Scalable Multi-Source Video Streaming Application over Peer-to-Peer Networks

Carlos Hernández Gañán

Department of Telematics Engineering

Polytechnical University of Catalonia

A thesis submitted for the degree of

Master of Science

9th September 2009

I thank the Father, Lord of heaven and earth, for hiding the things I learned during this thesis from the learned and wise, and revealing them to me.

Acknowledgements

This thesis arose in part out of years of research that has been done before I came to Telematics Engineering group. By that time. I have worked with a great number of people whose contribution in assorted ways to the research and the making of the thesis deserved special mention. It is a pleasure to convey my gratitude to them all in my humble acknowledgment.

First of all, I would like to express my gratitude to my supervisor, José Luis Muñoz, whose expertise, understanding, and patience, added considerably to my graduate experience. I appreciate his vast knowledge and skills in many areas, and his assistance in writing reports. Above all and the most needed, he provided me unflinching encouragement and support in various ways. His truly scientist intuition has made him as a constant oasis of ideas and passions in science, which exceptionally inspire and enhance my growth as a student, a researcher and a scientist want to be.

I am deeply indebted to Jorge Mata. Without his guidance, support and good nature, I would never have been able to develop this thesis successfully. I benefited greatly from his ideas and insights. His involvement with his originality has triggered and nourished my intellectual maturity that I will benefit from, for a long time to come.

Some debts are hard to put into words. My research colleagues Javier Parra-Arnau, Óscar Esparza, Juan José Alins all know why their names are here.

My last, but not least gratitude is for my parents, it is difficult to find words to express my gratitude and thanks to both of you.

I realize that not all people who contributed either directly or indirectly to my study are mentioned in this page. From the deepest of my heart, I would like to thank all of you...

Abstract

Live Streaming consists in distributing live media (video and audio) to large audiences over a computer network. Providing a live streaming service over the Internet presents many challenges: the application must respect the timing and quality constraints imposed by the nature of live media and by user expectations while struggling with the practical limitations due to the best effort properties and unpredictable dynamics of the Internet. Because of the limited deployment of native IP multicast, an Internet-based live streaming application with a global scope can only rely on end-to-end network primitives, such as unicast connections. The traditional client-server approach to live streaming has a serious scalability limit, as the upload capacity requirement at the server grows linearly with the user population.

A peer-to-peer (P2P) solution has the big advantage of seamlessly scaling to arbitrary population sizes, as every node that receives the video, while consuming resources, can at the same time offer its own upload bandwidth to serve other nodes. In theory, if every node contributed on average at least as much as it consumed, the P2P system would have enough resources to grow indefinitely. The P2P revolution permitted to overcome the last bandwidth bottlenecks represented by the limited server capacities, favoring the diffusion of even more bandwidth demanding applications. It is an attractive idea to spread media data using a peer-to-peer scheme; however it remains unclear whether this technology can really deliver real-time media to a large number of users in global scale. In this work we firstly present an overview of

the existing P2P approaches to live streaming, highlighting their features, with all their advantages and drawbacks. Then, we introduce Peercast, a tree-structured P2P system for audio streaming quite well extended in some countries, focusing in its implementation. Based on this framework, in this thesis we design and implement NeuroCast an unstructured P2P application for live video streaming.

We list the aspects that are addressed in the NeuroCast definition in order to realize a real world prototype of the system. We describe the researches done about the issues we have to deal with in a real implementation, like resources management, multi-source downloading and load redistribution mechanisms. We continue with the protocols and algorithms defined to solve these problems. We also present the modifications brought to the core algorithms in order to make them working in a real world environment.

In P2P live streaming using unstructured mesh, packet scheduling is a determining factor on overall playback delay. In this thesis, we propose and optimize a scheduling algorithm to minimize scheduling delay. Our scheduling is predominantly push in nature (and hence achieving low delay), and the schedule needs to be changed only upon significant change in network states due to, for examples, bandwidth change or parent failure. Given heterogeneous contents, delays and bandwidths of parents, we formulate the subchannel assignment problem to assign subchannels to parents maximizing the total bandwidth.

Moreover, NeuroCast enables a receiver peer to orchestrate a video stream from multiple congestion controlled senders. The primary challenge in design of a multi-source streaming mechanism is that available bandwidth from each peer is not known a priori, and could significantly change during a session. In NeuroCast , the receiver periodically performs quality adaptation based on aggregate bandwidth from all senders to determine (i) overall bandwidth that can be collectively delivered from all senders, and more importantly (ii) specific

subset of packets that should be delivered by each sender in order to gracefully cope with any sudden change in its bandwidth.

At the end, we describe NeuroCast's performance, analyzing the results of the experiments we run with the prototype. We also evidence the NeuroCast ability to accommodate heterogeneous peers advantaging more generous ones. Moreover, we show that NeuroCast is able to exploit the resources available in the system by distributing the available bandwidth among peers. We also validate the algorithms and protocols we defined in order to carry out the NeuroCast prototype.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In the nineties Internet was mainly a channel to spread information, services and contents under the control of industry. At the beginning Internet access capabilities where very limited and not so spread: everybody remembers the 56 Kbps modem plugged to the traditional telephone line. Since 2000, thanks to the development of new connection technologies, the bit rate of Internet access capacities started to grow. Moreover the monopolies of National Telecommunication providers disappeared in different countries favoring the birth of different private Telecommunication providers. Thus broadband Internet access became accessible to more and more people. This has completely changed the way to look at Internet together with the IT technical improvements. Today Internet is no more a simple channel to spread information, but it offers a new range of services no more under industrial control.

The growth of popular web sites serving multimedia contents has led to the increase of video streaming applications. However, video streaming over the Internet is complicated by a number of factors: 1) the Internet provides only the best effort service, and nothing is guaranteed about bandwidth, delay, and packet loss rate; 2) it is difficult to predict the bandwidth, delay, and loss rate information, since it is unknown and time-varying; 3) the heterogeneity of receiver capabilities is a significant problem when video streams are distributed over a multicast network; and 4) a congestion/flow control mechanism has to be employed to avoid the congestion collapse of the Internet.

It is important to mention the peer-to-peer revolution, a radical changing of the the Internet concept. In this manner it is possible to retrieve information and services by a simple exchange of data and a share of resources among users, without.anymore. the control of the enterprises. The first, and also the today most popular, p2p applications are the ones for file sharing. However lot of other p2p solutions have been proposed for different purposes.

In the last years two main phenomena have taken more and more relevance: VoIP telephony and media streaming.

The possibility to transport a voice call upon the IP network has changed the traditional view of telephonic communications. Previously, voice calls needed a traditional telephone line to be carried on; on the contrary now there is the possibility to have a voice communication with a simple Internet connection. This opened new research and business fields. This technology is not very recently but it has known an explosion after the recent development of several VoIP systems; among them the first one was Skype. With its ten millions users it allows to communicate all around the world for free by simply having an Internet connection.

Another phenomenon, even more recent, is the media streaming. More and more radio and television broadcasters provide on-line (often real-time) access to their programs. This means that now it is possible to access through Internet to services that were previously possible only with TV or radio devices.

In the last year another interesting trend came out: spread personal and original videos worldwide. This was possible thanks to companies like YouTube or DailyMotion which permit sharing videos of everybody with everybody. With its 70 millions videos watched everyday on its site, YouTube attests the relevance this phenomenon is acquiring on the Internet stage.

The system proposed in this work (named *NeuroCast*) perfectly fits the current trends: to give to everybody the possibility to show its video live. For example users would like to share a party with their friends which are on the other side of the world, or somebody would like to share with its family an important event of its life. Our system has been studied exactly for this purpose: to give to everybody the possibility to have a channel where he can live broadcast their videos.

1.1 Multimedia content distribution

The simplest way to transfer a multimedia file is the bulk file transfer. The content is simply a common file of known size available at a source. Every client has first to download the complete file and then it can reproduce its multimedia content. There are several ways to get the file. For example, the file could be stored at a server and a client only has to download it as in the traditional client-server model. Otherwise the content could be spread through a p2p file sharing application and so on.

Differently, the multimedia content could be delivered in the form of a media stream. The content is no more distributed as a simple file, but it is delivered as a continuous ordered flow of data from a source. A user does not need to get the complete content but just to reproduce the flow of information as it is received.

There are two ways to distribute a media stream: on-demand or live.

On-demand streaming consists in the delivering of a multimedia content of a known size, that is available at a source and that can be download in any moment. When a user wants to reproduce the content, it contacts the source that starts the stream. The most important challenge is the start-up delay, that is to say the time a user has to wait after a request in order to start reproducing the content. However, this requirement is not a strict real-time requirement since users can wait some instants before to start the content play-out.

Live streaming consists in the delivering of a multimedia content that has to be reproduced as soon as it has been created or few moments later. So we are dealing with the distribution of a tile of unknown and unpredictable length in which the data are only available for a small period of time. In this case the most important challenge is the play-out delay, that is to say the time elapsing between the content production and its play-out. It is clear that live streaming presents strict real-time requirements due to the volatility of the content.

1.2 Live streaming

The traditional client-server model is suitable for live streaming, but it is impracticable to a big audience because of scalability problems. In fact, a server has a

limit bandwidth and cannot serve more than a limited number of clients. Since the nature of the media content, it is very probable that lot of users want to see the stream at the same time and so the server cannot support them.

The best way to distribute a content from a source to a group of hosts at the same time is the IP multicast. However the deployment of IP multicast has been limited due to a variety of technical and non-technical reasons. First of all it requires changes in the network machines increasing the complexity and the overhead at the routers. Thus not all Internet equipments are able to support IP multicast. But, above all, it presents commercial problems since lot ISPs disable it. This is because they don't want to carry multicast traffic of other ISPs' users without gaining any money.

Since network level multicast does not offer an applicable solution, the idea of implementing multicast functionality at application level came out. The packets are no longer replicated at routers inside the network but at the application level or at the end hosts. This solves the problem of changing the network infrastructure, overcomes the ISP and the traditional client-server model limitations. This solution is known as Application Level Multicast and lot of systems have been developed for this purpose.

Another possibility is to use peer-to-peer networks for the live streaming. The stream receivers act as clients and servers at the same time replicating packets they receive. Peer-to-peer streaming is a method for multicasting or broadcasting streaming media, for example audio or video, over the Internet using a peer-to-peer network. It can be seen as a combination of traditional television or radio broadcast type of media delivery over a new kind of delivery medium, the Internet. The aim for these techniques is to allow bandwidth-consuming streaming media to be delivered to a large number of consumers without unnecessary network congestion. There are special requirements for the access networks and peer-to-peer streaming applications when they are used in a mobile environment. For example delay, jitter and throughput in the access network, used content encoding format, stream bitrate and buffer size affect to the quality of experience and the usability of the application. P2P live streaming systems organize nodes in an overlay in order to address the following issues:

- **Decentralization**. There is not a central authority that performs operations or that coordinates peers, but all the tasks have to be done locally. This may lead to sub-optimal performances but it permits to obtain a scalable and reliable system. In fact there is not the problem of the limited resources of a central entity, and a node failure doesn't kill the whole system.

- **Quality of the delivery data path**. Since the data replication is done at the end-hosts, in these systems the paths from the source to the receivers are different from the unicast paths between them. An important challenge is thus to obtain data paths that differ as less as possible from the unicast paths between source and receivers. Another interesting aspect regards the node degree that indicates the number of peers an host serves at the same time.

- **Robustness of the overlay**. End-hosts are less stable than routers or network equipments and so they can join, leave or simply fail. These systems have to limit the effects of peer arrivals/departures.

- **Control overhead**. The overlay has not a static structure; on the contrary it continuously evolves during time. There is the need of control messages to permit this evolution and to keep the system working. An important challenge is to limit the volume of the messages exchanged.

- **Adaptiveness**. This point is related to the heterogeneity of peers. Hosts may differ one from another in the amount of resources they have: for example they can have different processing powers, different access capacities, etc. These systems have to maximally exploit all the available resources and in particular the available bandwidth.

- **Fairness**. There is not a unique way to define fairness because its meaning changes according to context where it is used. A p2p system has to address the problem of selfish peers that try to download the content without serving other nodes. These peers are called free-riders and the system must be able to penalize them. Ideally a peer should only get as much as it contributes.

However, this is not compatible with a live streaming system where the download rate should be constant and equal to the stream rate. A live streaming system could try to favor more generous peers offering them advantages in term of reliability and latency.

- **Content nature**. The live streaming content has to be reproduced at the receiver either at the reception or shortly after, and it is available at source only for a limited period of time. Such a content is thus very sensible to delays and losses.

Next chapter presents the results of the research that the scientific community has done about live streaming problem and about already implemented p2p streaming systems. This research allowed us to understand the most important choices done in the NeuroCast's design and to propose possible improvements to the system. The research's results could be considered a good overview of the different existing p2p approaches to the live streaming problem.

1.2.1 Peer-to-Peer Live Streaming

We decided to focus on the problem of live streaming for several reasons. First, scalable data distribution is a fundamental need in the Internet today: while the distribution of static (i.e. pre-stored) content has been a topic of widespread interest during the last decade, the attention to live content distribution has been more recent. The rise of peer-to-peer (P2P) architectures in the context of static content distribution has provided a definite improvement in scalability over the previous server-based architectures. Extending a P2P approach to live media streaming allows to address the scalability issues of centralized systems, presents interesting challenges, and still constitutes an open research problem.

Second, the understanding of live media distribution is much less consolidated than the classic case of static file distribution, as it is subject to timing and ordering constraints. Third, live streaming is a problem with clear boundaries and requirements: as new data are constantly generated (i) all the viewers are loosely synchronized in receiving roughly the same stream segment, (ii) the system as a whole has a short memory and (iii) it can operate with a larger independence on

the user behavior patterns than bulk data distribution or VoD. Fourth, scalable live streaming applications have today a huge practical interest, as the equipment required (powerful computers, digital cameras or web-cams, Internet connections) is available off-the-shelf and is usually cheap: this constitutes a fundamental premise for the commercial success of a new application or service, enabling it to spread and succeed among the general population. Finally, we believe that there are several aspects of the existing practical P2P live streaming systems that can be improved, and we make several contributions in order to do so

It is quite interesting to notice that the distribution of media over the Internet has not taken off thanks to the explicit intervention of the industrial or academic world. Rather, the small players - such as single individuals, start-up ventures, and technically educated Internet users as a whole - have often been the first to explore the possibilities of the existing network infrastructure, anticipating the so-called innovations later introduced by the established players of the telecommunications field.

The development and success of P2P systems is a good example of user-driven innovation. These small players were in fact the ones confronted with serious practical issues, such as lack of economic resources, unreliable hardware and limited connectivity - they could not afford powerful server machines, nor large data storage facilities, nor fast Internet access. For them, P2P was more a necessary evolution than an incremental optimization over the server-based architectures.

In a world where servers were required to support every kind of on-line activity - from data download, to search, to instant-messaging, etc. - the adoption of a distributed approach that enabled the exploitation of the resources provided by the users was a definite breakthrough. Initially, there was widespread skepticism over the effectiveness of P2P networks as a replacement of server-based technology. Especially the earliest P2P software such as the first iterations of Gnutella were not meant to support large scale systems (94), and were expected to be deployed in local contexts with small user populations.

The astonishing growth in popularity of these applications, coupled to the reduced need for resources at the "server", quickly spurred a considerable interest. Suddenly, P2P became a sort of buzzword, which was supposed to grant either

the immediate success of a new application, enormous cost savings at the service provider, or both. This phenomenon sometimes reached unreasonable and hilarious proportions before finally giving way to more rational behaviors.

Today, the design of large-scale services and applications benefits from a large body of lessons learned from the evolution of P2P applications. The most important insight is that a P2P approach works well to solve few specific issues but is impractical for some others. Problems such as large-scale data distribution, data storage, keyword-based or pattern-matching search can enjoy significant benefits when distributed techniques are adopted. On the other hand, features such as user presence tracking and authentication are extremely challenging to reproduce in a pure P2P fashion.

1.3 Thesis Contributions

This thesis makes several contributions. It first approaches the problem of live media streaming from a practical point of view. The requirements of large-scale P2P streaming applications are presented and discussed starting from the set of challenges enumerated above, which reflect the current technical limitations of Internet technology (and their likely evolution in the foreseeable future). An in-depth survey of the related work is then performed, whose aim is to evaluate the existing live streaming architectures in light of their suitability to a large-scale deployment over the Internet. Insights on the architectural features that help toward this goal are also provided.

The second contribution is the design of NeuroCast, a P2P live streaming system that satisfies the previous requirements. The NeuroCast system is among the first systems to rely on an unstructured mesh-based design, and introduces a mechanisms for the selection of neighbor nodes. The advantages of NeuroCast over existing systems can be summarized as:

- Support for very high levels of churn (node arrivals and departures).
- Support for strongly heterogeneous distributions of peer upload capacity.

- Multi-source capabilities to use efficiently the available upload capacity, especially under scarcity of resources.
- Fast adaptation and recovery from abrupt changes in network conditions.
- Implicit awareness to network locality through latency measurements

The third contribution is an implementation using the VNUML (15) philosophy that models the complex behavior of a NeuroCast system. Based on the insights obtained by experimenting with the emulated algorithms, a stand-alone network was also emulated. These pieces of software have been used to improve our understanding of the emergent global behavior of NeuroCast systems that operate under a variety of bandwidth distribution scenarios and network environments.

The fourth and last contribution is the qualitative and quantitative analysis of NeuroCast based on emulation results. We first validate that the NeuroCast algorithms are operating as expected, and then assess their performance in a large range of challenging scenarios in which structured systems would hardly be able to operate. We specifically evaluate the awareness of the resulting overlay mesh to resource availability in the system and describe the average characteristics of the data paths that connect the source to the nodes.

1.4 Thesis Organization

The rest of the thesis is structured as follows. Chapter 2 presents live streaming over the Internet in an historical and technical perspective. The current solutions for P2P live streaming are then introduced: after restating in better detail the main challenges of this problem, the various available design options are compared and conclusions on their viability are drawn. Chapter 3 introduces the transport protocols used for streaming as well as the different issues that emerge when streaming over P2P networks.

Chapter 4 introduces the Peercast system: the basic and the implementation. As Peercast is the base of our system, we analyze its main advantages and drawbacks in depth. Chapter 5 describes the NeuroCast system in its entirety - the

basic insights, the terminology, the algorithms, and the implementation. This chapter approaches the problem of understanding a dynamic mesh-based system like NeuroCast: we illustrate the empirical techniques that we had to adopt to study our system. Chapter 6 introduces an original set of tools and software techniques used to describe the behavior of a generic data-driven system. An additional set of parameters that correlate the availability of node capacity and data reception performance is defined: these metrics will be the fundamental tools of our subsequent analysis of NeuroCast by way of emulation on medium-scale network testbeds. In chapter 7 we explore the security threads that could affect NeuroCast and propose possible approaches to mitigate them. Chapter 8 concludes this work.

Chapter 2

State of the art

As the amount of media delivered in the Internet seems to be ever-growing and the speeds of end-users' access network connections are getting faster day by day, network capacity continues to be a scarce resource. As opposed to the traditional client/server architecture, some solutions to relieve network congestion have been proposed and different peer-to-peer techniques being the most interesting and popular ones. In a peer-to-peer network, a single host acts as a server and a client simultaneously. Although peer-to-peer techniques do not seem to directly reduce network load compared to client/server approach, it has been observed that network load is distributed more evenly to the whole network when using peer-to-peer techniques. This leads to single links within the network being less congested.

Peer-to-peer streaming is a method for multicasting or broadcasting streaming media, for example audio or video, over the Internet using a peer-to-peer network. It can be seen as a combination of traditional television or radio broadcast type of media delivery over a new kind of delivery medium, the Internet. The aim for these techniques is to allow bandwidth-consuming streaming media to be delivered to a large number of consumers without unnecessary network congestion.

The first attempts in using P2P systems for distributing a streaming content date back to 1997 when a first P2P solution for on-demand data streaming had been proposed. Few years later (2000) some systems for live streaming, working

on two-tired infrastructure (networks where only the core is a P2P overlay, normally of powerful nodes, and the end-users are just receivers), appeared. From 2001, the P2P live streaming subject became very popular and researchers started to focus great attention to it.

Some P2P live streaming systems have been proposed. They can be classified in three main categories:

- Structured. In these systems peers are hierarchically organized in a tree or in multiple-tree overlays and data is forwarded from the source to peers upon these structures.
- Unstructured. The content is divided into pieces that are spread in the network by the source without follow a pre-defined structure. Peers have then to established relationships among them to retrieve their missing pieces.
- Hybrids. These systems use an approach that exploit both previous techniques mixing them.

Next, the related technologies are discussed. In the following we are going to describe the related work focusing in P2P systems and these three categories by presenting some examples, and by discussing their properties and performances.

2.1 Related Technologies

In this section we give an overview of the technologies that are commonly used for delivering digital content to a large number of users in the Internet.

2.1.1 Multicast

Multicast is an elaboration from the simple unicast delivery scheme. While still utilizing the nature of client/server architecture, the amount of data delivered within the network can be remarkably reduced compared to the unicast delivery. This is achieved by using point-to-multipoint delivery. Multicast delivery scheme is presented in Figure 2.1.

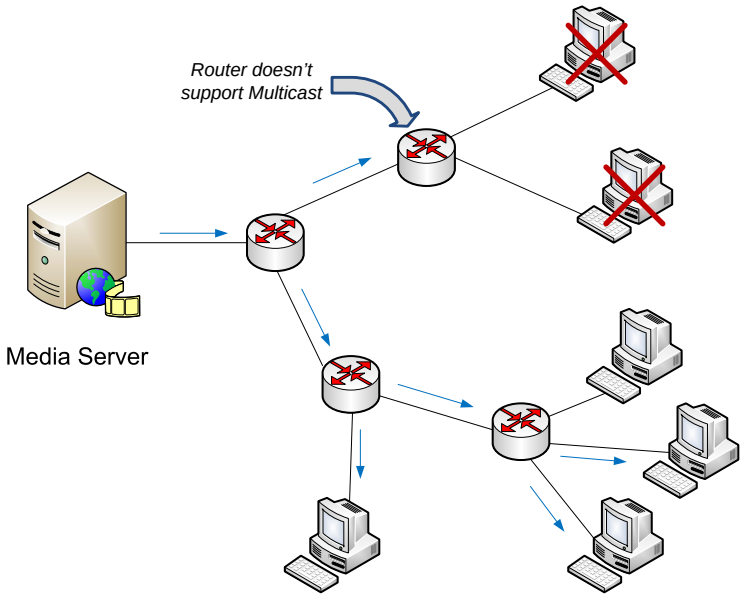


Figure 2.1: Muticast delivery scheme.

When using multicast, an IP datagram does not need to be replicated at the server end, but the same datagram is delivered for each receiver with a minimum amount of replication. The receivers must register to "listen" the multicast traffic in order to receive the data delivered by the server. Also, the network infrastructure (i.e., routers) must support multicast traffic to make it possible to forward the data stream to the end-users. As presented in Figure 2.1, one of the routers does not support forwarding multicast traffic, so the users located in the network behind the router are not capable of receiving the data, even if desired. So the network infrastructure sets a restriction to service availability for the users wishing to receive the service.

2.1.2 Application Layer Multicast (ALM)

In application layer multicast IP datagrams are replicated at the end hosts, compared with native multicast delivery where IP datagrams are replicated at the routers. In practice, the end-hosts form an overlay network, and the goal is to construct and maintain an efficient overlay for data transmission. Since application layer multicast protocols possibly send identical packets over the same link

(depending on the overlay network topology), they are less efficient compared to native multicast protocols. In contrast there is no need to change routers, so the network infrastructure does not set a restriction to service availability. Figure 2.2 shows a comparison of the former scheme compared with the ALM scheme.

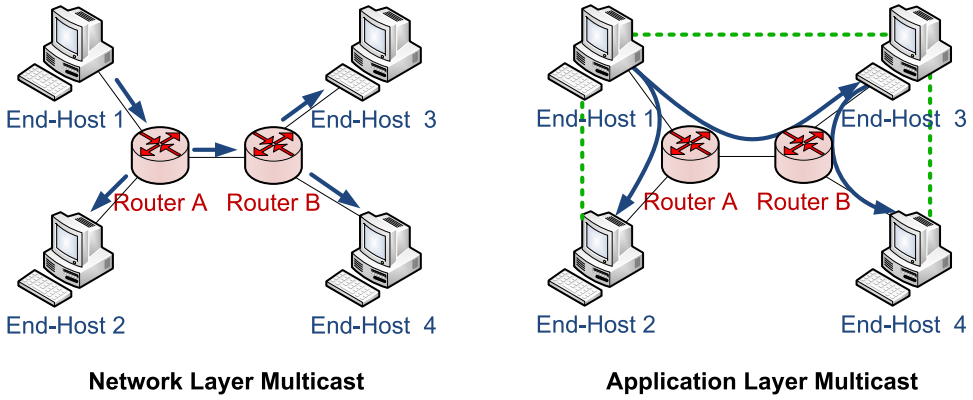


Figure 2.2: Multicast scheme (on the left) vs. Application Layer Multicast (ALM- on the right).

2.1.3 Peer-to-Peer Overlay

A peer-to-peer network utilizes a client/server architecture between number of hosts in a network. That is, each host acts as a server and client simultaneously. The network is depicted in Figure 2.3 and, as it is obvious, there are no routers in this architecture. This is because a peer-to-peer network is actually an overlay network. This means that the logical connections between hosts, peers, are formed on higher level than the network (IP) level. Typically the peer connections are formed using TCP.

Each host acts as a server and a client and the actual traffic is unicast (point-to-point) in nature. But because the hosts are distributed all over the network, the actual traffic load is distributed more evenly over the whole network. In contrast to an ordinary client/server architecture, where there is one centralized server (or more if the server system is distributed) serving all client hosts, a peer-to-peer network utilizes multiple ”servers” distributed over the whole network.

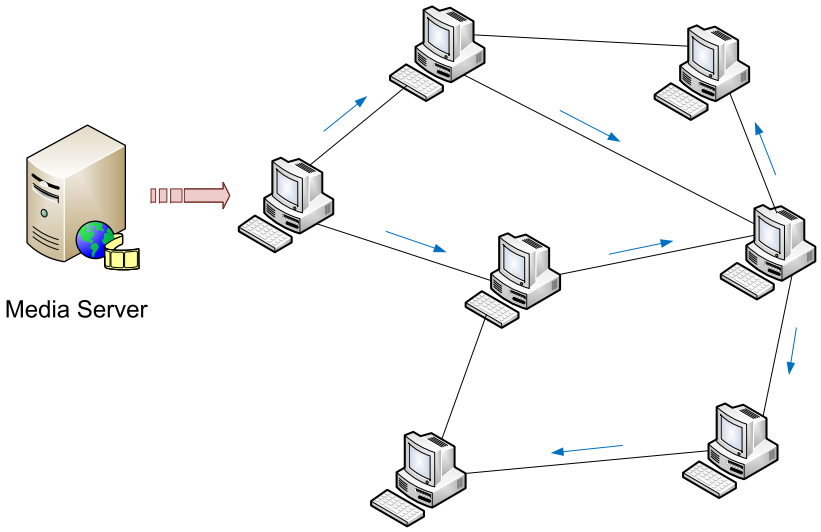


Figure 2.3: Peer-to-Peer overlay network.

Before being able to transfer data in a peer-to-peer network, a host must first somehow join the overlay network, and the means to do that vary between different peer-to-peer protocols. There is one major advance compared to multicast, though: because peer-to-peer traffic often relies on the existing network infrastructure capable of unicast traffic, there are no (or only a few) users left outside of service from the traffic reachability point of view.

Because of the lack of actual media server in a peer-to-peer network, there must be some way to inject the content for delivery to the network. This is usually done by making the data available on one or more hosts within the network (media source), allowing the content to be delivered to the users. This makes content controlling difficult in the network, when compared to centralized content servers used in multicast and unicast networks.

2.2 Peer-to-Peer Systems Overview

Peer-to-peer (P2P) systems are a special type of overlay network where there is little or no dedicated infrastructure and the peers, or clients, act as potential forwarders and, more generally, contribute their resources to the system (20). This

concept has many applications which range from grid computing, to distributed storage or, as considered in this thesis, real-time media delivery.

Today's most popular systems are mostly driven by media file sharing. The most popular ones, e.g., eDonkey (63) and BitTorrent(1) (or any client running these protocols) have millions of users and represent a large portion of Internet traffic (9). One of these clients, eMule (2), has reportedly been downloaded over 300 million times as of April 2007, which makes it the most successful open source project to date (13). In these systems, different peers coordinate to download files, without requiring costly web-servers to host and transmit content. The distributed nature of these systems require specific protocols to locate, store or download content, and has fostered a large amount of research and development in both the academic and open source communities. All these goals are shared by P2P streaming systems which incorporate, in addition, a latency constraint. In this section, we give an overview of the widely deployed BitTorrent protocol, describe recent advances in the area of P2P file sharing protocols, and finally focus on P2P streaming systems which will be a central topic of this thesis.

2.2.1 BitTorrent

BitTorrent (1) is an open source protocol which was created in 2001 by Bram Cohen, to overcome the shortcomings of prior P2P systems. In particular, BitTorrent aims at fully utilizing the up-link throughput of peers which have downloaded or are downloading a particular file. The main characteristics of the BitTorrent protocol include:

- Dividing files into small chunks of data, which can be downloaded by a peer independently of each other, and providing a simple system to check the integrity of each chunk.
- Incorporating simple uploading rules in the clients to enhance the performance of the system in terms of download speed, and of lifetime of a file in the P2P network. These include downloading the rarest chunk first and reciprocation, i.e., "tit-for-tat" (76) uploading.

The original BitTorrent system was successful at creating a large community of users, moderated by some active members, in charge of inspecting each new file in the P2P system, in order to maintain high quality and avoid pollution.

A BitTorrent system is composed of four parts. The first component is the *seed*, a user who has a complete copy of a file she wishes to share with other peers. The second component is the *.torrent* file. This file is created by a seed and published on a regular web-server. It contains enough meta-data to describe the file: the number of chunks into which data is separated and the SHA-1 (47) signature of each chunk, used to verify the integrity of the file, as it is being downloaded. In addition, it also indicates the address of a tracker. Trackers are the third component of BitTorrent systems. They are hosts which continuously monitor the download of the file by storing the addresses of the different peers which have downloaded or are downloading the file, as well as some additional information reflecting their performance. Finally, the fourth component are the actual peer nodes. The peers initially access the *.torrent* file, register with the corresponding tracker, and use it to obtain a list of connected peers. They locate missing chunks of the file by exchanging their buffer map with other active peers. These maps indicate the chunks they hold and the ones they are missing. Uploads and downloads between peers are negotiated following rules which we describe in the next paragraph. Periodically, peers report their download status to the tracker of the *.torrent* file. A detailed description of the syntax and implementation of the BitTorrent protocol has been made available to the public (1), and has resulted in a flurry of BitTorrent compliant clients, which compete on the quality of their content, of their client, and of their community. Some of these, notably, include Azureus, μtorrent, ABC, TurboBT, BitComet, as well as the original BitTorrent client.

Once a BitTorrent client has obtained a list of active peers from the tracker, it locates different chunks of the file by exchanging its buffer map (initially empty) with the other members of the session. It then requests the rarest chunks from the peers which hold them. This strategy decreases the likelihood that a torrent will die as a consequence of one of the chunks disappearing from the network. This, obviously, would preclude any download from completing. Peers serve a small number of incoming requests simultaneously (typically 4). The speed at which

the chunks are uploaded is dictated by TCP (61). When choosing among different requests, priority is given to requests from peers which themselves have provided chunks previously. This prioritization is known as the "tit-for-tat" rule (76). A small number of connections (typically 1) is also reserved for serving peers, regardless of the amount of data these peers have provided in the past. This is particularly advantageous for peers which have recently joined and which have not yet been able to provide any data. The motivation for this altruistic "optimistic unchoking" rule is that it may result in establishing a high-speed connection to a new peer which may be worthwhile in the future, as a consequence of tit-for-tat. Finally, when almost all the pieces have been downloaded, a peer will try to get the same remaining chunks from multiple peers at once to avoid being held up by a peer with a slow connection.

The emergence of large P2P file transfer systems with large amount of users and traffic has enabled researchers to investigate and attempt to model user behavior (23), or to study the evolution of large distributed systems (39; 55). Several improvements to systems like BitTorrent have also been proposed (56; 94; 99).

In one particularly interesting direction one tries to avoid any centralized index whatsoever. In the case of Napster, a centralized index was used to hold a list associating the files present on the network to the address of computers. In BitTorrent, a centralized list of peers is held by the tracker. The lack of any centralized index (largely motivated by piracy) creates an interesting challenge in terms of content discovery. A significant amount of work has addressed the problem of how to locate a file among a large set of users given its name and the IP addresses of a very limited set of connected peers. Beyond the simpler flooding approach (94), proposed solutions rely on distributed hash tables (DHT) which map a key, representing a file, to a value, indicating a peer, and are distributed among the users. As each peer only maintains a small part of the table, the algorithms also indicate how queries can be conducted on the whole table efficiently, through message exchanges. Such systems, including the popular Chord, CAN, and Pastry, are analyzed in (24; 51; 54; 62; 73; 84), where their scalability, efficiency and resilience are examined .

The BitTorrent protocol has proven to be very reliable as a result of its simplicity. One of its main characteristics is that it is data-driven. Peers look for particular pieces of data, regardless of the peers which hold them. This differs from other approaches where clients try to identify peers which hold the data and establish lasting connections to them, as, for example, in Kazaa, one of BitTorrent predecessors. This data-driven approach avoids the need of precisely measuring the amount of available throughput of different peers and monitoring the different connections. This distinction between data-driven protocols and connection-driven protocols, or, as these approaches are also referred to, between un-structured and structured systems also exists in P2P video streaming systems.

2.3 Peer-to-Peer Streaming Taxonomy

In P2P streaming systems, a critical requirement is to operate the media distribution continuously. Hence, the difficulty resides not only in content location, but also in resource location, as peers need to discover which other connected hosts have enough throughput to act as forwarders and relay the media stream they have received. To the best of our knowledge, one of the earliest proposals for this type of system is by Sheu et al. (90), which focuses on building a distributed video-on-demand system for ATM networks. More recent work on P2P asynchronous video streaming, and a good deployment example is the P2P client *Joost*, which improved the progressive download feature originally incorporated in Kazaa (4), to offer on-demand viewing of pre-recorded television shows.

The concept of live P2P multicast was made popular by Chu et al. (26) who suggested taking advantage of the resources of the users to form a dynamic delivery network which would offer the same viewing experience as live television. The idea is appealing as it does not require any infrastructure and is, in theory, self-scaling, as the number of peer servers and peer clients increase at the same rate. Even though this field is still in its early stages, it has become, in the last few years, a very active area of research. Many proposed systems rely on distributed protocols to construct one or several multicast trees between the media source and the different users to distribute the stream. Another approach lets peers self-organize in a mesh and request different portions of the video from their

neighbors, with no particular emphasis on the structure of the distribution path. Along with these early research experiments, many applications have appeared on the Internet, such as PPLive, PPStream, TVU networks, Zattoo, etc. One of the main goals of these applications is to enable the largest possible set of users to connect to each other by integrating to their protocol Network Address Translator (NAT) and firewall traversal techniques. These problems have been partially resolved via protocols which employ third-party rendez-vous servers. All these implementations constitute very exciting progress and demonstrate the feasibility of large-scale P2P streaming. As an example, both PPLive and Gridmedia have been reported to support over 100,000 peers simultaneously with a small number of servers. However all these systems typically suffer from long startup delays (possibly on the order of minutes) and often cannot sustain constant video quality.

2.3.1 Structured P2P live streaming systems

In these systems peers are organized in a fairly static structure upon which the stream is spread. This structure is a hierarchical (single or multiple) tree , with the source as root, where every node receives the whole data content from its parents and transmit it to its children. Differences between systems belonging to this category mainly concern the way peers are organized and the algorithms used to create, to maintain and to repair the tree structure.

These systems focus their attention on the quality of the data paths from the source to the receivers. In particular, they try to minimize the differences between the unicast paths linking the source to the receivers and the paths actually followed by the data. There are three main parameters to evaluate the quality of data paths:

- length of the data path.
- link stress. It quantifies the load on the network. It is normally computed by counting the number of packets that cross the same link.
- node stress. It quantifies the load at nodes. It is normally computed by counting the number of packets that a node receives.

The goal of such systems is obviously to minimize as much as possible these three parameters. The data paths can be modeled through mathematical analysis since they are built following some determined rules. It is thus possible to study the structure properties. For example, it is possible to estimate the time needed by a peer to receive a chunk or the number of peers a node serves at the same time (node's degree) etc.

To determine the play-out time of a stream is thus not a problem, since it is possible to estimate the time required by a peer to receive data from the source. The data always follows the same route and thus the delay is almost constant.

These systems has to focus also great attention to the placement of peers in the structure. It is necessary to place the peers with more resources close to the source. By doing this, they could receive the most recent data and distribute them among a wide range of peers. On the contrary, if peers with scarce resources are placed close to the source, they will not have enough resources to serve a wide range of peers slowing down or interrupting the stream distribution.

Structured systems suffer the peers' transiency; this because if one node covering a key role in the structure fails, lot of peers will lose data. For example if a node close to the source fails, all the peers belonging to its sub-tree will lose data until the failure will be repaired. In these systems the control overhead is mainly devote to the structure building and maintenance. In the next sections we are going to present single tree and multiple-tree structured systems. In the last section an improvement to single tree solutions will be presented.

2.3.1.1 Single-tree

Single-tree P2P live streaming systems are today the most populars since they represent the most intuitive way to reproduce the IP multicast structure at application level across many tunneled unicast connections. In these systems peers are hierarchically organized in a tree structure where the root is the stream source (see Figure 2.4). The content is spread as a continuous flow of information from the source down to the tree. Systems belonging to this category mainly differ in the algorithms used to create, maintain and repair the tree structure.

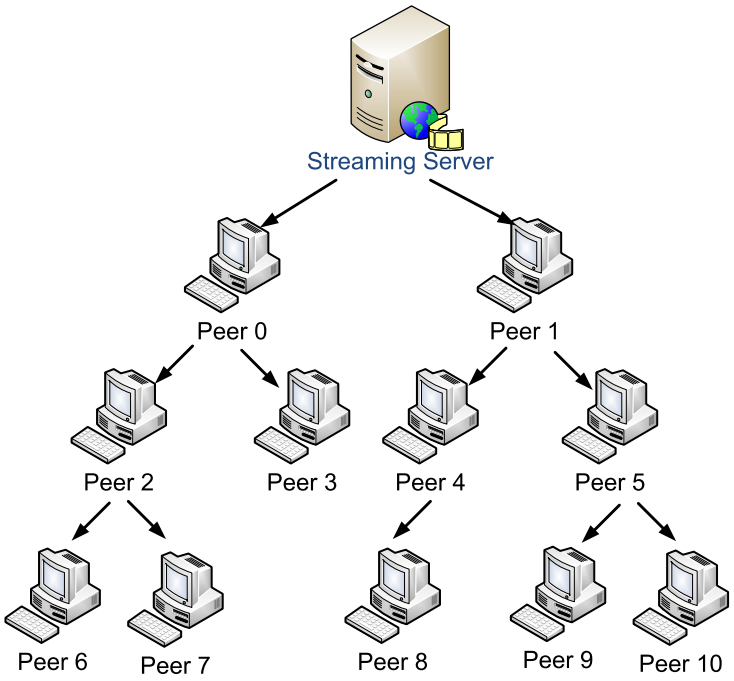


Figure 2.4: Single-tree model.

Since these systems are very close to IP multicast, trying to emulate its tree structure, they are able to achieve data paths that don't differ too much from IP multicast paths. In fact these systems are able to achieve link stress and data path lengths that are 1.5-2 times bigger than the ones of IP multicast.

The stream reception delay of a peer is bounded to the number of hops required to reach the peer from the source. In such systems the number of hops only depend on the position the peer occupies in the tree. This position is fairly static and thus the delay is almost constant. In single-tree systems all the load is supported by the interior nodes of the tree while the leafs are just receiving data. Thus, if an interior node has not the required computational or bandwidth resources to serve all its children, peers in its sub-tree will suffer of high delays in data reception or will never receive the stream. These systems don't seem to exploit very well all the available peers' resources and in particular the available bandwidth; in fact only few peers are in charge of the data forwarding while the others are just receiving data.

Single-tree systems suffer nodes' transiency. If a leaf of the tree fails or leaves,

the system will not suffer. On the contrary, if an interior node fails, peers on its sub-tree will lose data until the tree structure will be repaired. The amount of data lost varies from one system to another and depends on the repairing mechanism adopted. These systems implement data recovery mechanisms, such us source coding and so on, in order to face this problem. The data play-out of different systems is thus affecting in different degrees by peers' transiency, according to the data recovery mechanism adopted by each system. These systems present lack of control about peers' behavior; there is not a mechanism to penalize free-riders and to incentive peer cooperation. The most popular system using a single tree approach is NICE.

2.3.1.1.1 NICE

NICE (22) is an acronym that stands for "NICE the Internet Cooperative Environment". It is a project of the university of Maryland aiming to create a group of collaborative applications and to prove that such applications can achieve better performances than ones that don't collaborate. People working on this project proposed in 2002 the NICE application-level multicast protocol. This protocol has been studied to support applications with large receiver sets, low bandwidth and soft real-time data stream. It can also be extended to applications with high bandwidth requirements. In order to build the tree overlay, NICE assigns hosts to different levels that are sequentially numbered from the lowest. Members of each level are organized in clusters which size is bounded between k and $3k-1$ (k is a constant). Every cluster has a leader that is chosen to be the center (with respect to distances) between all its members.

The mapping of peers to levels is done as follow:

- All hosts are part of the lowest level where they are partitioned in different clusters.
- Cluster leaders of the lowest level (level 0) are also part of level 1 where they are newly organized in clusters.
- Cluster leaders of level L_i are also part of layer L_{i+1} where they are partitioned in clusters.

This procedure is done recursively until there is only one peer in a level (see Figure 2.5.

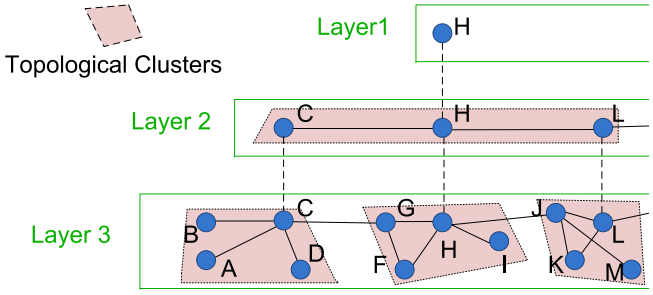


Figure 2.5: Hierarchical arrangement of hosts in NICE.

By implementing this mapping it is possible to achieve the three following properties:

- A host belongs to only one cluster at each level.
- If a host is present at level L_i, it is also present at every level L_j where $j < i$, and it is the cluster leader for these levels.
- There are at most $log_k N$ levels, where N is the number of peers and k is a constant.

The peer access to the system is provided by a Rendezvous point that gives to the new peer a list of peers of the upper levels. The new peer contacts them and it is inserted in the tree by coming down in the hierarchy. At every level the peer is inserted in the cluster whose leader is the closest (distance-wise) to the peer. Mechanisms for cluster maintenance, that foresee heartbeat messages among peers of the same cluster, and mechanisms for cluster refinement, that foresee merge or split of clusters as needed, have been defined.

The data delivery path is a tree and more specifically a source-specific tree. The source forwards the packet to all member of its cluster at level L_0. Each cluster leader is then responsible to forward the data to all members of clusters for which it is the leader, and to all the member of its upper level cluster.

Some simulations and real-world experience results are reported in (22), and show a low link stress, short path lengths, and limited control overhead even in

presence of node departures. However, some weaknesses are shown in the amount of data lost in presence of node failure. In this case the 20% of peers don't receive a part of the stream. Another problem is represented by a cluster leader departure; in fact the amount of control overhead generated in this situation is quite high ($O(logN)$ where N is the number of peers).

2.3.1.1.2 Other existing systems

Other live streaming systems, implementing a single-tree structure, have been proposed. Two systems have been defined by the university of Standford. A first system, called SpreadIt (29), has been proposed in 2002. It is a single-rooted tree with bounded fan out, which limits the nodes' bandwidth usage while making the maximum propagation delay optimal. Its goal is to provide an unreliable multicast infrastructure. A second system have been proposed by the same group and it is built upon the previous one; it is called PeerCast (10). This system is a developed application mostly used for Internet radio broadcast. As Peercast is at the core of the application developed in this thesis, it will be described in more detail in Chapter 4.

A system has been proposed by the CS department of Carnegie Mellon University and it is called End System Multicast (ESM). It aims more to overcome practical issues of a real system implementation than to propose new models or algorithms. Its system structure is in fact really similar to one of SpreadIt and PeerCast.

2.3.1.1.3 An improvement to the single-tree approach: ZigZag

There are some systems that, even if belong to the single-tree family, introduce some tricks to face the majors drawback of a single-tree solution. We are going to describe ZigZag (95) that has been proposed by the University of Central Florida and that improves the NICE protocol.

The tree organization is very close to the one proposed by NICE. Peers are organized in levels and clusters in the same way as NICE does. The difference is that each cluster has not only a leader (here called cluster head) but also an associate head. The algorithms for structure building and maintenance are

quite similar to NICE and all the NICE structure's properties are still valid. The main difference from NICE is however the data delivery path. While in NICE everything if forwarded by the cluster leader, here the responsible for data forwarding is the associate head. It recovers data from a non-head member of the above level and sends it to all its cluster members. The most interesting improvements with respect to NICE are :

- The worst case node degree is bounded to $6k - 3$ (where k is a constant). This offer an higher level of scalability than NICE.
- The recovery in case of head or associate head departure is easier than the recovery after the cluster leader departure in NICE. This is possible thanks to the existence of two head roles. In fact, if the head fails, the associate-head will immediately take the role of head and select a new associate-head. The same happens in case of associate-head failure: the head can immediately select a new associate-head. Problems arise in case of contemporary failure of head and associate-head. This solution offers a structure that is thus more robust than the NICE structure.

This version of ZigZag is called I-ZigZag ("I" stands for indirect) in order to differentiate it from a second version called D-ZigZag ("D" stands for Direct). In D-ZigZag members of a cluster directly download data from a non-head member of the above level removing the role of the associate head. This second version is more suitable for low latency applications but it is less robust and has higher bandwidth requirements for member nodes.

2.3.1.2 Multiple-tree/Forest

Conventional tree-based multicast is inherently not well matched to a cooperative environment. The reason is that in any multicast tree, the burden of duplicating and forwarding multicast traffic is carried by the small subset of the peers that are interior nodes in the tree. Most of the peers are leaf nodes and contribute no resources. This conflicts with the expectation that all peers should share the forwarding load. To address this problem, forest-based architecture is beneficial, which constructs a forest of multicast trees that distributes the forwarding load

subject to the bandwidth constraints of the participating nodes in a decentralized, scalable, efficient and self-organizing manner. A typical model of forest-based P2P streaming system is SplitStream (24) which is explained in the next point.

One of the main issues of single-tree approach is that the load burdens only few peers in the network. As seen in the previous section, this aspect also leads to resiliency problems. In order to address these issues a multiple-tree approach has been proposed. The idea is to built N different trees, sharing the same source, among peers. The stream content is divided in N complementary stripes and every stripe is spread upon a different tree (see Figure 2.6). It is important to place a node at different levels in the different trees. In particular, the key challenge of these systems is to build a forest of interior-node-disjoint trees. This mean that these systems try make a node an interior node for only one tree and a leaf for all the others.

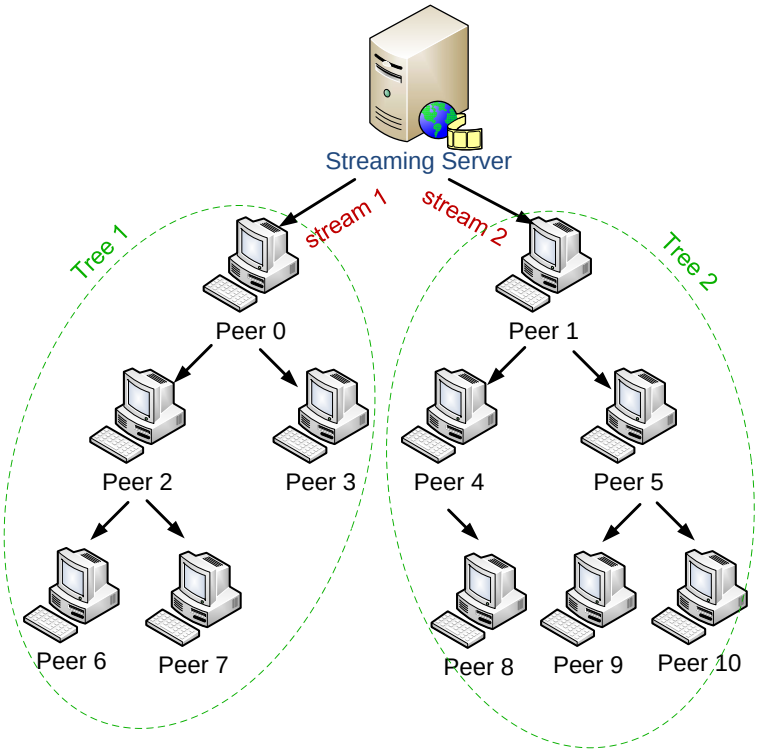


Figure 2.6: Multiple-tree model.

Since all peers are involved in the data distribution, the load is spread among

all nodes. This also leads to reduce the link stress and the node stress (number of packet that are sent by a node) with respect to a single-tree approach.

Another important improvement,with respect to single-tree approach, is the robustness. In fact a node failure causes losses on only one stripe. Peers thus loose only a small amount of data until the failure is repaired.

A drawback of these systems is the bigger control overhead with respect to the single-tree approach. This because there are many trees to build and maintain.

This approach is relatively new and is very fashion now. One of the first proposal that uses this approach is SplitStream; we are going to describe it in the next section.

2.3.1.2.1 SplitStream

SplitStream (24) system has been proposed in 2003 by the Microsoft Research center. It is able to build interior-node-disjoint trees that accommodates peers with different bandwidth capacity limiting the overhead due to tree construction and maintenance. SplitStream is built upon Scribe (83) which is in turn based on Pastry (82). In particular, Scribe builds a multicast structure by the union of all the Pastry routes from the receivers to the node responsible for the chosen multicast ID. The Pastry routes have the interesting properties that every node belonging to the route shares at least the first digit of its ID with the destination node ID. Thus the multicast tree built by Scribe involve nodes whose IDs have in common at least the first digit.

The key idea of SplitStream is to split the original media data into several stripes, and multicast each stripe using a separate tree. Peers join as many trees as there are stripes they wish to receive and they specify an upper bound on the number of stripes that they are willing to forward. The challenge is to construct this forest of multicast trees such that an interior node in one tree is a leaf node in all the remaining trees and the bandwidth constraints specified by the nodes are satisfied. This ensures that the forwarding load can be spread across all participating peers. For example, if all nodes wish to receive k stripes and they are willing to forward k stripes, SplitStream will construct a forest such that the forwarding load is evenly balanced across all nodes while achieving

low delay and link stress across the system. Striping across multiple trees also increases the resilience to node failures. SplitStream offers improved robustness to node failure and sudden node departures like other systems that exploit path diversity in overlays. SplitStream ensures that the vast majority of nodes are interior nodes in only one tree. Therefore, the failure of a single node causes the temporary loss of at most one of the stripes (on average). With appropriate data encodings, applications can mask or mitigate the effects of node failures even while the affected tree is being repaired.

SplitStream is able to build interior-node-disjoint trees exploiting this properties. It builds different trees exploiting the functionality of Scribe. The tricks consists in choosing one different group ID for each stripe and these IDs have to differ for at least the most significant digit. By doing this one node can be an interior node for only one stripe.

Moreover SplitStream allows nodes to declare the number of stripes they would like to receive/transmit and it has techniques to accommodate these requirements without loosing the previous property.

2.3.1.2.2 Other existing systems

Besides SplitStream, there are many other forest-based systems. Examples include building mesh-based tree (Narada and its extensions (26), and Bullet (65)), leveraging layered coding (PALS (81)), and multiple description coding (CoopNet (79)).

Moreover, it is worth to mention P2PCast (78), a system developed at CS department of New York University. It is largely based on the SplitStream architecture. It aims to improve SplitStream performances by removing the dependencies from the DHT overlay and by better allocate the spare bandwidth among nodes.

PrefixStream (40) is a system developed at INRIA of Rocquencourt. It is built upon a De Bruijn graph where the graph's node are replaced by disjoint clusters of nodes. CrossFlux (87) is a system developed at the Computer Science department of the Neuchatel University. This system, beyond the multicast tree links, introduces some backup links for fast failure recovery. Moreover, ClossFLux

implements a mechanism to optimize the overlay based on the nodes' available bandwidth.

.

2.3.2 Unstructured P2P live streaming systems

In conventional tree-based P2P streaming architectures, at the same time a peer can only receive data from a single upstream sender. Due to the dynamics and heterogeneity of network bandwidths, a single peer sender may not be able to contribute full streaming bandwidth to a peer receiver. This may cause serious performance problems for media decoding and rendering, since the received media frames in some end users may be incomplete.

In forest-based systems (multiple-tree systems), each peer can join many different multicast trees, and receive data from different upstream senders. However, for a given stripe of a media stream, a peer can only receive the data of this stripe from a single sender, thus results in the same problem like the case of single tree.

Multi-sender scheme is more efficient to overcome these problems. In this scheme, at the same time a peer can select and receive data from a different set of senders, each contributing a portion of the streaming bandwidth. In addition, members of the sender set may change dynamically due to their unpredictable on-line/off-line statuses. Since the data flow has not a fixed pattern, every peer can send and also receive data from each other, thus the topology of data plane likes mesh (see Figure 2.7). The main challenges of mesh topology are how to select the proper set of senders and how to cooperate and schedule the data sending of different senders.

To accommodate bandwidth heterogeneity among participating peers, the delivered stream could be encoded with a multiple description coding (MDC) scheme at source. All pairwise connections for content delivery between peers are congestion controlled (e.g., (68)) to properly share resources with coexisting traffic.

In a mesh-based P2P streaming mechanism participating peers form a randomly connected and directed mesh (i.e., unstructured overlay) that is used for

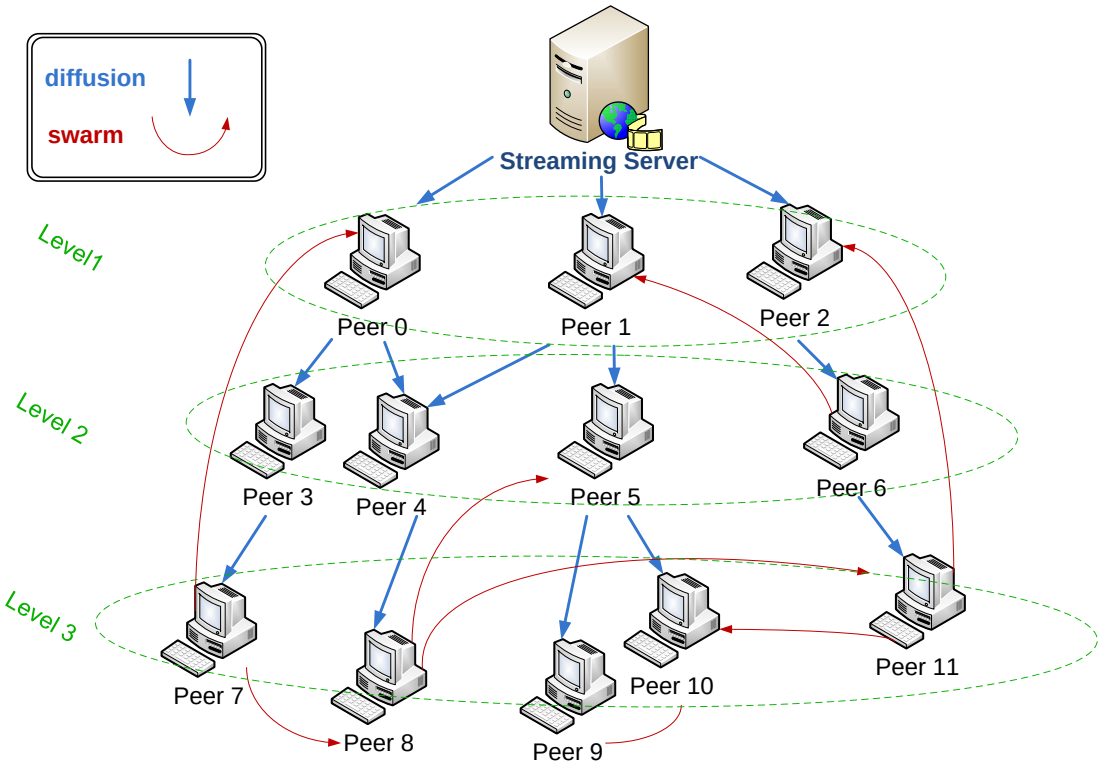


Figure 2.7: View of a mesh-based overlay with 12 peers.

content delivery to individual peers. The connection between each pair is unidirectional which means that data is delivered from a parent to a child peer. Except for the source, each peer in the overlay has multiple parents and multiple children. Maintaining such an overlay for content delivery has several advantages as follows: (i) overlay construction and maintenance are very simple, (ii) connections from different parents to each child peer are more likely to have a diverse path which in turn reduces the probability of a shared bottleneck between these connections, (iii) the resulting overlay is very resilient to churn. There are several approaches to form such an overlay. The simplest alternative is to use a bootstrapping node that maintains a list of participating peers and provides a random subset of participants to each new peer.

Content delivery among peers is performed using push reporting by parents coupled with pull requesting by child peers. Each peer receives content from all of its parents and provides content to all of its child peers in the overlay. As a

parent, each peer progressively reports its new packets to all of its child peers. As a child, each peer periodically requests a specific set of packets from each parent. Each parent peer simply delivers requested packets by individual child peers in the provided order at the rate that is determined by a congestion control mechanism. The requested packets from each parent are determined by a packet scheduling mechanism at each child peer. Given a peer's playout time as well as the available content and available bandwidth among its parents, this receiver-driven packet scheduling mechanism should select requested packets from each parent in order to maximize its delivered quality, i.e., accommodating in-time delivery of requested packets while effectively utilizing available bandwidth from all parents.

In these systems peers are no more organized in a hierarchical structure where the stream is forwarded as a continuous flow of data. Here, the source splits the stream in a series of pieces (often called chunks) and distributes them to different peers. The relations established among nodes are "data driven" in the sense that a peer establishes relations with potential providers in order to get its missing pieces.

In these systems there is not a real data path since every chunk follows a different route to arrive at peers. It is not possible to model these systems through mathematical analysis since the unpredictability of data exchanges. Some papers presents mathematical models where they can assure a bound for the number of hops a chunk does before to arrive at peers. However these mathematical models are only valid under great assumptions that are not often realistic.

As a consequence, it is not possible establish a precise time bound for packet reception. It is thus necessary to adjust the stream play out time according to the download rate. This is not a trivial task since the download rate can fluctuate during time.

Unstructured systems offer good performances in term of robustness. Since there is not an overlay structure a node failure doesn't affect the system. Peers that cannot get data from a failed node, will download their missing pieces from another provider without the need of repairing mechanisms. In an unstructured system, where peers are not organized in a fairly static structure, it is not nec-

essary to place peers with more resources close to the source in order to achieve better performances.

Here the peers naturally adjust their position in the overlay according to the network changes. This means that peers with lot of resources are naturally closest to the source and, if the network conditions change, peers will naturally move far/close to the source according to the change happened.

By doing this, these systems can accommodate peers with different resources and in particular with different access capacity. These systems try to minimize the control overhead; however it is not possible to verify if the overhead generated by these systems is lower or bigger than the overhead generated by a structured system.

In these systems it is possible to introduce the concept of fairness. It is in fact possible to evaluate the behavior of the other peers and to penalize or advantage them according to it. Examples of mesh-based multi-sender P2P streaming system include CollectCast (71), GnuStream (97), and DONet (CoolStreaming) (101). CollectCast puts its emphasis mainly on the judicious selection of senders, constant monitoring of sender/network status, and timely switching of senders when the sender or network fails or seriously degrades.

CollectCast operates entirely at the application level but infers and exploits properties (topology and performance) of the underlying network. Each CollectCast session involves two sets of senders: the standby senders and the active senders. Members of the two sets may change dynamically during the session. The major properties of CollectCast include the following: (i) it infers and leverages the underlying network topology and performance information for the selection of senders. This is based on a novel application of several network performance inference techniques; (ii) it monitors the status of peers and connections and reacts to peer/connection failure or degradation with low overhead; (iii) it dynamically switches active senders and standby senders, so that the collective network performance out of the active senders remains satisfactory.

GnuStream is a receiver-driven P2P streaming system which is built on top of Gnutella (3). It features multi-sender bandwidth aggregation, adaptive buffer control, peer failure or degradation detection and streaming quality maintenance. GnuStream is aware of the dynamics and heterogeneity of P2P networks, and

leverages the aggregated streaming capacity of individual peer senders to achieve full streaming quality. GnuStream also performs self-monitoring and adjustment in the presence of peer failure and bandwidth degradation.

However, the most famous system that implements such approach is CoolStreaming/DONET and we are going to present it in the next section.

2.3.2.1 CoolStreaming/DONET

DONET is a data-driven overlay network for media streaming presented in (101). Every peer of the systems has to focus its attention on three main aspects: the knowledge of other nodes in the network , the selection of peers to exchange data with and the chunk scheduling.

In DONET the first problem is solved thanks to the exchange of knowledge messages using a gossip membership protocol to distribute them. When a node receives a membership message, it will update/create the entry in the membership list. Then it forwards the message to another randomly selected peer until the message is spread to all nodes. When an entry is not update for a certain time is discarded.

A node keeps M partners with who exchange chunks among the peers present in the membership list. It periodically randomly selects new partners and gives them a score. The score is calculated for each peers as the maximum value between the number of chunks sent and received from it. After exploring new partners the one with the lowest score among the new and the old ones is discarded.

The scheduler uses a *Rarest first* policy between the chunks owned by the partners to select data to require to each of them. The sender simply sends the required chunks to its partners. Information about the chunks a peers own is spread among partners using a bitmap; the chunk number of the first chunk of the bitmap is also sent with it. Receiving these 2 information a peer can select what chunks download and from what node. The playback delay in DONET is semi-synchronized.

2.3.3 Other approaches

There exists some systems that cannot be easily classified in the previously presented approaches because they utilize hybrid solutions. We are going to present Bullet that is probably the most significant example among them.

These systems try to exploit the positive features of both structured and unstructured approaches mitigating their drawbacks. In particular they are able to well exploit the resources present in the network achieving a high total bandwidth. However they could probably have an high overhead due to both tree management and missing pieces recovery.

2.3.3.1 Bullet

Bullet (65) is a data distribution system targeting large-file transfer or real-time multimedia streaming to a large number of receivers. This system is to utilize both a tree structure and a mesh overlay to distribute the content among peers.

The data content is divided into chunks and they are sent in different points of the network. Peers receive data from their parents of the tree but they have to find other peers to download missing chunks from.

To summarize there are 3 main strengths to solve for a system like this: an efficient tree building and maintenance, a way to spread nodes' information in the network (membership management), an efficient peer selection strategy.

As concern the first aspect Bullet is designed to work with different kind of tree overlays. The authors assume that the slowest overlay link is the one that determine the throughput of the entire tree. So the best possible tree overlay to use would be the one with the maximum bottleneck link (OMBT). Build this kind of overlay is a NP-hard problem and the authors have proposed an off-line greedy OMBT algorithm. It works attaching an entry node to the node, already in the tree, with the higher throughput overlay link to it.

As concern membership management Bullet uses RanSub (66) a scalable approach to distribute uniform random subset of global state to all nodes of an overlay tree. This is done by collect and distribute messages. Collect messages start from the leaves of the tree and go up to the source leaving state information at each node. Distribute messages start at the source and propagate down to the

tree distributing random subset of participants to all nodes exploiting information of collect messages. These operations are done at every constant period of time (called epoch) and the messages propagated among nodes contain information about peers and packets they have. So receiving these messages a node is able to find and locate interesting peers from which download data.

Every node periodically evaluates its sender and receiver peers. Providers are evaluated by counting the number of duplicated packets they sent to the local node with respect to the total number of packets received. If this ratio is bigger than a threshold a new possible provider will be tested. If this new peer offers better performance, it will replace the old one. In the same way a peer evaluates its receiver peers by counting the number of useful (not duplicated) packets the peer is sending to the receiver. Periodically every peer drops the worst receiver and inserts a new one. Bullet can achieve a total bandwidth that is twice with respect to a simple tree solution and 60% more than a simple gossip protocol. Bullet well performs in lossy network and in presence of transient peers.

2.4 Outline

In the previous sections we have presented different approaches for P2P live streaming systems. It is not easy to make a comparison among them since the evaluation can vary according to the assumptions we make on the application environment.

In 2.1 we make a comparison between different systems supposing as application environment the today's Internet. Thus we are dealing with a large set of heterogeneous hosts that can join/leave the system in an unpredictable way. Hosts mostly differ in their access capacity and present an unpredictable behavior.

In such environment unstructured systems seems to better exploit peers' heterogeneity. Moreover they show a better resilience to transient peers. However the quality of the data path is unpredictable since the unforeseeability of the data exchanges. Structured approaches are able to achieve better performances in term of quality of the data paths but they suffer the peers' transiency.

Hybrid solutions perform well in exploiting heterogeneity and responding to peers' transiency. However they present an high overhead and high management

Approach	Quality of data paths	Robustness	Adaptiveness	Fairness	Control Overhead
Structured	Very good	Bad	Medium	-	Medium-Low
Unstructured	-	Very good	Very good	Good	Medium-High
Hybrid	Good	Good	Good	-	High

Table 2.1: Comparison of main P2P steaming approaches

complexity since they have to maintain both a tree structure and a mesh overlay. There is not the "best" approach; every solution has its advantages and its drawback. An approach may thus be suitable or not according to the goals a system want to achieve and the environment where it will be used.

At the moment peer-to-peer streaming is securing its position among users and the implementations are more or less unfinished or oriented to a small group of broadcasters. Compared to traditional unicast media transfer, peer-to-peer streaming offers a lot of improvements concerning the bandwidth usage and server requirements. Although the improvements are great and help small organizations to build, e.g., Internet radio to a large audience rather easily, people are probably unaware of the opportunities.

Chapter 3

Media Streaming over P2P

Because of the scalability issue, client-server architecture has been replaced by P2P architecture in some networking applications, such as file sharing and media streaming. In P2P streaming architecture, each peer not only downloads media packets from other peers as a client, but also uploads its media packets to other peers as a server. This effectively enlarges system's serving capacity, as more peers contribute their bandwidth and resources.

Measurement studies of current P2P products, such as Peercast (10), found that they predominantly use TCP as their transport protocol. However, it is commonly accepted that UDP, a light-weight transport protocol without connection setup and congestion control mechanisms, is better suited to be used by real-time streaming applications.

In the previous chapter the focus was on analyzing the existing applications for streaming media over p2p networks. In this chapter we follow the data flow to investigate issues in streaming the media data over the network to the client hosts. We can separate the media streaming problem into two aspects: protocol and scheduling. The former covers the issues in the design of the transport/application layer protocols between the media server and the media client. Some key issues include resource identification, playback controls, media data synchronization, authentication, and digital rights management, etc. The latter covers the data transmission issues such as scheduling media data transmission to sustain continuous media playback, transmission of variable-bit-rate media streams, and adaptation of the media stream to changing network conditions.

3.1 Transport Protocols related with Streaming

This chapter addresses the protocol issues, discusses the feasibility of streaming media data using the existing Internet transport protocols (TCP/UDP), and gives a brief overview of the recently standardized Internet streaming protocols RTSP, RTP, and RTCP. The scheduling issues are not dicussed in this chapter.

3.1.1 Streaming over TCP/UDP

Before discussing the specialized streaming protocols, let us first investigate the feasibility of using the existing Internet protocols for streaming applications. If we consider the transport layer protocols, then the Internet already supports the Transmission Control Protocol (TCP) (52) and the User Datagram Protocol (UDP) (53). TCP is the transport protocol used by most of the Internet applications, including the WWW, FTP, telnet, and so on. It is a connection-oriented protocol that has built-in error control, flow control, and congestion control (58; 70). In other words, TCP shields the application from much of the complexities in managing traffic flowing through the Internet. This greatly simplifies application development and TCP also possesses a desirable property – it shares network resources with other competing traffic flows in a fair manner (25). So, given the many desirable features of TCP, the natural question is, can we simply stream media data over TCP as depicted in Figure 3.1?

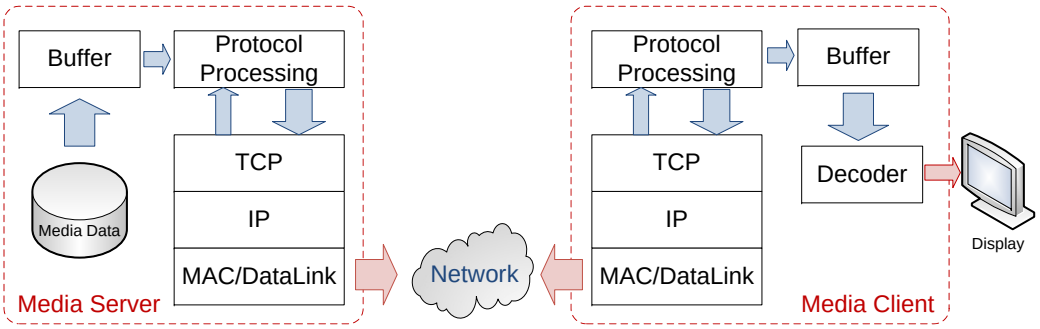


Figure 3.1: Media streaming over the Transmission Control Protocol (TCP).

The answer depends on the bandwidth requirement, network characteristics, and the desired quality of service. For example, if network bandwidth is abundant

compared to the media data rate, then streaming media data over TCP will likely work well. In fact, many web portals simply host media contents using ordinary web servers. Thus, when the client application requests the media content using the Hyper-Text Transfer Protocol (HTTP) (80), the web server simply sends back the media content over HTTP, which in turn makes use of TCP for data delivery.

In this case the web server does not explicitly stream the media content as it simply sends the media object as fast as TCP will allow, regardless of the media object's intrinsic data rate. The client application after receiving certain amount of data often can begin playback without waiting for the whole media object to be completely received. As long as the media data flow can keep up with the playback data rate, the end result is very much like streaming.

Streaming over HTTP/TCP has many obvious advantages. First, since the web server is used to serve media contents, the service provider will not need to invest in expensive specialized media servers. Second, deployment is simplified as the traffic is treated in the same way as ordinary web traffic, thus enabling them to transparently traverse firewalls and gateways. Third, the wide support for HTTP enhances compatibility with the client applications. Most media player software supports pseudo-streaming over HTTP/TCP protocol in addition to their own proprietary streaming protocols.

The downside of HTTP/TCP streaming, however, is in performance. At the application level, the web server is not designed to deliver time-sensitive media data and thus it may not always be able to sustain a smooth and jitter-free media playback, e.g., when the web server load is high. At the transport level, TCP's features have been developed for generic applications and thus have no provision for time-sensitive and bandwidth-sensitive applications such as streaming media.

For example, TCP's congestion control algorithm ramps up the transmission speed slowly after connection set-up (i.e., the slow-start algorithm), regardless of the bandwidth demand of the application (58; 70). Moreover, the error control feature in TCP enforces correct and in-sequence data delivery. This means that if a TCP segment is lost, the TCP sender will simply keep retransmitting the lost segment until either an acknowledgement is received from the TCP receiver (see Figure 3.2) or it gives up trying and shuts down the connection. In a media streaming application this may not always be desirable as media data have

intrinsic timing information, i.e., they have to be played back at a certain time or the data will become useless. Consequently, if the retransmitted data arrive after the playback deadline, the data will no longer be useful and are discarded by the receiver. In this case the bandwidth consumed in retransmitting the data is simply wasted.

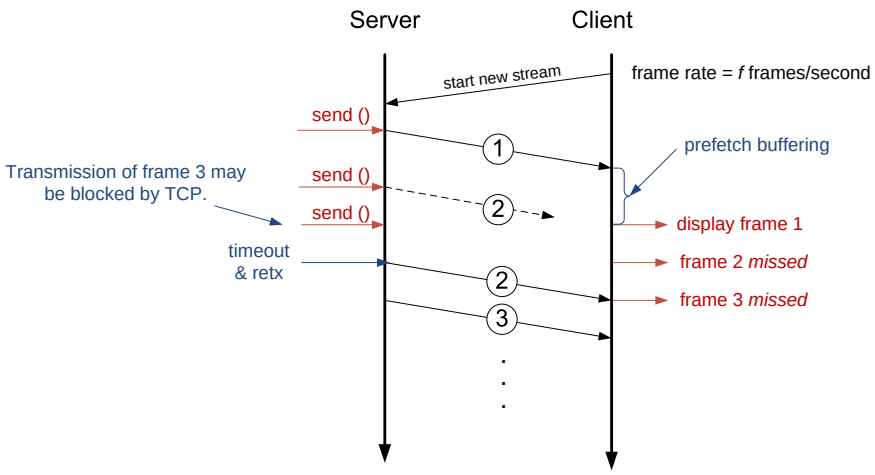


Figure 3.2: Retransmission of lost packet under TCP may cause media data to miss playback deadline at the client.

Worst still, TCP's congestion control algorithm will interpret packet loss as an indication of network congestion and thus throttle back its transmission rate by means of reducing the congestion window size (58; 70). This could end up stalling the sender from sending any more data until the congestion window grows back to normal after receiving a number of acknowledgements from the receiver. Again, this sender throttling will cause problems in media streaming as deferring transmission of the media data may cause them to miss the playback deadline, thus rendering them useless even if they are eventually received by the client.

The User Datagram Protocol (UDP), on the other hand, does not suffer from the problems of TCP as it is a relatively simple protocol that transfers datagrams without flow control, congestion control, or any error control at all. Therefore, the protocol itself will not introduce additional delay (ignoring processing time and packetization delay) like the flow control and congestion control algorithm in TCP, making it suitable for delivering time-sensitive media data. Nevertheless,

in media streaming it is sometimes still necessary to perform flow control, to react to network congestion, as well as handling packet losses. The key is that when performing these functions the timing and bandwidth requirement of the media data must be taken into account. This can be achieved by implementing another layer of streaming protocol on top of UDP, where the streaming protocol will handle the streaming-specific functions while UDP is simply used to deliver the data and control messages. We review in the next section some of the more popular streaming protocols in the Internet.

3.1.2 Specialized Streaming Protocols

Over the years a number of streaming protocols have been developed both by commercial companies and the Internet community. On the commercial side, streaming solution companies often develop their own proprietary streaming protocols for use in their streaming products. For example, Microsoft developed a Microsoft Media Services (MMS) for use in its Windows Media streaming solution. MMS employs TCP for the exchange of control messages and can send the media data over either UDP or TCP. RealNetworks also developed their own Real-Networks Data Transport (RDT) for use in their streaming solution. Because of the proprietary nature of these protocols we will not cover them further in this chapter.

On the other hand, the Internet community has also developed open standards for media streaming. This includes the Real Time Streaming Protocol (RTSP) defined in RFC 2326 (45), the Real-time Transport Protocol (RTP) and the RTP Control Protocol (RTCP), first introduced in RFC 1889, later revised in RFC 3550 (46), which became an official standard in May 2004.

3.1.2.1 Real-Time Streaming Protocol (RTSP)

The Real-Time Streaming Protocol (RTSP) is an application-layer protocol designed to control the delivery of media data (e.g., play, pause, and seek) with embedded timing information, such as audio and video. The protocol is independent of the lower-layer protocol. Thus, RTSP can be carried over TCP, UDP, or other transport protocols. The syntax of RTSP shares many similarities with

HTTP/1.1, thus simplifying implementation and deployment. However, besides the syntax similarities, RTSP differs from HTTP in many important ways.

First, unlike HTTP, RTSP is a stateful protocol, thus requiring the host to maintain state information of a streaming session across multiple RTSP requests. Second, both the RTSP server and client can issue RTSP requests. Finally, the media data are to be delivered out-of-band, i.e., using a separate protocol such as, but not limited to, the Real-time Transport Protocol.

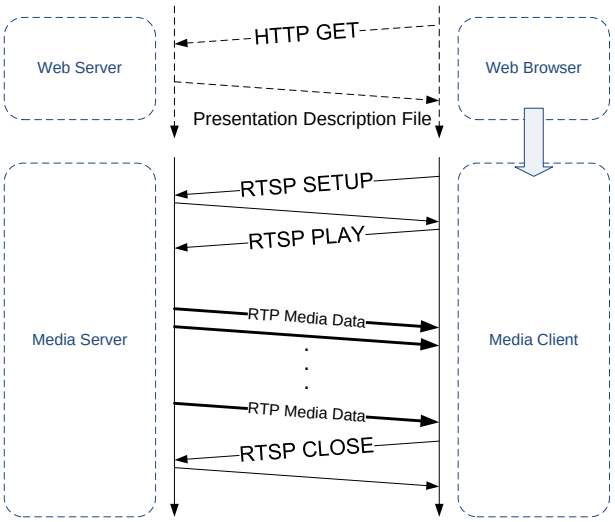


Figure 3.3: Protocol exchanges in a media streaming session.

In a typical streaming application (see Figure 3.3), the client will first obtain a presentation description file using out-of-band methods (e.g., through the web using HTTP). The presentation description file describes one or more presentations, each composed of one or more synchronized media streams. The presentation description file also contains properties of the media streams, such as the encoding format, to enable the client to select and prepare for playback of the media. Each controllable media stream is identified by a separate RTSP URL, which is similar to HTTP URL in that it identifies the server hosting the media stream and the logical path identifying the media stream. Note that the media streams in a presentation may come from multiple servers and each stream is controlled via a separate RTSP session. For more details refer to RFC 2326 for the specification of the protocol (45).

3.1.2.2 Real-Time Transport Protocol (RTP)

RTP is designed for transporting data in real-time applications such as audio and video conferencing. The protocol has been designed to be independent from the lower-layer protocol which ultimately carries the RTP packets. In the Internet RTP packets are often carried over UDP datagrams, which provides multiplexing of RTP flows within the same host (using different UDP port numbers for different flows). RTP also supports data delivery over both unicast and multicast network transports (e.g., IP multicast). A control protocol – RTP Control Protocol (RTCP) – is defined as part of the standard to provide control functions such as synchronization, reception statistics reporting (e.g., loss and delay jitter), participants monitoring, etc.

It is worth noting that RTP/RTCP on their own do not provide quality-of-service control/guarantee or perform network resource reservations. The protocols are designed to provide the necessary framework, such as header fields (sequence number, payload identification, etc.) in RTP and quality feedbacks (loss and delay jitter) in RTCP, for developers to implement their own quality-of-service mechanisms which are likely to be network and application-specific. Thus, the RTP/RTCP protocols are often extended and integrated into the application instead of existing as standalone general purpose transport protocols like UDP and TCP.

Another point worth noting is that the standard RFC 3550 does not define how the media data is to be stored inside the RTP payload. This is specified in a profile specification in separate RFCs, such as RFC 3551 (a profile for audio and video data) (88).

Figure 3.4 depicts the header format for RTP packets. The header is divided into a 12-byte fixed header that exists in all RTP packets, and a variable part containing optional headers such as the contributing source identifiers (typically inserted by RTP mixers), followed by the media data payload. A payload type field is included in every RTP packet to enable the sender (or a RTP mixer) to dynamically switch to a different encoding format in the middle of a session. This feature will be useful in media adaptation, e.g., switching to a lower bit-rate codec when available network bandwidth drops. Every RTP packet also includes

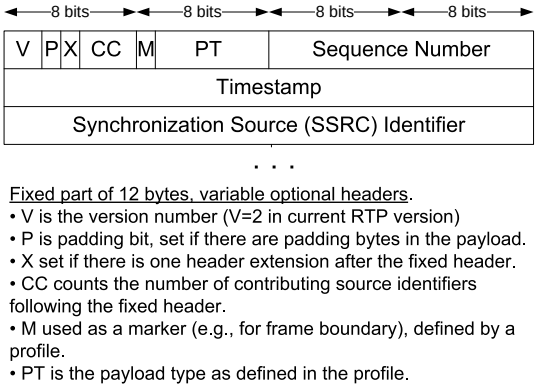


Figure 3.4: Format of the RTP packet header.

a sequence number and a time-stamp. The sequence number specifies the position of the payload within the media stream being carried by the RTP flow. Thus, the receiver can always re-sequence the incoming data into a proper media data stream even if out-of-order data delivery occurs in the underlying network.

If a multimedia stream contains multiple media streams, such as an MPEG system stream that includes an audio stream and a video stream, then the individual media streams should be delivered over separate RTP flows, in this case, one for audio and one for video. Note that the time-stamp used in each RTP flow is not measured in real time but a sampling instant derived from a clock that increments monotonically and linearly in time. The clock frequency is application and payload format dependent. Thus, the timestamps from, say, an audio stream and a video stream may not be directly comparable. Instead, the sender will periodically send Sender Report packets using RTCP to communicate to the receivers the proper interpretation of the timestamps for synchronization purpose. There are many other features in RTP/RTCP and it is beyond the scope of this chapter to cover all the features. For more details refer to RFC 3550 (46).

The RTSP/RTP/RTCP protocols have since gained increasing support from the Internet community as well as from commercial vendors. Many commercial streaming products now also support RTSP/RTP/RTCP streaming in addition to their proprietary streaming protocols. Nevertheless vendors have kept some advanced features within their proprietary protocols, such as multi-rate-encoded media streams used in adaptive media streaming.

3.1.3 NeuroCast Transport Protocol

Let us first conceptually analyze TCP and UDP protocols from an implementation point of view in the NeuroCast environment. The global Internet is a best-effort network, and it is not possible to predict which packets will get lost, duplicated, or re-ordered during transmission. At first glance, TCP may seem to be a good protocol choice, as TCP is a reliable protocol, and all the sent packets are guaranteed to be received. This reliability is achieved through acknowledgments and retransmissions. Meanwhile, using TCP in P2P streaming applications also has a number of disadvantages.

First, TCP is a point-to-point protocol. For each communication between two peers, a new TCP connection is required, and usually two system threads would be created to manage the connection, one in charge of sending operations, and the other responsible for receiving packets. In P2P applications. we have observed that the more peers that a peer is connected to, the better streaming performance it is able to achieve. As a result, it needs to create lots of system threads to mange the communications with those peers.

Second, P2P streaming data are usually time sensitive. If a requested streaming segment is received after its scheduled playback time, this segment will be discarded. A segments playback time is the time that the segment is supposed to be decoded and played by the streaming player. Since TCP guarantees that all packets must be reliably transmitted, it results in a lot of unnecessary retransmissions that have never been used at the receiver.

Third, TCP's built-in connection setup and congestion control mechanisms introduce long and variable delays in media packets, which is not ideal for real-time applications with constant playback intervals. These control mechanisms are likely to cause some packets to be "late" for their playback.

On the other hand, UDP is a connection-less transport protocol. No connection is required between two communicating peers. Peers communicate with other peers using the senders IP address attached in each UDP datagram. To this end, UDP is a one-to-many protocol. It can reuse the same UDP end point to communicate with multiple peers. If a peer would like to connect to and to receive media packets from multiple packets, it would create only one system

thread to send streaming segment requests. A second system thread can be created to receive and handle received segments. Based on the experiments, the processing speed of each receiving thread can handle the requests sent from six to eight peers.

As a result, using UDP in P2P streaming application decreases the number of system threads created and the system resources used. Moreover, since UDP does not require that all streaming packets to be received by the receiver, when a streaming segment is lost during transmission, it will not be retransmitted, thus avoiding unnecessary retransmissions of all other later packets.

Therefore, since TCP is the dominant protocol in the Internet and enjoys a more stable and reliable streaming quality than UDP, it is reasonable to employ TCP for video streaming: recent measurement study in (96) has reported that 44% of video streaming flows are actually delivered over TCP. Especially, in many situations, video streaming servers are located behind firewalls that permit only pre-specified port numbers. In this scenario, video streaming over TCP is the only choice to get around the firewalls using well-known port numbers (e.g., HTTP or RTSP). Also, the reliable packet delivery of TCP is important, when error resilience is not implemented in a video codec. While the use of TCP provides reliable video stream delivery, it is difficult to provide good quality of streaming video over TCP: i) the sawtooth behavior of additive increase and multiplicative decrease (AIMD) incurs significant data rate variability, and ii) the use of retransmission timeouts may introduce unacceptable end-to-end delay, and the retransmitted data may be delivered too late for display.

These drawbacks of TCP can be mitigated to some extent through the use of receiver-side buffering (e.g., see (28; 61)). The buffer size has to be large enough to insure that video data is not lost. In current practice, however, there are no guidelines for the provisioning of the receiver buffer, and smooth playout is insured through over-provisioning. We are interested in memory-constrained devices (e.g., mobile phones or PDAs) where it is desirable to determine the right playout buffer size. While the use of TCP provides reliable video stream delivery, the bursty nature of TCP requires buffering at a receiver for smooth video playout. Since it is desirable to determine the right size of playout buffer in memory-constrained devices, we quantify buffering requirements to achieve

desired buffer underrun probability by analytically modeling a video streaming system.

The use of TCP in NeuroCast, as in any other video streaming application has several limitations that are studied in the following.

3.1.3.1 Impact of TCP Throughput Under-Provisioning

When a streaming flow is congested inside network, and TCP throughput is decreased less than the video encoding rate, it is difficult to achieve desired video quality. The only solution is to employ a large amount of buffering delay so that a playout buffer not only accommodates TCP rate variability but also prefetches part of a video stream. In this section, we investigate the effect of TCP throughput under-provisioning and derive the buffer size requirement to achieve desired buffer underrun probability.

Note that slight under-provisioning of available bandwidth incurs significant increase of buffer underrun events. Conversely, slight increase of the video encoding rate is critical to perceived visual quality: it is expected from experimental results that, when streaming a CBR video, 10% under-provisioning of available bandwidth will incur more than twice of buffer underrun events, and therefore perceived quality will be degraded significantly. Therefore, end users should try to make a video streaming application operate on a well provisioned environment for the maximum video quality.

3.1.3.2 Impact of Window Size Limitation

TCP throughput is sometimes limited by the receiver or sender side buffer size. The window size advertisement from a receiver was designed for end-to-end flow control, such that a sender should not transmit more data than can be accommodated by a receiver's capability.

Some video streaming applications transmit packetized video at the data rate at which it was encoded. In this scenario, a sender does not exploit the whole available bandwidth, even the amount of available bandwidth is greater than the video encoding rate. Note that this behavior exhibits the same effect as the

sending buffer limitation, since packets are trickled into the network at the data rate of the window size per RTT.

In our study, we investigate buffer underrun characteristics. Experimental results show that, when TCP throughput matches video encoding rate, measured buffer underrun probabilities of more than 96% of streaming flows are smaller than desired buffer underrun probabilities. In the case of TCP throughput under-provisioning, measured buffer underrun probabilities are increased significantly. When TCP throughput is limited by the maximum window size, it is observed that measured buffer underrun probabilities are decreased, since TCP throughput variability is reduced. Still, measured buffer underrun probabilities are smaller than desired buffer underrun probabilities.

3.2 P2P Streaming Characteristics

Live video broadcasting over the Internet requires an infrastructure capable of supporting a large number of simultaneous unicast connections. Since the costs of providing this service grow with the number of viewers, television networks have been reluctant to offer it to their customers on a large scale. Peer-to-peer architectures are an alternative where viewers contribute their resources to the network to act as relays, hence overcoming the need for a dedicated content delivery infrastructure.

Peer-to-peer video streaming systems offer the same advantages as peer-to-peer file transfer networks but face additional challenges since data transfer needs to occur continuously to avoid playout interruptions. This is particularly difficult since the peers are connected to the Internet by links which may have different capacity and reliability. Moreover, data delivery paths may simply disappear without prior notice, e.g., when a peer leaves the broadcast. This challenging environment is a perfect field of application for recent advances in compression, streaming, and networking and a catalyst for new progress. Remarkably, functioning solutions have emerged and the research community now expects that in the future peer-to-peer video streaming system will be used for large-scale live television distribution over the Internet.

Peer-to-peer streaming is a concept for distributing streaming content over a peer-to-peer network. Also the term application layer multicast is sometimes used because application layer connections are used to form the peer connections. The streaming media needs to be injected to the network for delivery, and it is further being delivered through the whole network to the clients wishing to receive the data. Sometimes the users may not want to receive the data (e.g., a TV channel), but will act as a relay node, also referred to as "reflector", to other clients within the network. Reflectors are hosts that pass the streaming data to other hosts without consuming it themselves.

As these networks are peer-to-peer networks, the user wishing to receive (or act as a reflector) needs to join the network before the actual data traffic can occur. After the user has joined the network, there is a varying warm-up time before any data can be consumed. This is because of the initial buffering of data before a stream can be presented in order to ensure seamless viewing or listening of streaming media. The length of the warm-up time depends on the amount of users attending in the network, as well as the users' network capacity and the overall network latency. Also the software used for receiving and playing the stream and stream encoding format affects to the duration of the warm-up time. When a larger reception buffer is used in the software, the warm-up time is longer. A higher quality (low compression ratio and higher bitrate) stream usually takes a longer warm-up time than low bit rate stream of low quality.

In P2P multicast, a media stream is sent to a large audience by taking advantage of the up-link capability of the viewers to forward data. Similar to file transfer networks, data propagation is accomplished, via a distributed protocol, which lets peers self-organize into distribution trees or meshes. The striking difference is that this should happen in real-time, to provide all connected users with a TV-like viewing experience. Compared to content delivery networks, this approach is appealing as it does not require any dedicated infrastructure and is self-scaling as the resources of the network increase with the number of users.

To become widely adopted, P2P streaming systems should achieve high and constant video quality, as well as low startup latency. Three factors make this a difficult task. First, the access bandwidth of the peers is often insufficient to support high quality video. Second, the peers may choose to disconnect at

any time breaking data distribution paths. This creates a highly unreliable and dynamic network fabric. Third, unlike in client-server systems, packets often need to be relayed along long multi-hop paths, each hop introducing additional delay, especially when links are congested. This unique set of challenges explain why early implementations, although they constitute remarkable progress and demonstrate the feasibility of large scale P2P streaming, fall short of the goals.

3.2.1 Network Layout

The network layout varies depending on the technique in use. Typical layouts are tree and mesh. In a tree layout network, the stream is divided to several hosts in each node, so that after each node the amount of receivers within the network is multiplied. So, each host in the network acts as a point-to-multipoint server, so that one host receives one stream and delivers it down to several hosts. In a mesh network, each node is connected to several other hosts, and each can receive and send out multiple streams.

On a tree layout network there exists the single point of failure type of problem. When considering strictly a tree layout network, every node of the tree (excluding leaf nodes) is a root node having one or more child nodes. A stream is always passed through the root node to its children. If any of the root nodes happens to fail, the whole network originating from the failed root node fails to receive the stream. The worst case scenario is that the primary root for the whole tree fails, leading to denial of service for the whole network.

Such single point of failure issues can be avoided, or at least reduced in a combined tree-mesh or a mesh layout network. In these kind of networks all or at least some of the peers (nodes) have more than one connection to other nodes, so that instead of just passing the stream from root to children, also the children to root direction is used. This allows other nodes to receive the stream when a single (root) node fails. Thus compared to tree layout, mesh layout requires more complicated routing algorithms or request mechanisms between peers due to the increased number of peer connections.

3.2.2 Push and Pull Methods

One common feature shared by some peer-to-peer streaming systems is that they are push-based systems. This means that after a peer has received data (the stream) it sends it on to other peers in the network, without explicit requests for data from other peers. The forwarding decision is based on some predetermined routing algorithm, and the same algorithm is globally used over the whole network. This also leads to the network layout to be somewhat rigid, at least to some extent, because it is determined by the routing algorithm.

The problem with push-based systems is that they are poor in recovering from transmission losses, caused by the lack of requests for data. For example, if a peer connection is broken between two peers, a sending peer fails to forward the data to the receiving peer across this broken connection. This leads to the receiving peer never receiving data, because of the broken connection, thus experiencing a corrupted stream. Another problem in a push-based network is the amount of duplicate data. Because of the routing algorithm used for "blindly" forwarding (pushing) the data, it may well be that one or more peers may be sending the same packets to a common destination host. This can be avoided using requests to get the desired packets from the sending peers, leading to pull-based system.

In a pull-based system, peers wishing to receive the stream request the missing packets from other peers. That is, a peer wishing to receive a packet from other peers must request it prior to receiving. After receiving a packet, peer must notify other peers about the packet it received in order to pass the stream along in the network, thus enabling other peers to request the data.

However, if for some reason a packet is not received by a peer, it may request it from one or more peers announcing to have that packet. This results in better resilience against packet loss in reception, because in case of a failure the receiving peer can redirect request packets to another peer having the desired data. Also, when using a request based method, there is no need for using predetermined routing algorithms as in the push-based approach. The amount of duplicate data sent within the network is reduced, because the requests from the peer wishing to receive a packet may only be addressed to one sender who provides the packet to the receiver.

An obvious weakness with the pull-based method is the issue of dealing with free-riders. A free-riding peer is only requesting and receiving packets from other peers, without uploading anything to others. It is obvious that this affects to the performance of the network, because free-riders do not send requested packets to other peers. In a pull-based network the layout may well be more ad hoc in nature, because there might not be any predetermined way to form peer-relations, but the requests and announcements sent within the network define peer-relations.

3.2.3 Multiple Stream Approach

To deal with, e.g., the free-riding problem apparent a in single stream based delivery utilizing peer-to-peer streaming networks, there is an alternative approach to deliver streaming data: using multiple streams. In this approach a single stream (full stream) is divided into several sub-streams, also referred to as "descriptions". Each sub-stream is then forwarded separately in the network. A peer receiving all of the sub-streams can reconstruct the full-quality original stream from the sub-streams received, and therefore enjoys the best quality.

Using multiple streams in a peer-to-peer streaming network is a potential way for implementing an incentive mechanism in such a network. The incentive mechanism can be considered similar to "scoring" in traditional peer-to-peer file sharing networks, where the uploading peers are scored by the amount of data they upload. The more scores a peer has, the more privileged are the peer's download opportunities. In a streaming network, a peer uploading more streams has better "scores", thus being entitled to download more sub-streams and getting a higher quality stream itself. For example, for every uploaded stream, a peer is entitled to download one stream, thus enabling fairness among the peers in the network.

In addition to free-riding, the multiple stream approach helps to relieve the churning phenomenon also. If a peer sending out one sub-stream leaves the network, the overall stream quality does not collapse remarkably, because satisfactory quality of the overall stream can be achieved from the sub-streams that still exist. In churning, multiple (probably hundreds) peers rapidly attach and detach to the network within a short period of time, making sub-streams rapidly

available and unavailable. If we compare the multiple stream approach to the single stream approach in a churning, we avoid the following problematic case. Suppose that a peer sending out full stream leaves the network. If it is the only provider for that stream, the reception of the stream will be cut off from all the peers listening the stream sent by the detached peer. Hence, large amount of peers joining to and departing from the network in a single stream system may cause full loss of stream at times.

3.2.4 Mobile Aspects

There are special requirements for the access networks and peer-to-peer streaming applications when they are used in a mobile network environment. Delay in the access network has an effect on the buffering period at the beginning of the stream. Furthermore, the role of the application buffer size is much more important. If it is big enough, the effects of delay and jitter can be ignored. The access network throughput limits the quality of the stream: the bigger the used stream bitrate is, the bigger the throughput should be. If the used stream bitrate is too high then it is not possible to use the application over mobile network at all. Currently used content encoding formats do not have much effect, but when optimized solutions for mobile devices will exist then CPU capabilities might exclude some content encoding formats.

All these aspects affect to the quality of experience and the usability of the peer-to-peer streaming application and should be taken into account in a mobile network environment.

3.3 Conclusions

Media data transmission comprises two separate issues – protocol and scheduling. With the standardization of RTSP/RTP/RTCP set of protocols, more and more media streaming applications will support this set of Internet standards, thereby enhancing the inter-interoperability of media servers and clients from different vendors. Scheduling, on the other hand, is a more complex problem involving issues in admission control, resource allocation/reservation, and data transmission

scheduling. Ideally, if the media data have a fixed playback data rate, and the network supports resource allocation (i.e., ability to allocate and guarantee a given amount of bandwidth from a source to a destination for the duration of the media streaming session), then the problem of transmission scheduling is nothing more than keeping the transmission rate the same as the playback data rate.

In practice, both assumptions may be invalid – the media playback data rate may not be constant and the available network bandwidth may also fluctuate. For example, the choice of media encoder (CBR versus VBR codec) determines if the media playback data rate will vary; and the underlying network architecture, e.g., whether it is a best-effort network or one supporting QoS guarantee, determines if the available network bandwidth will fluctuate.

In this thesis we skip the scheduling problem for streaming variable bit-rate media over a mixed-traffic network that supports resource allocation/reservation, though we are aware of its importance. This scenario is representative of residential broadband networks operated by a service provider, who have control over the design and operation of the physical network. For best-effort networks such as the Internet, resource reservation is obviously unavailable and thus we need to approach the media streaming problem from another angle – adapting the media to the network bandwidth available. We decided to use TCP as the transport protocol of NeuroCast . The choice of TCP connections for data transfers is motivated by the need of reliability, as data chunks span over several IP datagrams, making an unreliable transport protocol such as UDP unsuitable for the task.

On the other hand, regarding the P2P characteristics, compared to YouTube (16) and other similar techniques, peer-to-peer streaming solutions are still far from being mature. A lot of work needs to be done for example on usability, in order to gain massive public support. People do not always want to, or even cannot, download the client software or share their bandwidth for security reasons, which prevents effective spreading of these techniques. For the average user, YouTube allows easy-to-access and easy-to-use services in order to spread user content and to share experiences in a community-like environment. But if the intention is that of professionally broadcasting high quality content over the Internet to a large-scale audience, then peer-to-peer streaming is the best choice. The currently existing peer-to-peer streaming applications are not implemented

with a mobile network environment philosophy but, as this chapter shows, some of them are already suitable to be used in the mobile environment.

Still there are many issues to be solved before optimized solutions for mobile devices will be available. One way to enhance the mobile usage with current applications is to implement support for multiple stream approach, e.g., with Multiple Description Coding (MDC) (44) or Advanced Video Coding (AVC) (17). In this way clients with low throughput access network connections are also able to play the stream and the overall quality of the stream increases.

Chapter 4

PeerCast

PeerCast is an open source streaming media multicast tool which is generally used for streaming audio and makes use of a bandwidth distributing approach where users can choose the relay to connect in the downstream (10). Another peer-to-peer media distribution system has also been named PeerCast (100). This particular implementation employs a complex IP multicast approach focused on disseminating data throughout a peer-to-peer file sharing network. In this chapter, first an overview of the main features of an improved version of this application is carried out and then we stress the implementation and performance of the open-source version.

4.1 PeerCast Features Overview

4.1.1 Peer-to-peer Network Management Protocol

Peers in the PeerCast system are end-system nodes on the Internet that execute multicast information dissemination applications. Peers act both as clients and servers in terms of their roles in serving multicast requests. Each end-system node in a PeerCast overlay network is equipped with a PeerCast middleware, which is composed of two-tier substrates (see Figure 4.1): *P2P Network Management* and *End System Multicast Management* (ESMM).

The P2P network management substrate is the lower tier substrate for P2P membership management, lookups, and communication among end-system nodes.

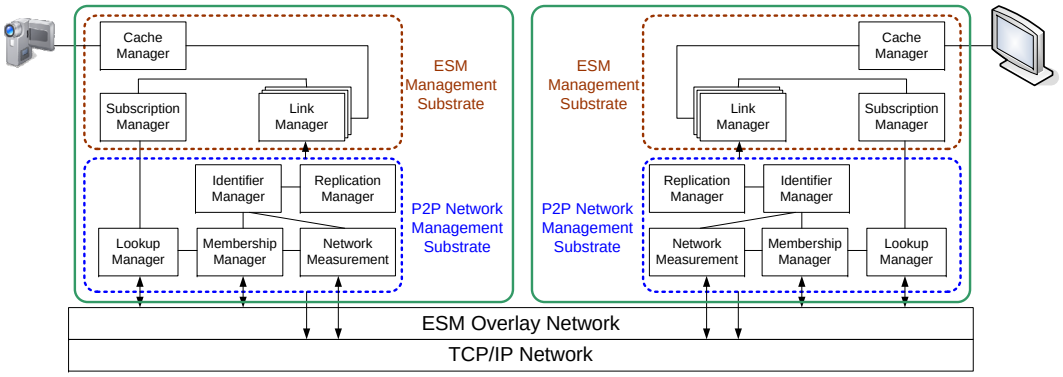


Figure 4.1: PeerCast system architecture.

It consists of *P2P membership protocol* and *P2P lookup protocol*.

PeerCast system uses the P2P membership protocol to organize the loosely coupled and widely distributed end-system nodes into a P2P network that carries the multicast service. PeerCast peer-to-peer network is a *Distributed Hash Table* (DHT) based structured P2P network. A peer p is described as a tuple of two attributes, denoted by p: ($peer\_ids$, ($peer\_props$)). $peer\_ids$ is a set of m-bit identifiers and are generated to be uniformly distributed by using hashing functions like MD5 and SHA-1 (47). Each identifier is composed of $\lceil m/b \rceil$ digits with m bits each. $peer\_props$ is a composite attribute which is composed of several peer properties, including IP address of the peer, resources such as connection type, CPU power and memory, and so on.

Identifiers are ordered on an m-bit identifier circle modulo 2^m, in a clockwise increasing order. The distance between two identifiers i and j is the smallest module-2^m numerical difference between them. Identifier i is considered as "numerically closest" to identifier j when there exists no other identifier having a closer distance to j than i. Given an m-bit key, the PeerCast protocol maps it to a peer whose peer identifier is numerically closest to that key.

A peer p invokes its local function $p.lookup(i)$ to locate the identifier j that is numerically closest to i. The lookup is performed by routing the lookup request hop-by-hop towards its destination peer using locally maintained routing information. Each hop, the lookup request is forwarded to a peer sharing at least one more identifier digit with i. In a P2P system composed of N peers, the forwarding

is of $O(log_{2^b} N)$ hops.

Each identifier possessed by a peer is associated with a *routing table* and a *neighbor list*. The routing table is used to locate a peer that is more likely to answer the lookup query. It contains information about several peers in the network together with their identifiers. A neighbor list is used to locate the owner peer of a multicast service and the replication peers of the multicast subscription information. The neighbor list points to immediate neighbors on the identifier circle. Initialization and maintenance of the routing tables and the neighbor lists do not require any global knowledge.

Network Proximity Awareness in PeerCast. A unique feature of PeerCast P2P management protocol is that it takes into consideration of the network proximity of end-system nodes, when organizing them into ESM overlays. This feature ensures the efficiency of the multicast services built over the P2P network, and distinguishes PeerCast from the other existing ESM scheme.

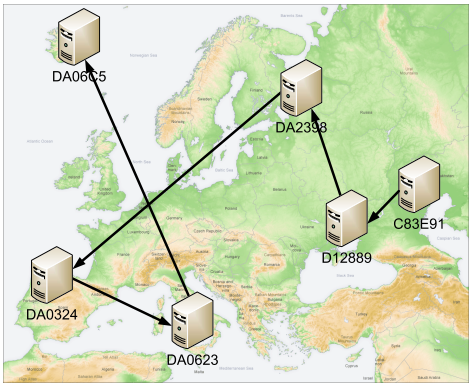


Figure 4.2: Logarithmic deterioration of routing in structured P2P network.

The basic P2P network shares with Pastry (82) of the same problem known as "logarithmic deterioration of routing". The length of each lookup forwarding hop increases logarithmically as a request is forwarded closer to its target peer, as shown in the example of Figure 4.2.

In PeerCast, it is proposed a scheme named "Landmark Signature Scheme" to tackle this problem, which we briefly describe below. A set of randomly distributed end-system nodes are chosen as the *landmark points*. The distances of an end-system node to these landmark points are recorded into a vector named

as *landmark vector*. The intuition behind this scheme is that physical network neighbors will have similar landmark vectors. This similarity information is used to twist the numerical distribution of peer identifiers such that peers physically closer will have numerically closer identifiers. Concretely, a new peer obtains a set of landmark points through the bootstrapping service when it joins the P2P network. Using this set of landmark points, the new peer generates its landmark signature by encoding the relative order of landmark vector elements into a binary string. The *landmark signature* is then inserted into its identifiers at a certain offset called *splice offset*. As the new peer joins our P2P network, it aligns itself along with the other peers that have similar landmark signatures.

Using this scheme, PeerCast system can bound more lookup forwarding hops to be within each others network vicinity, and reduce the number of long distance hops, as depicted in Figure 4.3.

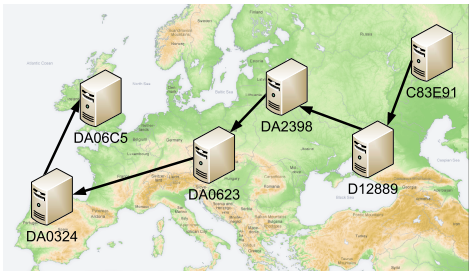


Figure 4.3: Routing regarding network proximity in PeerCast P2P network.

4.1.2 End System Multicast Management Substrate

The ESM management substrate is the higher layer of PeerCast middleware, responsible for ESM event handling, multicast group membership management, multicast payload delivery, and cache management. It is built on top of the PeerCast P2P network management substrate and uses its APIs to carry out ESM management functions. It consists of three protocols. The *Multicast Group Membership Management* protocol handles all multicast group creation and subscription requests, and adds new subscribers into the multicast tree. The Multicast Information Dissemination protocol is responsible for disseminating multicast payloads through unicast links among end-system nodes. When some peers

depart or fail in the ESM overlay, end-system nodes use *Multicast Overlay Maintenance protocol* to re-assign the interrupted multicast services to the other peers, while maintaining the same objectives – exploiting network proximity and balance the load on peers.

In PeerCast every peer participates in ESM service, and any peer can create a new multicast service of its own interest or subscribe to an existing one. There is no scheduling node in the system. No peers have any global knowledge about other peers. PeerCast organizes multicast service subscriber into multicast trees following the ESM management protocols, taking into account factors like peer resource diversity, load balance among peers, and overall system utilization. In PeerCast, the establishment of an ESM multicast service involves the following steps.

Creating Multicast Group and Rendezvous Node. An ESM service provider first defines the semantic information of its service and publishes a summary on an off-band channel. Potential subscribers could locate such information using the off-band channel. Each multicast group in PeerCast is uniquely identified by a m-bit identifier, denoted as g. Using the PeerCast P2P protocol, g is mapped to a peer with an identifier that is numerically closest to g. An indirect service is then setup on this end-system node. We refer to this end-system node as the *rendezvous node* of the ESM service. The rendezvous node re-directs subscribers to the service provider (the ESM source), who will actually inject the ESM payload into the multicast group.

Managing Subscription. Peers that subscribe to an ESM service will form a group, which the authors refer as the *multicast group*. Subscribers check those established multicast groups using the off-band channels, and identify the services that they want to subscribe. Through the rendezvous node, they learn the identifier of the ESM source. An end-system node joins a multicast group by starting the subscription process at one of its virtual nodes closest to the multicast source. The subscription request is handled in a way similar to the lookup request in PeerCast. It will be forwarded until it reaches the multicast source or a peer that is already in the multicast group. The reverse path will be used to carry multicast payload and other signal messages for multicast tree maintenance.

Efficient Dissemination using Multicast Groups. One unique feature of PeerCast is the *Neighbor Lookup* technique. Using this technique, each peer initiating or forwarding a subscription request will first check and try to subscribe to its P2P network neighbors before sending or forwarding the request. The landmark signature clustering scheme ensures that a peer can reside close to its physical network neighbors in P2P network with high probability. The neighbor lookup scheme thus let the new subscriber directly subscribe to its physical network neighbor, if it is already in the multicast group. PeerCast system can then take advantage of this property and optimize the multicast tree in various ways. Figure 4.4 gives an example of how the neighbor lookup scheme works. Peer $S_{k,1}$ first check if its P2P network neighbors have already joined the multicast group, before it forwards its subscription request to the next hop peer. It finds that peer $S_{k-1,1}$ is already in the multicast tree. It then directly subscribes to peer $S_{k-1,1}$ and terminate the subscription. Similarly, peer $S_{k,j}$ subscribes to $S_{k-1,1}$, and both $S_{n,1}$ and $S_{n,m}$ are connected to their physical network neighbor $S_{n-1,1}$.

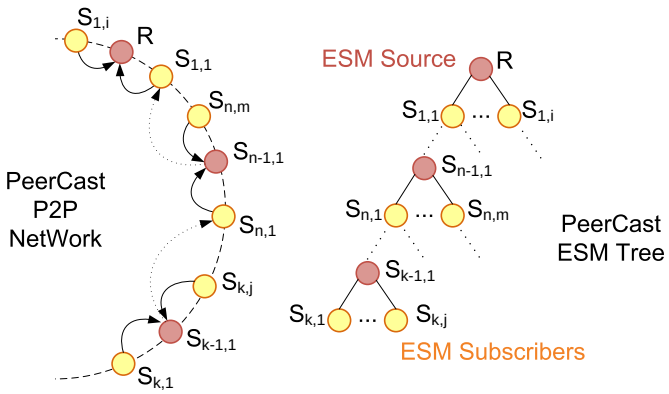


Figure 4.4: PeerCast with Landmark Signature and Neighbor Lookup schemes.

The authors also describe in more detail in their technical report (57) the overlay maintenance and optimization, which we are not going to deal with in this thesis.

4.1.3 Reliability in PeerCast

End-system multicast services are built upon the overlay network composed of widely-distributed and loosely coupled end-systems. The network infrastructure and end-systems are subject to service interruptions caused by perturbations like unintentional faults or malicious attacks. A reliable ESM system should deal with both kind of perturbations. The designers of PeerCast only focused in designing reliable ESM system against non-malicious failures.

Failure Resiliency in PeerCast. Failure resiliency is the system's capability to tolerant unintentional faults and non-malicious failures. Maintaining uninterrupted multicast service in highly dynamic environments like P2P network is critical to the reliability of an ESM system. To design such a system, following situations that may cause the interruption to ESM services are considered:

- When a peer p departs the network abnormally, say the user terminates the application before p hand-off all its workload, the ESM services to peers downstream to p will be interrupted.
- When a peer p fails unexpectedly, it will stop provide ESM services to its downstream peers. And thus they will experience service interruption.
- Even when a peer departs the system normally, if the hand-off takes longer time than the ESM overlay needs to recover the multicast service, the service of the leaving peer's downstream peers will be interrupted.
- When a peer p fails, the service replica can not be activated soon enough such that the service of its downstream peers will be interrupted.

Handling the failure of multicast sources it is not considered because, according to the authors, in the case such as video conference or on-line live broadcast, the failure of multicast sources can hardly be compensated. They assume that the ESM source is reliable and focus their efforts on building the reliable ESM overlay network.

Departures and Failures. Two types of events that depart a peer from ESM overlay are identified. A *proper departure* in PeerCast is a volunteer disconnection of a peer from the PeerCast overlay. During a proper departure, the

PeerCast P2P protocol updates its routing information. The leaving peer notifies the other peers to actively take over the multicast services that it was handling. A *failure* in PeerCast is a disconnection of a peer from the network without notifying the system. This can happen due to a network problem, computer crash, or improper program termination. Failures are assumed to be detectable (a fail-stop assumption), and are captured by the PeerCast P2P protocols neighbor list polling mechanisms. However, in order to recover a lost multicast service promptly with less overhead, a replication mechanism is needed. In both cases, multicast service can be restored by letting the peers whose services are interrupted to re-subscribe. This is the approach adopted by (83), and is also implemented in PeerCast as the "plan B" for service recovery.

Notice that once there is a replication mechanism, which enables the continuation of the multicast service from the service replica, the proper departures are very similar to failures in terms of the action that needs to be taken. This will eliminate the explicit re-subscriptions during peer departures. The main difference between a proper departure and a failure is that, a properly departing peer will explicitly notify other peers of its departure, whereas the failure is detected by the P2P protocol.

4.1.4 Service Replication Scheme

The failure of an end-system node will interrupt the ESM services it receives from its parents and forwards to its children. To recover the interrupted multicast service without explicit re-subscribing, each end-system node in PeerCast replicates the multicast service information among a selection of neighbors. The replication scheme is dynamic. As peers join and depart the ESM overlay, replicas are migrated such that there are always a certain number of updated replica exist. This property is a desirable invariable to maintain.

The replication involves two phases. The first phase is right after the ESM group information is established on a peer. Replicas of the ESM group information are installed on a selection of peers. After replicas are in place, the second phase keeps those replicas in consistency as end-system nodes join or leave the

ESM group. We denote this phase as the *replica management phase* (see Figure 4.5).

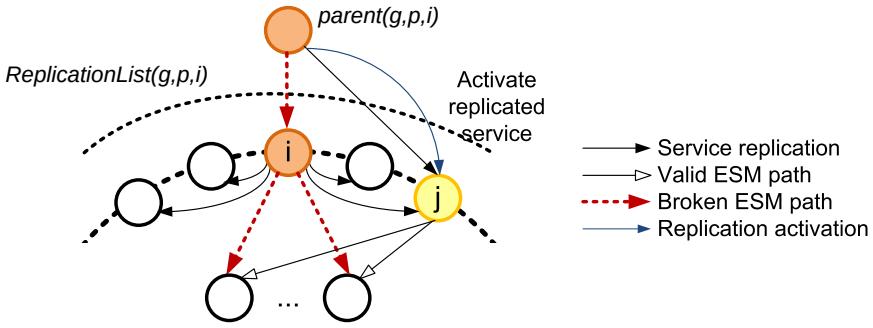


Figure 4.5: Multicast Service Replication with $r_f = 4$.

Given an ESM group identified by identifier g, its group information on a peer p with identifier i is replicated on a set of peers denoted as $ReplicationList(g, p, i)$. We refer this set as the replication list of group g on peer (p, i). The size of the replication list is r_f, which is referred as the replication factor and is a tunable system parameter. To localize the operations on the replication list, it is demanded that $r_f \leq 2 \cdot r$, which means that all the replica holders in $ReplicationList(g, p, i)$ are chosen from the neighbor list $NeighborList(p, i)$ of peer (p, i).

For each ESM group g that a peer p is actively participating, peer p will forward the replication list $ReplicationList(g, p, i)$ to its parent peer $parent(g, p, i)$ in group g. Once p departs from group g, its parent peer $parent(g, p, i)$ will use $ReplicationList(g, p, i)$ to identify another peer q with identifier j to take over the ESM multicast forwarding works of p. q will use the group information that p installed on it to carry out the ESM payload forwarding for group g. It is said that q is *activated* in this scenario. Once q is activated, it will use its neighbor list $NeighborList(q, j)$ to setup the new $ReplicationList(g, q, j)$, and use it to replace $ReplicationList(g, p, i)$ on $parent(g, p, i)$, which is equivalent to $parent(g, q, j)$ now.

The replication scheme is highly motivated by the passive replication scheme of (19). The active participant of an ESM group acts as the 'primary server' and the peers holding replicas as the 'backup servers'. However, the PeerCast's

scheme is difference in that the active peer could migrate its ESM tasks when it discovers a better candidate to do the job in terms of load balancing or efficiency.

4.1.5 Replica Management

In this section, it is explained how the described dynamic replication scheme is maintained as end-system nodes join or depart from the ESM system. Since the active replication scheme works for both peer departure and failure cases, the authors use the term departure to refer to both scenarios. It is assumed that the replication factor r_f is equal to $2r$. According to the designers, in case that r_f is less than $2r$, their arguments still hold with some minor modifications to the description.

When a multicast group g is added to the multicast group list on a peer p with identifier i, it is replicated to the peers in the $ReplicationList(g, p, i)$. PeerCast P2P protocol detects the later peer entering and departure event fallen within $NeighborList(p, i)$. Once such an event happens, an up-call is triggered by the P2P management protocol, and the replica management protocol will query the peers in $NeighborList(p, i)$ and update the replication list $ReplicationList(g, p, i)$. The reaction that a peer will take under different scenarios is described.

Peer Departure. A peer's departure triggers the update of $2r$ neighbor list. Once a peer p with identifier i receives the up-call informing the departure of peer p', it will perform the following actions:

- For each group g that p is forwarding ESM payload, p adds p'', which is added into $NeighborList(p, i)$ by the P2P management protocol, to the replication list $ReplicationList(g, p, i)$.
- For each group g that p is forwarding ESM payload, p removes the departing peer p' from the replication list $ReplicationList(g, p, i)$.
- For each group g that p is forwarding ESM payload, p sends its group information to p''.

- For each group g that p is forwarding ESM payload, p sends the updated replication list $ReplicationList(g, p, i)$ to its parent peer $parent(g, p, i)$ in multicast group g.

Peer Entrance. A peer's entrance also triggers the update of $2r$ neighbor list. Once a peer p with identifier i receives the up-call informing the entrance of peer p', it will perform the following actions:

- For each group g that p is forwarding ESM payload, p adds p', to the replication list $ReplicationList(g, p, i)$.
- For each group g that p is forwarding ESM payload, p removes peer p'', which is removed from $NeighborList(p, i)$ due to the entrance of p', from the replication list $ReplicationList(g, p, i)$.
- For each group g that p is forwarding ESM payload, p sends its group information to p' as replicas.
- For each group g that p is forwarding ESM payload, p sends the updated replication list $ReplicationList(g, p, i)$ to its parent peer $parent(g, p, i)$ in multicast group g.

Updating Replicas. As end-systems subscribe or unsubscripted from ESM groups, their subscription or unsubscription requests will be propagated up in the ESM tree and change the group information on some peers. Once the group information of group g is changed on peer (p, i). p sends its group information to all the other peers in $ReplicationList(g, p, i)$.

Replica Management Overhead. Assuming in average a peer participates k multicast groups, we can summarize the replica management overhead as:

- Average storage cost for the replicas stored per peer $\sim k \cdot r_f$.
- Average update cost for replicas stored per peer $\sim k \cdot r_f$.
- Average number of new replications required for entrance/departure per peer $\simeq k$.

4.1.6 Load balancing in PeerCast

The third major limitation of the basic ESM scheme, namely the load imbalance due to node heterogeneity, is also addressed by PeerCast. Measurement studies (86) have shown that end-hosts exhibit noticeable heterogeneity in large-scale P2P networks. For a distributed system in which end-hosts rely on one another to provide services like end-system multicast, balancing workloads among heterogeneous end-hosts is vital for utilizing the full system capacity and for providing efficient services. One way to address the node heterogeneity problem is to place end-hosts of a P2P network into different service layers depending upon their capabilities. A number of systems have adopted this approach to achieve better performance. For example, KaZaA (4) and Gnutella v.0.6 (3) have the notions of super node and ultra peer, respectively. Similar strategies have also been proposed for structured P2P networks (98) and (102).

However, the above approach has a significant drawback: the predetermined hierarchical system architecture of such schemes introduce vulnerabilities into the overlay network. Usually, nodes at higher levels of the hierarchy do not use the lower level nodes to relay traffic to one another. Rather, the supernodes interact directly with one another. When these supernodes drop out of the network, the overlay network is likely to be partitioned into a number of disconnected smaller networks, thereby leading to large-scale service disruptions.

The PeerCast system adopts an alternative approach. This approach is based on the concept of *virtual nodes*. A virtual node is a conceptual entity that encapsulates all the functionalities of an individual peer. Each virtual node has its own identifier in the PeerCast system. An actual peer in the PeerCast system is assigned one or more virtual nodes based upon the resource availability at the peer. In other words, an actual peer in the system hosts multiple identifiers and is responsible for performing the functionalities of the corresponding virtual nodes. Thus, this scheme implicitly assigns more workloads to powerful nodes.

The resource availability of a peer p_i, represented as $RA(p_i)$, denotes the resources available for the PeerCast application at p_i. In the current design of the PeerCast system, the resource availability of a peer p_i is computed as the

weighted sum of three components, namely bandwidth availability, CPU availability, and memory availability. Mathematically, $RA(p_i) = W_{BA} \times BA(p_i) + W_{CA} \times CA(p_i) + W_{MA} \times MA(p_i)$ where $BA(p_i)$, $CA(p_i)$, and $MA(p_i)$ denote the bandwidth availability, CPU availability, and the memory availability at pi, respectively, and W_{BA}, W_{CA}, and W_{MA} denotes the corresponding weights such that $W_{BA}+W_{CA}+W_{MA} = 1$. The bandwidth, CPU, and memory availabilities of a peer p_i may be set by the end-user, or they may be limited by other applications running on p_i. In either case, the resource availabilities are estimated through techniques similar to those proposed by Gedik and Liu (42). The weight of a particular parameter signifies its importance to the overall performance of the application executing on the PeerCast system. Weight assignments can be based upon the specification of the publisher of the contents, and will be passed down to subscribers when they join the multicast tree. For example, when PeerCast is handling an application demanding more bandwidth, the content publisher will set W_{BA} to a high value.

The PeerCast system incorporates three load-balancing operations:

- *Generate virtual nodes:* Each end-host joins the PeerCast ESM overlay with a set of identifiers generated with different random seeds. Each identifier represents a virtual node with one unit of resource. Each virtual node maintains its own overlay state information like routing table and neighbor list.

- S*subscribe with virtual nodes:* An end-host subscribes to a multicast group by starting the subscription process at one of its virtual nodes with an identifier numerically closest to the group identifier (g_{id}) of the service. Statistically, a more powerful end-host would have higher probability to own an identifier that is closer to any service identifier.

- *Virtual nodes promotion:* The virtual node subscription scheme assigns shorter multicast path to more powerful end-hosts with high probability. However, because the scheme uses randomly generated leading digits to control the distribution of identifiers, there is still a non-negligible probability of a weak end-host owning an identifier that is numerically close to

the multicast root. Such a node would be placed closer to the root of the multicast tree. Hence, it would have to serve a large number of subscribers and can become a bottleneck.

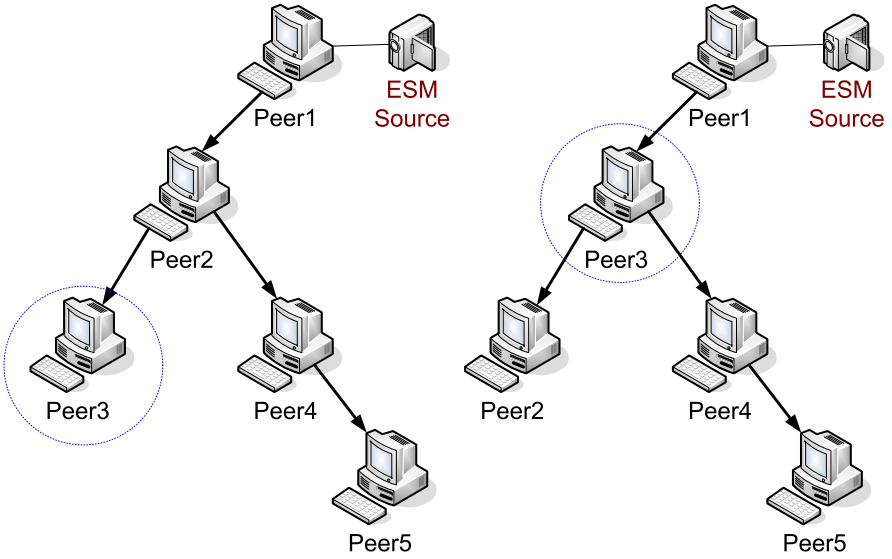


Figure 4.6: Virtual node promotion technique—node $Peer3$ is promoted by one level since it has more resources than $Peer2$.

In order to mitigate this problem, a technique called *virtual node promotion* is designed. In this technique, each node in the multicast tree periodically probes its child nodes. It chooses the child that has the most available resources (child with the maximum value of RA) as its *potential replacement*. Whenever a node detects that its potential replacement has more resources than itself, it informs the potential replacement to subscribe to its parent and informs its children to subscribe to the potential replacement. On receiving the promotion notification, the potential replacement will inform its children to subscribe to its current parent. Thus, end-hosts contributing more resources will be gradually *promoted towards* the root of the ESM tree, and they obtain better multicast service than other end-hosts. Periodic monitoring of the various resources enables the end-hosts to respond to changes in their resource availabilities by initiating a local reorganization of the multicast tree. Note that the whole process is transparent to the end users.

Figure 4.6 illustrates the virtual node promotion technique. Node $Peer2$ detects that $Peer3$ has more resources than itself, and it initiates the promotion of $Peer3$. $Peer2$ then becomes a child of $Peer3$.

One of the concerns with the virtual node promotion technique is whether it would adversely impact the physical network locality property of the multicast tree, which would in-turn impact the multicast latency. However, note that the node promotion technique guarantees that the nodes are promoted within a sub-tree of the multicast tree. The subscription management and multicast tree maintenance protocols ensure that all the nodes in the same sub-tree share the same node identifier prefix with the root node. Since, the landmark-based clustering mechanism provides the property that the nodes sharing the same identifier prefix are more likely to be close to each other in the IP network, the powerful nodes are well contained within certain network proximity, even after being promoted.

4.2 Peercast Implementation Analysis

In this section, we will analyze the core of the Peercast implementation developed Standford in 2002 by Hrishikesh Deshpande et al. (30). First, we will start observing the program flow. Thus we will show the way the different entities which compose PeerCarst interact among them. Peercast has been developed using the programming language C++ (35). C++ is a statically typed, free-form, multi-paradigm, compiled, general-purpose programming language where compilation creates machine code for a targeted machine. It is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features. C++ introduces object-oriented (OO) features to C. It offers classes, which provide the four features commonly present in OO (and some non-OO) languages: abstraction, encapsulation, inheritance, and polymorphism. Objects are instances of classes created at runtime. The class can be thought of as a template from which many different individual objects may be generated as a program runs. In Section 4.2.5 we will analyze in more detail the main Peercast classes.

Before describing the performance of the application, we introduce the most relevant entities that Peercast uses. In the following sections, this entities will be detailed.

- **Channel:** Shared content among the different peers in the network. This content can be either video or audio.
- **Servent:** Peercast object which serves the streams and handles different connections. This Peercast class performs the server functions as well as the client functions.

4.2.1 Peercast processes

Peercast is a multi-thread application, so that different processes can be executed at the same time concurrently. The multi-thread implementation makes much simpler the design of peer-to-peer application where a peer can act as server as well as client.

The number of processes in Peercast is variable and depends on the amount of connections that are established. The main thread deals with the creation of its child threads. In addition to these taks, this thread becomes passive waiting for incoming connections.

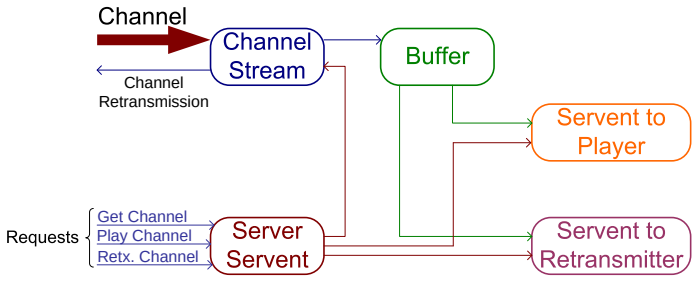


Figure 4.7: Peercast Code Architecture.

Figure 4.7 shows the architecture of the Peercast application with its inputs and outputs. Peercast receives requests and then it processes them. In general, for each request that it receives, a thread is created which will serve the request till its death.

The main thread is the *Server Servent* which handles several requests. These requests can be either internal (from the same host) or external. In the following, the more frequent requests are explained:

- **Get Channel:** Internal request. When a user wants to receive a particular channel, by means of a browser she provides to the Peercast application the channel identifier and, optionally, the IP address of any peer sharing the channel. Hence, the server creates a thread in which a `ChannelStream` object will handle the handshaking and the channel reception. During the handshaking, it is requested to retransmit the channel to another Peercast instance in any other machine sharing the channel.

- **Play Channel:** Peercast cannot play streams by itself. So, it needs to use a player which is connected to the Peercast application through the HTTP protocol. When Peercast receives this type of request, it creates a thread in which a "direct" servent will perform as a traditional streaming server. These requests are internal in general, but it is also possible to connect a player to a Peercast instance running in another peer.

- **Retransmit Channel:** External request. This request is the one we get from a peer which wants to get the channel that another peer is sharing. Thus a thread is created to handle this request in which a "relay" server sends the channel to another peer. In this type of server, the relay servent performs a PCP encapsulation, as there are two Peercast instances communicating one with each other.

4.2.2 PCP: Packet Chain Protocol

PCP is the protocol used in Peercast to allow communication among different clients. The main goal of this protocol is to reuse the same data-flow which is used to send the multimedia information, to send the control information too. If we already have a process in the application that sends/receives a channel it is not necessary to create a new connection only for sending the signaling data.

The first packet of a group indicates the type of the packets that are being sent and the number of packets that come after it. This type of packets that

indicate the type of the group are called *parent packets*. The great potential of this protocol lies in the way the packets are chained, as each one of the packets inside a group can be at the same time parent of another type of packets. Some examples of *parent packets* are:

- `PCP_HELO`: *Handshaking* packet group.
- `PCP_CHAN`: Packet group related with channel information.
- Other examples: `PCP_ROOT`, `PCP_HOST`, `PCP_GET`, `PCP_BCST`, `PCP_PUSH`...

The reason why the parent packets indicates the number of packets ahead is to ease the way the receptor processes the packets. First, it reads the *parent packet* and, afterwards, it processes a fix number of packets known a priori. Thus, it is easier to send and process variable length group of data packets. Moreover, chaining the packets allows to send the information sorted. Hence, for each type of packet there are different packets that can also perform as parent packets and, in this way create a new subgroup inside the upper group. For example, in the case of the `PCP_CHAN` group there exist the following *parent packets*:

- `PCP_CHAN_PKT`: Packet group with information from the stream: packet type, position, header, ...
- `PCP_CHAN_INFO`: Packet group with information related with channel: channel type, URL, name, bitrate, ...

Next, we will show an example describing the use of this protocol in Peercast. First, we will focus in the handshaking phase, analyzing the packets exchange between two Peercast instances. In first place, we introduce a brief description of the packet nomenclature used by the protocol:

- `PCP_HELO_AGENT`: Information about the Peercast client that the user is running. It is used during the handshaking to determine the compatibility among the users.
- `PCP_HELO_VERSION`: Information about the Peercast version.
- `PCP_HELO_SESSIONID`: Session number.

- `PCP_HELO_REMOTEIP`: Sender's IP address. It is necessary to include this address just in case the packets have gone through a NAT protocol.
- `PCP_HELO_PORT`: Port number used to send the stream.
- `PCP_HELO_NUMP`: Chunks number in which the stream is divided.
- `PCP_HELO_SUBC`: Packet group type related to the chunks that a subchannel has to send.
- `PCP_HELO_CNUM`: Specifies the stream chunks that a relay peer has to send.

Figure 4.8 shows an example of the packets that are being sent by a relay peer during the handshaking phase. In this example, the peer splits the stream in four chunks, and sends two of them (the first and the third) to the requesting peer.

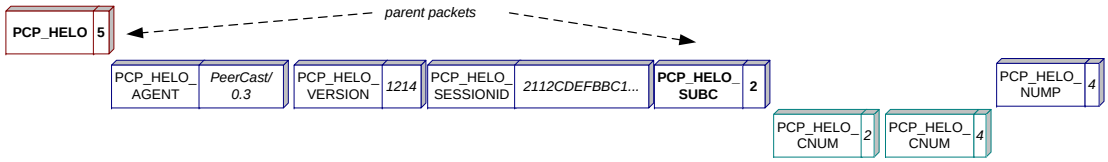


Figure 4.8: Handshaking PCP message requesting some specific stream chunks.

Continuing the example, the requested peer responds with five control packets, where it indicates its IP address and the port through which the stream will be sentd (see Figure 4.9).

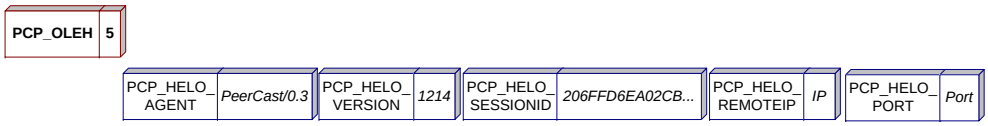


Figure 4.9: Handshaking PCP message responding the request.

4.2.3 Peercast Interaction with the user and other applications

In this section, we will analyze the process since a channel is requested until it is played. Peercast interacts with the user, with the browser, with the player and with other Peercast instances.

Imagine that a user running a Peercast instance in a host A wants to play a channel that another Peercast user is sharing in a host B. First, B has to give A a link in order to allow A to get the stream. This type of link is shown in Figure 4.10. As it can be seen, the first IP address corresponds to the address from where the peer will get the stream (at least some chunks) and the port of the emitter. Thus, the channel identifier consist of 16 hexadecimal digits. In this example, the server servent listens to the 7144 port and the user B is asking for a channel player list (*pls*). This playlist is a text file that some players use and contains one or several URL's to which the player can connect in order to download the stream.

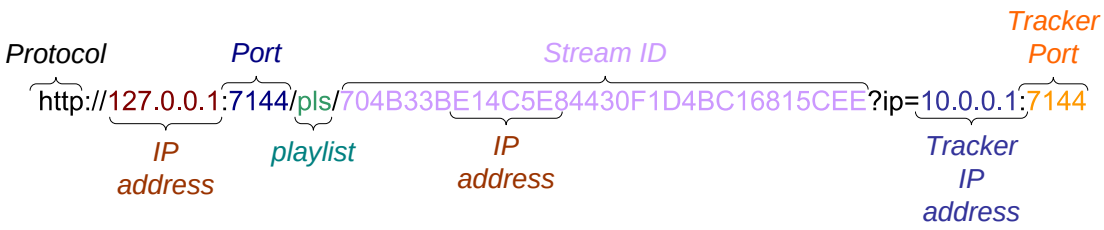


Figure 4.10: Example of a playlist link.

Once the server servent receives the playlist request through the 7144 port, it looks for the channel identifier. If the server does not found the channel in its own host, then it looks for it in the network. This checking is carried out because it is possible to close the player and continue downloading a channel. However, in this example, the host A includes in its request the IP address of the host B where the channel is available. In the case a user omits the IP address of a known sharing peer, then the server will perform a search through the `yp.peercast.org` directory.

Next, once the server has resolved where the channel is located, it communicates the link to the requesting peer. Thus, A servent creates a channel object and proceeds to perform a download request to the server servent at host B. When servent B receives the request, it creates a new relay servent which will be in charge of sending the stream to the requesting peer. At the same time, the user at host A could open the playlist in a browser and check who are the servents that are sharing the channel.

In this playlist, there will be several links like the one show in Figure 4.11. In this link example, it can be seen that for playing the channel, the player requests to the Peercast application the link. Then when servent A receives this message, it will create a new direct servent which will deal with the channel sending to the player.

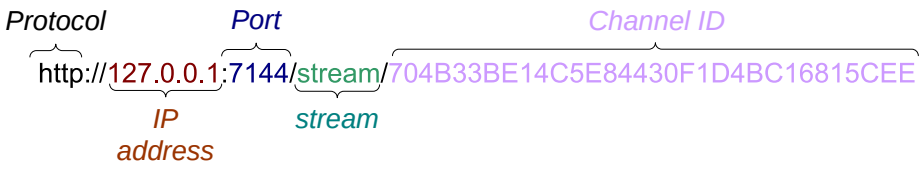


Figure 4.11: Example of a stream link.

This performance looks quite complex for a simple action like downloading a stream. However, the application is taking advantage of the oriented object programming as it is reusing the same functions in different calls to the application. Moreover, this apparently complex operation is completely transparent to the user. Thus, the user only chooses in a browser which channels wants to watch. By means of the browser, the user sends a request for the playlist, and once it gets this file, the stream is played.

The previous explained process can be summarized in eight steps (see Figure 4.12):

1. The user gets the playlist link through a browser. For instance, this link would look like:

   `http://127.0.0.1:7144/pls/704B33BE14C5E84430F1D4BC16815CEE?ip=10.0.0.1:7144`

2. With this link, the user requests to the Peercast application the playlist file.

3. The Peercast instance at the client side handshakes with its equivalent at the servent side at host B which has the IP address `10.0.0.1:7144`.
4. Host B starts sending the channel by means of a relay servent to host A.
5. Host A receives the playlist file which contains the stream link of the requested channel. In this case, it would look like:

   `http://127.0.0.1:7144/stream/704B33BE14C5E84430F1D4BC16815CEE`

6. Using the browser, the user is able to open the playlist and play the stream.
7. When the stream link is opened in a player, the Peercast instance receives a `HTTP` request asking for the channel.
8. The Peercast instance sends the channel to the player in the direct mode.

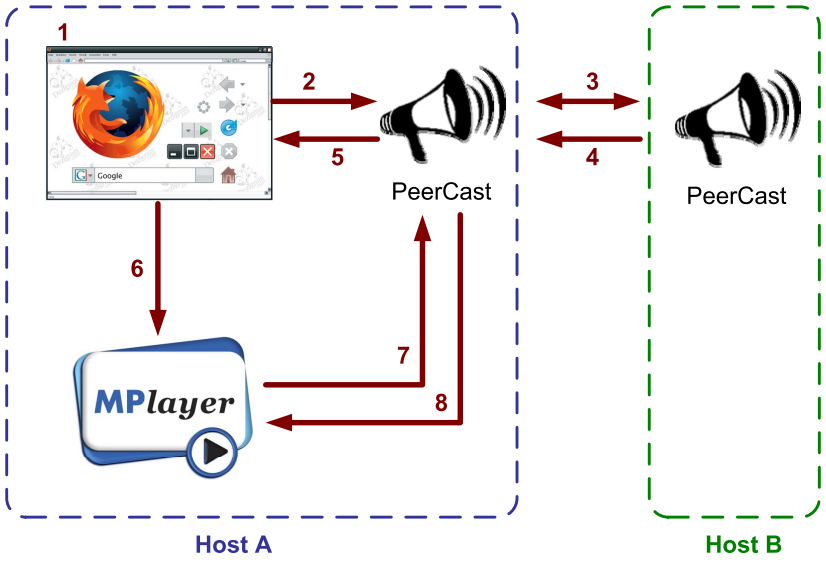


Figure 4.12: Channel downloading start process.

4.2.4 Buffer Management

Peercast adopts the ring-shape buffer mode, the architecture is as the following figure 4.13:

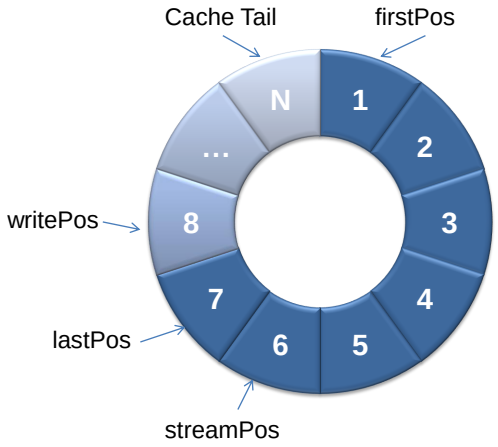


Figure 4.13: Peercast buffer architecture.

The space of each package is 16 KB, the buffer holds a total of 64 packets. The `StreamPos` is the current position which the player is reading for streaming service. `WritePos` is the position to being written into the new package. `FirstPos` is the position of the first coming package, `lastPos` is the position of the last package. All the `Pos` are the unsigned long integer structure.

Taking *ASF* (Advanced Streaming Format) streaming media as an example, the size of each package is related to the media playing rate. Broadcasters receive the package coming from the encoder and then take repackaging. It will add the `Pos` to the new package with name of "Sync". This `Pos` value is used to identify the global storage location of this package. After the client receives the package, it can get the storage position through modeling "sync" with 64, and then update other relevant `Pos`. This kind of cache approach is somewhat similar to the LRU algorithm (Least Recently Used). In the live streaming scenario, the cache structure of every client is similar, so the Peercast cache design is fit for the every client. But the main shortcomings of this approach is the cumulative delay impact to the real-time feature in live streaming.

Actually, in Peercast there is not only a single buffer, but a set of buffers. The thread that reads the packets arriving from the source, stores these packets in an auxiliary buffer named `inData`. Then, this thread analyzes the packets that has been inserted in the `inData` buffer. If it is a multimedia packet (`CHAN-PKT` type), then it is stored in the main buffer called `rawData`. The rest of the packets

(channel information, error messages, user messages, etc.) are processed but not inserted in the main buffer. Figure 4.14 show the previous explained process.

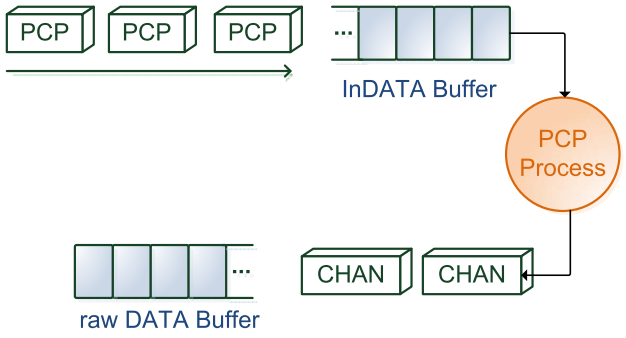


Figure 4.14: Data Flow between buffers in Peercast.

The main buffer, where the packets that contain the channel data are stored, is a buffer shared among the different threads of a Peercast instance. On the one hand, the source stream that copies the channel packets from the `inData` to the `rawData` buffer. On the other hand, the servents read the buffer and send the packets queued in this buffer to another Peercast user or to a player.

Moreover, it should be noted that the stream information is renewed completely in less than a second. That means that if the source process cannot write during a second, it loses a chunk of the streaming data which affects to the user directly. Therefore, this buffer should be managed in an efficient manner, in order to avoid losing packets through the path or overwriting the one that have not been read yet.

The fact that more than one thread can access to the buffer at the same time, forces Peercast to have some kind of mutual exclusion mechanism in order to avoid the information to be modified while some thread is still reading it. Peercast implements this mutual exclusion mechanism through a *monitor*. A monitor is an object intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared with code that may be executed in parallel. Monitors also provide a mechanism for threads to temporarily give up exclusive

access, in order to wait for some condition to be met, before regaining exclusive access and resuming their task. Monitors also have a mechanism for signaling other threads that such conditions have been met. On the other fact, using a mutual exclusion mechanism implies that the action that are performed during the exclusion period must be optimized in order to minimized delays.

When a Peercast instance reads from a buffer it calls the `CHANPacketBuffer` class which has the following interface:

```
bool ChanPacketBuffer::findPacket(unsigned int spos, ChanPacket &pack)
```

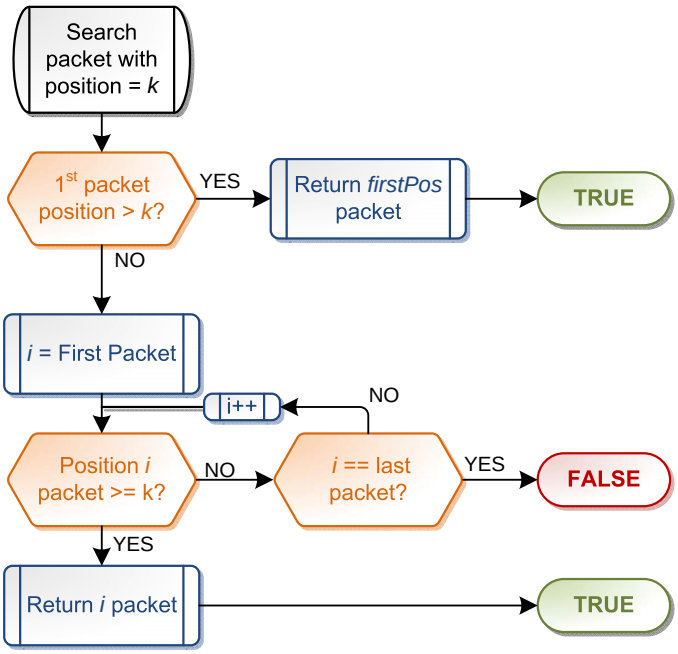


Figure 4.15: Search Algorithm Flow.

This function takes as arguments the packet to get and the memory address where this packet has to be stored. If the searched position is smaller than the oldest packet position that we have in the buffer, then the oldest packet will be returned. When a query about a packet that is in the buffer is launched, then the buffered is covered from the oldest packet towards the latest until it is found. If during the buffer covering, a packet with a higher numbering than the one requested is found , that means that the requested packet was lost. Finally, the interface return `true` if it is found the packet or one with a higher position, and

returns `false` otherwise. Figure 4.15 shows the flow diagram of the previous explained algorithm.

This simple algorithm performs efficiently, but after a deep analysis it can be concluded that it is not optimal from a performance point of view. Let us suppose that a Peercast client is receiving a channel and playing it, but this user is not broadcasting the channel to anyone. The only servent that searches packets in the buffer is the one that sends them to the player. Taking into account that packets are sent in a sequential way, as they are sorted by number, generally after sending a specific packet, the buffer is covered and, when the next packet is requested, the buffer will be covered again unnecessarily. We will handle this issue in the NeuroCast design by storing the last position of the last searched packet so that next time a packet is requested it will be find without covering the whole buffer.

In the case where apart from the servent sending packet to the player there is another one retransmitting the channel to another user, then the buffer managing will be more complex. In this scenario, the function should know which servent has called it and store the last searched position for each servent.

4.2.5 Peercast Entities

In this section we will describe in depth the different entities that compose Peercast. As mention before, Peercast was designed using the object oriented philosophy, so each one of the entities have been implemented in different classes.

4.2.5.1 Channel

The channel is the element shared among the different peers using Peercast, so it constitutes a fundamental part of the application. A channel is originated when a user starts to retransmit new multimedia information to the Peercast network. This user, the first one in uploading the channel to the network, is named the *emitter* or *broadcaster*, and it becomes the unique contact between the original source and the Peercast network of clients. At the instant when the broadcaster disconnects from the network, the channel is stopped. That means a basic difference with tradition content delivery networks where once the file has been delivered, even the original source is disconnected, the distribution

continues.

The channels are identified with a 16 hexadecimal code generated uniquely from the channel name, the broadcaster identity, the mount point and the channel rate. The main element of a channel is the buffer where the multimedia packets are stored. A channel has also a source stream which constitutes the Peercast input. Another indispensable element for a channel is the source host that identifies the machine which is serving the source stream.

From the physical point of view, a channel is a sort sequence of packets of 8192 bytes. Each packet has a position number that indicates the bytes distance from the first packet. As mentioned before, in figure 4.12, this packet is encapsulated in a PCP packet. The 8192 bytes belong to the data of the `CHAN-PKT-DATA` packet.

In figure 4.16, the main methods and attributes of the above commented class are cited:

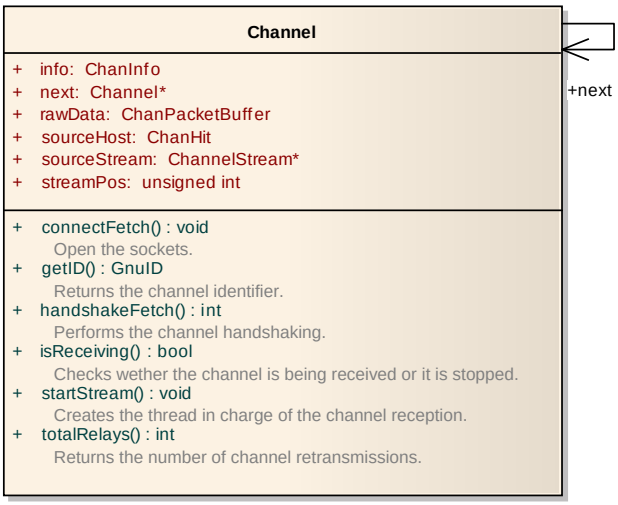


Figure 4.16: Main Channel Class Attributes and Methods.

4.2.5.2 Source Stream

Every channel needs a source stream which is the input stream that provides data to the channel. In Peercast, it has been implemented a class that represents the input stream concept: `ChannelStream`. In section 4.2.1, it has been described the process that uses the `ChannelStream`. As the source stream can have varied

origins, there are `ChannelStream` subclasses that represents the different types of sources. These subclasses are the ones who are instantiated in fact, while the `ChannelStream` class only groups the common characteristics to all of the input streams. Therefore, when a channel is received, the `ChannelStream` rules the thread dealing with the reception.

The most commonly instantiated `ChannelStream` subclass is the `PCPStream`. The methods of this class perform the PCP packet processing. The original source, the one from the broadcaster uses other stream types i.e. `MP3Stream` or `RawStream`. Peercast allows to create a channel from a multimedia file or from a stream that could come from a stream server located in the same host or in any other point in the network. The `RawStream` streams contain only multimedia information, namely, they do not deal with meta-data.

In figure 4.17 an example of the different stream types used during this thesis are shown. When a stream from another Peercast instance is received, a `PCPStream` is created. If the sending is performed by any other type of source, the reception is managed, for instance, by a `RawStream` object that does not perform PCP processing. As shown in this figure, there is a stream server, though Peercast allows to create stream from the file as mentioned above. However, there are plenty of stream servers quite extended and a lot of users, that have experience working with them, prefer to use them instead of Peercast to pump the media. In the case of audio streaming, it is worth to mention *Southcast*, developed by Nullsoft (11).This server offers a more pleasant GUI than Peercast and it allows to edit the playlist like in a simple player.

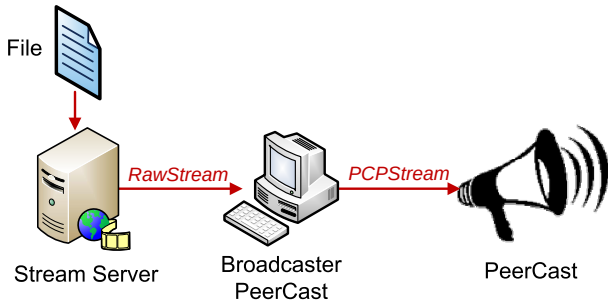


Figure 4.17: Stream types regarding peer types.

The `ChannelStream` methods are overloaded by the `PCPStream` and `RawStream`

methods which are the actual instantiated objects. The latter is in fact a very simple class, as it does not perform any processing and stores directly the arriving packet into the `rawData` buffer. Figure 4.18 shows some of the attributes and methods of the `PCPStream` class. The shown methods process the PCP packet chains. First, every packet goes through the `readAtom` function, and then, if they are channel packets the method `readChanAtoms` is called. Finally, those packets that contain data or the header of the channel are processed by the `readPktAtoms` method, which at the end of the processing puts the packets into the `rawData` buffer.

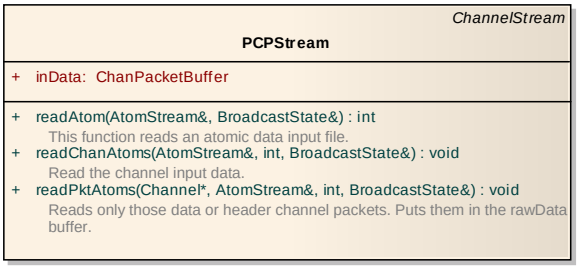


Figure 4.18: `PCPStream` class methods and attributes.

4.2.5.3 Servent

The servent class manages the channel sending and listens to requests coming from the network or from the same host. Servents exert the server function in Peercast, as every peer in a P2P network is client and server at the same time. Servents are classified according to the transmission type. Thus, the most used servents are:

- **Direct:** Is the transmission mode that goes directly to a player without using any Peercast instance. This mode is not interesting under the sharing point of view as the peer that receives the channel cannot retransmit it. If the Peercast instance is configured to work only in this mode, then it becomes a unicast streaming software with the same characteristics that Shoutcast (11). When someone plays a channel in background, a servent in direct mode is opened which sends the channel to the player.

- **Relay:** This is the natural Peercast transmission mode that takes place during the communication among different Peercast instances. When this mode is enabled, then the PCP encapsulation is used. In this mode, Peercast allows to limit the maximum number of relays per channel and the total number of relays.

- **Server:** A servent in this mode acts as server for the same host as well as for any other clients. It handles requests from different entities like other Peercast instances, players or browsers. As the Peercast GUI is based in web, the servents in this mode are in charge of delivering data to those webs. In the same manner, the users request channels through the browser which connects to their own Peercast instance.

As it has been mentioned in section 4.2.1, in the main thread there is a server servent. This is not the only server servent in Peercast. Any other instance has always two active servents in this mode. By default, one server is listening to the 7144 port and the other to the 7145. This second server servent is used in order to make Peercast compatible with ShoutCast which uses a different port number to send the meta-data.

The other two types of servent are created on demand. If someone asks to a server servent a channel, the servent creates another servent to handle the request and goes back to listen any other request. Figure 4.19 shows some of the attributes and methods of the `Servent` class which are worth to highlight.

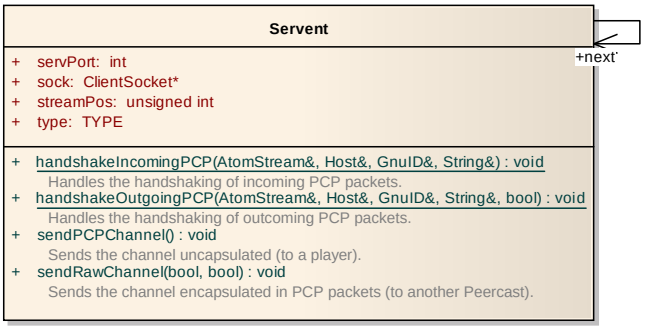


Figure 4.19: `Servent` class methods and attributes.

4.2.5.4 Channel and Servents Managers

In any OO-application, it is common that some methods from one class need to access to some attributes or methods from another classes outside its inheritance. This is a typical problem that can be solve in different manners. Peercast deals with it instantiating these shared classes globally. It also uses static methods which do not need a class instance to be invoked, but this is not always suitable in term of code efficiency.

An example of the recurrent OO-issue, can be seen in the `PCPStream` class, which represents the input stream. When a new packet arrives through a method of this class, then it is necessary to call a method belonging to the `buffer` class to put the packets into the buffer. The problem is that the buffer as well as the `PCPStream` object belong to the channel and therefore the member classes need a reference to the container class in order to access to other member classes. This relationship among these classes is shown in Figure 4.20.

In order to avoid passing as argument references to each type of channel in all the methods, Peercast uses a channel manager (`ChanMgr` class). This manager is instantiated globally so that any class can access it. The channel manager has a reference to a channel and all the channels have a reference to the next one, namely, the one created subsequently. The set of references make up a chained list.

The channel manager apart from giving a pointer to the desired channel, also adds functions that perform actions over the channel, i.e. create, close, check the status, etc. Figure 4.21 shows some of the attributes and methods of the `ChanMgr` class which are worth to highlight.

A similar task is carried out by the servents manager. In this case, in addition to return the servents references, it also includes methods that control the total number of connections. When the main thread starts, it loads a configuration file just before creating the first server servent. The global `ServMgr` object deals with the configuration loading and also the servents creation. Figure 4.22 shows some of the attributes and methods of the `ServMgr` class which are worth to highlight.

Figure 4.20: UML relationship diagram among `Channel`, `PCPStream` and `PacketBuffer` classes.

4.2.5.5 Buffers

The `ChanPacketBuffer` class represents a buffer in Peercast. It is obvious the necessity of a buffer in Peercast due to the fact that each node can perform as

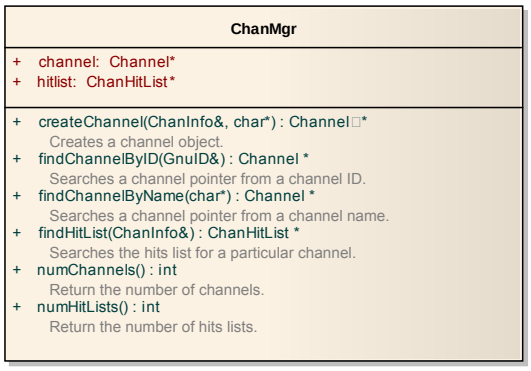


Figure 4.21: `ChanMgr` class methods and attributes.

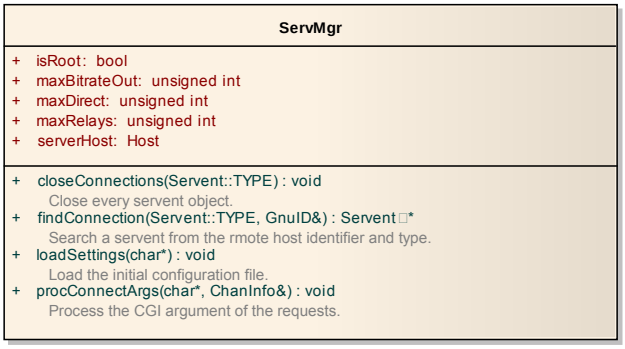


Figure 4.22: `ServMgr` class methods and attributes.

client as well as server, so it needs to store in memory the information in order to be capable of retransmit it. Without a buffer, a node can only reproduce the received packets in a player but cannot send them to any other client in the network.

The main component of the buffer is a packet array. The total number of packets that the buffer can store is 64. Any video packet has a length of 8192 bytes, without taking into account the length of the header. This class is used to create the main buffer (`rawData`) that stores the video and audio packets, as well as the auxiliary buffer (`inData`) that stores any type of arriving packet. The buffer has a series of indexes in order to allow the access to a particular position. Figure 4.23 shows the attributes and methods of the `ChanPacketBuffer` class.

Figure 4.23: `ChanPacketBuffer` class methods and attributes.

4.2.5.6 Root Nodes

In chapter 2, there have been mentioned the main P2P architectures nowadays. It can be inferred one common tendency in the research community: decentralization. Despite the fact that Peercast was designed to run in a decentralize way, the actual situation is centralized. The developers implemented Peercast in order to create a network without servers. Thus, Peercast only has clients.

A group of users that want to share a video among them, only need to know the IP address of some of them and the channel identifier. Hence, they will create a download tree like the one shown in figure 4.24.

$User1$ uploads the video to the network in first place. Then, $User1$ tells to two other users the channel identifier that Peercast created. These last users ($User2$ and $User3$) can download the video running her Peercast clients and calling the Peercast server by means of a simple browser. Now, if any of these users sharing the stream tells its IP address and channel identifier to a fourth user, they could resend the stream. Thus, through a contact network, theoretically it could be possible to generate an infinite tree without adding any server.

Now, let us assume that a user wants a video but it does not know any user from the sharing group. In order to be able to join the network, this user needs the IP address of any user and the channel identifier. This is a common situation

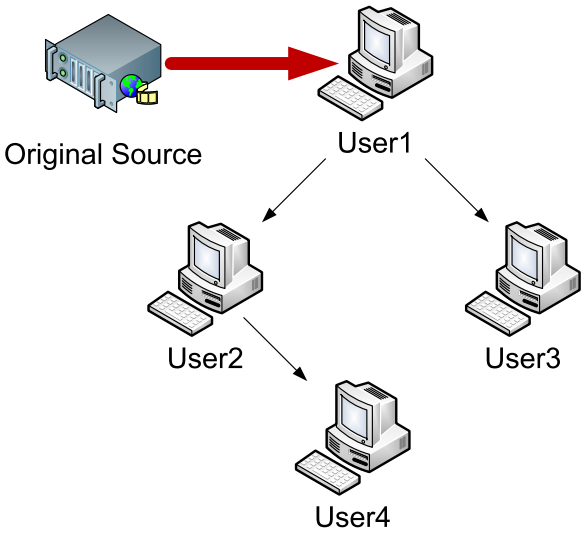


Figure 4.24: Peercast Tree architecture example.

that makes the tree model of Peercast unfeasible in the field.

Peercast deals with this issue with the concept of *root nodes*. For every broadcasting network, one node will act as root, being the primary source of the data flow, while the others will receive it and possibly retransmit it. The code difference between a normal client node and a root node, lies on the configuration file. Root nodes performs all the normal client functions, and in addition, they gather information about other clients in order to create an information directory. This directory will relate the channel name with the identifier and will provide the corresponding IP address to any request.

Each client has to point out in her configuration file (*peercast.ini*) who is her root. It is not compulsory and only one root node is allowed. When this client starts emitting a channel, the root adds it to the directory. Moreover, if a user retransmits a channel, the root node also takes that into account to update the directory.

In the same way, if a user ask for a channel and does not specify any IP address, it will connect to the root node and request the channel identity. A root node only has a channel if some of the subscribed nodes is broadcasting it.

On the other hand, the root nodes provide statistics. These statistics are used to create a webpage where the users can search channels. Figure 4.25 shows a

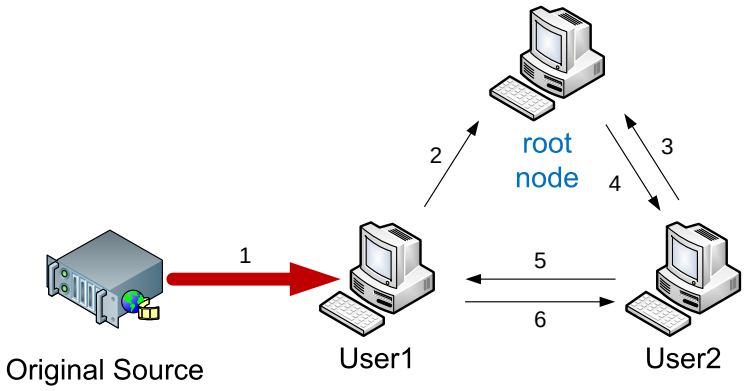


Figure 4.25: Peercast network architecture with a root node.

simple case of a network where there is a root node. The messages flow in this example is the following:

1. $User1$ gets the media content.
2. $User1$ publishes the content in the root node.
3. $User2$ requests a channel to the root node through a browser.
4. The root node sends to $User2$ the IP address of a node sharing the channel.
5. $User2$ requests the channel to $User1$.
6. $User1$ starts emitting the channel.

This example exposes that in root mode, any peer can become a directory. However, no all the peers are capable to perform the functions of a root node as a root node has always to be connected and broadcast its identity has to be known somehow.

Nevertheless, nowadays there is only one root node working: the official. So, by default the Peercast client has the root mode deactivated, and the associated root node is the official one. This fact causes that the worldwide Peercast network goes through the same node, so that the system has become centralized not achieving the designing goals.

Chapter 5

NeuroCast

The main goal of this thesis was to design and implement an application to distribute video over a P2P network. To achieve our aim we started off the open source application PeerCast, explained in detail in chapter 4, and developed an improved version we have named *NeuroCast*. NeuroCast is a simple and free way to listen to the radio and watching videos through Internet. Moreover, NeuroCast adds the necessary design to allow any user to become a broadcaster.

Among the different improvements made in NeuroCast, it is worth to mention the multi-source streaming capability. By means of the multi-source streaming, NeuroCast is able to overpass the PeerCast limitation which only allows to receive the stream from a single source. Therefore, in contrast to its predecessor, NeuroCast makes feasible the video streaming, as the common asymmetry of the link is solve using multiple sources. This limitation was the one which makes Peercast only suitable for audio streaming, as for the audio the bandwidth is lower than for video.

In this section, we will describe the general design and performance of NeuroCast, focusing in its main elements and the interaction among them.

5.1 System Architecture

Analyzing the evolution of P2P application for file distribution, it has been noted a leading trend to move from the typical tree/forest topologies to the mesh topology (see Figure 5.1). This evolution allows NeuroCast to use the bandwidth of

N users which cannot broadcast the video by themselves, but together they are able to achieve the broadcasting.

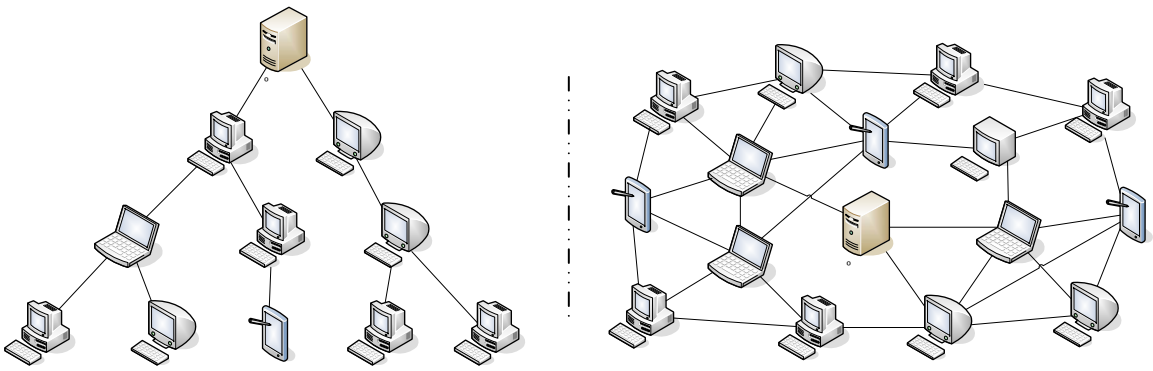


Figure 5.1: P2P tree structure network (left) and P2P mesh structure network (right).

In the same way, taking into account the P2P philosophy, each network user or peer is able to receive a stream from different peers, and at the same retransmit it to any other user. Bottom line, each client can perform as client as well as server. Therefore, in contrast to other approaches explained in chapter 2, NeuroCast only has a single type of client which deals with both functions.

As PeerCast, apart from the users, the main element in NeuroCast is the multimedia content shared in the network, called channel. Every channel has its original source. In general, this source is a user (*broadcaster*) who creates the channel and becomes the only physical link among the NeuroCast clients and the original multimedia file. Likewise, each channel is identified by a unique hexadecimal code which is obtained from the channel name, the broadcaster identifier, the mount point and the channel rate.

As it has been stated before, in order to transport multimedia content in real time it is necessary the use of a stream. Although NeuroCast allows to create directly a stream from a file, there are some applications (VLC (14), Miro (5), Ogg Streamer(8), ...) that make this task more simple for the user. Therefore, it is common to use these applications instead of NeuroCast to pump the stream.

Users interact with NeuroCast through a web interface quite simple and friendly (see Figure 5.2). Once the application is booted, the integrated server generates web pages. By means of these web pages, the user can manage the

application as well as get some information i.e. characteristics of the created channels, retransmission taking place or peers from where the stream is being downloaded. Like in PeerCast, our application also has a configuration file where the user can define different system parameters.

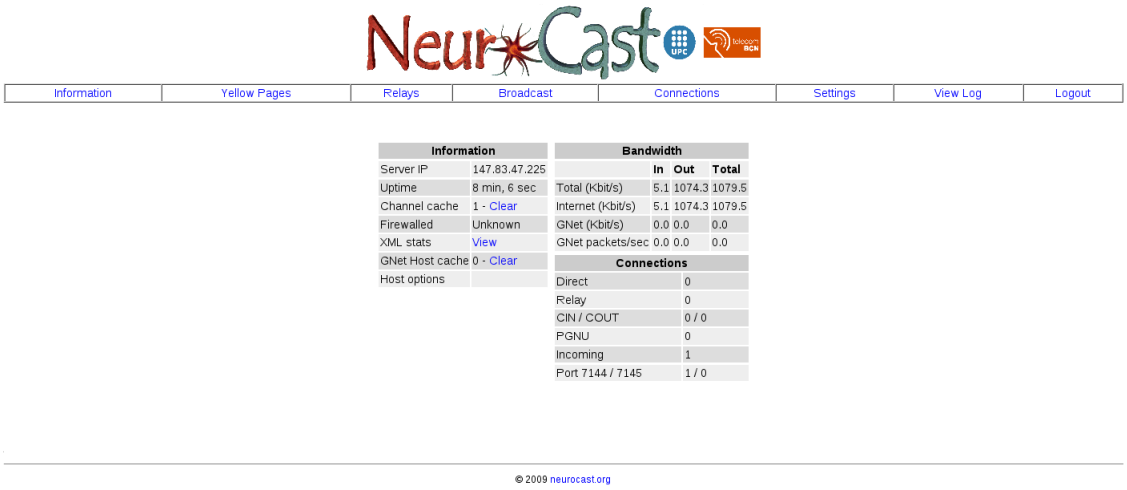


Figure 5.2: NeuroCast web interface snapshot.

Apart from the broadcasters who perform as stream sources, there are other type of clients in NeuroCast: the trackers. These clients deal with the gathering of information about the peers that are sharing a particular channel. Thus, when a user starts a session the tracker will provide her a list with the peers that are offering the requested channel. These peers that are sharing the channel are named *hits*. Therefore, it becomes essential that the trackers have an updated hit list.

NeuroCast has been implemented in such a way that all the users perform as trackers. Hence, any peer can share with another her peer list. Furthermore, the peers inform the tracker when a client is disconnected, so that they can maintain their list updated.

On the other hand, using a mesh approach forces us to introduce new concept: *substreaming*. Substreaming appears as consequence of the need of receiving a stream from several sources at the same time. In this way, the substreaming consist in dividing the original stream in N parts, which are delivered from the different sources that are available following certain criteria that are explained later.

Figure 5.3 shows a stream like a series of packets sent from the source. In NeuroCast, each sent packet has a length of 8192 bytes. The stream origin is the first packet received by the *broadcaster* from the source. Then, the application references this first packet from the SP (`Stream Position`) parameter: the first packet has $SP = 0$, the second $SP = 8192$, the third $SP = 16384$...

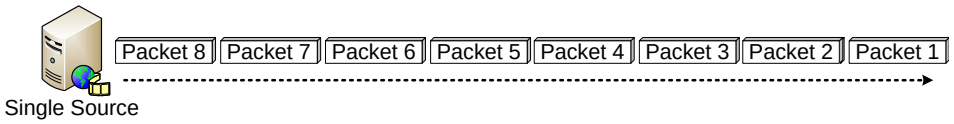


Figure 5.3: Stream from a single source.

When there are more sources the situation becomes more complex, as there are new issues to take care of like the arrival packet order or the packet distribution among the available sources. We start from the simplest scenario (Figure 5.4), where two sources send parts of the same stream but one send the even packets and the other the odd packets. In this scenario, we talk about two substreams as the original stream has been divided into two parts ($N = 2$). All the packets that the substream consists of are called *chunks*. Therefore, the first source has the chunk number equal to 0 and the second one equal to 1.

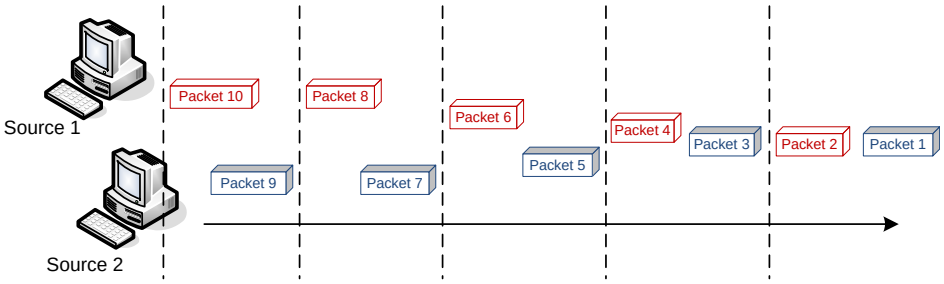


Figure 5.4: Substreams from two sources.

Thus, the sources in order to know which packet they have to send, they just have to divide the packet number between its chunk number and check the remainder. For example, in the scenario of Figure 5.4, *Source*1 in order to check whether it has to send the packet with number equal to 10, it just divides 10 (packet number) between 2 (chunk number). As the remainder of this division is null, then *Source*1 decides that it has not to send the packet, as it will be sent by

*Source*2. The algorithm is generalized to N sources performing a simple modulus operation, as algorithm 1 shows.

Algorithm 1: Stream Chunk Allocator Algorithm

ine **Input**: Stream
Output: Package Sending Condition
foreach *Package i* **do**
    Package_Number = GetPackageNumber(i)
    if *Package_Number modulus Chunk_Number* == *Assigned_Chunk_Number* **then**
        return *TRUE*
    else
        return *FALSE*

On the other hand, it could be interesting to receive more packets from one source than from another. For example, if a source can send packets at a rate of 100 kBytes/s and another can only send at 50 kBytes/s, then it seems more efficient to divide the stream in sets of 3 chunks and assign two chunks to the source with highest rate, and only one chunk to the source with the lowest rate.

Once it has been described the substream concept, now we focus on describing the performance of a standard NeuroCast client. The NeuroCast client is able to download the channel from different peers (acting as a common client) and, at the same time, it retransmits the channel to other peers (acting as server). So, it is feasible a scenario where a peer is downloading a channel from two sources and sending it to another three peers. Therefore, due to the substream concept, it appears another concept: the *subchannel*. In brief the subchannel is an identifier element of the substream, though this concept will be describe in depth in next sections.

It would be also common that a user wants to watch the video or listen to the audio stream that it is being downloaded. In order to do that, NeuroCast creates a new process to send the stream to a player directly.

Finally, as stated before, a client can also act as a sever retransmitting a channel. It is in these scenarios where the P2P architecture makes sense. Thus, a NeuroCast client will be always listening to any request to retransmit the channel. Once a new request arrives, the client creates a new thread in order to serve the requested stream. In this case, the packet sending is performed using the PCP

protocol (see section 4.2.2). The maximum number of retransmissions that a peer can perform can be customized in the configuration file through the `MAX_RELAYS` parameter. If a peer achieves this maximum, then it could only offer its peer list, like a tracker, but it will not be able to perform additional retransmissions.

Therefore, the basic actions that a user will perform to connect to the NeuroCast network are:

1. **Getting Hit List.**

   During the start, the NeuroCast application connects to the tracker using the `HTTP` protocol, and the tracker sends back a list of the possible peers that can retransmit the requested channel. This connection can be performed directly if the peer has the following information about the tracker: tracker's IP address, the channel identifier, the port from where the channel is being retransmitted. If the peer has not these data, then it could use other means in order to get this information. Usually, it could use the same web page that PeerCast uses like yellow-pages.

2. **Peer Selection.**

   Once a peer has the hit list, then it has to select the best set of peers from the list. In order to do that, NeuroCast implements an algorithm based on bandwidth measurements, network losses and channel availability. This algorithm will be discussed later (see section 5.4.2).

3. **Packet Allocation.**

   NeuroCast sends the file fragmented into packets of 8 kBytes. At the same time, in order to improve its delivery through the P2P network, these packets are grouped in specific chunk numbers which are assigned to the peer transmitting the stream.

4. **Network and load adaptation.**

   NeuroCast monitors the network status permanently while downloading the channel. Thus, it can better adapt to the Internet network and improve the application performance.

5.2 Basic Performance

NeuroCast is a multi-thread application, namely capable of performing different actions simultaneously. This way it becomes simpler to implement the client/server concept of P2P networks. Therefore, a single application is able to receive video from different sources of the network and, at the same time, retransmit it to other peers.

The number of processes in NeuroCast is variable and it depends on the amount of established connections. As in PeerCast, NeuroCast has a main thread which deals with the incoming requests and creates the children processes that handle these requests. In the same way, NeuroCast uses the Oriented Object programming, so it has several classes that represent the different system entities. In addition to the entities in PeerCast (see 4.2.5), NeuroCast also has the *subchannel* entity. Next we present a brief scheme of the main entities, which we will describe in depth in the following sections:

1. **Servent**: Handles requests from the system and sends the stream either to another NeuroCast instance or to the player.

2. **Channel**: Manages the multimedia content shared in the network. The channel has two main elements: the buffer and the channel stream. The buffer stores the incoming packets temporally to allow a retransmission posterior. The channel stream deals with the handshaking and the channel reception.

3. **Subchannel**: Represents a stream fragment. The subchannels allow to receive a channel from different sources or to retransmit channel chunks to other peers.

These entities are represented in Figure 5.5 where their basic performance as well as the main processes are captured.

NeuroCast has a GUI based on a web inherited in part from Peercast, and incorporates a server which generates HTML pages where the session status, the used bandwidth and the subchannels are shown. However, the environment is still too basic, and refreshing the pages each 5-10 seconds can bother the user. Recently, new graphic environments for different operating systems (Linux, Mac, Windows) have appeared, but are quite unstable.

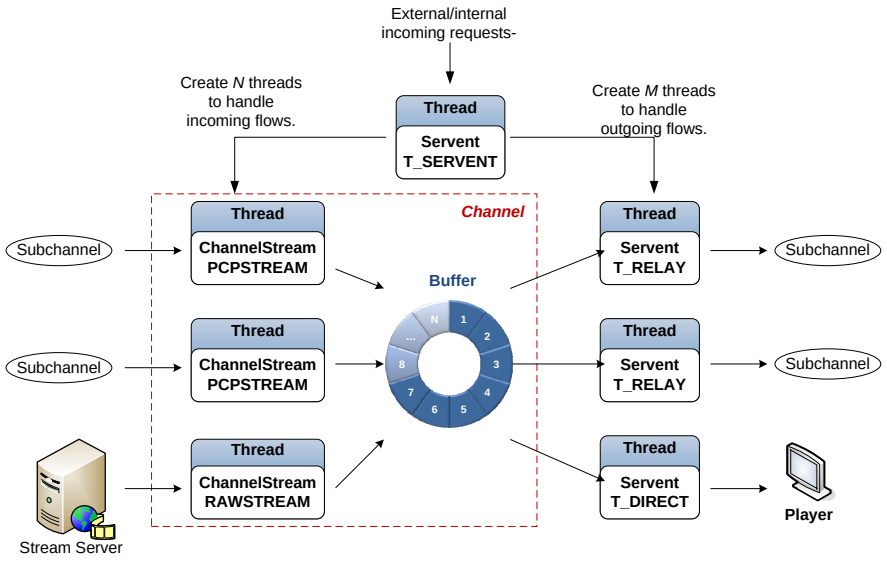


Figure 5.5: NeuroCast node internal architecture.

5.2.1 Working Modes for a Neurocast Node

As it has been explained before, every application with a P2P architecture aims the network decentralization. Likewise, NeuroCast is designed in such a way that clients centralizing data are not necessary. To achieve decentralization, Neurocast uses data directories which contain the channel descriptions, the channel identifiers and the IP addresses of the clients broadcasting these channels.

The original version of PeerCast incorporates an official directory by default. In this way, Peercast achieves to keep the channel information among the nodes that are sharing it. NeuroCast adds new capabilities to peers, allowing them to act as trackers and provide the list of the nodes sharing the channel easily. Moreover, when a hit leaves the network, the tracker is reported to update its hit list.

5.3 Software Architecture

The same way it was presented in section 4.2 an analysis of the PeerCast software architecture, in this section we will analyze NeuroCast software, stressing its main differences with PeerCast.

5.3.1 Servent

A servent is one of the main classes of the application. A servent acts as a server as well as a client. Its main function is to respond to the different requests to the application. These requests can come from other users who want the channel to be retransmitted, or from the NeuroCast application itself i.e. when a player request the channel or when a web page requests information about the application status. Thus, according to the request type, NeuroCast will create a new servent to deal with the request. This way, there exists three different type of servents:

- **`T_Server`**: It is the server of the main thread which handles any request whether it is external or internal. This subclass will create new server classes to manage the different requests. When the NeuroCast application starts, two new instances of this class are created. By default, these `T_Server` classes listen one to the 7114 port and the other to the 7145 port. This second port, is inherited from PeerCast, which uses it in order to be compatible with the Shoutcast stream server (11).

- **`T_Relay`**: This class manages the retransmission of a channel among different PeerCast instances. In this mode, the data are sent using the PCP protocol (see section 4.2.2).

- **`T_Direct`**: This class is used to retransmit the multimedia file directly, namely, without packing the data. In this way, this class is used to handle the packet sending to a player.

The scheme of figure 5.6 represents the different server subclasses that we have just introduced. At the center, we have the `T_Server` class, that manages the requests coming from our host. On the other hand, the `T_Direct` class is connected to the player directly and to the buffer from which it gets the packet to play. Finally, it is a `T_Relay` class, which is also connected to the buffer to get the packet, but in this case these packets are sent through the network to the requesting peer.

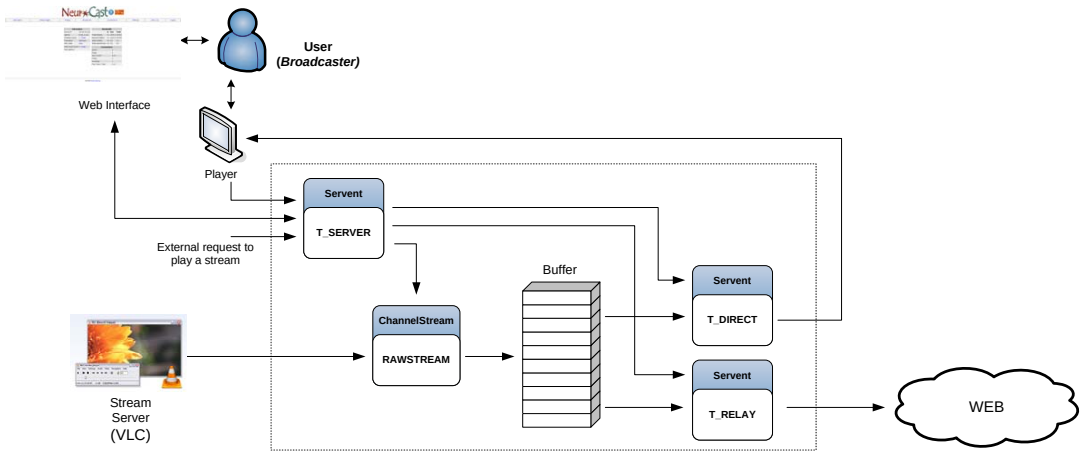


Figure 5.6: NeuroCast web interface and stream server architecture.

5.3.2 Channel

The channels are entities that NeuroCast uses to handle the different multimedia flows which are being shared in the network. Each one of them uses a different channel. NeuroCast only distributes parts of the stream not the whole stream (as the user is interested in the fragment which is being played at that instant), so instead of sharing a video what it is shared is a *channel*.

Channels are created when a user starts to retransmit new multimedia content in the NeuroCast network, and are identified by means of an unique 16-hexadecimal code. The first user sharing the stream is the one who creates the channel (called *broadcaster*), and becomes the only link between the application and the original source. Therefore, if the broadcaster is disconnected from the network the channel retransmission will be stopped. Thus, it is highly recommended that this first peer acts as tracker, as the channel exists as long as this user remains connected.

This class is the one that allows, among other things, to store the stream in a buffer or to control the packet arrival from the network. Hence, one main element of this class is the dynamic buffer. From this buffer the `T_Direct` and the `T_Relay` objects get the channel packets. Another important element is the `ChannelStreams` object that handle the incoming streams.

In the following, we analyze in detail these main elements that the class *channel* consists of.

5.3.2.1 Subchannels

As showed in the previous sections, originally PeerCast users downloaded the multimedia files from a single source. However, NeuroCast introduces the possibility of using several peers concurrently, dividing the stream among these, and making the network structure look like a grid.

The subchannels are the application elements that allow to identify the stream fragments. Thus, each one of the sources will be associated to a subchannel. This way, once the peer obtains the information about each subchannel, then one thread per subchannel is created. Each thread deals with the reception of one subchannel.

For example, when one of these threads wants to know which part of the channel has to request to the source, it will have to check the `subChannels` vector of the `Channel` class in order to obtain the number of the chunks associated to its subchannel number.

Each created channel has three attributes to manage the subchannels:

- **`NumSubChannels`**: Subchannels number into which the channel is divided.
- **`NumChunks`**: Total number of chunks into which the stream is divided.
- **`SubChannels[10]`**: Vector that contains the subchannel-type elements that have been created. This vector can contain a maximum of 10 subchannels.

The information that each subchannel contains is the following:

- **`ID`**: Subchannel identifier.
- **`Chunks[50]`**: Vector where the numbers of the chunks that the subchannel is downloading are stored. For example, if the channel is divided into 10 chunks, a possible subchannel could download the {1, 3, 5, 7} chunks.
- **`SourceHost`**: `ChanHit`-type attribute that contains information about the peer associated to the subchannel.
- **`NumSubChannelChunks`**: Number of chunks that a specific subchannel is transmitting. Following the last example, the value of this parameter would be equal to 4.

- **`OriginalSource`**: Parameter that allows to differentiate a subchannel coming from the original source from the other types.
- **`NewPosition`**: Position that the subchannel will have in the `SubChannels[]` vector once the vector elements are reordered.

5.3.3 Buffer (`ChanPacketBuffer`)

The `ChanPacketBuffer` manages the packet storage, namely, it performs as the buffer of the application. Moreover, it also has several methods to allow the interaction with it, such us writing new packets in specific addresses, or searching for a packet in the buffer, or reading the first position or controlling the last written position, etc.

Therefore, the main element of this class is the packet array that represents a buffer. The maximum number of packets that the buffer can contain is 64. This class becomes essential for the performance of the NeuroCast application. As any real-time P2P application, the application depends on the network conditions, so the packets can arrive out of order. Hence, it is basic to have a good system to order and store the packet efficiently.

On the other hand, when retransmitting the channel to other users, it will be necessary to access this buffer to get the stored packets and send them through the NeuroCast network.

5.3.3.1 Buffer Writing

In order to perform a deep analysis of this class, we present the main attributes that take part during a buffer writing as well as its main functionalities.

In first place, we introduce the attributes that deal with the positioning and indexing of the buffer. The value of these attributes are always modulus N, where N is the buffer capacity assigned to the buffer. These attributes are:

- **`lastWritePos`**: Position of the last packet that has been written in the buffer.
- **`firstPos`**: Position of the first consecutive packet in the buffer.
- **`lastPos`**: Position of the last consecutive packet in the buffer.

- **`writePosi[subchID]`**: Position where the next packet coming from a specific subchannel will be written in the buffer.
- **`nextSendPos[subchID]`**: Position of the next packet to be sent to a specific subchannel has in the buffer.

In second place, we introduce the variables related to the packet position in the stream. In this case, it is useful to divide these values by 8192 (packet length in bytes) in order to obtain the absolute number that the packet has in the stream taking the origin as reference. These variables are:

- **`lastWrittenPackPos`**: Last packet position that we have written in the buffer.
- **`pack.pos`**: Position of the packet regarding the stream to be written in the buffer.
- **`streamPos`**: Current channel position from the origin of the stream.

It should be noted that the `lastWrittenPackPos` attribute belongs to the `ChanPacketBuffer`, but the `pack.pos` attribute belongs to the `ChanPacket` class and the `streamPos` attribute belongs to the `Channel` class.

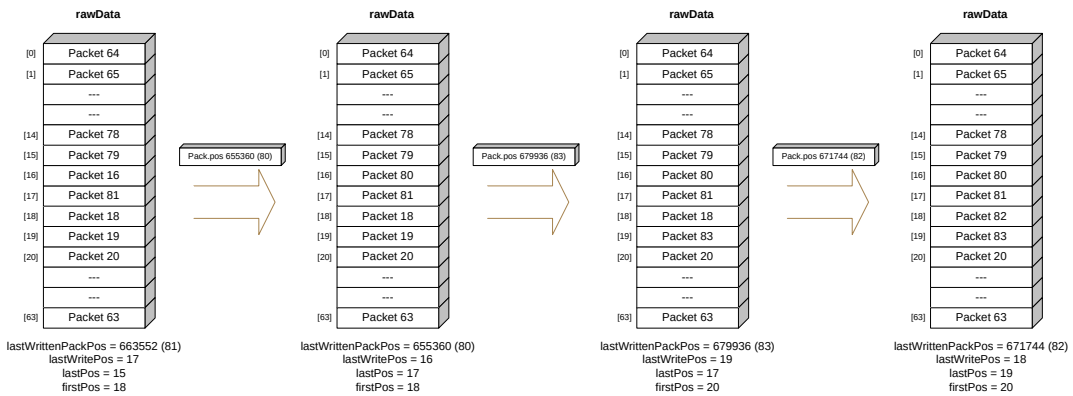


Figure 5.7: Evolution of the `rawData` buffer during the arrival of new packets.

In order to exemplify the usage of these attributes, figure 5.7 shows the evolution of the `rawData` buffer during the arrival of 3 new packets to the system.

In any case, the position of the packets stored in the buffer is determined by equation 5.1.

$$[lastWritePos + (pack.pos - lastWrittenPackPos)/8192] \ \ modulus \ \ N \quad (5.1)$$

So with the values related to figure 5.7, using equation 5.1 we obtain that the packet with number equal to 83 will be stored at position 19.

$$[16 + (679936 - 655360)/8192] \ \ modulus \ \ 64 = 19 \quad (5.2)$$

5.3.4 Channel Stream

As stated before, the `ChannelStream` class allows to control the incoming packets. Each object of this class is associated to a source and created by the main servent in order to manage a child thread that deals with the channel or subchannel reception.

Therefore, there are different subclasses of the `ChannelStream` class depending on the type of source:

- **`PCPStream`**: It is the most common type and is used to communicate with other NeuroCast instances. In this case, the packets arrive encoded with the PCP protocol (see section 4.2.2), so that before storing them in the buffer they must be preprocessed.
- **`RAWStream`**: It is used when the data flow comes from a stream server directly. Therefore, in this case, it is not necessary to preprocess the packets as they do not have any extra-header. So, as packets arrive, they are stored in the buffer `rawData` forthwith.
- **`MP3Stream, MMSStream, NSVSStream, WMAStream...`**: Each one of these subclasses are used according to the multimedia codec used to compress the stream.

As it is shown in figure 5.8, the `PCPStream` class has an internal `ChanPacketBuffer` object (`inData`) which is used as buffer and where the packets coming from the source are stored firstly. The goal of this buffer is to filter those packets that do not contain multimedia data and send the rest of them to the `rawData` buffer.

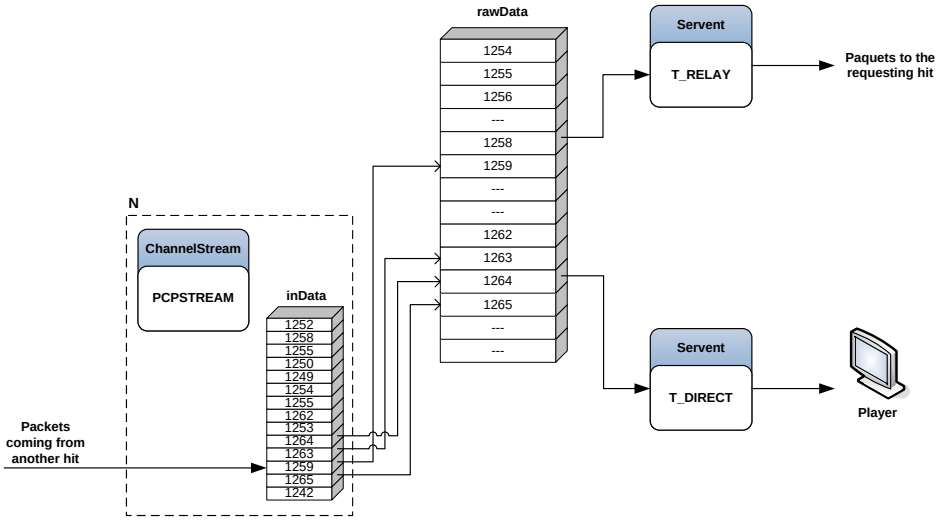


Figure 5.8: Relationship between the `rawData` buffer and the `ChannelStream`.

It is worth to mention that the application will have as many `PCPStream` elements as subchannels in use. For example, if the peer is dowloading from three different sources, NeuroCast will create three independent `PCPStream` objects, each one of them with its internal buffer.

5.3.5 Channel Hit

The `ChanHit` class has diverse information about the hits. In this way, the application will have a `ChanHit` object for each peer that could transmit the requested multimedia file. Therefore, if we are disconnected from any peer that was retransmitting the channel to us, then we will be able to reconnect to another hit. It should be mentioned that this class manages different attributes of the hits such as the IP address, the identifier hit number, the hit throughput, the hit availability or the link losses.

5.3.6 Channel Hit List

This class is used to manage the hits of each channel. It allows to perform basic operations like adding a new hit, deleting old hits, get the hit number, etc. Summarizing, it implements the typical methods related to lists. The list is made up of `ChanHit` objects.

Each channel has associated one hit list, though it is not in the `Channel` class, but in the `ChanMgr` class which is explained in the next section.

5.3.7 Server and Channel Manager

The channels as well as the servents need an upper element to control and manage the different created classes. That is the reason why NeuroCast creates at booting time an object of the `ServMgr` class to manage the servents and another of the `ChanMgr` class to manage the channels.

Some of the most important methods of the `ServMgr` class are the `loadsettings()` which allows to load the initial configuration of the application from a text file, or the methods that allow to control the maximum number of relay-type servers. The `ChanMgr` has several methods to create and to terminate channels, as well as methods to add or erase elements form the hit lists of any channel.

5.4 Load balance

As mentioned in the previous section, NeuroCast allows to balance the load according to the network conditions. In this way, Neurocast implements different algorithms to optimize the peer selection among the hits of the requested channel and the load distribution among the chosen ones. In this section we will analyze these algorithms. Before explaining the algorithms, first we introduce the nomenclature and definitions used in the development of this methodology.

On the other hand, we will show the empirical issues and how Neurocast is able to adapt to the network conditions and redistribute the load according to these conditions. These two factor are critical in a video streaming application. Differently from a file sharing P2P application, a link capacity decrease of a few seconds implies the stream reproduction to stop and if it is a live streaming this could result critical for the audience.

5.4.1 Definitions

Before analyzing in depth the NeuroCast performance, it is necessary to define some useful parameters. Some of these parameters are configurable through the

configuration file, and the others are parameters obtained from measurements carried out during the session.

- **Rp** (bps): It is the maximum *sending rate* that a peer can contribute anytime during the session.
- **A_BW** (bps): Available bandwidth of the path between the requesting peer and the peer retransmitting part of the video. This parameter is measured during the session.
- **Lp_r** (%): Expected packet loss of the path between the requesting peer and the peer retransmitting part of the video. This parameter is measured during the session.
- **Ap**: Probability of a peer to be available at an instant.
- **min** (bps): It is the minimum bandwidth that the application will require to new peers in order to consider them as possible hits. This way, the maximum number of peers to serve a stream is limited, and at the same time, it is also limited the number of active connections that the user should have to maintain. By default, the value of this parameter is equal to 100 Kbps.
- **Delta**: Number of chunks into which the stream is divided in order to work per groups with subchannels. In order to optimize the downloading among several users the packets of the stream are distributed in sets of *delta* packets or chunks.
- **Rl**: Lower limit of the total sending rate of a set of peers. By default, the value of this parameter is equal to 100 Kbps.
- **Ru**: Upper limit of the total sending rate of a set of peers. By default, the value of this parameter is equal to 2 Mbps.

5.4.1.1 Parameters Obtaining

Once the hit list has been received from the tracker, it is necessary to calculate which is the subset of peers that will allow to download the stream with the best conditions regarding the network. Therefore, it becomes essential to have some

parameters to relate our host with each one of the hits, and at the same time these parameters have to allow us to make a decision.

In first place, the requesting peer obtains the `peerRp` of each source. This parameter is introduced by each user in the configuration file, and represents the maximum sending rate that a peer is willing to contribute anytime during the session. As it can be seen in Figure 5.9, the requesting peer has not establish any kind of connection with the hits, so it has to communicate with them through sockets. This `peerRp` request is handled by the main servent of each hit.

As stated in section 3.1.3, NeuroCast uses TCP to transport the packets along the network. The socket methods used in NeuroCast to establish the connection among users, is developed in the C++ language programming. With these methods NeuroCast ensures:

- Data transmission without neither errors nor omissions.
- All the packets arrive to destiny in the same order in which they were transmitted.

In second place, NeuroCast calculates the throughput and the losses of the link with each one the hits. In order to do these measurements, NeuroCast takes advantage of a modified version of the Iperf (18) application (see section 6.2.5).

The basic process that takes place during the obtaining of these parameters is shown in the scheme of Figure 5.9. On the left, there is the thread related with the user who requests the stream chunks to the hits. On the right, it is shown the performance of these hits.

A common method to execute applications inside another one is to use the `fork()` call. With this system function, a process identical to the original is created to perform a series of tasks concurrently. In general, the application uses a pipe to communicate both processes. This way, we have an unidirectional means of communication where a process writes the results in one side of the pipe and, in the other side, the other process listens the written data.

The Iperf version that NeuroCast uses has been modified in order to adapt to its needs. Thus, it allows working using UDP to transmit in the standard output the measured throughput and the link losses. These two parameters are read by the `Channel` class, and are fundamental in order to select the best hits.

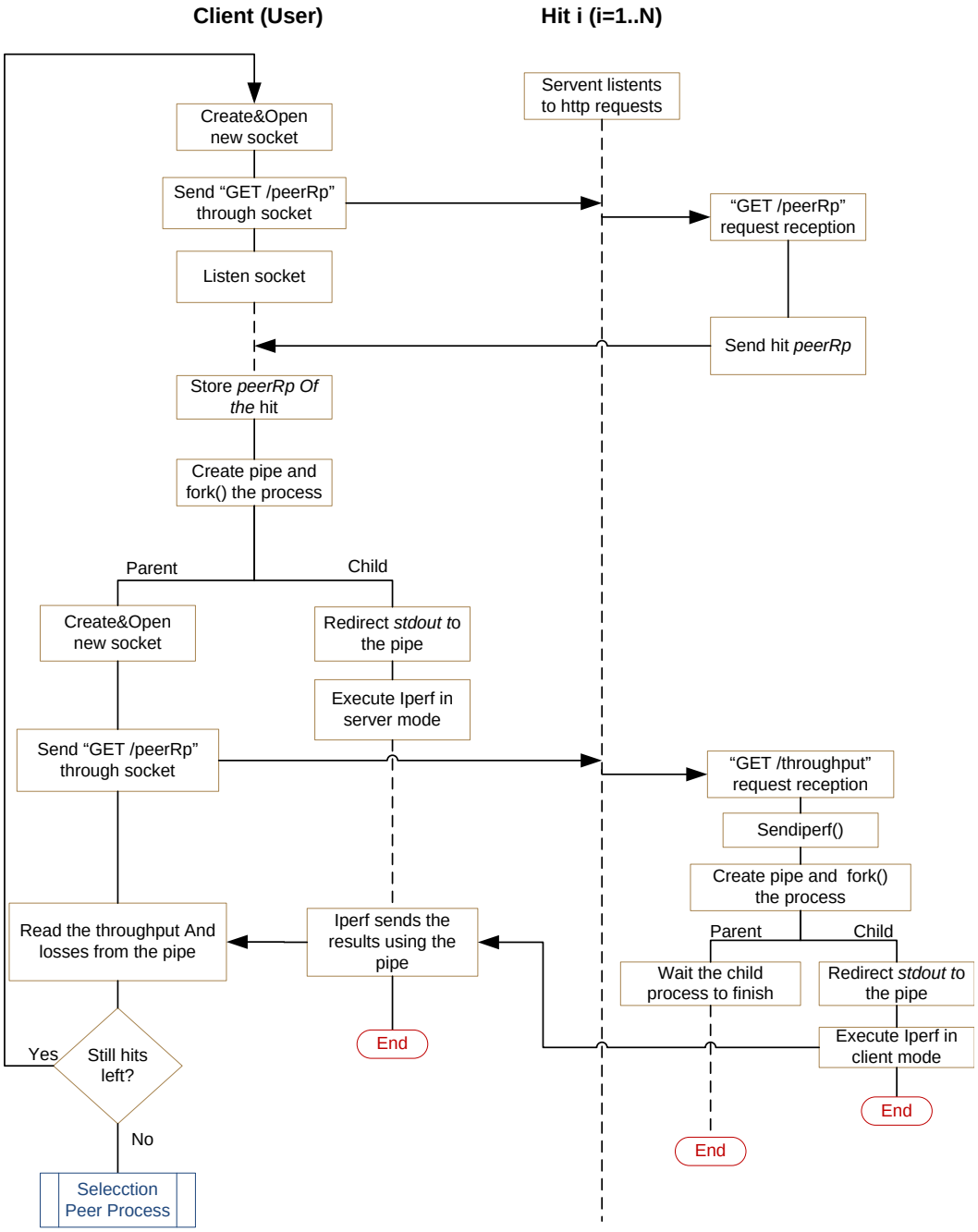


Figure 5.9: Flow diagram of the parameters obtaining.

In the following we show these modifications in the `reporter_printstats (Transfer_Info *stats)` method that Iperf uses to calculate and show the re-

```
1  void reporter_printstats(Transfer_Info *stats) {
2    double outputRate_d;
3    double transferred;
4    double packet_loss; //Percentage value.
5    transferred=(double)stats->TotalLen; //Transmitted number of
       bytes.
6    outputRate_d=transferred/(stats->endTime - stats->startTime);
       //Iperf returns the result Bytes/s
7    outputRate_d=8*outputRate_d; //Convert the result to Bits/s
8    packet_loss=(100.0*stats->cntError)/stats->cntDatagrams;
9    if (stats->mUDP!=(char)kMode_Server){
10     //TCP Reporting
11     write(1,&outputRate_d,sizeof(double));
12     exit(0);
13   }else{
14     //UDP Reporting
15     write(1,&outputRate_d,sizeof(double));
16     write(1,&packet_loss,sizeof(double));
17     exit(0);
18   }
19 }
```

Figure 5.10: `reporter_printstats` method of the `ReportDefault`-Iperf class.

sults.

In this report, we will not go further in the Iperf performance, as it has been used just like a tool to obtain the necessary parameters to optimize NeuroCast. In figure 5.10, lines 5,6,7,8,15 and 16 are the ones modified for our purposes. With these modification we achieve to send through the standard output the throughput as well as the packet loss rate values.

5.4.2 Peers Selection

5.4.2.1 Motivation

One of the most critical parts in any real-time player through a network based on P2P is the selection of peers. Unlike conventional P2P applications for file distribution, the fact of working in real time creates a hard and direct dependence

on the peers that are transmitting the stream. In this case, a connection loss or a decrease in the data flow from a single peer could mean in the best case the momentary stop of the video sequence, and in the worst case in could mean the total desynchronization with the player with the subsequent video freezing.

This reason makes it essential to carry out a good peer selection. NeuroCast will always try to prioritize these peer with a high available bandwidth, with the lower packet loss rate, and with a high availability. In this section we will explain the consequences of these choices and how the decisions are made.

Nowadays, one of the greatest limitation that the Internet connections have regarding the streaming is the asymmetry of the connection links. Most of the providers offer an upload link (up-link) whose capacity is quite lower compared to the download link (down-link). In the last years, it has increased the offered capacity of the down-link, evolving from the 56 Kbps of some years ago to the 20 Mbps of today. However, the up-link has not evolve accordingly, achieving a maximum of 256 Kbps for a common user. This asynchronism in the evolution makes the user able to download huge amounts of data, but she is only able to upload data at a rate even forty times lower. In order to exemplify this asymmetry, let us consider the playout of a video of a 1024 Kbps rate, a standard user (a connection of 6 Mbps down-link and 256 Kbps up-link) would be able to download and watch on-line up to 6 videos at the same time, while it would be necessary 4 users to upload a single video.

That is the main reason why NeuroCast uses the substreaming technique to download files from the network. Thus, NeuroCast is able to exploit the up-link capacity of several users.

5.4.2.2 Hits Availability

Apart from the bandwidth that a specific peer is willing to share, it is also interesting to know the availability of the set of hits. This parameter is being studied by several authors (69) due to the fact that P2P networks are more extended nowadays. Therefore, the availability has become one essential parameter to optimize the performance of the P2P networks.

In the literature, there are several definitions for the term availability. One which is most appropriate in terms of hardware or software systems is from Professor Malek teaching at the Humboldt University in Berlin: "Instantaneous avail-

ability is the probability that a system is performing correctly at time t and is equal to the reliability of non-repairable systems". Another definition by Professor Malek describes long term availability: "Steady-state availability is the probability that a system will be operational at any random point of time and is expressed as the fraction of time a system is operational during its expected lifetime".

So availability is mandatorily a value between zero and one. These definitions can be applied to most stand-alone hardware and software components, but cannot be taken over easily for peer-to-peer networks. In peer-to-peer networks availability is a function consisting of many factors.

Not only the availability of every single host has impact on this function, but also network and transient software faults have to be considered. Moreover users are constantly joining and leaving peer-to-peer networks, which results in availability being "...a combination of a number of time-varying functions ranging from the most transient (e.g., packet loss) to the most permanent (e.g., disk crash)...".

The main goal of a peer-to-peer network should be to provide a high video availability even if the network is compromised of hosts with very variable (thus also low) availabilities. Further it would be insufficient to consider only the permanent faults assuming that transient failures are transparently masked in peer-to-peer networks. Concluding there cannot be given a precise formula how to calculate the availability of a peer-to-peer network, as it would be much too complex. What can be given is a statistical analysis of a number of hosts availabilities.

In the NeuroCast case, it is impossible to determine the expected lifetime of the network, as it is not possible to predict when a node will be disconnected from the network, or the congestion of the network with the subsequent massive packet loss.

A possible implementation to calculate the availability in our environment is carried out in the following way. A parallel process to NeuroCast runs to check the hits status during a limited period of time (i.e. seven days). With these measurements, it will be generated a probabilistic function for each instant of the day, allowing to know the probability that a peer is connected at that time. Assuming that the hits are still willing to retransmit the stream, then we get

the peer availability empirically. So, each time a new user asks for a part of the video, apart from the `Rp`, we will also send the expected value of the availability.

Although with this method, NeuroCast is able to estimate the peer availability, a deep study will be necessary in future versions. The developed software has been implemented in order to allow a future change in the availability calculation. The experiments showed that availability in peer-to-peer networks is not easy to measure and there have to be taken many factors into account to calculate it. Most of all two time-varying distributions have influence on the availability of peer-to-peer networks. At first short-term daily joins and leaves of individual hosts and secondly long-term host arrivals and departures have significant impact on the availability of a peer-to-peer network.

Further host availability was underestimated in NeuroCast and it varies with time of day. Also there could be detected a significant host turnover in NeuroCast, however the number of hosts remained constant throughout the experiment. Finally it could be proven that there is significant independence between any pairs of hosts in NeuroCast.

5.4.2.3 Algorithm

NeuroCast peer selection algorithm is based in Collectcast algorithm (48). As stated in previous chapters, this algorithm is already implemented in other streaming application like PROMISE (71). The strengths of this algorithm are the sources selection, the network status monitoring and the periodic source redistribution in order to adapt to the time-varying network conditions.

The key component of NeuroCast is peer selection. Since the P2P environment is highly diverse and dynamic, selecting the best peers to serve a streaming session is critical to providing the desired high quality streaming. The selection technique should avoid peers that fail often and share congested network paths. This section presents three selection techniques: random, end-to-end, and topology-aware. The input to the selection technique is a set of candidate peers returned from the P2P lookup substrate. The output is a subset of the candidate peer set (called the active peer set) to start streaming the movie.

The random technique randomly chooses a number of peers that can fulfill the aggregate rate requirement, even though these peers may have low availability and share a congested path. The end-to-end technique estimates the “goodness”

of the path from each candidate peer to the receiver. Based on the quality of the individual paths and on the availability of each peer, the technique chooses the active set. The end-to-end technique does not consider shared segments among paths, which may become bottlenecks if peers sharing a tight segment are chosen in the active set. In contrast to the end-to-end technique, the topology-aware technique infers the underlying topology and its characteristics and considers the goodness of each segment of the path. Thus, it can make a judicious selection by avoiding peers whose paths are sharing a tight segment. However, it has been shown in (48) that both techniques performs similarly, but the end-to-end technique is easier to implement. Therefore, NeuroCast uses the end-to-end technique in its selection algorithm.

As stated before, the algorithm is based in the one of CollectCast, but it has been modified to reach our purposes. The selection process can be split up into three phases:

1st. Obtaining the hit list associated to the requested channel.

2nd. Enumerating the sets of hits that satisfy the constraints imposed by the equation:

$$R_t \leq \sum_{p=\mathbb{P}^{act}} R_p \leq R_u \tag{5.3}$$

3rd. Selecting the best set of peers among the solutions obtained in the previous phase.

In order to select the most suitable peers for each session, the `choosingSources()` method of the `Channel` class has been modified. New methods and variables have been added to achieve the purposes of the described algorithm. In the following, we analyze these methods and modifications.

In the first phase of the algorithm, the hits list associated to the channel is requested to the channel manager. The hits of the list already have the necessary parameters to run the algorithm: `A_BW` (available bandwidth of the link between the hit and the requesting peer), `Lp_r` (link losses) and Rp (maximum sending rate). With this information about each hit of the list, the requesting peer creates another list of the *candidates* according to the following constraints:

- If the tracker can act as peer (it is able to retransmit the stream), then it is included in the list. Otherwise, the tracker is discarded as candidate.
- If the peer `Rp` does not reach the minimum `Rp` defined in the configuration file, then it is not included in the list of candidates.
- If the available bandwidth measured by Iperf is lower than the `Rp` of a specific hit, then this parameter is updated with the measured one. Thus, NeuroCast uses the most restrictive value.

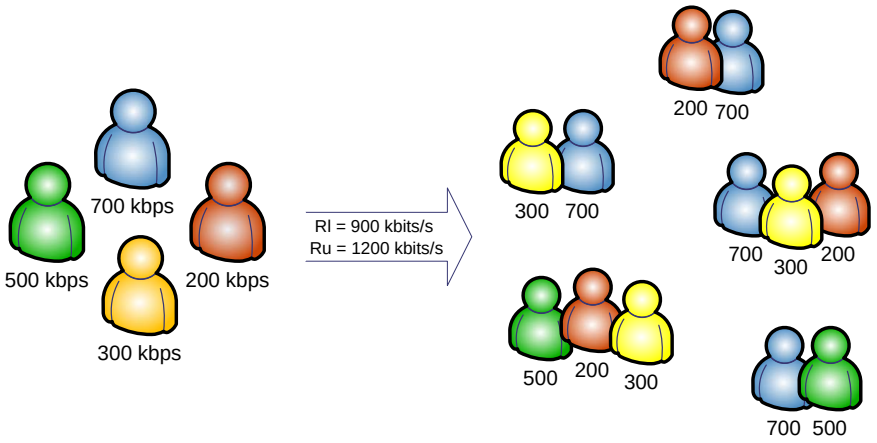


Figure 5.11: Peers combination that fulfill the equation constraints.

During the second phase of the algorithm, given the set of M peers of the candidates list, all the possible combinations of the candidates that fulfill equation 5.3 are calculated. In this equation Rl and Ru allow to limit the total bandwidth that a specific set of peers have to provide. This way, NeuroCast ensures that there is no set of peers with an excessive rate regarding the stream characteristics. For example, given 4 peers of figure 5.11 with their own Rp, there are 5 different way to group them to reach the constraints fixed by Rl and Ru.

These different sets are calculated by means of the `find` method, which is called by the `choosingSources` method. The former is a recursive method that returns a `solution` vector that contains the different sets of candidate peers to serve the video.

Algorithm 2: Pseudo code for selecting the best active peers set.

Input: Hits List

Output: Optimal Active Peer Set

Enumerate all possible sets that satisfy constraints in 5.3: $\mathbb{P}^1$, $\mathbb{P}^1$, $\mathbb{P}^1$,..., $\mathbb{P}^M$.

$\hat{\mathbb{P}}^{act}$ = null; $maxE = 0$

foreach $\mathbb{P}^m$, $1 \leq m \leq M$ **do**

  $E = 0$

  foreach $p \in \mathbb{P}^m$ **do**

    if $R_p \leq Available\_BW$ **then**

      $G_p = A_p \cdot (1 - \frac{Pathloss}{100})$

    else

      $G_p = A_p \cdot \frac{R_p - Available\_BW}{R_p} \cdot (1 - \frac{Pathloss}{100})$

    $E = E + G_p \cdot R_p$

    if $\overline{E} > \overline{maxE}$ **then**

      $maxE = E$

      $\hat{\mathbb{P}}^{act} = \mathbb{P}^m$

return $\hat{\mathbb{P}}^{act}$

Finally, during the third phase, it is chosen the best set regarding to the link error rate (`Lp_r`) and the available bandwidth (`A_BW`). This algorithm also takes into account the availability of each peer (`Ap`). If the availability has not been measured, NeuroCast will assume that all the peers are available and it will assume an `Ap` equal to 1. Algorithm 2 shows the pseudo-code of the peer selection used in NeuroCast.

The C++ code of part of this phase of the algorithm is shown in figure 5.12. In this code fragment, it can be seen how the `solution` vector is covered. For each element of the set i of the vector, it is calculated the expected "goodness" (G_p). This goodness depends on the `Rp, A_Bw, Lp_r` and `Ap` parameters.

Once the goodness of each candidate set is obtained, the maximum goodness is returned. This is obtained maximizing the expected goodness ($Gtemp$) of each set. The result is stored in the `active_peers` vector which contains the selected hits. Continuing with the previous example, the result that the algorithm returns is a vector which consist of the peer with `Rp` equal to 500 and 700 Kbps. Therefore, the implemented algorithm achieves to obtain the peers that maximize

```
1  for (int j=0;j<solution.set_peers[i].num_peers;j++)
2  {
3          Rp_temp=solution.set_peers[i].peers[j].Rp;
4          Ap_temp=solution.set_peers[i].peers[j].Ap;
5          Pathloss=solution.set_peers[i].peers[j].Lp_r;    //
               Pathloss is a percentage. i.e: 7,2%
6          Avail_bw=solution.set_peers[i].peers[j].A_BW;
7          if (Rp_temp<=Avail_bw){
8                  Gp=Ap_temp*1*(1-Pathloss/100);
9          }else{
10                 Gp=Ap_temp*((Rp_temp-Avail_bw)/Rp_temp)*(1-
                       Pathloss/100);
11         }
12         Gtemp += Gp*Rp_temp;
13 }
```

Figure 5.12: Loop inside the method `choosingSources` method performing the 3rd phase of the algorithm.

the throughput with the lower number of peers that fulfill the constrains.

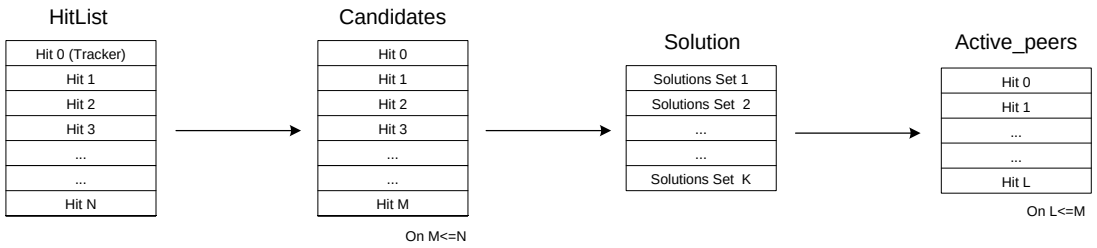


Figure 5.13: Lists during the peer selection process.

Figure 5.13 shows the evolution of the hits list during the whole selection algorithm. The algorithm starts with the *hitList* that it obtains from the tracker, and finally it gets the `active_peers` vector which consists of the hits that will serve the requested channel.

It is worth to mention that the maximum number of peers that a candidate set can consist of is limited to 10, as this is the maximum number of subchannels into which the channel can be divided.

5.4.3 Packet Allocation

The previous section detailed how NeuroCast selects the best active peers set to render good quality. This section describes how NeuroCast coordinates the active peers by assigning the appropriate rate and data portion to each one. The assignment is based on the offered rate of each active peer and the current loss rate in the network. In this section we will describe how NeuroCast distributes the load among the different hits that have been selected to retransmit the channel. Namely, it will show the way the packets into which the stream has been divided are distributed.

```
1  int Channel::sum_Rps(set active){
2          int temp_sum_Rps = 0;
3          for (int i=0;i<active.num_peers;i++){
4                  temp_sum_Rps += active.peers[i].Rp;
5          }
6          return temp_sum_Rps;
7  }
8  double Channel::assigned_Rp(set act, peer P, double alpha)
9  {       double temp;
10         temp = (alpha*chanMgr->Ro)/sum_Rps(act); //alpha&R0 are
               1.
11         temp = temp * P.Rp;
12         return temp;
13 }
```

Figure 5.14: Methods to calculate the assigned number of packets.

CollectCast (48) algorithm uses packet coding through FEC (Forward Error Correction) codes like Reed-Solomon or Tornado, which allow to detect and correct errors in the channel. Due to this fact, during the packet allocation the algorithm takes into account a new parameter related to these codifications: *alpha* (toleration level to packet losses of the network). So, they use erasure codes (also known as FEC in the network community) to tolerate packet losses due to network fluctuations and limited peers reliability. The media file is divided into equal-length data segments. Each segment has a size of Δ original packets and is protected using FEC separately.

In this first version of NeuroCast, although it is already implemented, the final application is simplified and does not use FEC codes. Thus, we set $\Delta = 1$ achieving a load distribution proportional to the sending rates of the different peers. The active peers collectively send the media file segment by segment: they all cooperate in sending the first segment, then the second one, and so on. Peer p is assigned a number of packets D_p to send in proportion to its actual streaming rate:

$$D_p = \left\lceil \Delta \cdot \frac{R_p}{\sum_{x \in \mathbb{P}^{act}} R_x} \right\rceil \tag{5.4}$$

Figure 5.14 shows the code implemented in NeuroCast to solve equation 5.4. The first method calculates the sum of rates of the peers that the selected set consists of. The second method invokes the first one to calculate the number of packets that is assigned to each peer of the set. Both methods are part of the `choosingSources` method of the `Channel` class.

```
1 for (int i=0;i<active_peers.num_peers;i++){
2         assign_Rp=assigned_Rp(active_peers,active_peers.peers[i],
             alpha);
3         temp_data_d = ((chanMgr->DELTA)/(2-alpha))*((assign_Rp)/(
             alpha*chanMgr->Ro));
4         temp_data = (int)temp_data_d;
5         if (temp_data!=temp_data_d) temp_data++;
6         active_peers.peers[i].num_peer_chunks=(int)temp_data;}
```

Figure 5.15: Loop which distributes the chunks among the selected peers.

Figure 5.15 shows the code that uses the D_p value to distribute the chunks among the different peers of the set. Note the importance of the parameters `active_peers` and Δ (Delta) in the implementation. Finally, it can be seen that the number of chunks that are assigned to each peer is stored in the `num_peer_chunks` attribute inside the corresponding element in the `active_peers` list.

Following the example (see Figure 5.11) and taking as example the following value of the involved parameters, we will have the following load distribution:

Initial parameters:

$\Delta = 10$ (stream split into sets of 10 chunks).

$\alpha = 1$.

$R_0 = 1$ Mbps.

	R_p	$\hat{R}_p$	D_p	$\lceil D_p \rceil$
Peer 1	700 Kbits/s	7/12	5.8333	6
Peer 2	500 Kbits/s	5/12	4.1666	5

Table 5.1: Example of chunks distribution to each peer.

Therefore, we obtain as final result that the peer of 700 Kbps will send 6 packets, while the one of 500 Kbps will send 5 packets. So, the stream will have to be divided into 11 parts.

5.4.4 Network adaptation

To accommodate the maximum load of the network, NeuroCast adds a new functionality so that it is able to adapt at any moment the number of chunks downloaded by a peer. The variable used to determine the most appropriate distribution is the time between arrivals of packets.

For example, in case you are viewing a video at a rate of 1024 Kbps, if you use packets of 8 Kbytes (64Kbits), we know that the rate of arrival of packets is 16 paq/s, or what is the same, we expect the arrival of a new packet every 62 ms or so. By knowing this value for each peer from which the video is being downloaded, in case of delays in receiving their packets we can decrease its contribution progressively until the time between arrivals is stabilized.

Thus, controlling the time between packet arrivals during the session, NeuroCast sets a threshold that allows to, once it is overpassed, redistribute the number of chunks that each peer retransmits. This threshold is calculated following next equation:

$$threshold = \frac{\frac{64}{bitrate} \cdot FACTOR\_SEP\_PACKETS \cdot numChunks}{numSubChannelChunks} \tag{5.5}$$

As it can be seen in equation 5.5, this threshold depends on different parameters:

- In first place, it depends on the bitrate of the played video.
- In second place we have the `FACTOR_SEP_PACKETS`, which is configurable by the user, and corresponds to the maximum allowed delay between packet arrivals. For example, with a factor of 1.2 the limit will be around the 74 ms, assuming a bitrate of 1024 Kbps.
- The third factor (`numChunks`) is related to the number of chunks into which the stream is divided among the different hits.
- The last factor, the `numSubChannelChunks` is the number of chunks that this subchannel retransmits.

In order not to depend on small variations of the network, Neurocast checks every x packet the `time_between_arrivals`, and works with the average value obtained during this period. The parameter that determines the number of packets used to compute the average is configurable by the user at the beginning of each session through configuration file parameter `SIZE_BUFFER_SEP_PACKETS`.

The following function is used to calculate the mean time, and we need to update the value of the variable each time it comes a new packet.

$$t_{between\_arrivals} = t_{initial\_between\_arrivals} + \frac{dif\_time - t_{initial\_between\_arrivals}}{samples} \quad (5.6)$$

The `dif_time` parameter is simply the difference between the arrival of the final packet and the current one. Since the time between arrivals is measured in seconds, also the difference is in seconds.

The `samples` parameter correspond to the number of samples we have used so far to calculate the average value. In our case we use a counter that when reaches the first packet then we reset it to 1. Then it is incremented each time a new packet arrives.

Next we see an example:

From the values of the table 5.2 we note that after receiving 5 packets the mean time between arrivals is 4.79 seconds. Assuming that the expected value

Samples	New Packet Arrival Time	`dif_time`	$t_{between\_arrivals}$
1	5	5	5.00
2	9	4	4.50
3	18	8	5.66
4	22	4	5.24
5	25	3	4.79

Table 5.2: Example of time between arrivals calculation.

was 4 seconds, but with a `FACTOR_SEP _PACKETS` of 1.25, the application will continue to run normally because it does not exceed 5 seconds.

However, note that the third package has arrived with a difference from the previous time of 8 seconds, and if we would have not used this method because we had to redistribute the load of the peers as we exceed the 5 seconds threshold.

In this way, with this algorithm we have in a single variable the mean time between arrivals and we do not have to store the values obtained previously. Therefore, we need very few resources and the computing time is greatly reduced.

5.4.5 Load Redistribution

As we have seen in previous sections, in a dynamic environment such as p2p networks is essential to have a system to allow rapid adaptation to changes in the network. However, it has to be taken into account that any changes we make in what would be the normal operation of the application, it will involve some delay in playing the video.

This is the main reason that, apart from calculating the time between arrivals of the packets, it will be necessary to overcome the fixed threshold (Equation 5.5) a certain number of times consecutively before redistributing the load among different peers. This number of times that we can exceed the threshold is also set up at the beginning of the session by setting the `delay_margin` in the configuration file.

The diagram in figure 5.16 shows the process prior to redistribute the load among peers.

The method used to alert the rest of hits is the same that we use when requesting the `peerRp` parameter to the hits: we take advantage of the implementation

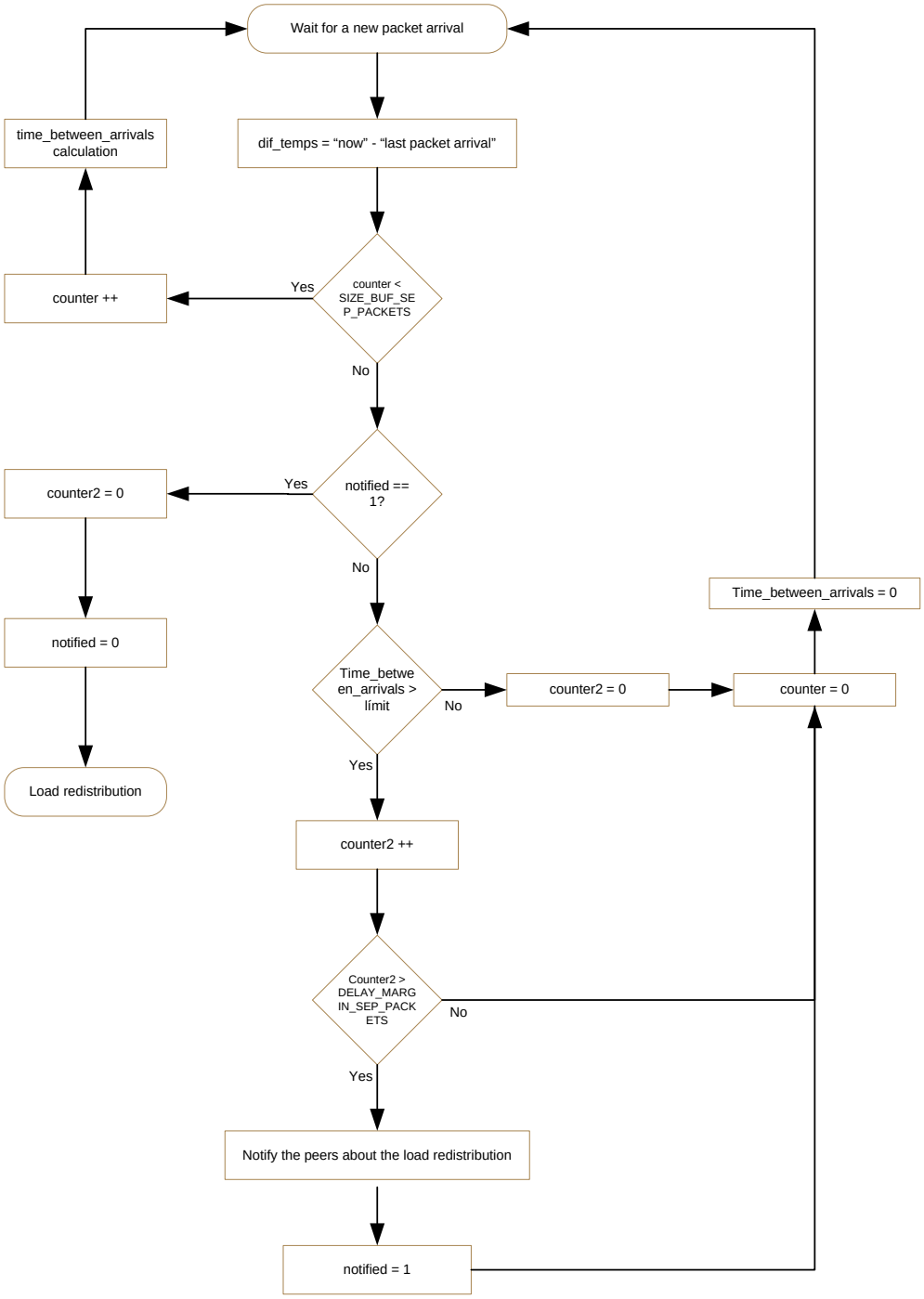


Figure 5.16: Flow of the process prior to the load redistribution.

of the handshaking via sockets.

Thus, we create a new socket for each one of our hits and send the message `CHANGE_PEER_CHUNKS` through it. Next, a NeuroCast server in the target machine reads the request and enables the flag associated with the redistribution, so that the `T_Relay` server that broadcasts the channel is ready to receive, if necessary a new number of chunks. In addition, the `servMgr` stores the IP address of the machine that has requested the change to ensure that it is the same to which the server sends packets.

Continuing with the diagram in the figure 5.16, once the subchannel is ready for the load redistribution, it follows these steps:

1st. Notifies the others subchannels about the redistribution.

2nd. Decreases by one the number of chunks that is downloading from its hit.

3rd. Searches for a subchannels with a lower `time_between_arrivals` in order to send the chunk it has you just subtracted.

4th. Waits until the subchannel with lower `time_between_arrivals` has redistributed the chunks.

The rest of subchannels once have been informed that a load redistribution is going to take place, they also enter into the process of redistribution. At this point is important the fact of having previously informed the hits sending us the chunks, so that we avoid those to notify the tracker that they have lost the connection with us and give us the low channel.

Here we must distinguish two different situations: on one hand the subchannels with lower `time_between_arrivals` and on the other hand the rest of subchannels. The latter simply expect that the redistribution is done, to continue with the download of the new allocated chunks.

And regarding to the subchannel with the lowest `time_between_arrivals`, it follows these steps:

1st. Checks that the channel has more than one subchannel currently. In case of failure to comply that, so that we are the only active subchannel, then it sets the number of chunks to 1 (the stream is not divided into different parts) and we start to download the full stream.

2nd. If there is more than one subchannel, it increases by one the number of chunks of the subchannel.

3rd. Finally, redistributes the load among all the hits from the list according to their number of chunks.

A different case from the one presented above, but that also requires load redistribution occurs when we do not receive packets from a certain subchannel. This case occurs in situations of heavy congestion on the network or with the collapse of any node between our machine and the one serving the stream. In these situations we would be waiting indefinitely the arrival of enough packets that allow us to calculate the mean time between arrivals, and thus stops the playback of the stream. For this reason, we have added a timer or time-out that is activated in case you are expecting a new packet for a time greater than N times the expected value between arrivals. Particularly the value of N is defined by the user at the beginning of the session with the parameter `arrivalsTimeout`. In these situations the load redistribution is forced immediately.

Another of the new features that have been introduced in NeuroCast is to get rid of the hits that are only sending a single chunk, and despite that, still send packets with delay. In these cases, we remove it from our list of hits and download the latest chunk from someone else, leaving the subchannels on standby until the channel becomes unavailable or we decide to disconnect.

The issue of redistributing the load is an open problem. There are other possibilities, so that the choice we have made here is not definitive and could always evolve to another in the future.

Chapter 6

NeuroCast Performance Evaluation

A first part of the experimental phase has been devoted to the validation of the NeuroCast implementation. These tests allowed us to find out some coding bugs and to check if the choices done to address a real implementation were correct. During this phase we continuously improved the protocols and mechanisms defined to provide to the core algorithms all the elements they need to correctly run in a real world environment.

Once we relied in our implementation, we started tests whose goals were to validate the core algorithms and the simulator implementation.

6.1 Evaluation Techniques

An important part of this project has undertaken the study of applications to measure the state of the network. This is why it was necessary to use a tool like VNUML (15) that allows us to have a virtual environment, completely adaptable to our needs, capable of simulating any type of network.

To deepen the analysis of different applications, it has been necessary to use NetEm (49) in the characterization of various parameters of the network, always trying to get the environment as closer to reality as possible.

6.1.1 VNUML

VNUML (15; 37) is a general purpose virtualization tool that brings the possibility of easily building complex network scenarios using virtual machines, interconnected through virtual networks of precisely specified topologies. These virtual machines are the functional equivalent to real ones, and can run inside just one physical host. VNUML is not a virtualization technique itself, but a front-end to UML (User Mode Linux) (32), the actual software that allows running a Linux kernel (using its own resources like memory, disk, process space, etc.) as a conventional process in the physical host.

The value added by VNUML (regarding the plain use of UML) is that the virtual scenario can be specified using a XML (eXtended Markup Language) file, that provides an abstract description of the network scenario to be created. Then, the VNUML parser reads the XML specification and sets up automatically the desired scenario, avoiding the need of knowing complex low-level details about UML and setting up the network manually (more time consuming and error prone). Section A.1 shows an example of the XML used to build an emulated NeuroCast network.

UML (in combination with `uml_switch`, the related tool to implement virtual networks) allows to set up the capabilities of the virtual machines and virtual networks at booting time (for example, the memory and disk filesystem of each virtual machine, along with the number of network interfaces and how they are interconnected between them). No tool-focused GUIs are required. In addition, UML virtual machines run stand-alone (no management daemons are required) and can be accessed out-of-band and managed from command line. Given these characteristics, UML integration in VNUML parser is quite direct and easy and makes it the preferable back-end among other virtualization alternatives (like Xen, Microsoft VServer or VMware).

From the point of view of the NeuroCast application considered in this thesis, all the network elements can be implemented using just a physical machine (actually, a conventional PC), with the equipment and management effort saving (it is clear that managing just a PC is easier than managing a network of devices) that this implies.

Using virtualization always implies a performance penalty comparing with

the implementation using real equipment (the overhead imposed by the virtual machine and virtual network layers).

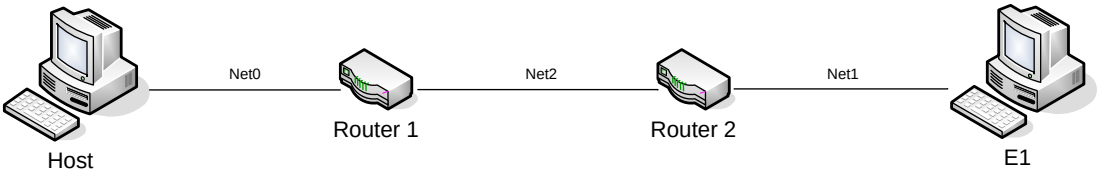


Figure 6.1: Network used during the tests.

The basic topology that has been used for the different tests is the one shown in figure 6.1. Through the configuration file of the VNUML (see A.1) we have defined the 3 virtual machines ($Router1$, $Router2$, and $E1$) and the 3 interfaces ($Net0$, $Net1$ and $Net2$). Apart from setting up the new "network" IP addresses allocated to different machines, we have limited the bandwidth of $Net1$ to 1.5 Mbits/s and $Net2$ to 1 Mbits/s to work. This way, with theses values of bandwidth close to what people have today in their homes we are able to get closer to a real scenario with the emulation.

Bottom line, with this scenario we have managed to create a simple but complete enough to get the results we are looking for.

6.1.2 Network Emulator

This chapter describes the usage of NetEm (49), the Linux network emulator module. NetEm is part of each standard linux kernel 2.6.7 and later. However some features are only available in kernel version 2.6.16 and later. The standard part of NetEm allows packet handling according to statistical properties. In addition a trace mode has been written though it is not part of the standard kernel. This trace mode allows the specification of an independent value for each packet to be processed.

NetEm provides functionality for testing protocols by emulating network properties. NetEm can be configured to process all packets leaving a certain network interface.

Four basic operations are available:

- **`delay`**: delays each packet.

- **`loss`**: drops some packets.
- **`duplication`**: duplicates some packets.
- **`corruption`**: introduces a single bit error at a random offset in a packet.

NetEm provides Network Emulation functionality for testing protocols by emulating the properties of wide area networks. The current version emulates variable delay, loss, duplication and re-ordering/corruption. In any 2.6 Linux distribution, (Fedora, OpenSuse, Gentoo, Debian, Mandriva, Ubuntu), NetEm is already enabled in the kernel and a current version of `iproute2` is included. NetEm is controlled by the command line tool `tc` which is part of the `iproute2` packet of tools. The `tc` command uses shared libraries and data files in the `/usr/lib/tc` directory.

In order to understand better the performance of this emulator, it becomes necessary to deepen the analysis of the Linux network traffic control.

6.1.2.1 Linux Network Traffic Control Overview

Traffic control is the name given to the sets of queuing systems and mechanisms by which packets are received and transmitted on a router. This includes deciding which (and whether) packets to accept at what rate on the input of an interface and determining which packets to transmit in what order at what rate on the output of an interface.

In the overwhelming majority of situations, traffic control consists of a single queue which collects entering packets and dequeues them as quickly as the hardware (or underlying device) can accept them. This sort of queue is a FIFO. The default `qdisc` under Linux is the `pfifo_fast`, which is slightly more complex than the FIFO.

There are examples of queues in all sorts of software. The queue is a way of organizing the pending tasks or data. Because network links typically carry data in a serialized fashion, a queue is required to manage the outbound data packets.

In the case of a desktop machine and an efficient webserver sharing the same up-link to the Internet, the following contention for bandwidth may occur. The web server may be able to fill up the output queue on the router faster than the data can be transmitted across the link, at which point the router starts to

drop packets (its buffer is full). Now, the desktop machine (with an interactive application user) may be faced with packet loss and high latency. Note that high latency sometimes leads to screaming users. By separating the internal queues used to service these two different classes of application, there can be better sharing of the network resource between the two applications.

Traffic control is the set of tools which allows the user to have granular control over these queues and the queuing mechanisms of a networked device. The power to rearrange traffic flows and packets with these tools is tremendous and can be complicated, but is no substitute for adequate bandwidth.

Linux offers a rich set of traffic control functions. In this section, we briefly give an overview of the design of the respective kernel code, describes its structure, and illustrates the addition of new elements.

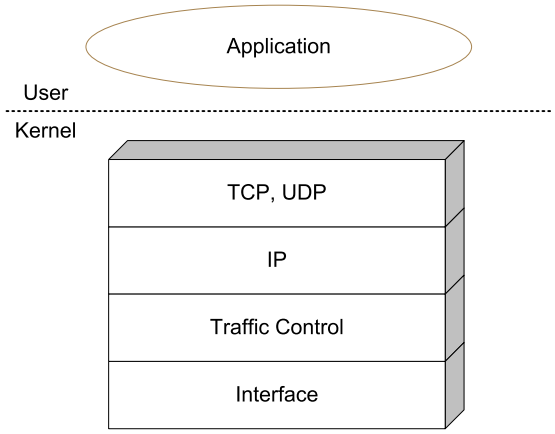


Figure 6.2: Traffic control layer architecture in Linux.

Figure 6.2 shows roughly how the kernel processes data received from the network, and how it generates new data to be sent on the network: incoming packets are examined and then either directly forwarded to the network (e.g. on a different interface, if the machine is acting as a router or a bridge), or they are passed up to higher layers in the protocol stack (e.g. to a transport protocol like UDP or TCP) for further processing. Those higher layers may also generate data on their own and hand it to the lower layers for tasks like encapsulation, routing, and eventually transmission.

"Forwarding" includes the selection of the output interface, the selection of the next hop, encapsulation, etc. Once all this is done, packets are queued on

the respective output interface. This is the point where traffic control comes into play. Traffic control can, among other things, decide if packets are queued or if they are dropped (e.g. if the queue has reached some length limit, or if the traffic exceeds some rate limit), it can decide in which order packets are sent (e.g. to give priority to certain flows), it can delay the sending of packets (e.g. to limit the rate of outbound traffic), etc. Once traffic control has released a packet for sending, the device driver picks it up and emits it on the network.

The traffic control code in the Linux kernel consists of the following major conceptual components:

- queuing disciplines.
- classes (within a queuing discipline).
- filters.
- policing.

Each network device has a queuing discipline associated with it, which controls how packets enqueued on that device are treated. A very simple queuing discipline may just consist of a single queue, where all packets are stored in the order in which they have been enqueued, and which is emptied as fast as the respective device can send.

More elaborate queuing disciplines may use filters to distinguish among different classes of packets and process each class in a specific way, e.g. by giving one class priority over other classes. Figure 6.3 shows an example of such a queuing discipline. Note that multiple filters may map to the same class.

Queuing disciplines and classes are intimately tied together: the presence of classes and their semantics are fundamental properties of the queuing discipline. In contrast to that, filters can be combined arbitrarily with queuing disciplines and classes as long as the queuing discipline has classes at all. But flexibility doesn't end yet classes normally don't take care of storing their packets themselves, but they use another queuing discipline to take care of that. That queuing discipline can be arbitrarily chosen from the set of available queuing disciplines, and it may well have classes, which in turn use queuing disciplines, etc.

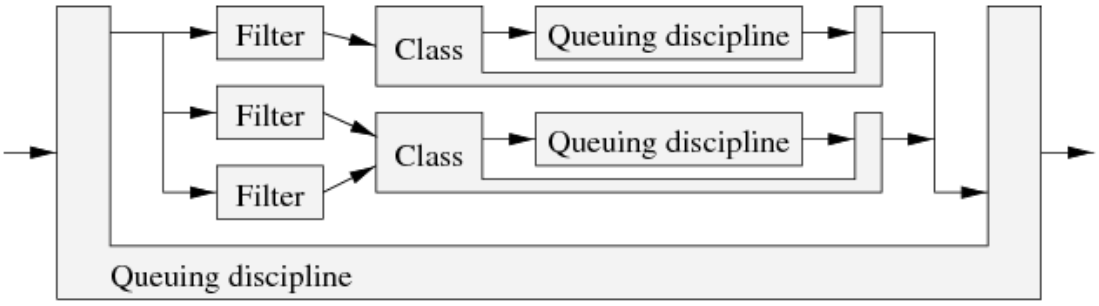


Figure 6.3: Queuing disciplines.

Figure 6.3 shows an example of such a stack: first, there is a queuing discipline with two delay priorities. Packets which are selected by the filter go to the high-priority class, while all other packets go to the low-priority class. Whenever there are packets in the high-priority queue, they are sent before packets in the low-priority queue (e.g. the `sch_prio` queuing discipline works this way). In order to prevent high-priority traffic from starving low-priority traffic, we use a token bucket filter (TBF), which enforces a rate of at most 1 Mbps. Finally, the queuing of low-priority packets is done by a FIFO queuing discipline.

Note that there are better ways to accomplish what it is done here, e.g. by using class-based queuing (CBQ). Packets are enqueued as follows: when the enqueue function of a queuing discipline is called, it runs one filter after the other until one of them indicates a match. It then queues the packet for the corresponding class, which usually means to invoke the enqueue function of the queuing discipline "owned" by that class. Packets which do not match any of the filters are typically attributed to some default class.

Typically, each class "owns" one queue, but it is in principle also possible that several classes share the same queue or even that a single queue is used by all classes of the respective queuing discipline. Note however that packets do not carry any explicit indication of which class they were attributed to. Queuing disciplines that change per-class information when dequeuing packets (e.g. CBQ) may therefore not work properly if the "inner" queues are shared, un less they are able either to repeat the classification or to pass the classification result from enqueue to dequeue by some other means.

Usually when enqueuing packets, the corresponding flow(s) can be policed, e.g. by discarding packets which exceed a certain rate.

Finally, we show two examples of NetEm usage:

1. In the case, we want to add a delay of 200ms to packets to going through the `eth0` interface of a `x` machine:

   ```
   tc qdisc add dev eth0 root NetEm delay 200ms
   ```

2. In the case we want to restrict the traffic rate to 1500 Kbits/s, and we also want to add a packet error probability of 10%:

   ```
   tc qdisc add dev eth0 root handle 1:0 NetEm loss 10%
   ```

   ```
   tc qdisc add dev eth0 parent 1:1 handle 10:  tbf rate 1500Kbit lat
   50ms burst 299999b
   ```

In both cases we use the `tc` (traffic control) instruction to create the `qdisc` (queue discipline), and we also indicate the device (`add dev eth0`) where the traffic control has to be applied. The difference is that in the second case we add two restrictions, which is why we need to create two different disciplines, a first NetEm discipline which adds a packet error probability of 10% , and a second TBF discipline (Token Bucket Filter) that "hangs" from the first (parent 1:1) and limits the speed of outgoing traffic.

6.2 Network Performance Measurement Tools

In data communication networks, high available bandwidth is useful because it supports high volume data transfers, short latencies and high rates of successfully established connections. Obtaining an accurate measurement of this metric can be crucial to effective deployment of QoS services in a network and can greatly enhance different network applications and technologies.

Several applications need to know the bandwidth characteristics of their network paths. For example, some peer-to-peer applications need to consider available bandwidth before allowing peers to join the network. Overlay networks can configure their routing table based on the available bandwidth of the overlay links. Network providers lease links to customers and the charge is usually based on the available bandwidth that is provided. Service Level Agreements (SLAs) between

providers and customers often define service in terms of available bandwidth at network boundaries.

Available bandwidth is also a key concept in congestion avoidance algorithms and intelligent routing systems. Techniques for estimating available bandwidth fall into two broad categories: passive and active measurement. Passive measurement is performed by observing existing traffic without perturbing the network. It needs to process the full load on the link and requires access to all intermediary nodes in the network path to extract end-to-end information. Active measurement on the other hand, directly probes network properties by generating the traffic needed to make the measurement. Despite the fact that active techniques inject additional traffic on the network path; it is more suitable to use active probing measurement in order to measure end-to-end available bandwidth.

Many different active probing techniques and tools for available bandwidth measurement have been developed and evaluated. However, a sufficient number of studies comparing the performance of these tools have not been carried out. In this thesis, a study of available bandwidth measurement methodologies is presented and a comparative analysis in terms of accuracy, intrusiveness and response time of active probing tools for this metric is achieved. We have collected our measurements using a simple testbed configuration that allows us to test measurement tools (Pathload, Pathrate, TTCP, NTTCP, Iperf, D-ITG) with the same parameters.

In practice, four bandwidth measurements can be performed, namely the capacity/raw bandwidth of a link, the end-to-end capacity of a path, the available bandwidth of a link and the available bandwidth of a path. In the following we will define available bandwidth parameters and present the techniques and tools for this metric.

P is a network path from source S to destination D. P is a sequence of N store-andforward links LI_1, LI_2,... LI_N. We assume that P is fixed and unique (no routing changes or multipath forwarding occur during the measurement).

- **Capacity of the link**: Denoted C_i, is the maximum possible IP layer transfer rate at that link. The end-to-end capacity of the path is then the maximum IP layer rate that this path can transfer from the source S to the sink D:

$$C = \min C_i \tag{6.1}$$

- **Available bandwidth of a link**: Defines the unused capacity of this link during a certain time period. We assume that link i is transmitting $C_i \cdot u_i$ bits during a time interval T. u_i is the utilization rate of this link during T, with $0 \leq u_i \leq 1$. The available bandwidth A_i of the link i is:

$$A_i = C_i \cdot (1 - u_i) \tag{6.2}$$

- **Available bandwidth a path**: The available bandwidth A of the path P during the time interval T is the minimum of the available bandwidth of all links that comprise P:

$$A = \min_{i=1..N} C_i \cdot (1 - u_i) = \min_{i=1..N} A_i \tag{6.3}$$

In our case it is particularly important to have a good knowledge of the available bandwidth of the links, to decide the best possible load distribution among peers, and from this information distribute the stream parts among them.

For this reason, different measurements tools have been analyzed, allowing us to measure the available bandwidth or in some cases the load of the network. In the next sections we introduce these tools focusing in its advantages and main drawbacks.

6.2.1 Pathload

Pathload (59) estimates available bandwidth. It is based on the use of a sequence of so called Self-Loading Periodic Streams. The basic idea is that the one-way delay of a periodic packet stream shows increasing trend when the stream rate is larger than the available bandwidth along the path.

Measurement starts with sending the first periodic packet stream of UDP packets from the sender to the receiver. The sender timestamps each packet. The receiver compares this timestamp with the arrival time and computes relative one-way delay. If time is not synchronized between the sender and the receiver, it is not a real one-way delay, but for the purposes of this measurement we just need to know how measured relative one-way delay changes for subsequent packets.

When the transmission rate of the packet stream is larger than available bandwidth, a short term overload should cause an increasing trend in measured relative one-way delay. A series of packet streams of different rates is generated until pathload iteratively finds an approximate available bandwidth.

The main advantages of Pathload are:

- "Non-intrusive".
- Relative high accurate.

However, the major drawbacks of Pathload are:

- It introduces large network traffic.
- Requires both ends of the link for available bandwidth estimation.
- The average measure time is relative long.
- Unsuitable for many real-time applications because of sending a lot of packet streams.

Therefore, Pathload is presented as a "non-intrusive" measurement tool, but with some limitations. The most important thing for us is the time of convergence of the algorithm which is quite high and makes it a useless tool for instant measurements.

6.2.2 Pathrate

Pathrate (34) is a tool that is able to measure the capacity of network paths. Two bandwidth metrics that are commonly associated with a path are the capacity and available bandwidth. Pathrate defines the capacity as the maximum throughput the path can provide when there is no competing traffic load (cross traffic). Likewise, it defines the available bandwidth as the maximum throughput the path can provide to a flow, given the current cross traffic load.

Pathrate is based on the dispersion of packet pairs and packet trains. It uses many packet pairs (with packets of variable size) to uncover the generally multimodal bandwidth distribution characteristic of the path. The local modes in this

distribution are possible values for the capacity of the path. Then pathrate uses long packet trains to estimate the so-called asymptotic dispersion rate (ADR). The capacity of the path is always larger than the ADR. Among the local modes that are higher than the ADR, pathrate chooses the strongest and narrowest mode as the final capacity estimate.

Pathrate was designed to be robust to cross traffic effects, meaning that it can measure the path capacity even when the path is significantly loaded. This is crucial, since the hardest paths to measure are the heavily loaded ones.

Pathrate differs from other bandwidth estimation tools, such as *pathchar*, *clink*, *pchar*, *nettimer*, and *pipechar*, which attempt to measure the capacity of each link in the path. The technique they use, however, often provides wrong estimates when the path includes "hidden" layer 2 switches.

Pathrate is publicly available with source code, documentation, and installation instructions. The tool is actively maintained and runs on all major UNIX systems, and does not require superuser privileges.

6.2.3 TTCP

Test TCP or TTCP (12), is a test tool to perform TCP/IP or UDP/IP performance tests. TTCP is a command-line sockets-based benchmarking tool for measuring performance between two systems. It was originally developed in 1984 by Mike Muuss and Terry Slattery for the BSD operating system. The original TTCP and sources are in the public domain, and copies are available from many anonymous FTP sites.

TTCP sends and receives TCP/UDP data. This tool have control over the number of packets sent, the packet size, the port number at which the transmitter and receiver rendezvous, and several other parameters. By changing the parameters, it can test various buffering mechanisms in the network equipment between the transmitter and receiver. Moreover, the destination does not have to be another router.

To test throughput to a remote server, start only a transmitter and specify the discard port (TCP port 9) on the remote server. TTCP reports the amount of data transferred, the transfer time, and the approximate throughput. By comparing the actual throughput with the theoretical bandwidth between the

transmitter and receiver, it can be told whether the network is operating as expected. Variations in throughput may indicate a significant amount of other traffic, overloaded network equipment, or perhaps communications errors which are causing packets to be corrupted or dropped. By performing tests on individual segments of the path, it can be determined which links or devices should be examined in more detail.

6.2.4 NTTCP

NTTCP (7), a `ttcp` variant, measures the time required to send a set number of fixed-size packets. NTTCP is a communications analysis and simulation tool developed by NSWC-DD was modified to serve as a network resource monitor by collecting the metrics of interest for the 27 paths required by the RTDS application. NTTCP was already capable of measuring all the metrics of interest (and many more) but trade-offs led to specific configuration option settings and a structural redesign. The NTTCP program measures the transfer-rate (and other numbers) on a TCP, UDP or UDP multicast connection.

Similar to Iperf (18), NTTCP uses a client and server to perform tests, with data being sent from client to server in typical unidirectional tests. To perform a basic throughput test, we have to install NTTCP on two machines reachable via an IP network. On one machine, we start a NTTCP server, by running the command `NTTCP -i`. After starting the server, we start the NTTCP client by running the command `NTTCP -T <server ip>`, where <server ip> is the IP address of the machine running the NTTCP server. This commence an untimed test that will complete when 8388608 bytes have been transferred. A test summary will be printed to screen on completion.

There are a whole range of options to tweak, which can be found by running `NTTCP` without any arguments. Of particular interest for TCP testing are the `-w`, `-l` and `-n` options. The `-w` option allows to specify the size of the send buffer used by the NTTCP TCP socket. The `-l` option allows to specify the size of the buffers transmitted by NTTCP during the test. The `-n` option allows to specify the number of buffers transmitted by NTTCP during the test. The product of the buffer size (specified using `-l`) and number of buffers (`-n`) equates to the number of bytes transmitted during the test.

6.2.5 Iperf

Iperf (18) is a bandwidth measurement tool which is used to measure the end-to-end achievable bandwidth, using TCP streams, allowing variations in parameters like TCP window size and number of parallel streams. End-to-end achievable bandwidth is the bandwidth at which an application in one end-host can send data to an application in the other end-host. Iperf approximates the cumulative bandwidth (the total data transferred between the end-hosts over the total transfer period) to the end-to-end achievable bandwidth.

We need to run Iperf for fairly long periods of time to counter the effects of TCP slow-start. For example, while running Iperf from SLAC to Rice University using a single TCP stream, with a TCP window size of 1 MB set at both ends, only 48.1 Mbps is achieved during slow-start (slow-start duration was about 0.9 seconds, the Round Trip Time (RTT) for this path was about 45ms), whereas the actual bandwidth achievable is about 200 Mbps. For the cumulative bandwidth to get up to 90% of the end-to-end achievable bandwidth, we need to run Iperf for about 7 seconds.

The most basic use of iperf involves running one instance in server mode and one instance in client mode, sending packets from the client to the server. Thus, to measure how the target TCP/IP stack performs on the receive (RX) side, first start iperf in server mode on the target: `iperf s`. Then, run iperf in client mode on the development host: `iperf <client ip>`. By default, the iperf client will generate packets as fast as possible for 10 seconds and report the average bandwidth.

6.2.6 Distributed Internet Traffic Generator

Distributed Internet Traffic Generator (D-ITG) (21) is a platform capable to produce traffic that accurately adheres to patterns defined by the inter departure time between packets (IDT) and the packet size (PS) stochastic processes. Such processes are implemented as an i.i.d. sequence of random variables. A rich variety of probability distributions is available: constant, uniform, exponential, Pareto, Cauchy, normal, Poisson and gamma. Also, D-ITG embeds some models proposed to emulate sources of various protocols: TCP, UDP, ICMP, DNS, Telnet and VoIP (G.711, G.723, G.729, Voice Activity Detection, Compressed RTP).

This means that the user simply chooses one of the supported protocols and the distribution of both IDT and PS will be automatically set.

D-ITG can perform both one-way-delay (OWD) and round-trip-time (RTT) measurements, packet loss evaluation, jitter and throughput measurement. For each generation experiment it is possible to set a seed for the random variables involved. This option gives the possibility to repeat many times exactly the same traffic pattern by using the same seed. Also, D-ITG permits the setting of TOS (DS) and TTL packet fields.

D-ITG allows to store information both on the receiver side and the sender side. It is thus possible to retrieve information on the traffic pattern generated. Additionally, DITG enables the sender and the receiver to delegate the logging operation to a remote log server. This option is useful when the sender or the receiver have limited storage capacity (e.g. PDAs, Palmtops, etc.). Also, it can be used to analyze log information on-the-fly, for example, in case the sender is instructed by a controller entity to adapt the transmission rate based on channel congestion and receiver capacity.

Another innovative feature is that the sender can be remotely controlled by using ITGApi. This means that the DITG sender can be launched in daemon mode and wait for commands that instruct it to generate traffic flows.

Moreover, D-ITG is currently available on Linux, Windows and Linux Familiar platform.

6.2.6.1 Usage Examples

6.2.6.1.1 Example 1

Single UDP flow with constant inter-departure time between packets and constant packets size.

```
1.start the receiver on the destination host (say it B):
./ITGRecv
2.start the sender on the source host (say it A):
./ITGSend -a B -sp 9400 -rp 9500 -C 100 -c 500 -t 20000
-x recv log file
```

The resulting flow from A to B has the following characteristic: the sender port is 9400, the destination port is 9500, 100 packets per second are sent (with constant inter-departure time between packets). The size of each packet is equal to 500 bytes, the duration of the generation experiment is 20 seconds at receiver side ITGRecv creates log file recv log file.

6.2.6.1.2 Example 2

Single TCP flow with constant inter-departure time between packets and uniformly distributed packet size between 500 and 1000 bytes with local sender/receiver log.

```
1. start receiver on the destination host (10.0.0.3)
[carlos@hernandez]$ ./ITGRecv -l recv log file
2. start the sender on the source host
[carlos@hernandez]$ ./ITGSend -a 10.0.0.3 -rp 9501 -C 1000 -u
500 1000 -l send log file
3. close the ITGRecv by pressing Ctrl-C
4. decode the receiver log file on the destination host:
[carlos@hernandez]$  ./ITGDec recv log file.
5. decode the sender log file on the source host:
[carlos@hernandez]$ ./ITGDec send log file
```

6.2.6.1.3 Example 3

Single TCP flow with constant inter-departure time between packets and uniformly distributed packet size between 500 and 1000 bytes with remote sender/receiver log.

```
1. start the log server on the log host:
[carlos@hernandez]$ ITGLog
2. start the receiver on the destination host:
[carlos@hernandez]$  ITGRecv
3. start the sender on the source host:
[carlos@hernandez]$  ITGSend -a 10.0.0.3 -rp 9501 -C 1000 -u 500
```

```
1000 -l send log file -L 10.0.0.3 UDP -X 10.0.0.3 UDP -x recv log file
4. close the receiver by pressing Ctrl-C
5. close the log server by pressing Ctrl-C
6. decode the receiver log file on the log host:
[carlos@hernandez]$  ITGDec recv log file
7. decode the sender log file on the log host:
[carlos@hernandez]$  ITGDec send log file
```

6.3 Tools Evaluation

Once we have introduced some of the tools that we have in the market, in this section we analyze some of their features in order to justify the choice of some of them for evaluating NeuroCast.

The desired features for evaluating NeuroCast are: low network load, high measurements rate and accurate information.

6.3.1 VNUML and NetEm Analysis

As we have shown before, D-ITG is a very comprehensive tool that allows us to perform any action on a network. This is why before starting to analyze the tools used to measure the throughput with NeuroCast, we check the characteristics of these virtual networks created with VNUML and with the constraints added with NetEm.

6.3.1.1 VNUML Scenario without constraints

Based on the scenario described in Figure 6.1, the following table 6.1 shows the results observed with D-ITG measuring the characteristics of the link $Net2$ between the virtual machine $Router1$ ($R1$) and the virtual machine $Router2$ ($R2$).

In the table 6.1 we can see how the measured bit rate varies around 932 bps, given the samples from the 1st second when the system is stabilized, while we have specified a VNUML available bandwidth of 1000 bps.

Moreover we have that the delay varies around 0.5 seconds, setting the jitter 0.008 seconds. Finally note that there is no packet loss. This results are shown

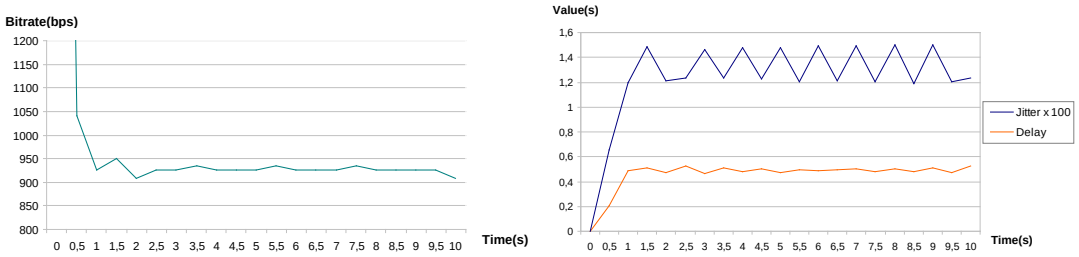
Time	Bitrate	Delay	Jitter	Packet Loss
0,000000	3760,128000	0,000151	0,000013	0,000000
0,500000	1040,384000	0,203760	0,004535	0,000000
1,000000	925,696000	0,488945	0,007059	0,000000
1,500000	950,272000	0,510741	0,009784	0,000000
2,000000	909,312000	0,472353	0,007359	0,000000
2,500000	925,696000	0,522727	0,007092	0,000000
3,000000	925,696000	0,462474	0,009982	0,000000
3,500000	933,888000	0,513308	0,007205	0,000000
4,000000	925,696000	0,476295	0,009997	0,000000
4,500000	925,696000	0,505333	0,007242	0,000000
5,000000	925,696000	0,475532	0,010024	0,000000
5,500000	933,888000	0,498914	0,007025	0,000000
6,000000	925,696000	0,486195	0,010055	0,000000
6,500000	925,696000	0,494668	0,007179	0,000000
7,000000	925,696000	0,499790	0,009958	0,000000
7,500000	933,888000	0,483294	0,007202	0,000000
8,000000	925,696000	0,499551	0,009985	0,000000
8,500000	925,696000	0,479758	0,007119	0,000000
9,000000	925,696000	0,507167	0,009948	0,000000
9,500000	925,696000	0,471051	0,007304	0,000000
10,000000	909,312000	0,523042	0,007123	0,000000
	932,2496	0,478745	0,008159	

Table 6.1: Measured parameters of the Net2 link with DITG.

graphically in Figure 6.4(a) and 6.4(b):

6.3.1.2 Analysis of the Delay and Losses introduced by NetEm

The delay is one of the characteristics of the P2P networks that can affect our application. Due to the the on-line playback of any multimedia stream, we are timely conditioned to the arrival of packets from our sources. Any delay that occurs at some point of the network will involve a gap in the playback of the stream, with the inconvenience this causes to the end-user.



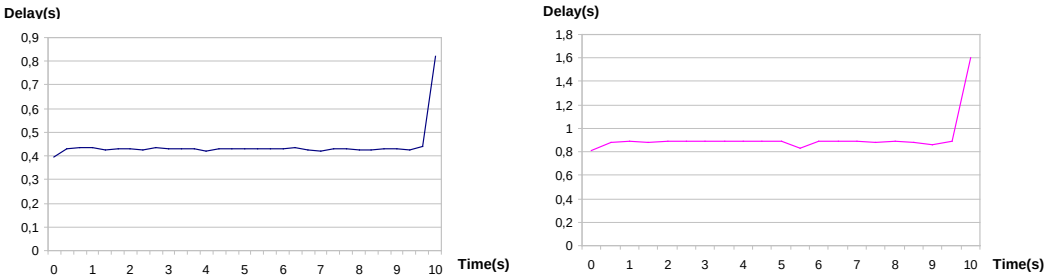
(a) Bitrate evolution.

(b) Delay and Jitter evolution.

Figure 6.4: D-ITG measurements.

To study the performance of the NeuroCast in network scenarios with different delays, first we measure the delays of 800 ms and 400 ms with the Iperf tool introduced to the *Net*2 link by NetEm .

Graphics in figures 6.5(a) and 6.5(b) show how the delay introduced by the NetEm is not constant but shows small fluctuations around the expected values in each of the situations. In the first case the values vary around 400 ms and in the second one around 800 ms, always a little above the value entered with NetEm.



(a) Delay of 400 ms.

(b) Delay of 800 ms.

Figure 6.5: Delay Iperf measurements.

In the following, we analyze the packet loss in several scenarios where different values of packet loss have been introduced with NetEm. We have limited the time for measuring of Iperf to 300 seconds, and we have obtained partial results every 30 s, which are represented by each of the bars in the graph of Figure 6.6. For example, in the first test we introduced a loss of 0.01% as follows:

- `Router1: ~ # tc qdisc change dev eth2 root NetEm loss 0.01\%`

Then, we executed the Iperf in server mode in the virtual machine *Router*2, with the option `-i 30` to get the partial results every 30 seconds, and the option `-u` to force the UDP transmission mode and to calculate the loss of packets.

`- Router2: ~ # iperf-s-and u-30`

Finally, we run the Iperf in client in the virtual machine *Router*1 with the option `-t 300` to indicate the measures will last 300 seconds.

`- Router1: ~ # iperf-c 10.0.1.2-u-t 300`

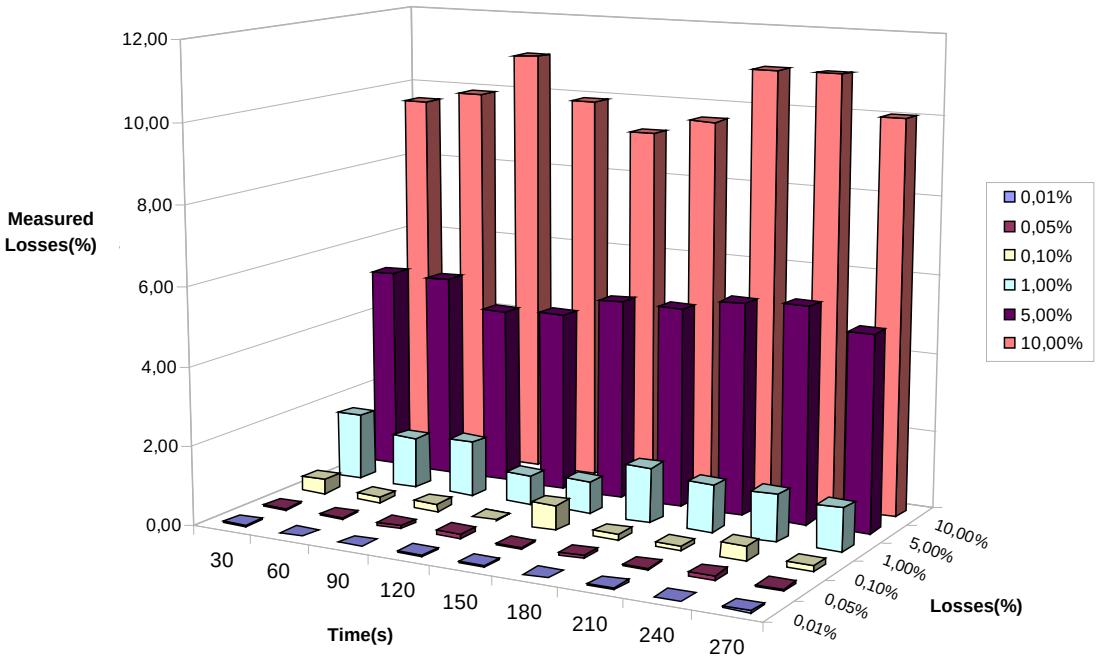


Figure 6.6: Losses measured with Iperf in different scenarios with incremental losses caused with NetEm.

The results with Iperf at *Router*2 are as follows:

```
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  105 KByte (default)
------------------------------------------------------------
[  3] local 10.0.1.2 port 5001 connected with 10.0.1.1 port 3073
[ ID] Interval       Transfer     Bandwidth       Jitter   Lost/Total Datagrams
[  3]  0.0-30.0 sec  3.76 MBytes  1.05 Mbits/sec  16.883 ms    1/ 2681 (0.037%)
[  3] 30.0-60.0 sec  3.74 MBytes  1.05 Mbits/sec  15.713 ms    0/ 2668 (0%)
[  3] 60.0-90.0 sec  3.76 MBytes  1.05 Mbits/sec  15.607 ms    0/ 2680 (0%)
[  3] 90.0-120.0 sec  3.75 MBytes  1.05 Mbits/sec  15.094 ms    1/ 2675 (0.037%)
```

```
[  3] 120.0-150.0 sec  3.75 MBytes  1.05 Mbits/sec  14.871 ms    1/ 2676 (0.037%)
[  3] 150.0-180.0 sec  3.75 MBytes  1.05 Mbits/sec  15.231 ms    0/ 2675 (0%)
[  3] 180.0-210.0 sec  3.75 MBytes  1.05 Mbits/sec  14.476 ms    1/ 2674 (0.037%)
[  3] 210.0-240.0 sec  3.76 MBytes  1.05 Mbits/sec  16.647 ms    0/ 2679 (0%)
[  3] 240.0-270.0 sec  3.73 MBytes  1.04 Mbits/sec  13.752 ms    1/ 2665 (0.038%)
[  3]  0.0-299.9 sec  37.5 MBytes  1.05 Mbits/sec  15.995 ms    7/26751 (0.026%)
```

With these results we look at the last line (the summary) and see that during the 300 seconds it took the test, it has been sent a total of 26,751 packets and it have been lost 7 packets, which represents a loss of 0.026%. As it is a small value compared to the measurement period of time (every 30 seconds one packet or any is lost), the measured result differs slightly from the 0.01% that we had set up. However, if we increase the time to measure, the Packet Loss values that we obtain are close to theoretical ones.

The graph in Figure 6.6 has been generated from the values obtained in 6 modified scenarios with NetEm, each one with a value of lost packets from 0.01% to 10%.

6.3.2 Iperf Evaluation

In this section, we analyze the measurements carried out with the Iperf tool at different instants. The tests are conducted on a link of 1 Mbps, and we also analyze the number of packets sent to obtain the result.

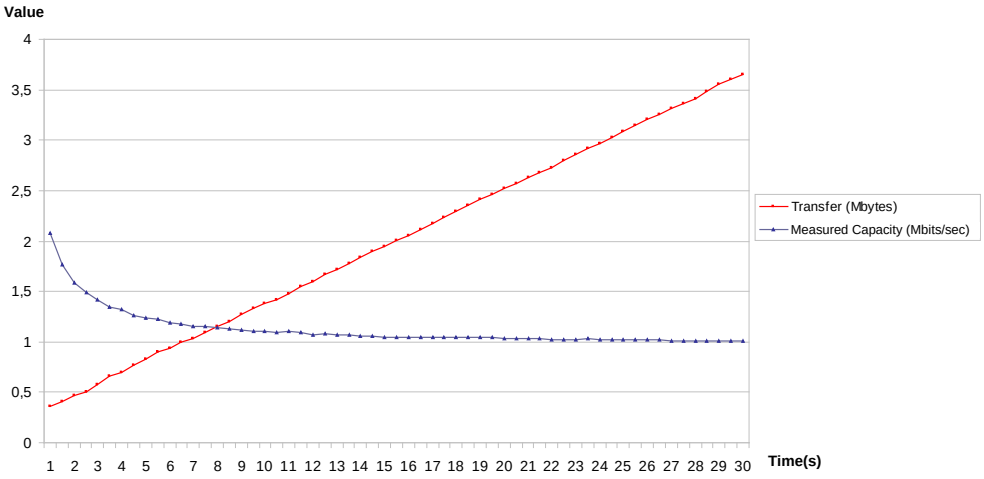


Figure 6.7: Iperf Algorithm Convergence.

Graph in Figure 6.7 gives a pretty clear idea of the convergence of the algorithm, since we observe that the measured capacity values tend to converge fast to the expected value. For example, it takes only 5 seconds to achieve a measured capacity of 1.2 Mbps, namely we got a measurement with an error close to 20%. After 15 seconds the error of the measurements is below a 5%.

On the other hand we see, of course, that the increase in the measurement time increases the number of sent packets. Hence, we increase the intrusive traffic in the network.

Thus when measuring, we have a trade-off among the duration of the measurements, the reliability of the results and the intrusive traffic that is being generated.

It is also interesting to analyze the measured capacity versus the network losses. Next chart in figure 6.8 shows that the Iperf estimation of the available bandwidth of the link hardly varies when the losses are less than 5%.

Figure 6.8: Capacity measured varying the packet loss.

In the same way, we analyze the measured capacity as a function of the network delay. Next chart in figure 6.9 shows that the estimation of the available bandwidth of the link hardly changes when the delay is lower than 150 ms. This reveals the low accuracy of the Iperf algorithm.

Finally, we also analyze the measured capacity versus the network load. To get a cross traffic that varies in time from the host, we send a stream of variable data (following the following script_file) the machine $E2$ with the D-ITG:

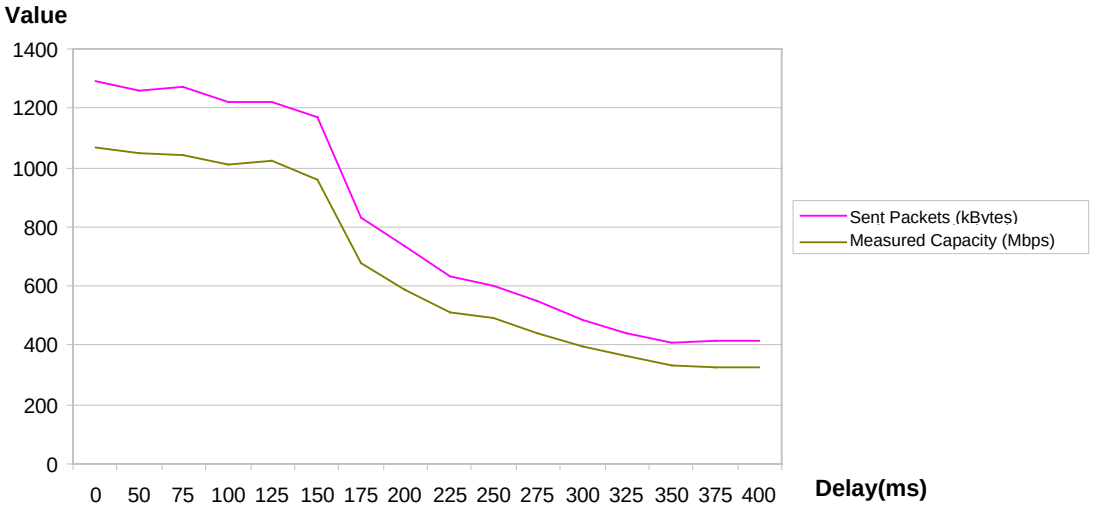


Figure 6.9: Capacity measured varying the delay.

```
-a 10.0.2.2 -rp 10001 -C 25 -c 1000 -t 20000
-a 10.0.2.2 -rp 10002 -C 50 -c 1000 -d 20000 -t 20000
-a 10.0.2.2 -rp 10003 -C 75 -c 1000 -d 40000 -t 20000
-a 10.0.2.2 -rp 10004 -C 100 -c 1000 -d 60000 -t 20000
```

First we define the IP address of the machine where to direct the traffic. Secondly, we define the port used to send data. Then we define the number of packets per second and the length of the packets. Finally, with the `-d` option we introduce the delay while sending the traffic and with the `-t` option the duration. We can see that we have set these values so that the flows are sent consecutively, each with a duration of 20 seconds.

Once the cross traffic is created, we run Iperf on *Router*1 and *Router*2. In order to be able to analyze the data after the program execution, we run it for 100 seconds. We observe the bandwidth available at intervals of 0.5 seconds and store the results into the file *result_r*2.

```
Router2: # iperf -s -t 100 -i 0.5 > result_r2
```

If we analyze the results (see Figure 6.10), we see that as the load increases (cross traffic introduced by the D-ITG) the available bandwidth measured with Iperf decreases, so that at any time the sum of the load and the bandwidth

remains fairly constant at around 1000 Kbits/s, which matches the capacity of the link set at the beginning of the session with VNUML. We refer the reader to section B.1, where the results from which we generated the graph of the figure 6.10 are detailedfor further analysis.

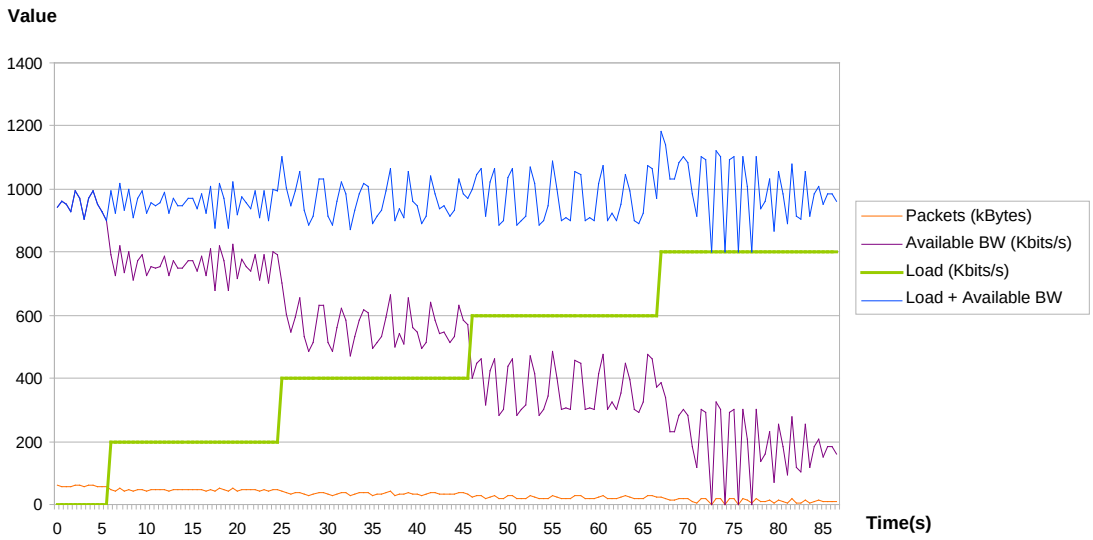


Figure 6.10: Capacity measured varying the load.

6.3.2.1 Iperf vs. NTTCP

Finally we present a comparison table between the Iperf (a program used to assess the state of the network in NeuroCast), and the NTTCP used in other P2P video streaming application. Table 6.2 has been compiled from the results obtained while measuring the available bandwidth in the *Net*2 link by both tools. The capacity of the link was of 1000 Kbits/s and in the tests we have varied the load with D-ITG.

Unlike NTTCP, Iperf permits to fix the duration of the measurements, which is why in this case we have performed the tests twice for each load value, the first time setting a measurement of 10 seconds and the other one without time constraints.

Analyzing Table 6.2, note that one of the most important results is the duration of the measurements. In all the presented cases the NTTCP duration values are much higher than the ones with Iperf. The duration obtained varies from 68

Transmitter	Receiver	Link Capacity fixed VNUML (Kbits/s)	Link Load (Kbits/s)	Tool	Time(s)	Sent Traffic (Mbytes)	Measured Av_BW(%)	Error(%)
R1	R2	1000	0	NTTCP	68	8	0,985	1,50
R1 (client)	R2 (server)	1000	0	iperf	10,4	1,38	1,11	-11,00
R2	R1	1000	200	NTTCP	93	8	0,716	8,40
R1 (client)	R2 (server)	1000	200	iperf	10,6	0,760	0,618	18,20
R1 (client)	R2 (server)	1000	200	iperf	20,4	1,65	0,676	12,40
R2	R1	1000	400	NTTCP	109	8	0,616	-1,60
R2	R1	1000	400	iperf	11,1	0,624	0,460	14,00
R2	R1	1000	400	iperf	21,3	0,976	0,376	22,40
R2	R1	1000	600	NTTCP	206	8	0,325	7,50
R1 (client)	R2 (server)	1000	600	iperf	12,6	0,472	0,308	9,20
R1 (client)	R2 (server)	1000	600	iperf	19,6	0,320	0,134	26,60
R2	R1	1000	800	NTTCP	234	8	0,285	-8,50
R1 (client)	R2 (server)	1000	800	iperf	10,5	0,262	0,193	0,70
R1 (client)	R2 (server)	1000	800	iperf	55	0,408	0,060	14,00

Table 6.2: Comparison between NTTCP and Iperf.

to 234 seconds when we have around 800 Kbits/s. While with Iperf, the duration varies from 10 to 55 seconds when we have no restrictions, and we have around only 10 seconds when the duration of the measurement is limited.

Regarding to the intrusive traffic that each tool injects, we can also see how the NTTCP in all cases sends 8 Mbytes of data, while Iperf adapts to the conditions of the network, and in any case it exceeds 2 Mbytes.

Finally, in the last column of the table we can see that in this case the NTTCP gets better results than Iperf. These results are quite logical given that the algorithms used to make measurements are more accurate and their duration are longer. However, the results with Iperf with the limitation of 10 seconds in any case have an error higher than a 27%.

However, our interest is to estimate the available bandwidth to distribute chunks, and also we want it to be fast so you can start downloading the stream as soon as possible. So with the values obtained for the comparison we opted for Iperf as the difference in duration of the measurements is not compensated by the precision offered by NTTCP. Moreover, the fact that NTTCP is more intrusive than Iperf is a fact to be considered for the second.

In any case, we lose the precision which is compensated with the network adaptation mechanism, and therefore, when load redistribution is necessary it is carried out once the stream download has started.

6.4 Evaluation Scenarios

In this section, we present emulation results by which NeuroCast characteristics are evaluated.

6.4.1 General Scenario Description

Most of the emulation carried out is based on the scenario described in Figure 6.11. We have varied the initial parameters of each machine and each network according to the scenario we wanted to emulate. Anyway, this virtual network topology in which the *Host* serves the video to a directly connected machine $E0$, and the latter sends the stream to 3 machines ($E1$, $E2$ and $E3$), agrees quite well with a simplified P2P network in the field.

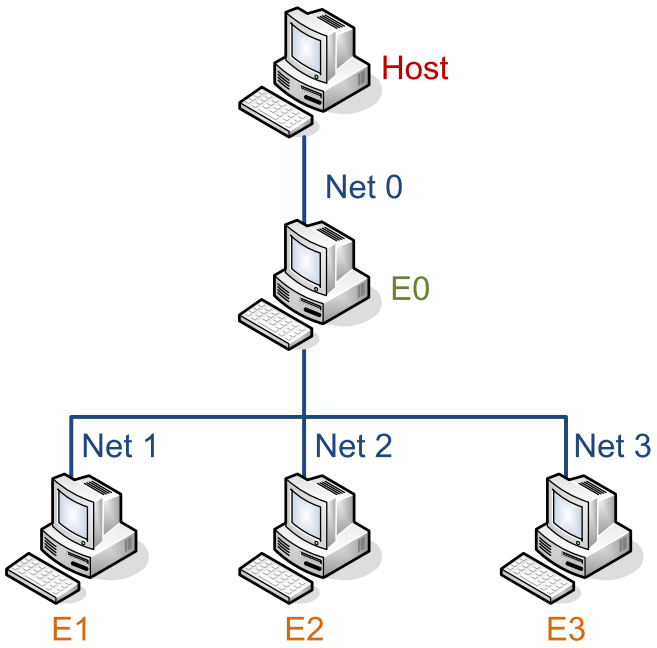


Figure 6.11: Network topology used for the tests.

The general parameters used during the tests are:

Video

- Encapsulation method: `MPEG PS`.

- Video Codec: `MP2V`.
- Bit rate: 512 Kbps.
- Output: `HTTP`, port 8080.

Virtual Network

Table 6.3 shows the main characteristics of the virtual network created using VNUML.

	Net0	**Net1**	**Net2**	**Net3**
Typology	lan	ppp	ppp	ppp
Bandwidth	2 Mbps	1 Mbps	1,5 Mbps	750 Kbps

Table 6.3: Parameters of the virtual network created with VNUML.

Virtual Machines

Table 6.4 shows the main characteristics of the virtual machines emulated using VNUML. These parameters are set up through the configuration file of NeuroCast.

	Host	**E0**	**E1**	**E2**	**E4**
maxRelays	1	4	4	4	4
peerRp	2 Mbps	1 Mbps	1 Mbps	700 Kbps	1 Mbps
minRp	100 Kbps	100 Kbps	100 Kbps	100 Kbps	100 Kbps
delta	0	10	10	10	15
factor_sep_pack	1	1.3	1.3	1.3	1.3
delay_margin	0	4	4	4	3
buffer_sep_pack	0	30	30	30	16
Rl	100 Kbps	100 Kbps	100 Kbps	100 Kbps	100 Kbps
Ru	2 Mbps	2 Mbps	3 Mbps	3 Mbps	3 Mbps

Table 6.4: Initial Parameters of the virtual machines.

In the following sections, the NeuroCast application is evalutated in the described scenario varying the behavior of the peers.

6.4.2 NeuroCast Performance Evaluation

In this section we will show how the application performs during its normal execution. In the following we described the different phases of the performed experiment in order to run and analyze NeuroCast.

1. **Booting the Server Stream and NeuroCast at the *Host*.**

   Firstly the server streams is booted (in this case, VLC (14)) and starts to deliver packets on port 8080 at a rate of 512 Kbps. Figure 6.12 shows a snapshot of the configuration parameters that where used to set up the server stream in the VLC 0.7.x.

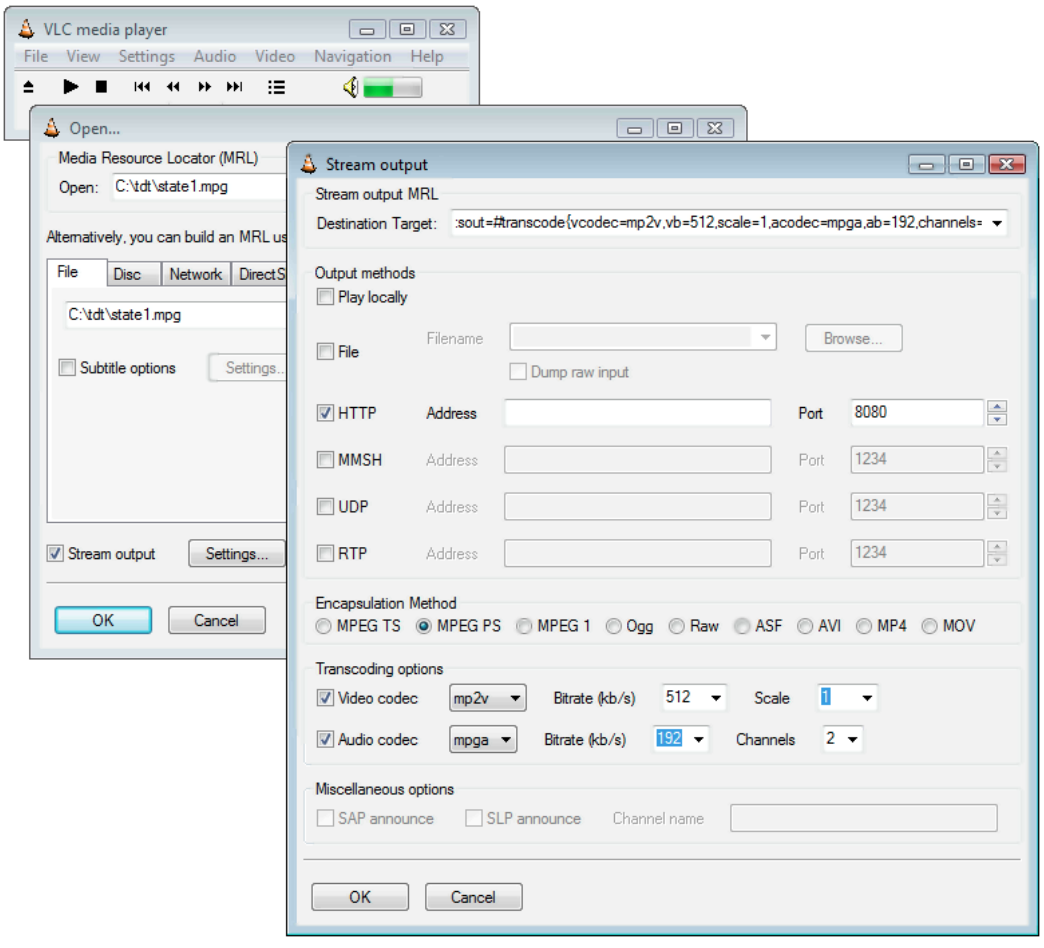


Figure 6.12: VLC configuration example snapshot.

Then, we run NeuroCast in the *Host* and we load its GUI through a browser. To do this we have to access it at `http://<Host Ip Address>:7144`. After that, we create a new channel through which the server stream will serve the video. We click in the "Broadcast" section and create a new relay based primarily on two parameters: the URL of the VLC server and the rate at which the video is transmitted (see Figure 6.13).

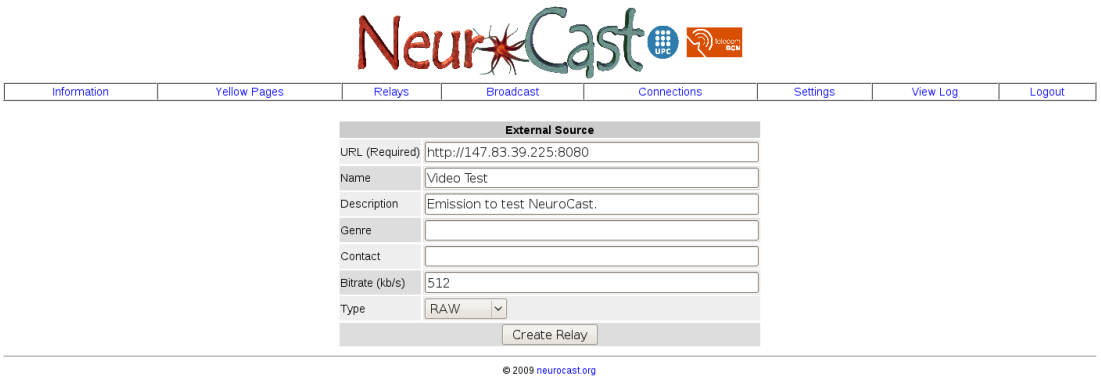


Figure 6.13: New channel creation through NeuroCast web interface.

Once created, the "relays" section (see Figure 6.14) shows that we are receiving the channel and we check that the IP address of the source (*Source*1) is `0.0.0.0`. This is normal because it is our own machine which serves the video, so the NeuroCast is not getting it from any external source.

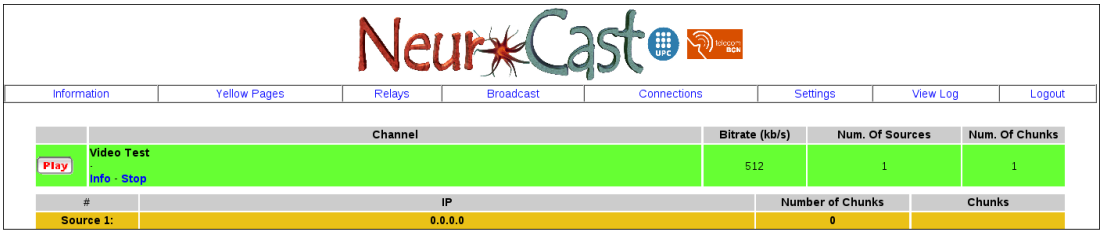


Figure 6.14: NeuroCast *Relays* Web Tab.

From the link to which the "Play" button points, we can get the identifier of the created channel, and the rest of the machines may request the channel to display the video. This link would look like:

`http://127.0.0.1:7144/stream/206FFDCE9822031D514B5A35334649CC.mpg`

Then the NeuroCast waits for new users to ask for the channel.

2. **Booting NeuroCast at machine $E0$.**

   From the tracker IP address (`10.0.0.1`) and the channel identifier where the desired video is being transmitted (`206FFDCE9822031D514B5A35334649CC`), we run NeuroCast which starts the negotiation process of the channel and then downloads the video:

   ```
   peercast-l & a.log

   lynx http://127.0.0.1:7144/pls/206FFDCE9822031D514B5A35334649CC?tip=10.0.0.1:7144
   ```

   The first instruction simply starts the NeuroCast application in background mode and saves the log (messages that the program leaves during execution) to the `a.log` file.

   With the second instruction we achieve to communicate with the NeuroCast process running on our machine through port 7144 . We ask for the playlist associated to the specified channel. Moreover, we provide the IP address of the tracker that can provide us the list of hits for the specified channel.

   As the tracker we ask for the channel has no hits sharing the video at that instant, it creates a new distribution playlist of the channel and adds to it our IP address (`10.0.0.2`). Then, as we only have one source (the tracker itself) and it has been indicated that it can act as a relay, we start downloading the full video from the tracker.

3. **Booting NeuroCast at machine $E1$.**

   The same way we have configured machine $E0$, we configure machine $E1$. Therefore, we execute the NeuroCast application and ask for the channel to the tracker. In this case, the tracker has a list of hits from which the channel can be downloaded. In addition, it adds our IP address (`10.0.1.2`) to the playlist with those hits. At the same time, the tracker notifies $E1$ that it has already reached its maximum number of retransmissions, and thus it cannot serve the video. Thus, once we analyzed the list, there is only a possible hit (`10.0.0.2`). So, in this case $E1$ proceeds to download the full video from machine $E0$.

4. **Booting NeuroCast at machine $E2$.**

   Following the same procedure than has been carried out in the two previous machines, $E2$ gets the playlist from the tracker. Unlike the previous machines where we only had one possible source, now $E2$ has two possible hits: machines $E1$ and $E0$.

   At this point, the selection peer algorithm 5.4.2 starts in order to define how to distribute the packets among the sources. First, the Iperf tool measures the available bandwidth and the link losses for each hit. In addition, we also get the `Rp` of each hit (see table 6.5).

	Rp	Measured BW	Measured Link Losses
E0	1 Mbps	1048779,78 bps	0
E1	1 Mbps	1048749,82 bps	0

Table 6.5: Measurements of the parameters of each hit.

   With these values it is clear that the distribution is symmetric, as both machines have almost identical parameters.

   From the configuration file value of $Delta = 10$, the algorithm decides to divide the stream into 12 parts, and from these, it gives 6 parts to each machine (see Table 6.6).

Subchannel	IP address	Chunks
0	10.0.0.2	0,2,4,6,8,10
1	10.0.1.2	1,3,5,7,9,11

Table 6.6: Chunks initial distribution between the channels.

   Once determined the load distribution, $E2$ notifies both machines which will serve the video about the number of chunks in which they divide the stream and which part has to distribute each machine. To do this, $E2$ uses the PCP protocol implemented in NeuroCast for communication between clients.

   Once completed the exchange of messages that we have seen in section 4.2.2, the hit splits the stream into the indicated number of parts and starts to transmit the requested chunks.

5. **Booting NeuroCast at machine *E*3.**

In this case, we get three possible sources once we have downloaded the list of hits from the tracker. Using the same methodology than in the previous case, we measure the interested parameters with Iperf (see Table 6.7)

	Rp	Measured BW	Measured Link Losses
E0	1 Mbps	857541,13 bps	0
E1	1 Mbps	744729,24 bps	9,24
E2	700 Kbps	739973,06 bps	14,96

Table 6.7: Measurements of the parameters of each hit.

Since both the throughput offered by hits and the link losses are different, the load distribution will not be symmetrical. Based on $Delta = 15$, the stream is divided into 18 different chunks (see Table 6.8).

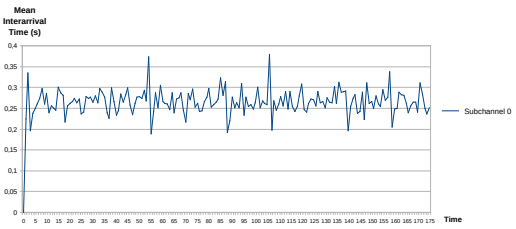
Subchannel	IP address	Chunks
0	10.0.0.2	0,2,4,6,8,10,
1	10.0.1.2	1,3,5,7,14,16
2	10.0.2.2	9,11,13,15,17

Table 6.8: Chunks initial distribution between the channels.

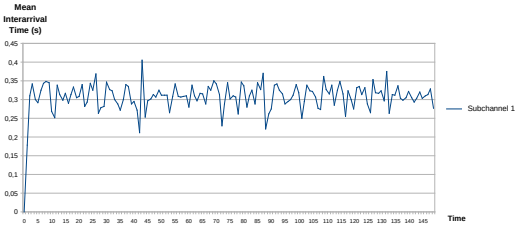
In the following graphs 6.15(a), 6.15(b) and 6.15(c), we show the arrival rates of packets for each subchannel. This arrival rate corresponds to the average value obtained from the arrival of 16 packets, and it is the same value used to determine if you exceed a certain threshold and thus begin the process of packet reallocation.

The expected values are:

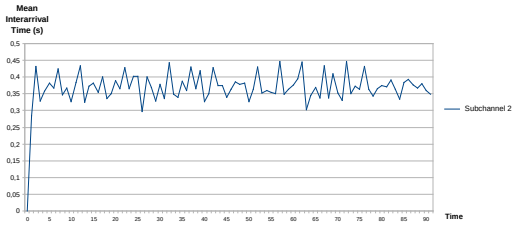
- *Subchannel 0*: We provide 7 chunks of 18. Thus, with a rate of 512 kbytes/s using the formula 5.5, we obtain the following threshold: 0.418 seconds. Since the parameter `factor_sep_pack` is equal to 1.3, we obtain that the expected mean time between arrivals is 0.321 seconds.



(a) Subchannel 0.



(b) Subchannel 1.



(c) Subchannel 2.

Figure 6.15: Evolution of the time between packet arrivals.

- *Subchannel 1*: We provide 6 chunks of 18. Thus, with a rate of 512 kbytes/s using the formula 5.5, we obtain the following threshold: 0.487 seconds. Since the parameter `factor_sep_pack` is equal to 1.3, we obtain that the expected mean time between arrivals is 0.375 seconds.
- *Subchannel 2*: We provide 6 chunks of 18. Thus, with a rate of 512 kbytes/s using the formula 5.5, we obtain the following threshold: 0.585 seconds. Since the parameter `factor_sep_pack` is equal to 1.3, we obtain that the expected mean time between arrivals is 0.450 seconds

.

In the graph 6.16 we can see the different obtained values of the mean time between arrivals for each subchannel. The values were obtained during the normal execution of the program: no losses or interfering traffic.

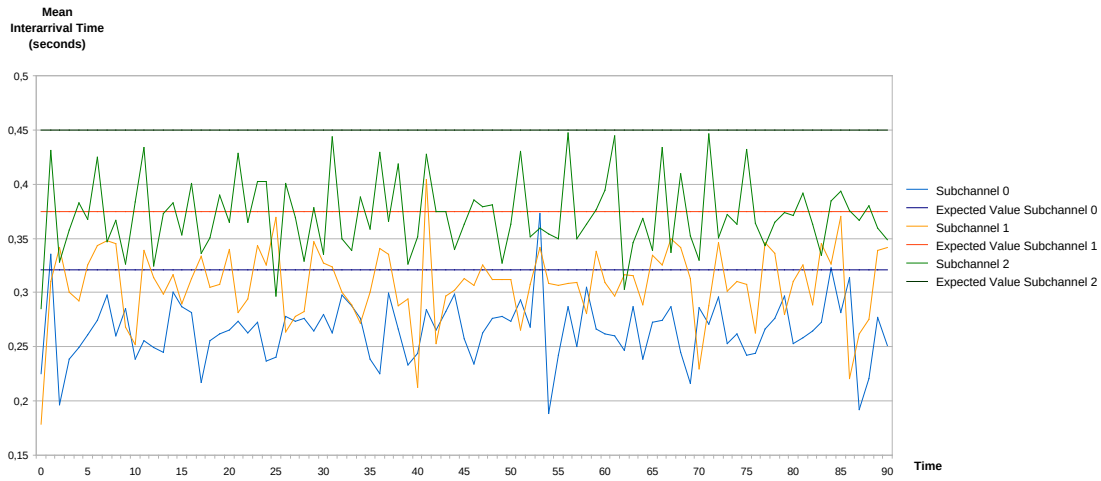


Figure 6.16: Evolution of the mean time between packet arrivals and the expected value for each subchannel.

6.4.2.1 Peer Leaving

In the case downloading a video from more than one source, and one of these sources leaves the network, then it redistributes the load and continues downloading the subchannels from other sources. The process that takes place consists in:

1. Check if there is a free peer in our list of hits. That is, a hit from which we are neither downloading any chunk will nor sending any part of the stream.
2. Request a a new list of hits to the tracker and check again if there are any free hit.
3. Try to download the "missing" chunks from one of the hits that we are already using.

In the latter case it requires a reallocation of subchannels. Depending on the situation, there must be completed a series of changes:

- If the client is downloading more than one subchannels from the same hit, then it groups the chunks in the same subchannel.
- If a subchannel that is not at the end of the list is deleted, then this position is filled with the last subchannel, so there are no empty spaces in the list.

- If a subchannel downloads all the chunks into which the stream is divided, then the value of `numChunks` is set to 1 and the stream is no longer divided into different parts.

Next, we show an example where some of these situations are discussed. Initially we have the following configuration (see Tables 6.9 and 6.10):

Hit List				
Hit	0 (tracker)	1	2	3
IP	10.0.0.1	10.0.0.2	10.0.1.2	10.0.2.2

Table 6.9: Initial List of Hits.

Subchannels List			
	IP address	**Number of Chunks**	**Chunks**
Subchannel 0	10.0.0.2	6	0 2 4 6 8 10
Subchannel 1	10.0.1.2	3	1 3 5
Subchannel 2	10.0.2.2	3	7 9 11

Table 6.10: Initial Chunks distribution among the 3 subchannels.

After some time, source `10.0.0.2` leaves the network. We check that there is no-free hits in our hit list, so we request a new hit list to the tracker (see Table 6.11)

Hit List				
Hit	0 (tracker)	1	2	3
IP	10.0.0.1	10.0.1.2	10.0.2.2	10.0.3.2

Table 6.11: New List of Hits.

As the only new addition to the list of hits is our own IP address, we have only the possibility of reusing the hits which are already transmitting the stream. In this example, as shown in Table 6.12 we selected the 10.0.1.2. The configuration we have is the following:

And, finally we reallocate the subchannels of the list (see table 6.13).

Subchannels List			
	IP address	**Number of Chunks**	**Chunks**
Subchannel 0	~~10.0.0.2~~	~~6~~	~~0 2 4 6 8 10~~
Subchannel 1	10.0.1.2	3 + 6	1 3 5 + 0 2 4 6 8 10
Subchannel 2	10.0.2.2	3	7 9 11

Table 6.12: New Chunks distribution among the 3 subchannels.

Subchannels List			
	IP address	**Number of Chunks**	**Chunks**
Subchannel 0	10.0.2.2	3	7 9 11
Subchannel 1	10.0.1.2	9	0 1 2 3 4 5 6 8 10

Table 6.13: Final Chunks distribution among the 2 subchannels.

6.4.2.2 Impact of Traffic Interference

Continuing with the network topology of figure 6.11, we start from the situation where we have 4 machines running NeuroCast and playing a video: *Host*, $E0$, $E1$ and $E2$.

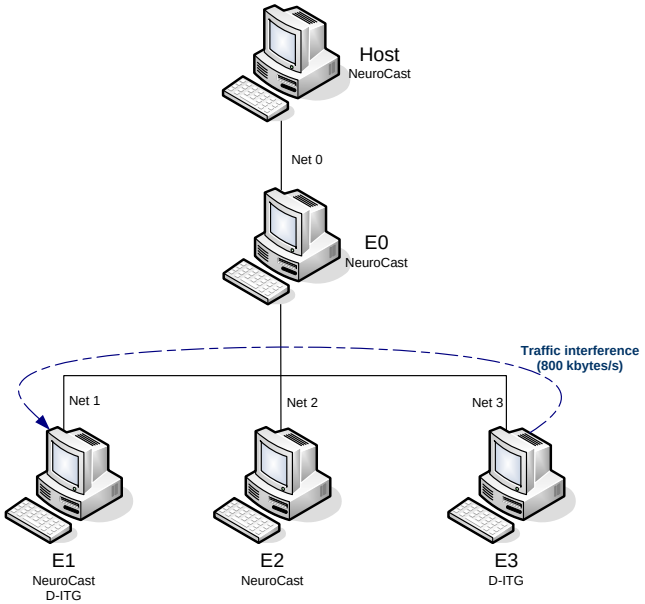
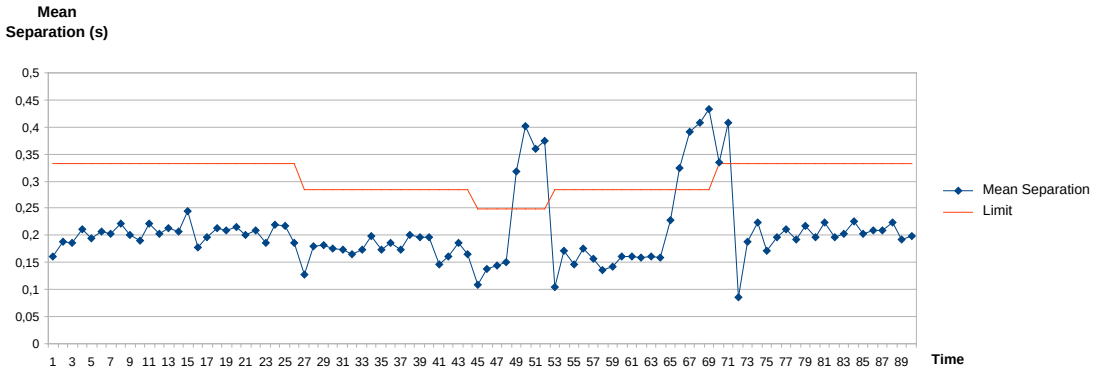
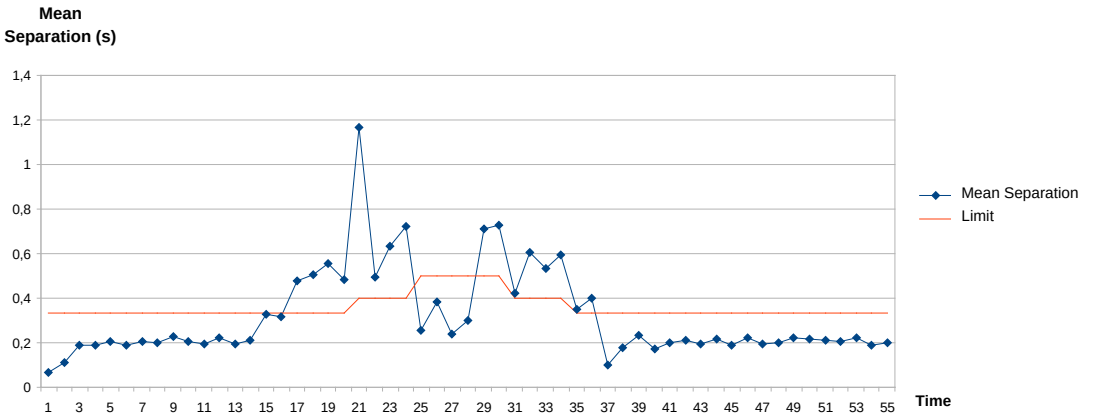


Figure 6.17: Scenario with traffic interference in Net1.

As we have seen in previous cases, in a "stable" situation machine $E0$ downloads the stream from $Host$, machine $E0$ downloads it from $E1$, and finally $E2$ downloads it from $E0$ and $E1$ simultaneously.



(a) Subchannel 0



(b) Subchannel 1.

Figure 6.18: Mean time between packets arrival evolution.

In this section, we focus on studying the performance of machine $E2$ while varying the network conditions. To achieve that environment, we introduce traffic interference in link $Net1$ using D-ITG in $E3$. This is the scenario shown in figure 6.17.

Prior to the injection the traffic interference, $E2$ is downloading 6 chunks from each source as the conditions of $Net1$ and $Net2$ are virtually identical. However, when we introduce a traffic of 800 kbytes/s in $Net1$ link, this balance is broken. Thus, chunks coming from the $E1$ are affected, and consequently, the

time between packet arrivals is increased. Precisely this is reflected in the graphs of figure 6.18(a) and 6.18(b), where we can see the mean time between packet arrivals, calculated from groups of 30 arrivals (this value is determined by the parameter `buffer_sep_packets` in the configuration file of NeuroCast).

Subchannel 0 provides us the flow of data coming from $E0$ and subchannel 1 the one coming from $E1$.

Figure 6.18 shows, apart from the mean time between packet arrivals, the theoretical limit that the program uses to determine when we need a redistribution of chunks. It is important to note that the redistribution occurs when that limit is exceeded 3 times, because this is the value of the parameter `delay_margin` defined in the NeuroCast configuration file running in $E2$.

6.4.2.3 Redistribution Time

As we have seen previously, some parameters in the configuration file directly affect the behavior of the application during variations of the network conditions. The following experiments have been carried out, varying two parameters such as `delta` and `buffer_sep_packets`, and we have observed in each case the time needed to redistribute the chunks.

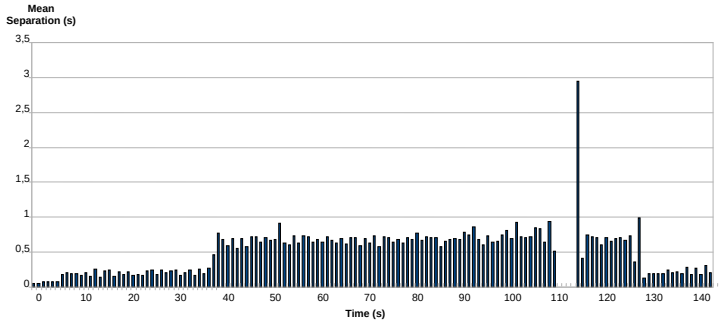
To force the redistributions, we use NetEm to change the link capacity of one of the hits that retransmits the stream. Thus we have switched from the 1000 Kbps available at the beginning of the session, to 112 Kbps.

	Test1	**Test2**	**Test3**	**Test4**	**Test5**
Delta	20	5	20	20	5
`buffer_sep_packets`	20	20	50	5	5
Time to redistribution	90s	86s	173s	31s	31s

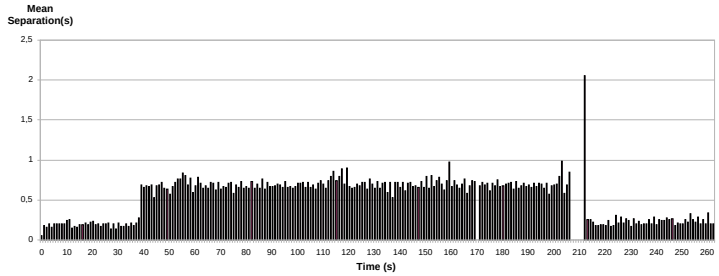
Table 6.14: NeuroCast performance with a traffic interference of 90% of the link capacity.

Analyzing the results of table 6.14 we see that the parameter `Delta`, i.e. the number of chunks into which the stream is divided does not affect significantly the obtained time, while the parameter `buffer_sep_packets` is determinant.

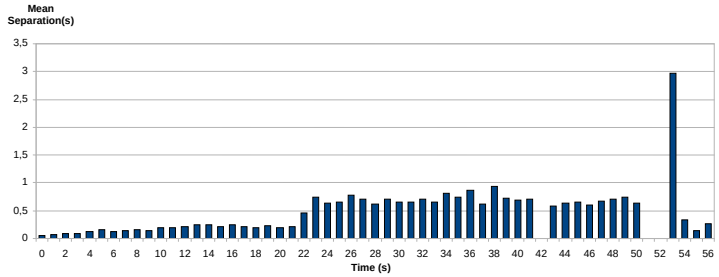
In figures 6.19(a), 6.19(b) and 6.19(c), we can see the evolution of the mean separation between packets arrivals of experiments 1, 3 and 5 graphically.



(a) Experiment 1.



(b) Experiment 3.



(c) Experiment 5.

Figure 6.19: Packets arrival evolution.

6.4.3 `RawData` buffer Evolution

As we have seen in previous sections, one of the core elements of the application is the `RawData` buffer. The `Channel` and `Servent` classes access this buffer to read, write or check any packet at any time. As we have seen, its capacity is limited to 64 packets. To get a better idea, this means that with a data flow of 1024 Kbps and 8192 bytes packets the buffer is completely refreshed every 4 seconds.

By means of two parameters such as `lastPos` (the buffer position to the

last row available packet) and `nextSendPos[i]` (the buffer position to the next packet we have to send to subchannels i), we can analyze the usage of the buffer during the session. Based on the value of the subtraction of the two parameters, 3 scenarios can be distinguished :

- Difference next 0: optimal performance. It means that the servent sending the packets uses the newly ones arrived to the `RawData` buffer. Thus we do not accumulate packets in the buffer.
- Difference increasing up to 63: Problems with the transmission. That means that we have pending packets to be sent. We have as many packets as the result of the subtraction, and therefore we are going to "saturate" the buffer. The maximum value that the difference can reach is 63. Once the threshold is exceeded, some packets will be sent out.
- Difference decreasing up to -63: Problem with the reception. That means that we have pending packet to be written in the buffer. We have as many packets as the result of the subtraction, as the servent `T_Relay` is requesting newer packets. As in the latter case, when you exceed the threshold of -63, the application will start to lose packets that can not be sent to other users or to the player itself.

6.4.3.1 Basic scenario: 1 Host & 1 peer

To observe the evolution of the buffer we have emulated a scenario where we have a host that sends the stream to a single peer. In this situation, packets arrive to the host in an orderly way and without delay, as we are directly connected to the stream server. Therefore, the only problem we could have regards to the packets transmission to hit $E1$.

The scenario emulated is the one shown in figure 6.20.

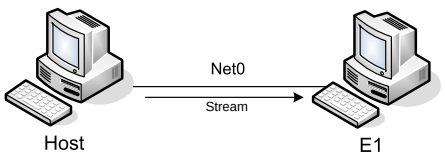


Figure 6.20: Basic Scenario: 1 Host and 1 Peer.

If we analyze the buffer `rawData` at the NeuroCast located in the *Host*, it looks like the one represented in figure 6.21. Analyzing figure 6.21, we note 3 basic elements that exists at the *Host*: the buffer `rawData`, the servent who sends packets to the machine $E1$ and the Channel Stream that reads packets from the stream server to write them to the buffer.

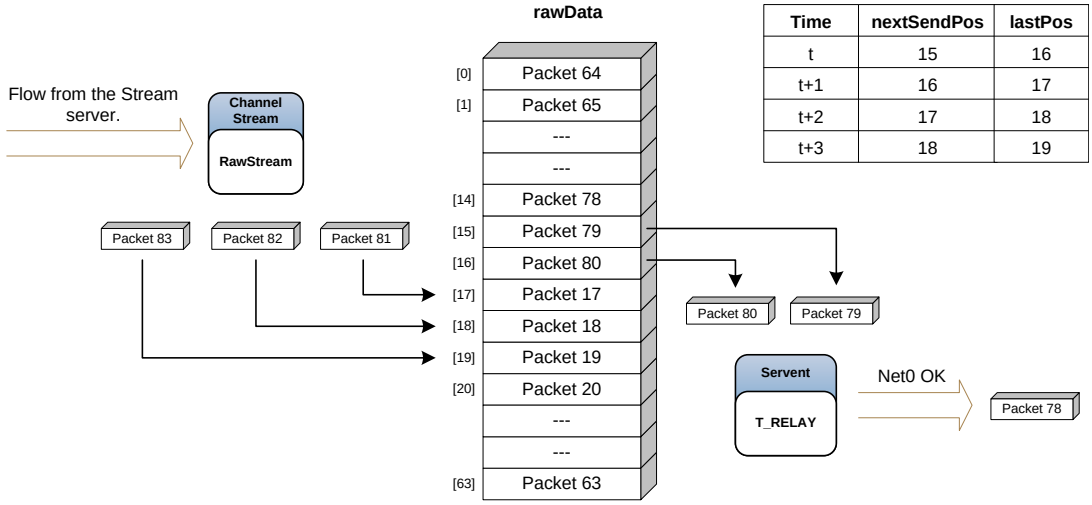


Figure 6.21: `rawData` buffer during the arrival of new packets and without transmission problems.

In the table in figure 6.21, we see the evolution of both parameters mentioned above: the `nextSendPos` and `lastPos`. At the beginning, we assume that we have just written the 80 packet at position 16 of the buffer , and the servent has just sent the packet 78 located at position 14. So that, next packet that will be sent at (instant $t + 1$) is 79 (position 15), and `lastPos` will point at packet 80 (position 6).

Then, to see how NeuroCast evolves when network conditions change, we saturate link $Net0$ so that the machine $E1$ cannot receive packets from the *Host*. The new situation is represented in figure 6.22.

Regarding to the packets arrival to the system, the situation remains the same, so the `lastPos` index evolves as in the previous case. The final packet which is written is number 80 at the position 16 and the following packets that will be added are 81, 82, 83... The problem is with the output, because once the servent ceases to send packets to the network, the index `nextSendPos` remains constant,

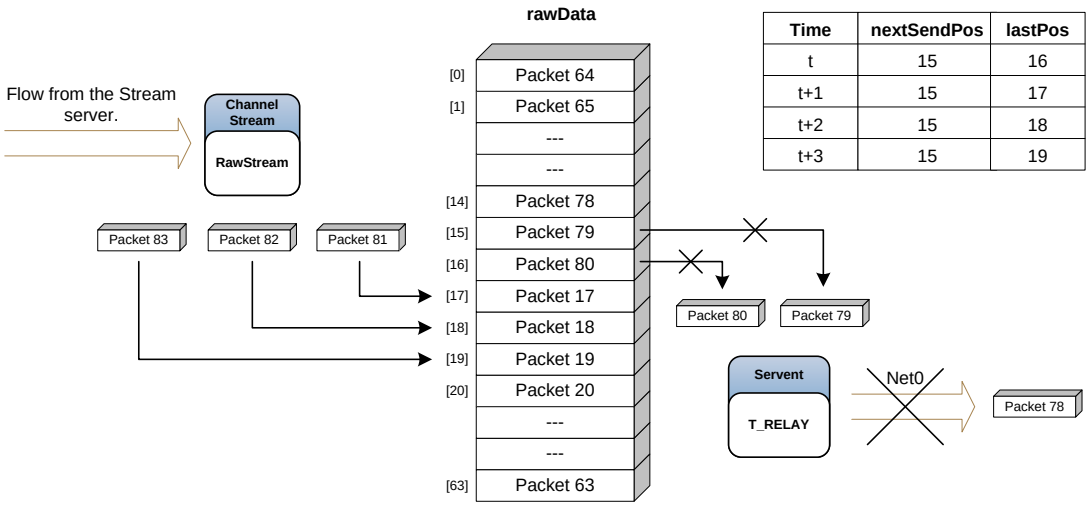


Figure 6.22: `rawData` buffer when transmission problems occur.

in the case of the example of the figure 6.22 at position 15. In this situation the number of pending packets to be sent to the buffer is progressively increased.

Graph in figure 6.23 shows the evolution of the difference between the two parameters under study, namely, the number of packets that are queued to be sent.

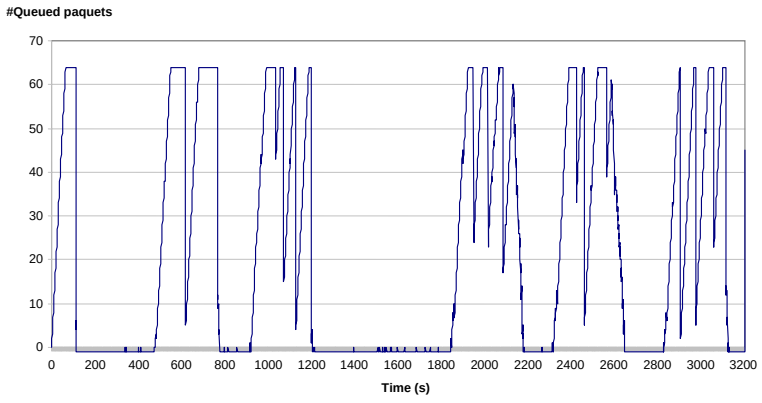


Figure 6.23: `rawData` buffer evolution when transmission problems occur.

In order to saturate the link, we send an intermittent traffic to $E1$ through $Net0$, so it takes all the bandwidth for 30 seconds. Thus the $Host$ can only send a packet to $E1$ sparingly, as the channel remains almost always busy.

The graph in figure 6.23 shows that at the beginning of the session as well as with intrusive traffic, the buffer queue is increased rapidly until 64 packets, the limit of the `rawData` buffer. The growth in these cases is constant in the 6 "bursts" that are represented, and as shown in Figure 6.23 the queue increases packet by packet as the `ChannelStream` introduces new ones.

However, we observed that the performance when retrieving packets pending to be sent is not the expected one. In some cases, we evolve from having 64 packets in the queue to have 0 packets, or from 64 packet to only 7 or 8 packets. The ideal case would consist in, once the channel is released, attempting to send out the last 64 packets that have been written into the buffer. For this reason the code has been modified to correct this situation, so performing again the emulation we have obtained the results graphed in figure 6.24.

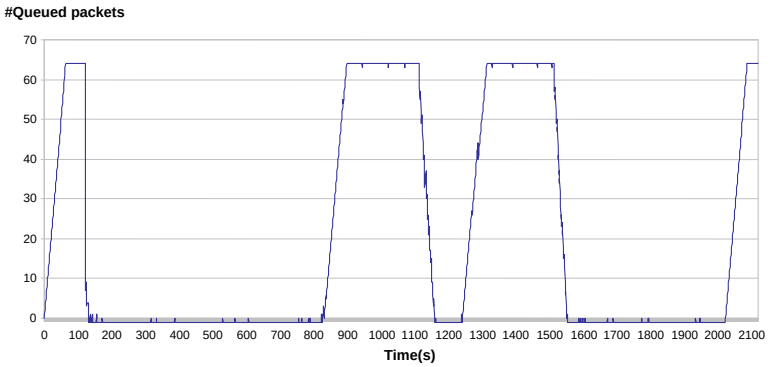


Figure 6.24: `rawData` buffer evolution when transmission problems occur (improved version).

The origin of the problem we encountered in the first emulation was caused by the servent who forwards the stream to the other hits. Specifically, when it looks for an old packet in the buffer, i.e., one that has been overwritten by the newest. In this case, instead of the desired packet, it will return the one placed in its position. The described situation is presented in figure 6.25.

If we analyze the buffer on the right of Figure 6.25 (when the application has spent some time without sending packets), we see that the request to the servent to get the packet at the position 16 will return packet 784. At this instant, the buffer queue becomes 49 $(833 - 784)$.

Once received the new packet, the servent updates the values of its `streamPos`

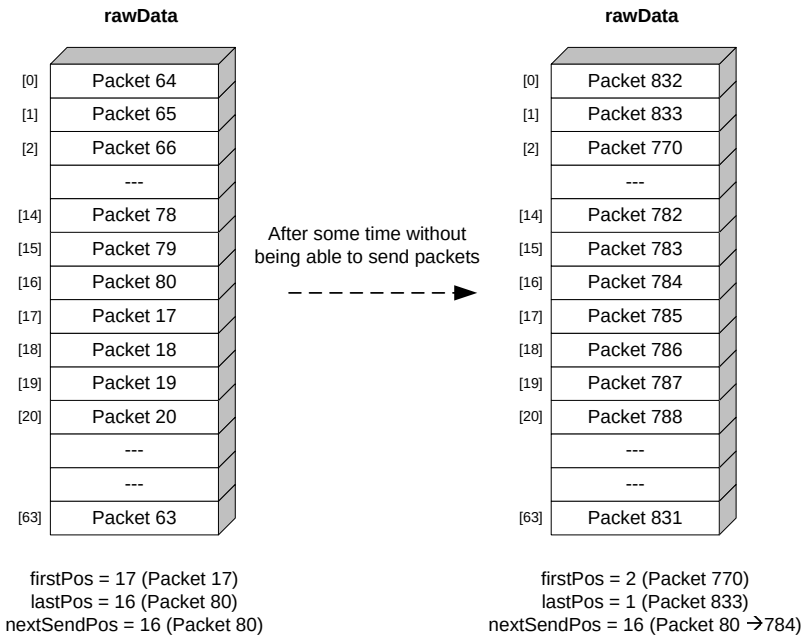


Figure 6.25: `rawData` buffer during the arrival of new packets and it is unable to send.

and `nextSendPos` according to the packet sent to the machine $E1$. It is important to note that in this case we lose all the packets that were in the buffer from the `firstPoint` (770) until the one we have just sent (784), 14 packets in total.

To improve the situation we have changed the code so that if the servent requests a packet located at any position previous to `firstPoint` in the `rawData` buffer, NeuroCast tries to send the first available packet regardless what the buffer had returned. This minimizes the losses we get, as we can send the last 63 packets written into the buffer.

Following the parameters of example of figure 6.25, the packets that we will send through the servent before and after the modification are shown in figure 6.26. Thus, with this code modification the servent can retransmit packets from 770 and so on, and this way, it minimizes the impact of possible packet losses during the playback of the stream at $E1$.

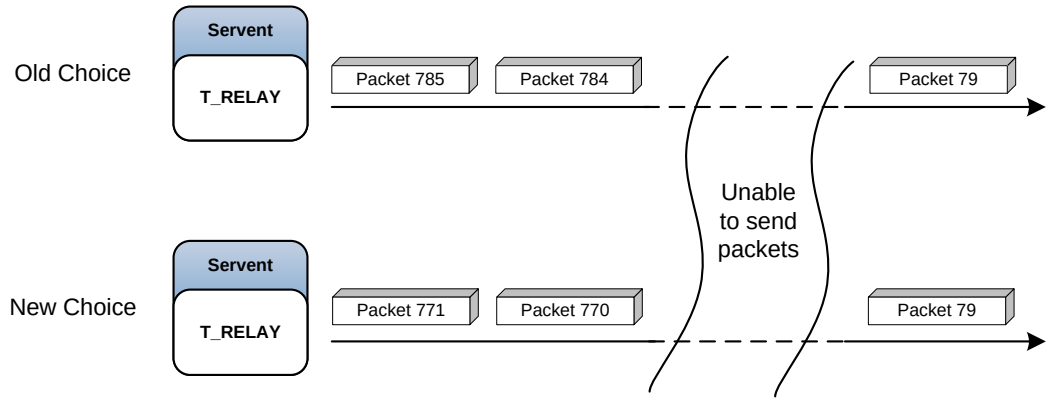


Figure 6.26: `rawData` buffer during the arrival of new packets and it is unable to send.

6.4.3.2 P2P Scenario: 1 Host & 3 Peers

In this section we studied the evolution of the buffer `rawData` in a more complex scenario. We consider the scenario previously presented in Figure 6.11. In this case we are interested in observing the performance of the `rawData` buffer at *Host* and at machine $E2$. Thus we consider a case where we have a machine that receives the stream from two different sources ($E0$ and $E1$), and simultaneously send the video to another machine ($E3$).

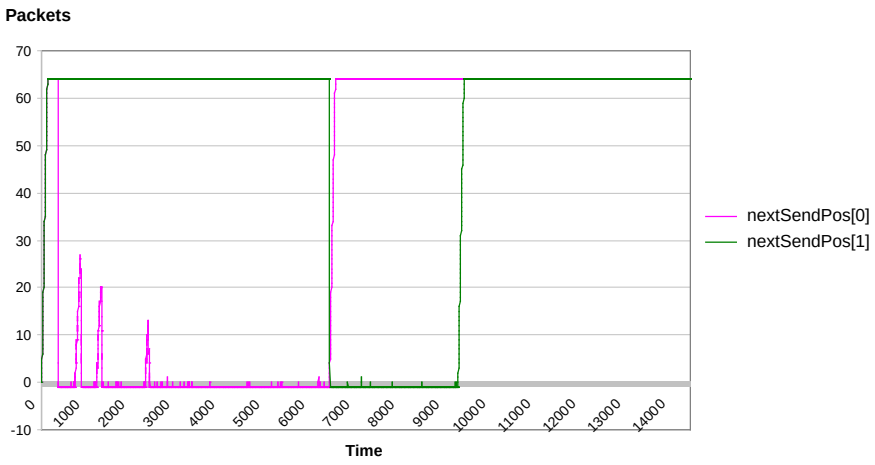


Figure 6.27: `rawData` buffer evolution at *Host*.

The sequence we followed was this: we booted (in this order) *Host*, $E0$, $E1$,

$E2$ and $E3$. Then, we have injected a traffic interference in machine $E1$ during approximately 10 seconds. After that, machine $E0$ ended its session. A few moments after, $E3$, $E2$, $E1$ and $Host$ also ended the session.

In the figure 6.27 we can see the evolution of the $Host$ buffer plotting the parameter `nextSendPos[i]`. The two represented variables give an idea of what we have in queue to be sent. In a first case regarding the first hit in the stream we serve: $E0$. In a second case respect the other machine $E1$ (once $E0$ has ended its session). As we have seen in section above, we have plotted the difference between the `lastPos` and `nextSendPos[i]` parameters.

Analyzing the results, we realize that, at the beginning, the buffer fills up quickly until it reaches 64 packets, and once $E0$ starts the session we send it the stream. Then, we observe 3 peaks, which in any case exceed 30 packets, and correspond to the joining of machines $E1$, $E2$ and $E3$ to the network. These are due to the calculation of the available bandwidth with the machine $E0$, causing a slight delay at the packets reception.

Approximately after 6500 s the buffer usage is measured respect to `nextSendPos[0]`, associated to the servent that sends packet to $E0$. But when we send the stream to $E1$ we use a new servent, so that the new reference parameter in this case is `nextSendPos[1]`.

On the other hand we can see the evolution of the `rawData` buffer of $E2$, represented in figure 6.28. In this case the analysis is more complex because in the buffer there are three different elements interacting:

- `writePos[0]`: related with the ChannelStream that writes packet in the buffer `rawData` coming from E0.
- `writePos[1]`: related with the ChannelStream that writes packet in the buffer `rawData` coming from E1.
- `nextSendPos`: related with the servant that sent chunks to $E3$.

The graph in figure 6.28 represents the difference of these three parameters with the `lastPos` value. We can see how at the beginning of the session the two sources start to send packets. Firstly, the subchannel 1 is slightly ahead of subchannel 0, but then the differences are stabilized with `lastPos` around 0 packets.

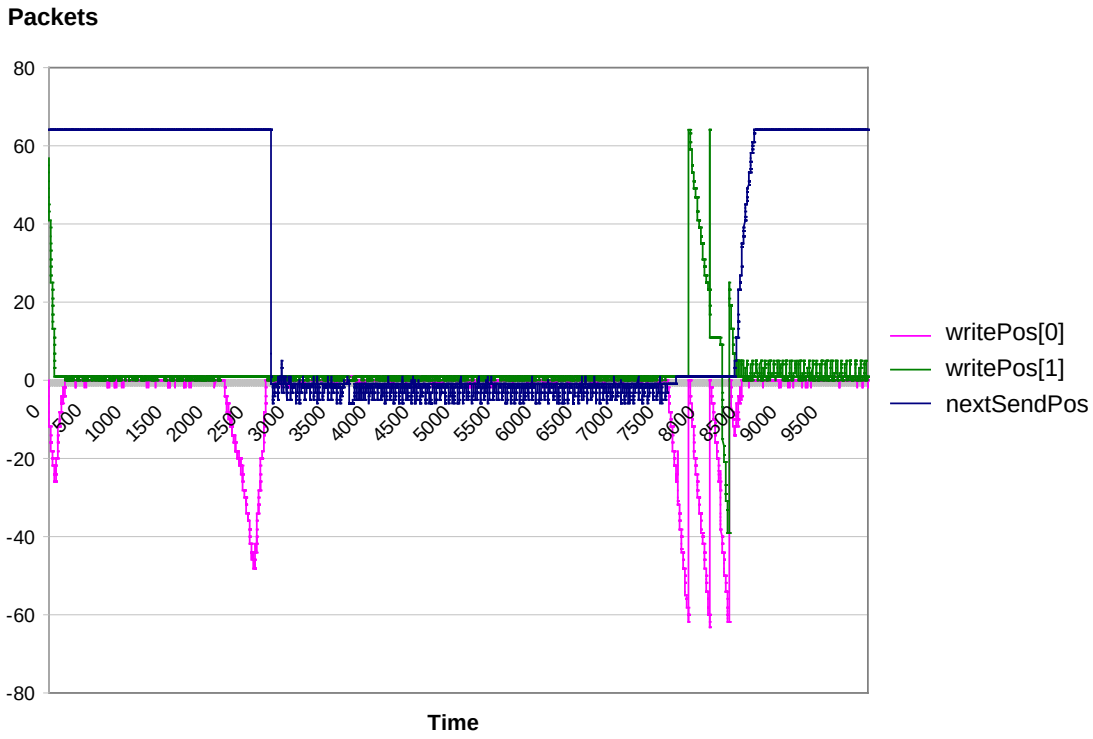


Figure 6.28: `rawData` buffer evolution at $E2$.

The first negative peak occurs after 2500 seconds due to slight delays caused by the joining of the machine $E3$. Some time later, we observe that the servent starts to take the necessary measures and the `nextSendPos` parameter is updated with the value of `lastPos`. From there, the three elements related to packet reading and writing continue to evolve at the same rate.

As we have discussed above, it has been introduced a traffic interference between $E1$ and our machine. This is why for a few seconds we do not receive packets form this machine, so the `lastPos` remains constant and consequently the `writePos[0]` decreases until -63 packets.

At this point, we start to overwrite packets in the buffer and `lastPos` is updated according to the last packet written by $E0$, so the roles are reversed, and now is the `writePos[1]` values 63, due to the packets still pending to be sent, while `writePos[0]` values again 0.

Approximately at instant 8500, the traffic interference disappears and the situation stabilizes with respect to both subchannels. We also note that the

`nextSendPos` related with $E0$ values 63, once it has ended the session and therefore we stop sending the stream.

6.4.3.3 Contribution of the Subchannels to the buffer

As we have seen in the previous section, from the `writePosi[subchID]` parameter we can analyze the contribution of each subchannel to the `rawData` buffer. Based on the scenario of Figure 6.17, we have analyzed the process of writing packets coming from $E0$ and $E1$ into the buffer, while changing the network conditions injecting interferent traffic between our machine and $E1$.

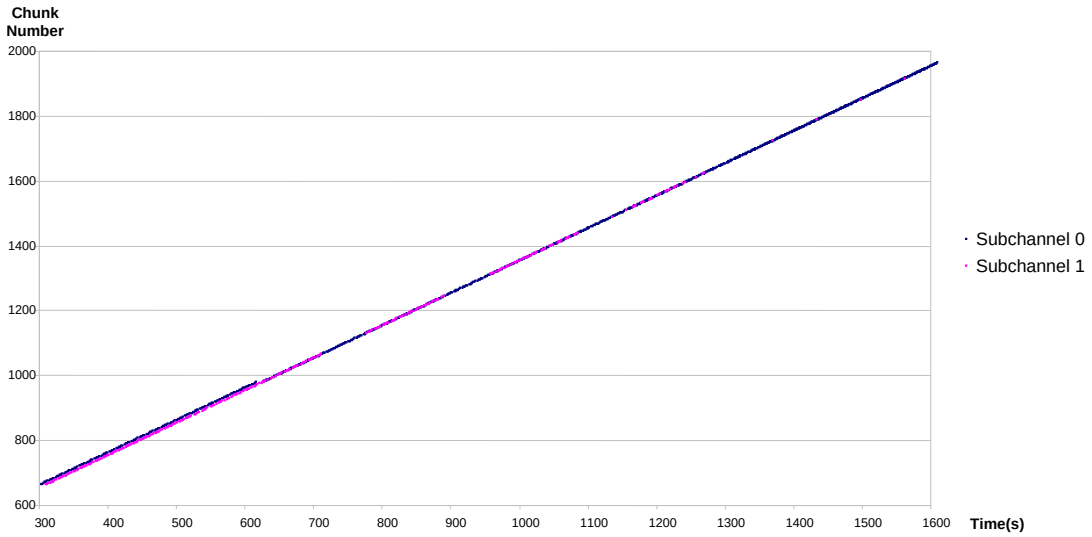


Figure 6.29: `rawData` buffer evolution at $E2$.

To give a more graphically idea of the importance that each hit has on the total number of sent packetst, we sorted the chunks provided by each one of the hits and we have obtained the graph in figure 6.29. Since the stream has been split into 12 chunks, we observe that at the beginning we download 6 chunks from each subchannel (in particular the even chunks from subchannel 0 and the odd ones from subchannel 1). But as time goes through, this distribution changes. Thus, since the arrival of the 515 packet, the subchannel 0 downloads 5 chunks and the subchannel 1 only 5 chunks. Hereafter, the difference increases until the arrival of the 1,250 packet, when the subchannel 0 downloads 11 chunks and the subchannel 1 only 1 chunk.

Chapter 7

NeuroCast Security Analysis

Security is an essential requirement and an important component of any communication and computing system. This is certainly true for a peer-to-peer system. In fact, security in P2P is an issue of particularly concern to many. With Napster's debut in 1999, P2P file sharing became immensely popular. The public's concern with information security has also increased tremendously in the past eight years. Web searches on keywords such as: P2P security news, P2P security concerns, and P2P security story; all return long lists of results with many news headlines. In a 2008 study, when asked to rank threats that p2p users believe would pose the largest problems over the next 12 months, bots and botnets again took the top spot, followed closely by DNS cache poisoning and BGP route hijacking (see Figure 7.1).

Recent studies (6; 60) focusing on information leakage and inadvertent disclosures through P2P networks found a surprising number of threats to both corporate and individual security, including a large number of searches targeted to uncover sensitive documents and data. In their study, P2P searches and files on three P2P networks Gnutella (3), Fast-Track, and eDonkey over a seven-week period (December 27, 2005, through February 13, 2006) were categorized. Sixteen thousand searches out of an estimated 800 million searches were found to be related to banking institutes. From relatively humble megabit beginnings in 2000, the largest Distributed Denial Of Service Attacks (DDoS) have now grown a hundredfold to break the 40 gigabit barrier in 2007. The growth in attack size continues to significantly outpace the corresponding increase in underlying trans-

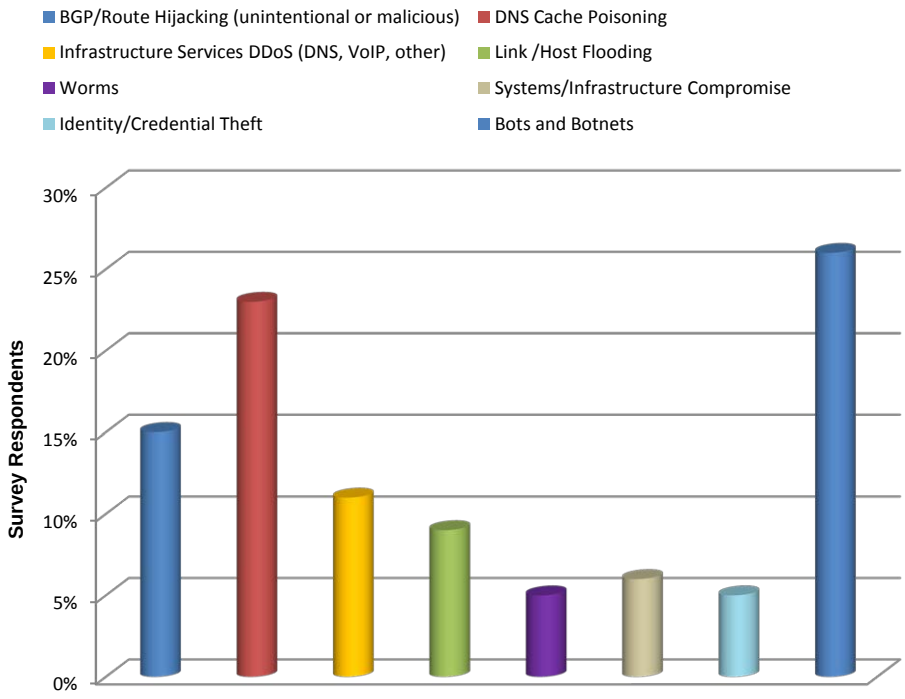


Figure 7.1: Most Concerning Threats. *Source: Arbor Networks, Inc.*

mission speed and ISP infrastructure investment. Figure 7.2 shows the yearly reported maximum attack size.

The increasing popularity of P2P applications, including P2P file sharing, P2P media streaming, P2PTV, and P2P gaming, could potentially instigate security risks that are more serious than those found in and beyond the much discussed content security and copyright issues. It could open up opportunities for cyber criminals to trawl the P2P network and steal or gather confidential information, to damage documents, content, or even devices, and to poison the network for criminal intents. It is believed that much of these vulnerabilities are rooted in the autonomous and decentralized nature of P2P systems and the relatively limited use of security techniques in existing P2P applications. In the following sections, we look at the security threats of P2P systems, discuss several existing solutions, and address some challenges in the P2P security domain.

This chapter scrutinizes the security issues of NeuroCast. We present a threat analysis and possible solutions to mitigate them. However, it was out of the scope of this thesis to implement NeuroCast as a secure p2p application. The goals of

this chapter are the following:

- highlighting the unique security requirements for P2P Video streaming applications.
- sketching approaches towards potential solutions, focusing on the specific characteristics of NeuroCast which can be used to develop decentralised security mechanisms.

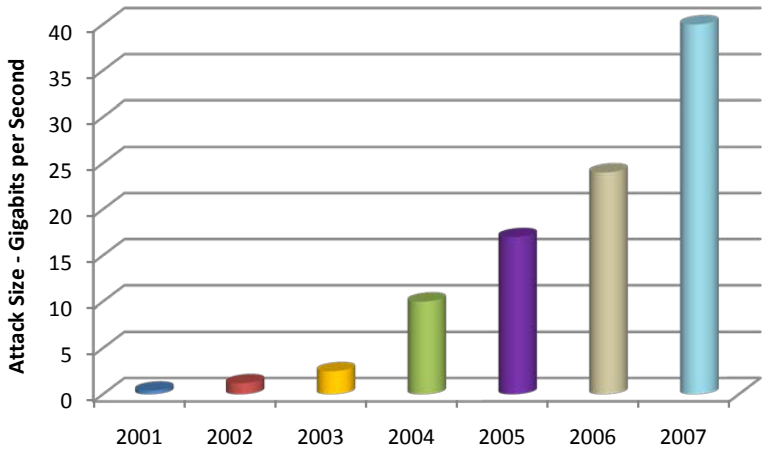


Figure 7.2: Time-Evolution of the Largest Attack Size. *Source: Arbor Networks, Inc.*

7.1 Security Risks and attacks

Due to the dynamic nature of P2P systems (e.g., nodes can join and leave the network frequently) and the lack of central entities on routing paths, many existing security solutions are not (or at least not directly) applicable to P2P networks. In principle, nodes in a P2P network must be regarded as not trustworthy and attacker nodes may drop, modify, or misroute messages.

The security of P2P systems has been studied by researchers, mostly considering file-sharing as the prototypical P2P application. Real-time communication applications, however, impose additional challenges which have not received a lot of attention in the literature. These specific challenges are due to the real-time

requirements, the type of data stored in the network, privacy considerations, and the risk of unsolicited communication. In this section we discuss these challenges in detail.

7.1.1 Authentication

Douceur (33) has shown that in any distributed system without some form of a central entity for identity assertion, attackers are able to create virtual, fake identities. Thus, decentralised authentication of participating nodes is a challenge for any P2P application. For live video streaming applications, the authentication of user-identities without a central authority which is trusted by all nodes in the system (e.g., a root certificate authority) is an additional problem. Authentication and encryption of real-time streams, on the other hand, is technically straightforward with SRTP (45). Thus, confidentiality of media streams is not a challenge but rather that proper mechanisms for authentication and key exchange exist.

We can distinguish different attacks regarding authetication: the sybil attacks and nodeId attacks.

7.1.1.1 Sybil attack

The Sybil attack (33) is a common threat in a system which allows user to generate their identities freely without any authentication mechanism. The P2P network which does not rely on a central authority to certify the participants is vulnerable for this attack. A hostile node may generate multiple pseudonymous identities for malicious purposes. For example, an attacker may create a large amount of identities to manipulate a segment of an overlay network. She can later eavesdrop and control the lookup messages traversed through this area, as well as perform many kinds of attacks.

The solution to the Sybil attack is to employ a trusted authority or reputation system to certify each node's identity. Nonetheless, this method takes away some advantages of a purely distributed architecture. Another defence is to make the process of joining the overlay more costly. For example, the overlay challenges participants with computational puzzles or requires a user to trade some resources in order to obtain an identity.

NeuroCast can suffer from this type of attack. Due to the decentralized solution that NeuroCast represents, it is not feasible to introduce a trusted authority. The most common way to deal with this attack in unstructured networks is the use of reputations mechanism to generate certain trust among nodes and hits. So, peers in NeuroCast could trust its hits depending on its reputation, and the selection algorithm could be improved introducing the reputation as another parameter to take into account in order to select the optimal set of hits.

7.1.1.2 Node Idetinfier Attacks

A `nodeId` attack can occur in an overlay when one node or a coalition of malicious nodes are able to obtain a specific `nodeId` (node identification) that maximizes its probability to appear in a victim peer's hit list. It could potentially compromise the integrity of a P2P network without the malicious party controlling a large fraction of nodes. For instance, one attacker may choose the `nodeId` of a particular peer to gain control of access to that object. `NodeId` attacks would give the attacker the ability to mediate the victim peer's access to the overlay or censor the object.

Although these type of attachs are typical of structured networks, it could also work in unstructures networks. For example, in NeuroCast a malicious node could fake its IP address in order to impersonate another peer with lower bandwidth, and thus send fake information to a resquesting node. This attack is also mitigated with a trusted authority or through reputation. However, the way NeuroCast identifies peers through their IP addresses could also be improve by using other kind of identifier difficulting the `nodeId` forgery.

7.1.2 Real-time Communication Availability Requirements

Real-time communication applications demand low delay and in the case of video additionally high, constant bandwidth. These characteristics make RTC applications more susceptible to attacks on availability than other P2P applications.

Attackers can severely degrade services simply by dropping or delaying messages transmitted over the P2P network, even if this gets eventually detected by the sender and messages are re-sent over a different path. The simplest case are so-called free-riders: nodes that consume services offered by other nodes but

which do not themselves contribute services to the P2P network. In addition to such passive attacks, active attackers can absorb even more resources from the P2P routing layer in vain by forwarding messages wrongly or with false content, potentially delaying the detection and retransmission of messages even further.

In the case of NeuroCast, not enough chunks might arrive at users' clients so that either the video cannot be played at all or with less quality (in the case of codecs which can compensate lost chunks by playing the video stream at a lower bitrate), degrading the overall quality of the application for users.

7.1.3 Routing Table Attacks

The routing attack is an action when a deceitful node, within the routing path, manipulates the lookup request. The bad node alters, drops, delays, or forwards the request to the wrong destination. This causes the service to become unavailable. The data storage in this thesis is based on an overlay network. In the overlay, routing attack is not relevant as the originator floods the request directly to all peers including the one who holds the desired information.

Routing table attacks include those attacks that manipulate routing table entries for malicious intent. Adversaries could take advantage of routing table updates to feed false updates and thus introduce faulty routing table entries to other peers. The effect cascades after subsequent updates. This could potentially trigger many problems in the P2P network

On the other hand, altouhg NeuroCast is an unstructured p2p netowrk, it could become vulnerable to routing attack since lookup requests are forwarded from tracker to traker without verifying their trust. A reputation-based routing can be used to handle this problem.

7.1.4 Denial of Service attack

Denial of service (DoS) is an ordinary threat on the Internet. We discuss two types of DoS attacks in the followings.

7.1.4.1 Flooding attack

An attacker can overload a target node with large amount of requests to make the overlay storage service unavailable. The classic TCP sync flooding (also known as halfopened TCP sync attack) is an action where a malicious node initiates a TCP session and skips sending the `ACK` response to cause the session unaccomplished. The malicious node can also forge multiple IP addresses to send loads of `SYN` requests to a victim node. Then, the victim replies with `SYN-ACK` to those bogus addresses and keeps waiting for `ACK` responses. The attacker repeats these steps until the victim gets exhausted and cannot serve any other legitimate user.

Some common defences against this attack are defined in RFC 4987 (36). The first defence is to reduce the `SYN-RECEIVED` timer. Secondly, the end-host may recycle the oldest half-open transmission when the entire resource is consumed. However, these solutions are not very effective when the attacking packet rate is high. Another method is to provide a `SYN` cache which limits the amount of resource allocation when initiating the session. Full resource is only allocated after the session has been successfully established. Similarly to `SYN` cache, `SYN` cookies control the amount of resource usage. However, there is no state allocation during the challenge process; the state will be allocated after the handshake has been complete. The request attack is another type of flooding attack where a malicious or several compromised nodes overwhelm the network with numerous request messages. This attack causes legitimate requests unable to be accomplished.

7.1.4.2 Refusal of action or service

Each node in the overlay stores some data objects related to its `nodeId`. Hence, there is a possibility that a particular user data will be stored at a malicious node who later denies providing data service when user asks for.

To prevent this attack, the overlay storage should not only store the data record on a single node but duplicate the record to the node's neighbours. Data replication resolves the problem of denial of service attack as it diminishes a single point of failure. Additionally, replication also reduces lookup latency especially for popular objects.

This attack hardly applies to NeuroCast unstructured overlay since data is managed by the respective data owner at each peer. In other words, we trust

the data owners to be responsible for their own lookup service. They have the privilege of whether to give their information to a requester or not. However, the hits retransmitting the chunks of the stream could refuse to send the stream and therefore dismish the application performance. Again, with a reputation mechanism this peer behavior could be mitigated.

7.1.5 Traffic analysis

The nature of the Internet allows a malicious node to monitor video transmissions, and obtain private information like the sender's and receiver's IP addresses. Although the IP address itself may not be enough to identify a person, it discloses user privacy by implying the geographical location of a user. Moreover, the packet's IP header indicates that two different hosts are communicating with each other. This information when correlated with some other knowledge might reveal user identity.

To mitigate this threat, NeuroCast should consider Privacy Enhanced Techniques (PETs) such as Mix Networks and Onion Routing which offer mutual privacy protection for sender and receiver nodes. Firstly, the sender node needs to randomly choose a set of relay nodes in the transmission, and negotiates the encryption keys. These keys are used to conceal routing information into multiple encrypted layers. The cipher message will be traversed along the selected path. Each relay decrypts one layer of the message and forwards to the next hop until the message reaches the destination. Thus, every node only knows the IP address of previous and successive nodes but not the IP address of sender and receiver nodes.

7.1.6 Eclipse attack

Before launching an eclipse attack (93), a malicious user must have control over a group of nodes in the network. It is also possible for a single node to perform this attack using the Sybil attack to generate great amount of identifiers and pretend to be multiple nodes. Then, the malign node compromises its neighbors by injecting false information into the routing table. This attack is also called routing-table poisoning attack. An attacker who manages the eclipse attack can

perform other types of attack such as routing attack, traffic analysis, or denial of service in an easy manner.

The eclipse attack can be detected by looking at the degrees bound to a node. An indegree means the number of incoming routes to a node whereas an out-degree means the number of outgoing routes from a node. If the in-degree or out-degree is higher than an average threshold, the node has been corrupted. Therefore, each node should be cautious when selecting a neighbour node.

7.1.7 Content Pollution Attacks

Content security and copyright protection have brought a constant battle between the content industry and P2P content-sharing companies and users in the last several years. On one hand, many users are enjoying free content distributed via P2P file-sharing and streaming applications. On the other hand, content creators and their representatives are trying to stop unauthorized sharing.

While taking file-sharing companies and individuals to court, the content industry also learned to take advantage of the distributed resources and other characteristics of P2P networks to sabotage P2P file-sharing systems themselves. One of the techniques they used is called *pollution*. The goal is to stop P2P file sharing by causing user frustration. The basic idea is to create bogus content, especially popular content, and store it in a P2P file-sharing system. When peers search for files on these systems, they could be directed to the bogus ones. The chaining effect of such sharing will result in a large number of peers getting constant bogus content so that they become frustrated with the system and thus abandon it voluntarily.

Content pollution and metadata pollution are the two most popular types used in these attacks. With content pollution, the actual content of a music or video files is modified significantly to generate a bogus copy. Metadata pollution alters the metadata of a fake copy of the content and makes the metadata resemble the metadata of the target music or video. Because many P2P systems do not have effective mechanisms to prevent pollution attacks, these attacks have become successful tools for the content industry to use in its fight against illegitimate content sharing over P2P networks. Overpeer, for example, is such a tool that uses pollution attacks to help content creators and distributors minimize sharing

over P2P networks. Based on a measurement study conducted by Liang et al. (67) over KaZaa (4), 50% of the copies of many popular songs were polluted at the time of the study. This shows the success of polluting systems for copyright protection.

Pollution is used for copyright protection in the preceding case; it can also be used for other purposes as well. To defend against pollution attacks, we can adopt many mechanisms discussed in this section. Authentication and fingerprinting may be used for peers to acquire only authenticated copies of data and thus reduce the probability of downloading bogus contents. Trust and reputation systems can also help reduce the spread of bogus files.

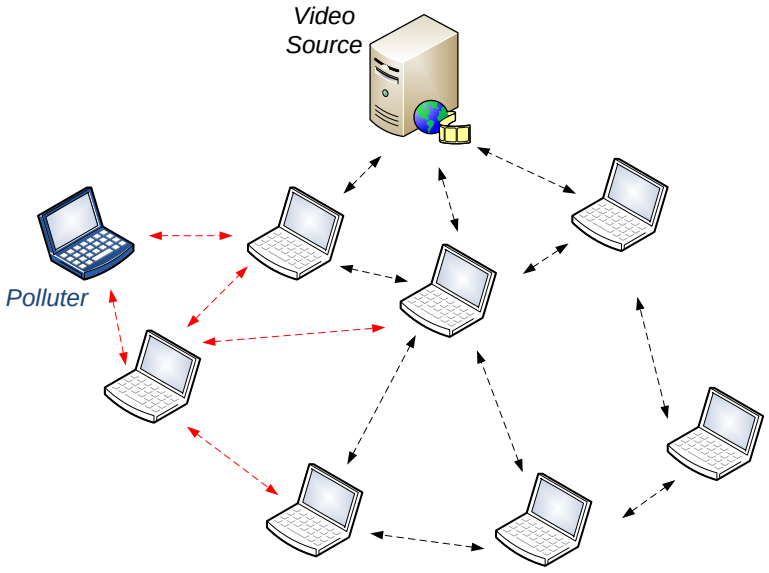


Figure 7.3: Pollution Attack in a P2P Live Video Streaming System.

Regarding to video streaming systems pollution attacks are also feasible, where attackers forward unsolicited chunks to other peers. Figure 7.3 shows a P2P network where a malicious node sends bogus chunks to other peers, falsely marking these chunks as legitimate. These corrupted chunks get propagated through the P2P network, thereby not only degrading performance of the polluter's direct neighbors but also of other peers. Content pollution attacks can severely impact the quality-of-service in P2P live video streaming systems.

7.1.8 Integrity of Data Items

Attackers can impersonate a target identity by redirecting requests through forged P2P messages. Likewise, without integrity protection attackers can modify chunks in P2P live video streaming systems. This can result in content pollution attacks with advertisements or offending content. However, in systems with a single or few sources it may be reasonable to assume that peers can obtain a certificate of the source (e.g., through an out-of-band channel).

7.1.9 Fairness in Resource Sharing

Many researchers have observed the problem of fairness in P2P overlay-based applications. In one study (50), 70% of free riders were reported. It is unfair to other peers in the overlay if some peers only want to use other peers' resources without fair contribution of their own resources. From a security point of view, this kind of behavior can be characterized as resource theft under the interception class of attack. Furthermore, with a large number of free riders in a P2P network, such as the 70% reported in (50), the rest of the peers more easily become congested or come under attack.

7.2 Trust and Privacy Issues

One important aspect of a P2P system is the way a peer trusts another peer in the system. The level of trust is the level of confidence of one peer toward another peer with which it is communicating. A P2P system relies heavily on a set of distributed peers working properly and fairly together. In a small P2P system, especially one that involves only known entities, establishing and maintaining trust between peers is easily achievable. However, today's P2P system can be tens of thousands to millions of peers in size, with peers interacting with many unknown peers. Unfortunately, free riders are a common phenomenon in P2P applications.

Performing peer authentication and authorization and thus establishing and maintaining reliable trust between peers play central roles in many aspects of P2P security. The main goal is to avoid interactions with nodes that do not lead to security risks. Noticeably, trust is not merely an issue of peer-to-peer trust.

Content and resource trust, being able to authenticate the content and resource of an accessing peer, is also an obvious security issue.

7.2.1 Reputation

One notable facet of research focuses on trust value establishment. The key idea is to use previous interactions to determine the reputation and thus the trustworthiness of a particular user in the P2P network. Reputation is the memory and summary of behavior from past transactions. Reputation score is the numerical representation of reputation. It can be calculated either via a centralized reputation server or distributedly using local or global trust metrics. Trust value can be a function of the reputation score that acts as a guide for peers to make choices in selecting transaction partners or neighbors in a P2P network. It also serves as an incentive for those peers that have good reputation scores.

According to Despotovic and Aberer (31), reputation systems can be split into those using probabilistic estimation and those based on social networks. The former considers only a small proportion of the globally available feedback concerning a node's behavior, using probabilistic methods to assess its trustworthiness. The latter aggregates all the feedback available to assess trustworthiness. The two solutions were found to have varying degrees of effectiveness depending on the nature of collusion that occurs within the P2P network. For example, the social networking technique was found to be most effective when the peer population was split into a group of colluding peers and a group of noncolluding peers of equal size, with probabilistic estimation performing better otherwise.

The implementation cost in terms of message exchanges of the probabilistic method was found to be $O(logN)$ in structured networks with logarithmic search costs and $O(E)$ for unstructured networks, where N represents the number of nodes and E the number of edges in the overlay. For social networks the entire network must be flooded, and the cost is therefore $O(N)$ and $O(E)$ for structured and unstructured networks, respectively.

In general, reputation requires mapping information about past transactions to peers and a metric that can bind the mapping to trustworthiness. When a peer solicits reputation information about a particular peer from other peers of the network, it faces the problem of trust; should the peer trust the reputation

information provided by other peers? Obviously, if any peer can easily refresh or modify its reputation score, the value of the reputation-based scheme degrades to negative. This implies that a reputation scheme needs to be secure to prevent it from turning into the weakest link. Today this is still a challenging problem.

7.2.2 Privacy

Many P2P networks today do not have built-in privacy mechanisms. Users of these P2P networks can be tracked or identified by others, including attackers. Obviously, user education, better software practice, and more secure protocols and system architecture can help reduce the risks. Adding privacy technology to a P2P network can also help offset this disadvantage.

Goldberg (43) believes that reputation interacts well with privacy-enhanced technologies since reputation can be calculated without disclosing private information of a peer. His view is shared by Kinateder and Pearson (64). They use a network of trusted agents on each client platform that exploit Trusted Computing Platform Alliance (TCPA) technology. A trusted agent forms recommendations and decides what is appropriate to send out, depending on who is asking for it. This, or another such trusted agent, can be used to formulate queries asking for recommendations from others in a peer-to-peer network and process the responses. Furthermore, the system is designed such that the agents are independent and may be trusted by entities other than the owner of the platform on which they are running, and the integrity of these agents is protected by the trusted platform against unauthorized modification.

Another approach attempts to improve privacy with anonymous communication. If a communication protocol can guarantee that the sender is indistinguishable from other peers in network to a receiver, and vice versa, sender and receiver anonymity can be achieved. P5 (89) achieves this goal via a broadcast-based protocol. It allows secure anonymous connections between a hierarchy of progressively smaller broadcast channels and allows individual users to trade off anonymity for communication efficiency.

Tarzan (38) serves as transport layer anonymizer. Each node selects a set of peers to act as mimics. Initial node discovery and subsequent network maintenance are based on a gossip model. Mimics are selected randomly but in some

verifiable manner from the available nodes. Each node exchanges a constant rate of cover traffic of fixed size packets with its mimics using symmetric encryption. The relays' public keys are used to distribute the symmetric keys. Actual data can now be interwoven into the cover traffic without an observer detecting where a message originates. A sender randomly selects a given number of mimics and wraps the message in an "onion" of symmetric keys from each node on the path. The sender passes the packet—indistinguishable from cover traffic—to the first node in the chain, which removes the outermost wrapper with its private key and then sends it along to the next node. With the exception of the last node, each node in the chain is aware of the node before and after it in the chain but cannot tell where it is in the chain itself.

The final node in the chain of mimics acts as the Network Address Translator for the transport layer and sends the packet to its final destination through the Internet. This final node gets the content and destination but has no information about the sender. Nodes store a record for the return path, so a reply can be received by the final node in the chain, rewrapped with its private key, and sent back to the penultimate hop. The message is then passed back through the chain, with each node adding another layer of encryption. The originating node can use the public keys of each node to unwrap the layers and read the message. Since it is the only node to know the public keys of each hop along the path, the content is secure.

7.3 Possible Security Approaches

Given the vulnerabilities of existing P2P overlays and the attacks we have described, it is not obvious to achieve a P2P overlay secure and dependable. Defending against the threats against P2P overlays requires careful planning and selection of P2P infrastructure and security mechanisms. Security policies are the foremost requirement in building a secure system. The set of rules defines and governs the control, use, and action entities of a system. With security policies in place, it is then possible to design suitable security mechanisms to enforce the security policies and ensure the security of the system.

In this section we describe a series of mechanisms that could make the NeuroCast application more secure, though it could make more complex the algorithms.

7.3.1 Cryptographic Solutions

Cryptographic schemes offer the most effective solutions for many information security issues. They are also essential to security in P2P. Among various crypto tools, encryption and authentication are two fundamental and the most frequently used crypto primitives.

Encryption is the process of disguising a message in such a way that its content is hidden and cannot be revealed without a proper decryption key. This is a fundamental security tool that implements confidentiality with coding. Symmetric key and asymmetric key encryptions are the two types of encryption algorithms. Symmetric key encryption algorithms, also called private key encryption algorithms, are a class of encryption algorithm that uses identical keys for both encryption and decryption. Popular examples of symmetric key encryption algorithms include Data Encryption Standard (DES) and Advanced Encryption Standard (AES), which are standardized by the National Institute of Standards and Technology (NIST) and are widely adopted by many applications. Asymmetric key encryption algorithms, a.k.a. public key encryption algorithms, such as the RSA encryption algorithm, are another class of encryption algorithm that employs different keys at encryption and decryption. Note that in general, asymmetric key algorithms are much more computationally intensive than symmetric algorithms.

Encryption can play many positive roles in P2P security. It makes it difficult for attackers to carry out interception and modification classes of attack. If all confidential information is encrypted, even if some is shared or leaked over some insecure P2P file-sharing communication channels, adversaries would have a hard time decrypting the information without a proper key. The security risks will be subsequently reduced. Therefore, in the ID theft case discussed early in this chapter, the offender would not have been able to get access to others' financial information that easily. The number of victims would have been greatly reduced in that case.

Authentication, another essential security tool in computer systems, is the process of verifying whether an object is in fact who or what that object declares itself to be. A one-way hash that is nonreversible, sensitive to input changes, and collision resistant is used for authentication as well as data integrity verifi-

cation. Authentication can also play many positive roles in P2P security. For example, combining secure authentication of each peer with message encryption, a P2P system can prevent eavesdropping attacks. With content authentication, information substitution and insertion attacks can not easily be realized.

7.3.2 DoS Countermeasures

The most popular countermeasures of DoS attacks include service/host backup, reactive detection, rate limiting, and filtering. Having a separate emergency block of IP addresses, for example, can be invaluable in surviving a DoS attack. Pattern detection is often helpful by storing the signature of known attacks in a database. Rate-limiting mechanisms impose a rate limit on a stream that has been characterized as malicious by the detection mechanism. These are often used as a response technique when a detection mechanism cannot characterize the attack stream. Effective filtering is another way to protect against DoS and DDoS attacks. For example, attacks originating from or going to bogus IP addresses can be filtered using Bogon filter (a bogus IP filter). Filtering traffic based on access control lists (ACLs), rate limiting of IP addresses, or ranges of IP addresses can also be employed. Some switches today offer deep packet inspection capability and Bogon filtering. These can help defend against DoS and DDoS attacks.

TCP splicing, also called delayed binding, is another widely used mechanism to prevent DoS attacks. By postponing the connection between the client and the server, sufficient information to make a routing decision can be obtained. Many application switches and routers implement this capability today. Delay-binding the client session to the server until the proper handshakes are complete can prevent DoS attacks.

Firewalls have been used to defend against many types of attacks. Today advanced firewalls have built-in capabilities for differentiating good traffic from DoS attack traffic. For example, Cisco PIX (Private Internet Exchange) uses stateful inspection to confirm TCP connections before proxying TCP packets to service networks. Certainly most simple firewalls have limited capabilities for differentiating good traffic from DoS attack traffic, making it difficult to defend against DoS attacks, let alone DDoS attacks.

Recall that Naoumov and Ross (75) show the distributed nature of two types

of P2P-DDoS attacks, which lead to the ineffectiveness of IP address filtering in defending such types of attack. Although pattern detection and advanced filtering mechanisms may be helpful in detecting these types of P2P-DDoS attack, the eruption of new types and large P2P-DDoS attacks make this type one of the toughest to defend against.

Sia (92) also exploited the DDoS vulnerability of the current BitTorrent (BT) protocol. Allowing a user to arbitrarily specify its own IP is one significant cause of the BT protocol vulnerability to DDoS attacks. Sia suggested fixing this issue using more strict protocols with source authentication and pattern-matching packet filtering mechanisms.

Mirkovic and Reiher (74) summarize DDoS defense mechanisms into a series of categories. For example, resource accounting and resource multiplication are preventive types of defense, whereas pattern detection, anomaly detection, hybrid detection, third-party detection, agent identification, rate limiting, filtering, and reconfiguration are reactive defense types. Most notably, deploying comprehensive protocol and system security mechanisms can substantially help improve resilience to DDoS attacks. Additionally, enforcing policies for resource consumption and ensuring that abundant resources exist can both help reduce the impact of DDoS attacks so that legitimate clients will not be affected.

7.3.3 Secure Node Identifier Assignment

The most straightforward scheme for secure nodeId assignment is to establish a centralized certificate authority to issue `nodeIds` and bind the `nodeId` to the IP address. In enterprise system or private networks, user authentication is already in place. Hence, central authority-based `nodeId` certification can be a preferred choice. Furthermore, imposing a nodeId certificate fee or physical identify bound `nodeIds` will greatly reduce the risk of Sybil attacks. Many believe, though, that trust is an important issue to improve a P2P system's capability to fight Sybil attacks.

A distributed `nodeId` binding scheme is another possible means to avoid malicious nodes inserting themselves at multiple `nodeId` locations with a single IP address. Every time a node accepts a particular binding <node id, IP address> from another node, it stores the binding information in the overlay using the IP

address as the key. Before storing this binding, it checks to see whether another binding for the same IP address is stored. If such a binding with a different `nodeId` exists in the overlay, the current request to establish a new binding is rejected. If, however, an existing binding in the overlay contains the same `nodeId`, the current request is accepted. If there is no existing binding information for that particular IP address in the overlay, again the current binding is accepted.

Noticeably, without additional measures this scheme is vulnerable to DoS attack, whereby a malicious node simply inserts random bindings for different IP addresses, preventing those IP hosts from joining the overlay. To avoid this problem, a return routability test, similar to the one developed for Mobile IPv6, can be employed. A return routability test should be conducted every time a binding is stored on the overlay, to ensure the validity of the binding. Obvious extensions to this solution in terms of adding redundancy and verifying the source IP address ensure that a single node (with single IP address) cannot hog a large portion of the `nodeId` space. Although there is additional cost in terms of the return routability message exchange, we believe that it is justified for the same rationale behind the adoption of such a solution in Mobile IPv6.

7.3.4 Secure Message Forwarding

Since peer-to-peer overlays rely solely on other peers for message routing, a message has to be properly routed without modification in transit. An adversary may try to alter the message when routing the message to its receiver, alter its routing table to disrupt the message forwarding, or take advantage of locality to control some routes. Secure message forwarding ensures that at least one copy of a message sent to a peer reaches the correct peer with high probability.

An obvious way to trade performance for security is to use multiroutes to send duplicated messages from the source to the destination. However, in a bandwidth-intensive application, message duplication will burden the system. An improvement on the scheme is to utilize message authentication. Only when a message fails an authenticity check, redundant routing is invoked to reforward a correct copy of the message from one or more different routes. Although this is not helpful in streaming media applications due to the real-time and continuous playback requirement, it can be used otherwise. In streaming media applications,

instead of redundant duplication, multiple substreams through multiple-route forwarding can be introduced. With a stream-splitting algorithm such as multiple description coding, a media stream can be split into multiple substreams. A receiver can still play back the media without interruption when receiving only some of all the substreams, although with degradation of quality. In this case, a fine-grained scalable message encryption and authentication scheme can help improve the security in streaming message forwarding, with minimal additional cost. One tradeoff of the stream-splitting scheme, though, is the added control cost in terms of multiple distribution routes (trees).

Carefully designed overlay routing algorithms may also help reduce security risks in the overlay layer. Assuming that there are an average of h routing hops in message delivery from the sender to the destination peer, the percentage of faulty nodes is γ, so between the sender and receiver, the probability of routing successfully to a correct destination peer is $p = (1 - \gamma)^h$. Obviously, the smaller h is, the higher p may be. This suggests that routing algorithms with fewer hops increase the probability of routing accuracy and hence reduce the probability of security risks at the overlay layer due to faulty message forwarding.

The hybrid P2P architecture that comprises supernodes in the P2P overlay may also help improve secure message forwarding. If only trusted nodes can be promoted as supernodes, those supernodes can serve as filers or servlets. Trusted messaging between supernodes can be established. Further, a supernode can serve as a "centralized" authority to those ordinary nodes connected to it or message filters. This adds another layer of protection.

7.3.5 Improving Fairness

Incentives are one way to improve fairness in resource shairing; a peer gets an "incentive" to use other resources by contributing its own resources on a pay-more/get-more base. A centralized broker or trusted centralized quota authority that monitors all transactions could accomplish the goal of fairness in resource sharing. However, scalability is a major limitation of this kind of scheme. The centralized broker can quickly become a bottleneck in applications with frequent requests.

In this case, distributed mechanisms are needed. A quota manager (77) ap-

proach, for example, is one distributed approach. For each peer in the P2P network, a manager set with a set of nodes, perhaps neighbors of the peer, acts as the quota manager. Each manager records the amount of resources consumed by the peers it manages. A remote node, when requesting a fair sharing, would seek an agreement with a majority of the managers of the peer agreeing that a given request is authorized. If a hybrid P2P network architecture is employed, the supernodes can naturally act as quota managers. In a fully distributed P2P network, though, selecting peers to act as quota managers is a challenging problem. Clearly, in either case, the process of approval can cause long latency.

Another class of distributed mechanism is distributed auditing. One approach is to ask each peer in the P2P network to maintain a record of its own usage and to publish it in the overlay. Other peers can audit these records to achieve fair sharing. In the case of a hybrid P2P network, auditing can be done in two ways: having the supernodes act as auditing authorities and collaboratively publish the auditing records, or making supernodes take on the task of auditing monitoring, whereby all peers publish records of their own, with the supernodes monitoring the publishing and auditing of peers. Of course, techniques to ensure that all peers will publish their records are needed, and this is most challenging in the fully distributed P2P network. Incentives are one way to achieve this goal.

Micropayment systems may also help improve the fairness issue. It is unclear, though, whether any existing micropayment system or simple auditing scheme could scale well to support large P2P overlay and/or high churn applications. Mojo Nation, an already tested P2P system, tried to use a credit and incentive-based scheme to improve fair sharing. It was a pseudo-currency micropayment-based system. In Mojo Nation, if you provided resources, computational, storage, or bandwidth, to the system, you earned Mojo, a kind of digital currency. If you consumed resources, you spent the Mojo you'd earned. This system was intended to keep freeloaders from consuming more than they contributed to the system. But if users are heavily consuming resources, it does not pose a real threat to most existing P2P system users, so Mojo Nation never really worked. Today, designing a P2P system that can take advantage of incentive-based mechanisms with efficient auditing is still a challenging problem that is being studied by many researchers.

7.3.6 Pollution Defense Mechanisms

Polluted chunks received by an unsuspecting peer not only effect that single peer, but since the peer also forwards chunks to other peers, and those peers in turn forward chunks to more peers, the polluted content can potentially spread through much of the P2P network. There are several possible defenses to the pollution attack such as blacklisting, traffic encryption, hash verification, and chunk signing.

In the blacklisting approach, we attempt to determine - in a centralized or decentralized manner - the peers that originate and relay pollution. All such peers are put onto a blacklist. Peers neither send chunks to nor receive chunks from peers on the blacklist. An alternative approach is for each peer to attempt to determine whether a chunk is polluted. If a chunk is determined polluted, then the peer that sent the chunk can be assigned a low reputation value. Again, the reputations can be shared and a distributed blacklist can be created. The critical step in this approach is accurately determining whether a chunk is polluted or not. In P2P live video streaming, a receiver typically obtains chunks from more than one peer. Therefore, by comparing characteristics of the received chunks, one might be able to distinguish between the fake and the legitimate copies. Video and audio processing techniques can possibly be used to detect polluted chunks. However, an attacker should be able to circumvent such an approach by creating chunks that resemble neighboring chunks (in the stream of chunks) but nevertheless significantly diminish the quality of the rendered video. For example, the attacker could insert duplicate chunks into the stream. In summary, it is unlikely that any of these reputation/blacklisting approaches will be able to consistently stop the pollution attack.

Regarding to the traffic encryption solution, to inject pollution into a stream, the attacker needs to send the correct messages to the other peers with the correct header and data format. This requires the attacker to first sniff some traffic specific to the streaming system and analyze the traffic to understand the protocol sequence and message formats. If all the messages a system uses were encrypted, it would be difficult for the attacker to determine the message structure in distributed application. This would prevent the attacker from inserting crafted messages into system, such as message containing polluted data. The

main disadvantage of using traffic encryption as a means to preventing pollution, however, is that it works well to protect the privacy of the application protocol and message formats until the system is subjected to a reverse engineering of the source.

In BitTorrent (1), before a peer begins to download a file, it obtains a torrent file which provides the hashes of all the chunks of the file. When a peer receives chunks from other peers, it compares the hashes of the chunks received with the corresponding hashes in the torrent file to verify their integrity. We now consider applying the same general technique for P2P live video streaming. The simplest approach for this would be for each receiver to get the hash of each chunk form the source itself. As in BitTorrent, this would allow each peer to verify the integrity of each chunk before forwarding it to other peers. However, the load on the source will be very high for a large number of receivers. Therefore, hash version, as done in BitTorrent, is not a viable solution for P2P live streaming.

The chunk signing mechanisms is considered the most appropriate for avoiding this type of attack. There are several techniques that permits to sign the chunks so that they can be authenticated by the receiver. In each technique, the so-called "authentication information" needs to be transmitted to the receivers along with the chunks. This authentication information can either be provided by the source (in which case the load on the source might be high) or could be distributed through the P2P system itself, in the form of a separate stream or be piggybacked onto chunks.

7.4 NeuroCast Security requirements and design principles

The basic requirements for secure Neurocast areas follows. Firstly, an established session only connects to the expected parties and not anyone else. Secondly, the retransmission between parties must be protected. Moreover, the user should be able to discard an unwanted requests. This means that the system should provide an option for user to manage whether he wants to limit incoming requests from only friends or also unknown users. Lastly, information about who are communicating with each other should not be revealed to an outsider. That is a

malicious user should not be able to monitor call history.

7.4.1 Privacy

Understanding privacy as: "The privilege of all people to have their streaming systems and content free from unauthorized access, interruption, delay or modification." In other words, users should be able to control what information will be given to others. The information also must not be corrupted and must reach the intended party successfully.

As the NeuroCast web interface do not provide privacy protection for user information, a malicious user can monitor and intercept all traffic between proxy and overlay storage. Lack of privacy protection presents many threats to the applications including message eavesdropping, tampering, etc.

To fulfil the privacy requirements, NeuroCast should implement a set of secure interfaces that provide authentication and authorization, confidentiality and integrity as described in the following sections.

7.4.2 Authentication and authorization

The system should authenticate users who access to the overlay video streaming service. Normally, the key for storing and retrieving a user registration record is the node identifier. Without authentication, anyone who knows the peer's identifier is able to query for user's information or overwhelm the video storage of target identifier with bogus records i.e. drowning attack.

This problem could be solve by introducing a shared secret between the data owner and recipients. Every user needs to create a shared secret with all of his broadcasting peers. This secret together with the `nodeId` produces a keyed-hash value which is used as a record key. Having only one secret for all peers is more convenient and easier to manage; however, it imposes some difficulties (restrictions) to the user e.g. when user wants to revoke the access right from one of the peers. Thus, in our design, user should create a different secret for each of the peers.

As a record key cannot be computed without the knowledge of a shared secret, a malicious user cannot easily put false data under a specific peer identifier. However, if an attacker wants to attack any random peer identifier, it is still

possible to replay a captured key with the bogus value. This can be handled by changing the shared secrets periodically to prevent replay and brute force attacks.

The new record key ensures that users are unable to query for someone's data unless they know the common secret. This prevents unauthorized access to other people's records.

7.4.3 Confidentiality

NeuroCast should provide confidentiality for both key and value of each record in the overlay. The value of a record contains personal information such as ip address, presence status, and neighbor list. Without confidentiality, the system is vulnerable to many kinds of attacks such as message tampering, and eavesdropping.

NeuroCast could protect the record values using symmetric and asymmetric encryptions (using shared secret and intended party public's key respectively) depending on the use case. Symmetric encryption is faster, but asymmetric encryption provides also user authentication.

7.4.4 Integrity

Integrity protection includes origin integrity which verifies the source of the data (often called authentication), and data integrity which refers to the trustworthiness of the data content. Without integrity control, any untrustworthy node that helps forwarding put and get requests or any node who broadcast a video is able to modify the content without being noticed.

NeuroCast should provide mechanisms to ensure integrity of the video by allowing a recipient to verify whether the information in the overlay is indeed submitted by the original party and is not later be tampered. We propose a set of signed interfaces to protect record integrity by signing all parameters (key, value, time-to-live, and hash of the removal secret) using the owner's private key. The signature is attached to the original record. End recipient then verifies the correctness and trustworthiness of the record using the public key of the owner. Signing also helps to prevent drowning attack.

7.4.5 Availability

Availability refers to the ability to access desired information or services; NeuroCast should ensure that users get information or service when they request for. However, providing availability is difficult due to various kinds of denial of service attacks. NeuroCast could provide availability by protecting users from sessions hijacking where an attacker modifies users' contact location causing the victims unable to obtain service. Furthermore, we protect the overlay storage from drowning attack using signed interfaces as mentioned earlier. With the signed interfaces, the user can obtain the correct information that she asks for and avoid the unnecessary effort to filter false records caused by drowning attack.

Chapter 8

Conclusions

P2P technology gives novel opportunities to define an efficient multimedia streaming application but at the same time, it brings a set of technical challenges and issues due to its dynamic and heterogeneous nature. Especially, real-time feature is key requirement for user. With increasing of the peer nodes number, the delay issue is serious problem. We must make a balance between the breadth and depth of a live streaming overlay tree. At the same time, the robustness issue must be considered carefully, as the dynamic feature and freedom of P2P network itself. P2P live streaming must maintain the stability of services, without restricting the P2P node freedom. These issues also should be dealt in P2P live streaming systems.

Designing and implementing an efficient and failure-resilient end system multicast service on top of highly dynamic overlay networks poses several research challenges. In this this work we have presented the implementation of NeuroCast, an unstructured P2P system for live steaming. We also analyzed the NeuroCast performance in emulated network environments. NeuroCast is born as an improvement of an already implemented platform, PeerCast. Through the contrastive analysis between PeerCast and NeuroCast, we have verified the better performance of our scheme for supporting more large scale and more stable live streaming requests.

A great part of the time of this thesis has been devoted to the implementation of the NeuroCast application. We first pointed out what were the issues addressed in the NeuroCast design in order to realize a real prototype. We made a research about existing solutions to problems like membership management, multi-source

and load balance mechanisms. Then we designed algorithms and protocols to face these issues in a system like NeuroCast. Moreover we modified some of the NeuroCast core algorithms in order to adapt them to a real world environment.

NeuroCast makes several contributions: its design combines an unstructured mesh-based architecture, which grants the nodes great freedom to associate, which are used as peer selection strategies. NeuroCast intends to optimize bandwidth allocation and combine dynamic peer selection strategies that rely on implicit feedback from data reception.

The presence of a global control process based on node lag allows NeuroCast to quickly react to changes in both, node resource availability and system membership. NeuroCast's algorithms are meant to redistribute node capacity in an useful way and, in case of system-wide resource shortage. We also argue about the risk of buffer starvation during live streaming, which becomes high for peers that offer an insufficient upload contribution.

The experiences we performed using large-scale testbed emulation, indicates that NeuroCast is capable to operate in a harsh environment such as the Internet. Also, they show that NeuroCast can easily take into account network conditions when establishing node relationships, introducing a further performance improvement. Our results also confirm the ability of unstructured mesh-based systems to withstand the high levels of transience that can result from user and network dynamics (churn, failures, congestion, etc.).

Moreover, we have collected and elaborated a number of metrics to evaluate both generic and resource-aware data-driven systems. Besides their immediate interest for the analysis of NeuroCast, we hope they will help to reach an improved understanding of unstructured streaming systems and to develop a more comprehensive comparison between structured and unstructured approaches to live data distribution.

We have also analyzed NeuroCast under a security point of view. Thus, serveral threats have been presented and some solutions to mitigate these threats have been proposed. However, the impemented version of NeuroCast has not take into account these security issues but we are aware of its importance for a real environment.

Finally, we presented some preliminary results about the NeuroCast performances analyzing the results obtained by our experiments. The NeuroCast pro-

totype performs well in different conditions. The peers distributed the overall bandwidth of the system among them in order to provide to all nodes a sufficient download rate for the playback of the stream. The nodes maintain a small average delay from the source and correctly playback the stream. The system well scales to a big number of nodes showing a logarithmic trend in response to the increasing number of peers.

8.1 Future Work

Recent work on peer-to-peer streaming has focused on designing systems which maximize the achievable streaming rate. Moreover, recent experimental studies have shown that tree-based designs can reach close-to-optimal rates in real-life conditions. However, until now, efficient mesh-based designs have only been evaluated analytically or through simulations. In this thesis we have presented NeuroCast which demonstrates that a pure mesh architecture can deliver near-optimal rates with low diffusion delays. In addition, we have identified some optimizations which help increase the efficiency of mesh-based designs.

The search for an optimal strategy to distribute live data over networks with arbitrary upload capacity distribution is not over yet: recent developments of theoretical models (27; 72) and the application of graph theory (41), game theory (76) and gossip algorithms (85) will certainly lead to an improved understanding of the live streaming problem and to the design of new, more efficient algorithms and solutions. It is our opinion that the improvement of live streaming techniques will also benefit applications for large-scale data diffusion, such as video on demand and bulk file distribution. The deliberate introduction of loose synchronization between receivers, once seen as an undesirable constraint of live media, could actually prove an effective method to increase the efficiency and data distribution performance of these systems.

The live streaming problem has mostly been studied in cooperative contexts so far, which implied either full compliance by the nodes to system policies (e.g. providing as much upload as required, connecting to a specific number of "entitled" sub-trees), or at least honest reporting of protocol information (e.g. correctly describing the number of children served, providing a truthful account of the

content a node buffer). NeuroCast has not take into account the consequences of non-cooperative behavior (as freeloading or insufficient upload contribution): the support for bandwidth heterogeneity naturally implies the ability to react to purposeful lack of cooperation.

Nevertheless, many avenues of attack emerge when the hypothesis of policy-compliant behavior is rejected: an interesting example is provided by the latest attempts to defeat other incentivebased mechanisms, such as the BitTorrent protocol (1). In the case of deliberate tampering with protocol information, however, the problem becomes way more complex, reaching to the realms of practical and theoretical computer security: furthermore, the assumption of player rationality may no longer hold if nodes try to actively disrupt the system without seeking any benefit from it. The concept of *faithfulness* (91) of an application as a form of resilience to misleading external information and behavior has been recently introduced to support the design and analysis of P2P systems.

On the other hand, given our encouraging results, we plan to extend our work and investigate a number of issues not covered in this work. First, we plan to further evaluate the use of source coding (such as erasure codes or multiple description coding) to minimize the impact of chunk misses on image quality.

Second, we will look into improving the mechanisms that determine whether the overlay has enough aggregate uplink capacity to deliver the stream with acceptable losses. An adaptive system could enable high-bandwidth helper nodes (independent from the source) when a capacity shortage is detected.

Next, we will implement and evaluate all the security approaches that have been proposed to mitigate the security issues of NeuroCast. We plan to evaluate NeuroCast in a non-cooperative environment where trust and reputation aspects will become essential.

Finally, we will study how we can modify algorithms to further increase their delay performance by favoring high capacity nodes not only at the source, but in all scheduling decisions.

Appendix A

VNUML Configuration Files

In this appendix we detail the XML files used to emulated the different scenarios used during the evaluation of the NeuroCast network using VNUML.

A.1 XML File Scenario Figure 6.1

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE vnuml SYSTEM "/usr/share/xml/vnuml/vnuml.dtd">

<vnuml>
  <global>
    <version>1.8</version>
    <simulation_name>Test_1</simulation_name>
    <automac/>
    <vm_mgmt type="none" />
    <vm_defaults>
       <filesystem type="cow">/usr/share/vnuml/filesystems/
          root_fs_tutorial </filesystem>
```

```
12        <kernel>/usr/share/vnuml/kernels/linux</kernel>
13        <console id="0">xterm</console>
14      </vm_defaults>
15    </global>
16
17    <net name="Net0" mode="uml_switch" />
18    <net name="Net1" mode="uml_switch" type="ppp">
19          <bw>1500000</bw>
20    </net>
21    <net name="Net2" mode="uml_switch" type="ppp">
22          <bw>1000000</bw>
23    </net>
24
25    <vm name="Router1">
26      <if id="1" net="Net0">
27        <ipv4>10.0.0.2</ipv4>
28      </if>
29      <if id="2" net="Net2">
30        <ipv4>10.0.1.1</ipv4>
31      </if>
32      <route type="ipv4" gw="10.0.0.2">10.0.0.1/24</route>
33      <route type="ipv4" gw="10.0.1.2">default</route>
34      <forwarding type="ip"/>
35    </vm>
36
37    <vm name="Router2">
38      <if id="1" net="Net2">
39        <ipv4>10.0.1.2</ipv4>
40      </if>
41      <if id="2" net="Net1">
```

```
42        <ipv4>10.0.2.1</ipv4>
43      </if>
44      <route type="ipv4" gw="10.0.1.1">default</route>
45      <route type="ipv4" gw="10.0.2.1">10.0.2.0/24</route>
46      <forwarding type="ip"/>
47    </vm>
48
49    <vm name="E1">
50      <if id="1" net="Net1">
51        <ipv4>10.0.2.2</ipv4>
52      </if>
53      <route type="ipv4" gw="10.0.2.1">default</route>
54    </vm>
55
56    <host>
57      <hostif net="Net0">
58         <ipv4>10.0.0.1</ipv4>
59      </hostif>
60      <route type="ipv4" gw="10.0.0.2">10.0.0.0/16</route>
61    </host>
62  </vnuml>
```

A.2 XML File Scenario Figure 6.11

```
1
2  <?xml version="1.0" encoding="UTF-8"?>
3  <!DOCTYPE vnuml SYSTEM "/usr/share/xml/vnuml/vnuml.dtd">
4
```

```
<vnuml>
  <global>
    <version>1.8</version>
    <simulation_name>Test_2</simulation_name>
    <automac/>
    <vm_mgmt type="none" />
    <vm_defaults>
       <filesystem type="cow">/usr/share/vnuml/filesystems/
          root_fs_tutorial</filesystem>
       <kernel>/usr/share/vnuml/kernels/linux</kernel>
       <console id="0">xterm</console>
    </vm_defaults>
  </global>

  <net name="Net0" mode="uml_switch"/>
  <net name="Net1" mode="uml_switch" type="ppp">
        <bw>1000000</bw>
  </net>
  <net name="Net2" mode="uml_switch" type="ppp">
        <bw>1500000</bw>
  </net>
  <net name="Net3" mode="uml_switch" type="ppp">
        <bw>750000</bw>
  </net>

  <vm name="E1">
    <if id="1" net="Net0">
      <ipv4>10.0.0.2</ipv4>
    </if>
    <if id="2" net="Net1">
```

```
34        <ipv4>10.0.1.1</ipv4>
35      </if>
36      <if id="3" net="Net2">
37        <ipv4>10.0.2.1</ipv4>
38      </if>
39      <route type="ipv4" gw="10.0.2.2">10.0.3.0/24</route>
40      <forwarding type="ip"/>
41    </vm>
42
43    <vm name="E2">
44      <if id="1" net="Net1">
45        <ipv4>10.0.1.2</ipv4>
46      </if>
47      <if id="2" net="Net3">
48        <ipv4>10.0.3.1</ipv4>
49      </if>
50      <route type="ipv4" gw="10.0.1.1">default</route>
51      <forwarding type="ip"/>
52    </vm>
53
54    <vm name="E3">
55      <if id="1" net="Net2">
56        <ipv4>10.0.2.2</ipv4>
57      </if>
58      <if id="2" net="Net3">
59        <ipv4>10.0.3.2</ipv4>
60      </if>
61
62      <route type="ipv4" gw="10.0.2.1">default</route>
63    </vm>
```

```
64
65    <host>
66      <hostif net="Net0">
67         <ipv4>10.0.0.1</ipv4>
68      </hostif>
69      <route type="ipv4" gw="10.0.0.2">default</route>
70    </host>
71  </vnuml>
```

Appendix B

Numeric Tests Results

B.1 Iperf Measurements Figure 6.10

In this table we detailed the results obtained during the measurement with Iperf in a load network.

Time(s)	Packets (kBytes)	Capacity	Load	Load+Capacity	Time(s)	Packets (kBytes)	Capacity	Load	Load+Capacity
0	59,3	945	0	945	50,5	28,3	463	600	1063
0,5	57,1	961	0	961	51	17,4	285	600	885
1	58	950	0	950	51,5	18,4	301	600	901
1,5	56,6	927	0	927	52	19,3	316	600	916
2	60,8	996	0	996	52,5	28,7	470	600	1070

Time(s)	Packets (kBytes)	Capacity	Load	Load+Capacity	Time(s)	Packets (kBytes)	Capacity	Load	Load+Capacity
2,5	59,4	973	0	973	53	25,5	417	600	1017
3	55,1	904	0	904	53,5	17,4	285	600	885
3,5	59,4	973	0	973	54	18,4	301	600	901
4	60,8	996	0	996	54,5	21,1	346	600	946
4,5	58	950	0	950	55	29,7	487	600	1087
5	56,6	927	0	927	55,5	24,4	401	600	1001
5,5	55	901	0	901	56	18,4	301	600	901
6	48,5	794	200	994	56,5	18,8	308	600	908
6,5	44,2	725	200	925	57	18,4	301	600	901
7	49,9	818	200	1018	57,5	27,8	455	600	1055
7,5	44,7	733	200	933	58	27,3	447	600	1047
8	48,9	801	200	1001	58,5	18,4	301	600	901
8,5	43,3	710	200	910	59	18,8	308	600	908
9	47,1	771	200	971	59,5	18,4	301	600	901
9,5	48,5	794	200	994	60	25,5	417	600	1017
10	44,2	725	200	925	60,5	29,1	477	600	1077
10,5	46,1	755	200	955	61	18,4	301	600	901
11	45,7	748	200	948	61,5	19,7	323	600	923
11,5	46,1	755	200	955	62	18,4	301	600	901
12	48,1	788	200	988	62,5	21,6	354	600	954
12,5	44,2	724	200	924	63	27,3	447	600	1047
13	47,1	771	200	971	63,5	24	394	600	994
13,5	45,7	748	200	948	64	18,4	301	600	901
14	45,7	748	200	948	64,5	17,8	292	600	892

Time(s)	Packets (kBytes)	Capacity	Load	Load+Capacity	Time(s)	Packets (kBytes)	Capacity	Load	Load+Capacity
14,5	47,1	771	200	971	65	19,8	324	600	924
15	47,1	771	200	971	65,5	29,1	477	600	1077
15,5	45,2	740	200	940	66	28,3	463	600	1063
16	48	786	200	986	66,5	22,6	371	600	971
16,5	44,2	725	200	925	67	23,5	385	800	1185
17	49,4	809	200	1009	67,5	20,7	339	800	1139
17,5	41,4	679	200	879	68	14,1	232	800	1032
18	49,9	818	200	1018	68,5	14,1	232	800	1032
18,5	47,1	771	200	971	69	17,4	285	800	1085
19	41,4	679	200	879	69,5	18,4	301	800	1101
19,5	50,3	824	200	1024	70	17,4	285	800	1085
20	43,8	718	200	918	70,5	11,3	185	800	985
20,5	47,5	778	200	978	71	7,07	116	800	916
21	46,1	755	200	955	71,5	18,4	301	800	1101
21,5	45,1	739	200	939	72	17,8	292	800	1092
22	48,5	794	200	994	72,5	0	0	800	800
22,5	43,3	710	200	910	73	19,8	324	800	1124
23	48,5	794	200	994	73,5	18,4	301	800	1101
23,5	42,8	702	200	902	74	0	0	800	800
24	48,9	801	200	1001	74,5	17,8	292	800	1092
24,5	48,5	794	200	994	75	18,4	301	800	1101
25	42,8	702	400	1102	75,5	0	0	800	800
25,5	36,8	602	400	1002	76	18,4	301	800	1101
26	33,4	547	400	947	76,5	12,6	207	800	1007

Time(s)	Packets (kBytes)	Capacity	Load	Load+Capacity	Time(s)	Packets (kBytes)	Capacity	Load	Load+Capacity
26,5	36,2	593	400	993	77	2,83	0	800	800
27	40	655	400	1055	77,5	18,4	301	800	1101
27,5	32,5	533	400	933	78	8,48	139	800	939
28	29,6	485	400	885	78,5	9,9	162	800	962
28,5	31,5	516	400	916	79	14,1	232	800	1032
29	38,6	632	400	1032	79,5	4,24	69,5	800	869,5
29,5	38,6	632	400	1032	80	15,6	255	800	1055
30	31,5	516	400	916	80,5	11,3	185	800	985
30,5	29,7	487	400	887	81	5,66	92,7	800	892,7
31	34,3	563	400	963	81,5	17	278	800	1078
31,5	38,1	624	400	1024	82	7,07	116	800	916
32	35,8	586	400	986	82,5	6,39	105	800	905
32,5	28,7	470	400	870	83	15,6	255	800	1055
33	32,5	533	400	933	83,5	7,07	116	800	916
33,5	35,8	586	400	986	84	11,3	185	800	985
34	37,7	617	400	1017	84,5	12,7	209	800	1009
34,5	37,2	609	400	1009	85	9,34	153	800	953
35	30,1	493	400	893	85,5	11,3	185	800	985
35,5	31,5	516	400	916	86	11,3	185	800	985
36	32,5	533	400	933	86,5	9,9	162	800	962
36,5	36,2	593	400	993	87	25,5	417	800	1217
37	40,4	663	400	1063	87,5	56,6	927	0	927
37,5	30,6	501	400	901	88	58	950	0	950
38	32,9	540	400	940	88,5	59,4	973	0	973

Time(s)	Packets (kBytes)	Capacity	Load	Load+Capacity
38,5	31,1	510	400	910
39	40	655	400	1055
39,5	34,3	563	400	963
40	33,4	547	400	947
40,5	30,1	493	400	893
41	31,5	516	400	916
41,5	39,1	640	400	1040
42	35,8	586	400	986
42,5	32,9	540	400	940
43	33,4	547	400	947
43,5	31,5	516	400	916
44	32,5	533	400	933
44,5	38,6	632	400	1032
45	35,8	586	400	986
45,5	34,8	571	400	971
46	24,4	401	600	1001
46,5	27,3	447	600	1047
47	28,3	463	600	1063
47,5	19,3	316	600	916
48	25,9	424	600	1024
48,5	28,3	463	600	1063
49	17,4	285	600	885
49,5	18,4	301	600	901
50	26,8	439	600	1039

Time(s)	Packets (kBytes)	Capacity	Load	Load+Capacity
89	58	950	0	950
89,5	56,6	927	0	927
90	60,8	996	0	996
90,5	58	950	0	950
91	56,6	927	0	927
91,5	60,8	996	0	996
92	57,9	948	0	948
92,5	56,6	927	0	927
93	60,8	996	0	996
93,5	56,8	930	0	930
94	60,2	986	0	986
94,5	57	933	0	933
95	58,3	956	0	956
95,5	58,4	957	0	957
96	58,4	957	0	957
96,5	57,8	947	0	947
97	58,4	957	0	957
97,5	58,8	963	0	963
98	57,9	948	0	948
98,5	58,4	957	0	957
99	58,4	957	0	957
99,5	58,8	963	0	963
100	57,9	948	0	948

References

[1] BitTorrent Protocol Specification. [online] http://www.bittorrent.org/protocol.html, Accessed on September 2009.

[2] eMule Homepage. [online] http://www.emule-project.net, Accessed on September 2009.

[3] Gnutella. [online] http://rfc-gnutella.sourceforge.net, Accessed on September 2009.

[4] Kazaa media deskto. [online] http://www.kazaa.com, Accessed on September 2009.

[5] Miro. [online] http://www.getmiro.com, Accessed on September 2009.

[6] News and events, Triversa. [online] http://www.tiversa.com/news/, Accessed on September 2009.

[7] NTTCP: New TTCP program. [online] http://www.leo.org/~elmar/nttcp/, Accessed on September 2009.

[8] Ogg streamer. [online] http://dir.visionair.tv/streamer.php, Accessed on September 2009.

[9] Peer-to-peer in 2005. [online] http://www.cachelogic.com/research/2005_slide07.php, Accessed on June 2006.

[10] Peercast: P2p broadcasting for everyone. made available under gnu general public license. [online] http://www.peercast.org, Accessed on September 2009.

[11] Shoutcast Homepage. [online] http://www.shoutcast.com, Accessed on September 2009.

[12] Test TCP (TTCP) benchmarking tool for measuring TCP and UDP performance. [online] http://www.pcausa.com/Utilities/pcattcp.htm, Accessed on September 2009.

[13] Top sourceforge downloads. [online] http://sourceforge.net/top/, Accessed on September 2009.

[14] Videolan project. [online] http://www.videolan.org, Accessed on September 2009.

[15] Virtual network user-mode-linux (vnuml). [online] http://www.dit.upm.es/vnuml, Accessed on September 2009.

[16] YouTube Homepage. [online] http://www.youtube.com Accessed September 2009.

[17] ISO/IEC 14496-10. Advanced video coding. *Information technology – Coding of audio-visual objects – Part 10*, 2005.

[18] F. Qin J. Dugan A. Tirumala, M. Gates and J. Ferguson. NLANR/DAST : Iperf - the TCP/UDP bandwidth measurement tool, [online] http://dast.nlanr.net/Projects/Iperf/, Accessed on September 2009.

[19] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 562–570, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[20] D. Anderson and J. Kubiatowicz. The worldwide computer. *Scientific American*, **286(3)**:28–35, 2002.

[21] S. Avallone, S. Guadagno, D. Emma, A. Pescapè, and G. Ventre. D-itg distributed internet traffic generator. *Quantitative Evaluation of Systems, International Conference on*, **0**:316–317, 2004.

[22] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. *Technical report, UMIACS TR-2002*.

[23] F. Bustamante and Y. Qiao. Friendships that last: Peer lifespan and its role in p2p protocols. 2003.

[24] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream:high-bandwidth multicast in cooperative environments. *9th ACM Symposium on Operating Systems Principles*, 2003.

[25] D.M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems*, **17(1)**:1–14, 1989.

[26] Y. Chu, S. G. Rao, S. Seshan, and H. Zhang. A case for end system multicast. In *Proceedings of ACM Sigmetrics*, pages 1–12, 2000.

[27] G. Dan, V. Fodor, and I. Chatzidrossos. On the performance of multiple-tree-based peer-to-peer live streaming. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications.*, pages 2556–2560, 2007.

[28] P. de Cuetos and K. Ross. Adaptive rate control for streaming stored fine-grained scalable video. In *NOSSDAV'02, International Workshop Network and Operating and Video , Miami, Florida*, May 2002.

[29] H.h Deshpande, M. Bawa, and H. Garcia-Molina. Streaming live media over a peer-to-peer network. Technical Report 2001-30, Stanford InfoLab, 2001.

[30] H.h Deshpande, M. Bawa, and H.r Garcia-Molina. Streaming live media over peers. Technical Report 2002-21, Stanford InfoLab, 2002.

[31] Z. Despotovic and K. Aberer. P2P Reputation Management : Probabilistic Estimation vs Social Networks. *Computer Networks*, **50**:485–500, 2006.

[32] J. Dike. *User Mode Linux*. Prentice Hall Ptr, 1st edition, April 2006.

[33] J. R. Douceur. The sybil attack. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 251–260, London, UK, 2002. Springer-Verlag.

[34] C. Dovrolis, P., and D. Moore. What do packet dispersion techniques measure? In *Proceedings of IEEE Infocom*, pages 905–914, 2001.

[35] B. Eckel and C. Allison. *Thinking in C++, Volume 2: Practical Programming*. Prentice Hall, us ed edition, 2003.

[36] W. Eddy. TCP SYN flooding attacks and common mitigations, 2007.

[37] J. Ruiz O. Walid F. Galán, D. Fernández and T. de Miguel. A virtualization tool in computer network laboratories. In *5th International Conference on Information Technology Based Higher Education and Training, Istanbul, Turkey*, 2004.

[38] M. J. Freedman and R. Morris. Tarzan: a peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS '02)*, pages 193–206, 2002.

[39] H. Zhang D. Towsley A. Venkataramani G. Neglia, G. Reina and J. Danaher. Availability in bittorrent systems. *Proceedings of IEEE Infocom, Anchor- age, AK, USA*, 2007.

[40] A. Gai. *Structuration en graphe de de Bruijn ou par incitation dans les reseauix pair à pair*. PhD thesis, Institut national de recherche en informatique et automatique (INRIA), 2006.

[41] A. Gai, D. Lebedev, F. Mathieun, F. de Montgolfier, J. Reynier, and L. Viennot. Acyclic preference systems in P2P networks. *Euro-Par 2007 Parallel Processing*, pages 825–834, 2007.

[42] B. Gedik and L. Liu. Peercq: a scalable and self-configurable peer-to-peer information monitoring system. Technical report git-cc-02-32, Georgia Institute of Technology, 2002.

[43] I. Goldberg, D. Wagner, and E. Brewer. Privacy-enhancing technologies for the internet. In *COMPCON '97: Proceedings of the 42nd IEEE*

International Computer Conference, page 103. IEEE Computer Society, 1997.

[44] V. K. Goyal. Multiple description coding: Compression meets the network. *IEEE Signal Processing Magazine*, **18(5)**:74–94, 2001.

[45] A. Rao H. Schulzrinne and R. Lanphie. Real Time Streaming Protocol (RTSP), RFC 2326, 1998. Available at http://www.ietf.org/rfc/rfc2326.txt.

[46] R. Frederick H. Schulzrinne, S. Casner and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications, RFC 3550,, 2003. Available at http://www.ietf.org/rfc/rfc3550.txt.

[47] H. Handschuh, L. R. Knudsen, and M. J. Robshaw. Analysis of sha-1 in encryption mode, 2001.

[48] M. Hefeeda, A. Habib, D. Xu, B. Bhargava, and B. Botev. Collectcast: A peer-to-peer service for media streaming. *ACM Multimedia 2003*, **11**:68–81, 2003.

[49] S. Hemminger. Network emulation with netem. In *Linux Conf Au*, April [online] http://linux-net.osdl.org/index.php/Netem/, Accessed on September 2009.

[50] D. Hughes, G. Coulson, and J. Walkerdine. Free riding on gnutella revisited: The bell tolls. *IEEE Distributed Systems Online*, **6**:2005, 2005.

[51] D. Karger F. Kaashoek I. Stoica, R. Morris and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. 2001.

[52] Information Sciences Institute. Transmission control protocol, RFC 793, 1981. Edited by Jon Postel. Available at http://www.ietf.org/rfc/rfc793.txt.

[53] Information Sciences Institute. User datagram protocol, RFC 768, 1981. Edited by Jon Postel. Available at http://www.ietf.org/rfc/rfc768.txt.

[54] M. Nahas J. Liebeherr and W. Si. Application-layer multicasting with delaunay triangulations overlays. *IEEE Journal on Selected Areas in Communications*, **20(8)**:1472–1488, 2002.

[55] D. Epema J. Pouwelse, P. Garbacki and H. Sips. The bittorrent p2p file-sharing system: Measurements and analysis. *4th International Workshop on Peer-to-Peer Systems (IPTPS), Ithaca, NY, USA*, 2005.

[56] V. Prabhakaran J. Wang, C. Yeo and K. Ramchandran. On the role of helpers in peer-to-peer file download systems: Design, analysis and simulation. 2007.

[57] C. Pu J. Zhang, L. Liu and M. Ammar. Reliable end system multicasting with a heterogeneous overlay networ. Technical report git-cercs-04-19, cercs, Georgia Institute of Technology, 2004.

[58] V. Jacobson and M. J. Karels. Congestion avoidance and control. *Computer Communication Review*, **18(4)**:314–329, 1988.

[59] M. Jain and C. Dovrolis. Pathload: A measurement tool for end-to-end available bandwidth. In *Proceedings of Passive and Active Measurements (PAM) Workshop*, pages 14–25, 2002.

[60] M. E. Johnson and S. Dynes. Inadvertent disclosure: Information leaks in the extended enterprise. In *Proceedings of the 6th Workshop on the Economics of Information Security*, 2007.

[61] C. K. Jonathan, J. Walpole, K. Li, and A. Goel. The case for streaming multimedia with tcp. In *8th International Workshop on Interactive Distributed Multimedia Systems (iDMS 2001)*, pages 213–218, 2001.

[62] S. Rao K. Hildrum, J. Kubiatowicz and B. Zhao. Distributed object location in a dynamic network. *Theory of Computing Systems*, **37**:405–440, 2004.

[63] A. M. Kermarrec, F. Le Fessant, and L. Massoulié. Exploiting semantic clustering in the edonkey p2p network. In *11th ACM SIGOPS European Workshop (SIGOPS)*, pages 109–114, 2004.

[64] M. Kinateder and S. Pearson. A privacy-enhanced peer-to-peer reputation system. In *E-commerce and Web technologies*, pages 206–215. Springer-Verlag, 2003.

[65] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. 2003.

[66] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Using random subset to build scalable network services. 2003.

[67] J. Liang, J. Liang, and R. Kumar. Pollution in p2p file sharing systems. In *IEEE INFOCOM*, pages 1174–1185, 2005.

[68] X. Liao, H. Jin, Y. Liu, L.M. Ni, and D. Deng. Anysee: Peer-to-peer live streaming. *Proceedings of IEEE Infocom*, 2006.

[69] P. Linga, A. Crainiceanu, J. Gehrke, and J. Shanmugasudaram. Guaranteeing correctness and availability in p2p range indices. In *Cornell Technical Report*, pages 323–334, 2005.

[70] V. Paxson M. Allan and W. Steven. TCP congestion control RFC 2581, 1999. Available at http://www.ietf.org/rfc/rfc2581.txt.

[71] B. Botev D. Xu M. Heffeeda, A. Habib and B. Bhargava. Promise: peer-to-peer media streaming using collectcast. *Proc. ACM Multimedia (MM'03), Berkeley, CA*, 2003.

[72] L. Massoulie, A. Twigg, C. Gkantsidis, and P. Rodriguez. Randomized decentralized broadcasting algorithms. In *Proc. of IEEE INFOCOM '07*, 2007.

[73] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information sys tem based on the xor metric. 2002.

[74] J. Mirkovic and P. Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *ACM SIGCOMM Computer Communication Review*, **34**:39–53, 2004.

[75] N. Naoumov and K. Ross. Exploiting p2p systems for ddos attacks. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, page 47, 2006.

[76] G. Neglia, H. Zhang, and Do. Towsley. A network formation game approach to study bittorrent tit-for-tat. Technical report, Towsley –EuroFGI International Conference on Network Control and Optimization, 2007.

[77] T. W. Ngan, D. S. Wallach, and P. Druschel. Enforcing fair sharing of peer-to-peer resources. In *2nd Int. Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[78] A. Nicolosi and S. Annapureddy. P2pcast: A peer-to-peer multicast scheme for streaming data. *1st IRIS Student Workshop*, 2003.

[79] V. N. Padmanabhan, H. J. Wang, P. A. Chou, and K. Sripanidkulchai. Distributing streaming media content using cooperative networking. In *Proceedings of the 12th international workshop on Network and*, pages 12–14, 2002.

[80] J. Mogul H. Nielsen R. Fielding, J. Gettys and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1, RFC 2068, 1997. Available at http://www.ietf.org/rfc/rfc2068.txt.

[81] R. Rejaie and A. Ortega. Pals: Peer-to-peer adaptive layered streaming. 2003.

[82] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for largescale peer-to-peer systems. *IFIP/ACM Intl. Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany*, 2001.

[83] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notidication infrastructure. 2001.

[84] M. Handley R. Karp S. Ratnasamy, P. Francis and S. Schenker. A scalable content-addressable network. 2001.

[85] S. Sanghavi, B. Hajek, and L. Massoulie. Gossiping with multiple messages. In *IEEE Transactions on Information Theory*, pages 4640–4654, 2007.

[86] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of MMCN*, 2002.

[87] M. Schiely and P. Felber. Crossflux: An architecture for peer-to-peer media streaming. *Emerging Communication: Studies on New Technologies and Practices in Communication*, 2006.

[88] H. Schulzrinne and S. Casner. RTP profile for audio and video conferences with minimal control, RFC 3551, 2003. Available at http://www.ietf.org/rfc/rfc3551.txt.

[89] R. Sherwood, B. Bhattacharjee, and A. Srinivasan. P5: A protocol for scalable anonymous communication. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 58, Washington, DC, USA, 2002. IEEE Computer Society.

[90] S. Sheu, K. A. Hua, and W. Tavanapong. Chaining: A generalized batching technique for video-on-demand systems. In *ICMCS '97: Proceedings of the 1997 International Conference on Multimedia Computing and Systems*, page 110, Washington, DC, USA, 1997. IEEE Computer Society.

[91] J. Shneidman and D.C. Parkes. Specification faithfulness in networks with rational nodes. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 88–97, 2004.

[92] K. Cheung Sia. Ddos vulnerability analysis of bit-torrent protocol. Technical report, UCLA, 2006.

[93] A. Singh, T. W. Ngan P., Druschel D. S., and Wallach. Eclipse attacks on overlay networks: Threats and defenses. pages 1–12, 2006.

[94] K. Sripanidkulchai. The popularity of gnutella queries and its implications on scalability. *Technical report, Carnegie Mellon University*, 2001.

[95] D. A. Tran, K. A. Hua, and T. Do. A peer-to-peer architecture for media streaming. *IEEE Journal on Selected Areas in Communications, Special Issue on Service Overlay Networks*, 2004.

[96] Y. Wang and M. Claypool. An empirical study of realvideo performance across the internet. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop*, pages 295–309, 2001.

[97] D. Xu X. Jiang, Y. Dong and B. Bhargava. Gnustream: A p2p media streaming prototype. *Proc. the 2003 IEEE International Conference on Multimedia and Expo (ICME'03)*, 2003.

[98] Z. Xu, M. Mahalingam, and M. Karlsson. Turning heterogeneity into an advantage in overlay routing. In *Proceedings of INFOCOM*, pages 1499–1509, 2003.

[99] D. Wu Y. Tian and K.-W. Ng. Modeling, analysis and improvement for bittorrent-like file sharing networks. 2006.

[100] K. Zerfiridis and H. Karatza. Large scale dissemination using a peer-to-peer network. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 421, Washington, DC, USA, 2003. IEEE Computer Society.

[101] X. Zhang, J. Liu, B. Li, and T. Yum. Coolstreaming/donet: A data-driven overlay network for efficient peer-to-peer live media streaming. *Proceedings of IEEE Infocom*, 2005.

[102] B. Y. Zhao, Y. Duan, L. Huang, A. Joseph, and J. Kubiatowicz. Brocade: Landmark routing on overlay networks. In *IPTPS*, **2429** of *Lecture Notes in Computer Science*, pages 34–44. Springer, 2002.