DServer code for this purpose.

When a PPPC driver, regardless of whether the PPPC driver also forwards the stream to other clients or not, receives the stream, it will forward it to the media player pretending to be the original media server using multicast or a fake IP if necessary. This is not real network traffic, but local traffic on the local host blocked in the kernel. Then, the media player will receive the stream and will act as if it received the stream directly from the media server. The PPPC driver will send a stream just like a media server.

Thus, the media server sends a standard (usually multicast) data stream, and the media player receives a standard stream. This enables the system to work with any media server, any media player and any codec, etc., without the need to have any specific integration.

20.3.2 Detailed Description of the System Components

One instance of the server handles one media stream. Multiple instances of the server are possible in order to handle more than one stream. Parts (entities) within the server communicate with each other by TCP enabling them to run on several computers.

20.3.2.1 Distributing Server (DServer)

The distributing server transfers the stream contents to root clients and serves as a backup source for clients (the DServer is also a backup server). It contains two physical components:

1. A single *receiver*, which gets the raw data from a media server via multicast or UDP. The DServer Receiver then encapsulates the data arriving from the media server in PPPC packets (recognized by PPPC clients).

2. One or more *distributors* which receive the information directly from the receiver and serve the root clients.

The distributors share packet relay and a connection code with the drivers, but they are the only entities that receive the stream directly from the receiver. This division is suggested in order to receive optimal scalability, and it allows the deployment of the distributors across several CDN sites.

20.3.2.2 Coordinating Server (CServer)

The coordinating server maintains the control connection with every client. It decides which clients connect between them, i.e., it constructs the PPPC tree. Our algorithm is implemented within the CServer. The CServer updates dynamically the PPPC tree on such events as connection/departure of clients, unexpected disconnections, etc.

The CServer, similar to the DServer, also contains two components:

1. A single centralized *main* module where all the users' (of a single stream) data is saved. The main module provides all the logic and decisions to the system.
2. One or more *proxies* who receive client connections and pass requests and responses to/from the CServer's main module.

In a large-scale PPPC system, where several proxies can exist, each maintains a connection to a large group of clients. The main module is the only place where complete information and decision making regarding all clients is kept for decision making regarding the clients' tree. Reports on the clients' connections and disconnections are handled by the main module.

20.3.2.3 The PPPC Driver

The PPPC driver is a very light client which consumes very little system resources apart from the relatively free upstream. It is the only client side software and it communicates with the CServer and DServer components and the PPPC drivers in other clients.

20.3.3 Viewing a Stream with PPPC—Life Cycle

This life cycle assumes that the clients select the stream using WWW:

1. The user accesses a page on the WWW which provides him with stream information.
2. The file is silently downloaded to the user's computer.

3. The PPPC driver parses the file which includes a standard media player activation file. The PPPC driver reads the CServer and DServer IP address as well as other parameters and invokes the standard media player to handle the stream.
4. The client connects simultaneously to the CServer and DServer.
5. The DServer sends data to the client which is immediately displayed to the user.
6. In a certain event[2] a CServer decides to rebuild the PPPC client trees.
7. The CServer sends the client messages with information about its new stream source (another client or the DServer) and possibly the address of other clients that should be served the stream.
8. A client connects to specified clients and starts to receive information from the parent and relays it to its children. The arrival of data is viewed through the media player to the user.
9. The CServer may decide during the operation to rebuild the tree and sends again the corresponding messages to the client, which disconnects its older connections and creates newer ones.
10. When the user decides to stop viewing the stream, the PPPC client recognizes it and sends the message "I'm going to disconnect" to the CServer and quits. Meanwhile, the CServer updates the clients' tree if necessary.

20.3.4 Maintaining a Certain Level of QoS

In order to maintain and guarantee a certain level of QoS, we will add a stream rate detection unit to every client. The stream rate is published by the CServer when clients join the stream. If a client detects that the stream rate has dropped below a certain level, he will be connected to the DServer to complement the missing packets or as an alternative stream source. Numerous reasons cause the packet rate to be dropped: parent disconnection (the packet rate drops to zero), a sudden drop in the packet rate when the parent uses his upstream to send an email, or a high CPU load on the parent machine. He might also report that his previous parent was a "bad parent"; then the CServer will not assign new children to a "bad parent".

The switch between parents and going to the DServer should be done very fast (within the streaming buffer time found in the client). If all packets arrive before the buffer expires, the user will never notice the switch between the parents.

We will describe the exact method in which bad parents are detected in Sect. 20.5.

20.4 Avoiding Flat Trees, Distress, and Guaranteeing a Certain Level of QoS

In this section we describe all the system engineering issues which are connected to the appearance of what we call "flat trees". Flat trees are trees that have a very

[2]For example, after some other new clients have arrived, or old clients have disconnected.

large number of root clients compared to the total number of clients and a very small number of peer-2-peer connections. We will also describe how these issues are solved.

We encountered several reasons for having extremely flat trees, and most of them were related to our goal to achieve a high level of QoS. This section describes our solution, which provides high streaming quality to clients who can receive the stream. This is done while we maintain a peer-2-peer connection with a high bandwidth saving ratio. We realized that QoS and the flat trees problem are closely connected. Several problems have been identified:

- Parents that could not serve clients constantly received new clients which caused a decrease in QoS.
- Parents that were declared bad parents never received new clients and caused flat trees.
- Clients that were unable to view the stream pass from parent to parent declared them all to be bad parents (hereby bad clients). Such a client can easily mark all clients in a tree as bad parents which will surely result in a flat tree.
- In case of a transmission problem in the upper layers of the tree, many clients in the lower layers of the tree did not receive the stream and reported their parents to be bad parents. This caused the multiplicity of bad parents.
- Clients that detected problems were generally not the direct children of the clients that caused the problem. Thus, many of the clients were declared to be bad for no apparent reason. (Same as above!)
- Due to the above conditions, the lower layers of the tree received poor-quality stream.

As we can see from above, most of the problems occurred due to faulty behavior when served by an unsatisfying packet rate. We shall hereby call this situation *distress*.

20.5 Bad Parents and Punishments

When a client reports to the CServer that his parent does not provide him with a sufficient packet rate, the CServer will mark the parent as a bad parent. In this case the bad parent's maximum number of children is set to its current child number.

The client that reported the bad parent will also connect to the DServer either to compliment the missing packets or to replace its current bad parent with the DServer. Therefore, a bad parent cannot have any more children. We will not disconnect any of the other children he already had. We will allow new children to replace one of the old ones if they were disconnected. If the previous client did not have any children then he will not have children anymore.

We "punished" bad parents so harshly to prevent any connection of new clients to them. Thus, we avoided a situation where a client connects to several "bad parents" before receiving the stream. Thus, the QoS is degraded. We provide another punishment function that either produces no result, i.e. a client kept connecting to bad parents regardless of the punishment or the same result. The goal was that the client will never connect to bad parents.

The isolation of bad parents plays a very important role in guaranteeing a high QoS. We realized that a stream is never disrupted in real-world scenarios by the sudden disconnection of parents or fluctuations in their upstream. However, bad parents were often one of the main reasons for having flat trees. The clients could not find themselves a suitable parent because all possible parents were marked as bad parents and could not accept any new children.

Therefore, we give a chance for a bad parent to recover. We set a punishment time stamp where the punishment time is assigned to each of the bad parents. To recover from this situation we introduce bad parents' rehabilitation process (see Sect. 20.5.1). There are many temporary situations such as sending and e-mail which hogs the upstream, starting Microsoft Office, which causes a CPU surge for a couple of seconds, and many more. A "bad parent" can recover from the "temporary" situations. This should not prevent him from future productive service to clients.

20.5.1 Bad Parent Rehabilitation

There are many reasons for punishing a client and turning it into a bad parent. Nevertheless, we realized that the punishment mechanism on the PPPC network should be temporary. We shall associate a time stamp with the punishment time when a client is punished. After a period of time we will rehabilitate the parent and allow it to receive new connections.

The rehabilitation thread is in charge of bad parents rehabilitation. The suggested time period for rehabilitation is between 5 and 15 minutes.

20.5.2 Distress Logic: Marking of Bad Parents

A distress state is the state in which a client does not receive enough information within a PACKET_RATE_CALCULATION_PERIOD. There are two variables that dictate a distress state:

1. Parent distress is a boolean variable that indicates whether the parent sent any indication of entering into a distress state.
2. Current distress is a variable that may be equal to either no-distress, light distress, or major distress.

These variables introduce six different distress states:

No distress: The standard state. The packet rate is fine and the father has not informed otherwise.

Light distress: The state that occurs when a client receives less packets than DISTRESS_MIN_PACKET_RATE and there is no notification from the parent that he reached a similar state.

Parent distress: The parent indicates that he is in a light distress state but the information still flows fine.

Parent and light distress: Indicates that both the current client and its father experienced light distress.

Major distress: Indicates that the current packet rate is below MIN_PACKET_RATE.

Major and parent distress: Indicates that the current packet rate is below MIN_PACKET_RATE and the parent is also experiencing difficulties.

20.5.2.1 Entering into a Distress State

A packet rate threshold, DISTRESS_MIN_PACKET_RATE, is used to determine the upper bound of entering into a "light distress" state. A client in "light distress" does not complain about a bad parent, but opens a connection to the DServer to complement missing packets from there. The client only sends a "Bad Parent" message when the packet rate reaches MIN_PACKET_RATE, then it connects to the DServer (hereby major distress).

When a client enters into a distress state it will inform its children about its state. When a client enters into a major distress it will not report his parent as a bad parent if his parent is also in a distress state.

20.5.3 Bad Client

Some clients, for whatever reasons may be, are simply unable to receive the stream. Reasons may vary from insufficient downstream, congestion at the ISP or backbone, busy CPU, poor network devices or others.

Those clients will reach a major distress state regardless of the parent they were connected to. An "innocent" parent will be marked as a "bad" parent. In order to prevent this from happening we add new logic to the PPPC driver.

The client should stop complaining about bad parents when the problem is probably in its own ability to receive the stream.

20.6 The Algorithm

20.6.1 The Structure of the Internet—from Peer-2-Peer Streamer Perspective

Each of the Internet nodes viewing the stream comes from a location with an Internet connection. Often such organization is the user's home. The system we developed is capable of detecting multiple nodes in the same location (such as two users in the same LAN or home) via multicast messages. The system ensure that at any location only one user will stream in or out of the location. This way we eliminate congestion and as a by-product guarantee that only one user in each location is visible to the algorithm.

Connections between Internet nodes tend to be lossy (it is typical to have about a 1 % packet loss) and add latency. Furthermore, not all connections are equal. When connecting to a "nearby" user, we can expect significantly less latency and packet loss then when connecting to a "far" user. Latency specifically is increased and can differ from a few milliseconds to hundreds of milliseconds depending on the distance.

We will now define *nearby* and *far* in Internet terms. Each Internet connection belongs to an "autonomous system". An autonomous system is usually an ISP[3] and sometimes a large company (such as HP or IBM) that is connected to at least two other autonomous systems. Two users from the same autonomous systems will be considered to be nearby each other.

We have created two additional levels of hierarchy. Below the autonomous system we have created a "subnet" level. Each IP address belongs to a subnet that defines a consistent range of IP addresses. Autonomous systems get their IP range as a disjoint union of subnets. Often each subnet belongs to different location that can be very far from each other (such as the east and west coast of the US). Thus, when possible, we prefer to connect to a user from the same subnet.

Autonomous systems are interconnected. Some autonomous systems can be considered "hubs" connected to many other autonomous systems. We have created "autonomous system families" centered on the hubs (containing all the autonomous systems that connect to the hub). When a user from the same autonomous system cannot be found, we will prefer a user from the same autonomous system family.

An autonomous system usually belongs to more than one autonomous system family. Thus, when choosing clients to connect to each other, we prefer clients that share a subnet. If none is found, we prefer clients that belong to the same autonomous system. If none is found, we prefer clients that belong to the same autonomous system family. Clients that have no shared container will be considered far and will not connect to each other.

20.6.2 Minimum Spanning Trees of Clients

The algorithm uses containers that hold all clients in a certain level. Since we can consider all clients that share a container and does not share any lower level container to be of an identical distance from each other, we can store all clients in a container in "heap-min" and only consider the best client in each heap to connect

[3]Internet service provider.

to. The best client will be considered using the distance from the source and the best available uplink (the best uplink considering other peers served by the client).

Algorithm 20.1 strives to maintain the graph as close to the MST as possible while responding to each new request (vertex joining, vertex removal) in nanoseconds. Indeed, our solution often involves finding an immediate solution such as connecting directly to the source and improves the solution over time until it reaches the optimal state. The proposed algorithm can handle very large broadcast trees (millions of listeners) in a nearly optimal state. As the server load increases (with more users), we may be further away from the optimal solution but we will not be too far and the stream quality for all users will be well tested.

Algorithm 20.1 Real time graph analysis

```
 1: Read subnet to autonomous systems and autonomous systems to autonomous
    systems family files. Store information in a map.
 2: Create global data structure spawn interface and parents rehabilitation thread
    and interface thread.
 3: while Main thread is alive do
 4:    if There are new requests then
 5:       handle new request, touch at most 100 containers.
 6:       Inform interface thread when you are done.
 7:    else
 8:       if there are dirty containers then
 9:          clean at most 100 containers
10:          inform interface thread when you are done
11:       else
12:          wait for new request
13:       end if
14:    end if
15: end while
```

Clean and *dirty* in the algorithm sense are containers that are optimal and containers that are sub optimal for any reason.

For example, let us assume a client has disconnected. We will try to handle the disconnection by touching no more than 100 containers, let's say by connecting all the "child" nodes directly to the source. We will mark all the containers containing the nodes as dirty. At some point we will clean the container and fix any non-optimal state.

20.7 Related Systems

The authors have been involved with peer-2-peer streaming company vTrails Ltd that operated in peer-2-peer streaming scene in 1999–2002. (vTrails no longer operates.) Many of the concepts and system design may have originated from the authors' period with vTrails though the system has been written from scratch.

In recent years several peer-2-peer streaming systems have been proposed, some with a similar design. Our main contribution in this work is the peer-2-peer algorithm designed to calculate graph algorithms based on a real-time approach. Some features of the system approach such as the handling of distress state are also innovative.

ChunkySpeed [14] is a related system that also implements peer-2-peer multicast in a robust way. Unlike PPPC, ChunkySpeed does not take Internet distances into account.

ChunkCast [2] is another multicast over peer-2-peer system. ChunkCast deals with download time which is a different problem altogether. In streaming, a guaranteed constant bitrate is required. This requirement does not exist in content download which is not vulnerable to fluctuation in download speed and only the overall download time matters.

Climber [9] is a peer-2-peer stream based on an initiative for the users to share. It is our experience that user sharing is not the main problem but rather broadcasters willing to multicast their content on peer-2-peer networks. Thus Climber does not solve the real problem.

Microsoft [8] researched peer-2-peer streaming in a multicast environment and network congestion but had a completely different approach which involved multiple substreams for each stream based on the client abilities.

Liu et al. [6] recently researched peer-2-peer streaming servers' handling of bursts of crowds joining simultaneously which is handled by the algorithm easily thanks to its real time capabilities.

References

1. Bhushan AK (1971) File transfer protocol. RFC 114, April. Updated by RFC 133, RFC 141, RFC 171, RFC 172
2. Chun BG, Wu P, Weatherspoon H, Kubiatowicz J (2006) ChunkCast: an anycast service for large content distribution. In: Proceedings of the 5th international workshop on peer-to-peer systems (IPTPS)
3. Cohen B (2003) Incentives build robustness in BitTorrent. BitTorrent Inc., May. http://www2.sims.berkeley.edu/research/conferences/p2pecon/papers/s4-cohen.pdf
4. Feng C, Li B (2008) On large-scale peer-to-peer streaming systems with network coding. In: Proceedings of the 16th ACM international conference on multimedia, pp 269–278
5. Fielding R, Gettys J, Mogul J, Frystyk H, Masinter L, Leach P, Berners-Lee T (1999) Hypertext transfer protocol—HTTP/1.1. RFC 2616 (draft standard), June. Updated by RFC 2817, RFC 5785, RFC 6266
6. Liu F, Li B, Zhong L, Li B, Niu D (2009) How P2P streaming systems scale over time under a flash crowd? In: Proceedings of the 8th international workshop on peer-to-peer systems, IPSTS 09
7. Liu Y (2007) On the minimum delay peer-to-peer video streaming: how realtime can it be? In: Proceedings of the 15th international conference on multimedia, pp 127–136
8. Padmanabhan VN, Wang HJ, Chou PA (2005) Supporting heterogeneity and congestion control in peer-to-peer multicast streaming. In: Voelker GM, Shenker S (eds) Peer-to-peer systems III: third international workshop (IPTPS 2004). Lecture notes in computer science, vol 3279. Springer, Berlin, pp 54–63

9. Park K, Pack S, Kwon T (2008) Climber: an incentive-based resilient peer-to-peer system for live streaming services. In: Proceedings of the 7th international workshop on peer-to-peer systems, IPTPS 08
10. Postel J (1980) User datagram protocol. RFC 768 (Standard), August
11. Postel J (1981) Internet protocol. RFC 791 (Standard), September. Updated by RFC 1349
12. Postel J (1981) Transmission control protocol. RFC 793 (Standard), September. Updated by RFCs 1122, 3168, 6093
13. Postel J (1982) Simple mail transfer protocol. RFC 821 (Standard), August. Obsoleted by RFC 2821
14. Venkataraman V, Francis P, Calandrino J (2006) ChunkySpread: Multi-tree unstructured peer-to-peer multicast. In: Proceedings of the 5th international workshop on Peer-to-Peer systems, IPTPS 06
15. Zaidenberg NJ (2001) sFDPC—a P2P approach for streaming applications. MSc thesis, Tel Aviv University