Daedalus: A media agnostic peer-to-peer architecture for IPTV distribution

ATHANASIOS MAKRIS
and
ANDREAS STRIKOS

KTH Information and
Communication Technology

Master of Science Thesis
Stockholm, Sweden 2008

COS/CCS 2008-11

Daedalus: A media agnostic peer-to-peer architecture for IPTV distribution

Athanasios Makris & Andreas Strikos

June 23, 2008

Examiner
Prof. Gerald Q. Maguire Jr.

Supervisor
Andreas Ljunggren

Department of Communication Systems (CoS)
School of Information and Communication Technology (ICT)
Royal Institute of Technology (KTH)
Stockholm, Sweden

Abstract

IPTV is gaining increasing attention. It is an expanding field where a lot of people are working hard to solve the problems that delay its wide-spread use. One major problem is that the existing IPTV distribution mechanisms do not seem to work well when applied on a large scale. Especially, IP multicast does not seem to meet the requirements of highly demanding IPTV services. In contrast, peer-to-peer architectures for distributing content have been available for a number of years (since the late 1990's), and their success suggests that this is a promising alternative means of distributing content. Although peer-to-peer architectures are well known for file transfer, this kind of architecture has been used in this thesis for distributing streaming video flows. We combine results from two different approaches - IPTV and peer-to-peer systems - as part of our design and implementation of a new solution for distributing IPTV. Our proposal aims to avoid any weaknesses that the existing solutions have, whilst offering a viable solution for distributing live content.

Sammanfattning

Intresset kring IPTV ökar hela tiden, och många människor arbetar på att lösa de problem som hindrar området från att växa snabbt. Ett av huvudproblemen är att den existerande IPTV-distributionstekniken inte fungerar bra då den appliceras på stora lösningar. Bland de största problemen är att IP-Multicast inte möter de krav som marknaden ställer på global distribution av material. I motsats till detta har peer-to-peer teknik, som funnits sedan 90-talet, visat sin styrka för fil distribution på en mycket global skala på existerande infrastruktur. I denna uppstats kombinerar vi dessa två områden för att utröna vilka möjligheterna som finns för att optimera kostnaden för distrubition av live-tv samtidigt som vi försöker att undvika de svagheter som normalt associeras med de olika arkitekturerna. Vårt mål är att utnyttja de bästa egenskaperna från de olika teknikerna för att skapa en livsduglig och långsiktig lösning för TV-distribution.

Acknowledgements

First of all, we would to thank our Professor Gerald Q. Maguire Jr., our academic supervisor and examiner in KTH, for all his support and his essential comments throughout the different parts of the project. His continuous guidance was helpful so as to broaden our scientific horizons.

We would also like to thank Andreas Ljunggren and Robert Skog for giving us the opportunity to work on this interesting topic within Ericsson AB. We were given the opportunity to work in a state of the art laboratory and gain valuable experience.

Last but not least, we would like to thank our own people for their continuous support not only during this project, but for all these years of our studies. A great thank you to everyone.

Athanasios Makris & Andreas Strikos
Stockholm, June 23, 2008

Contents

List of Tables

List of Figures

Abbreviations

ALM	Application Level Multicast
DHT	Distributed Hash Table
DMOs	DirectX Media Objects
DONet	Data-driven Overlay Network
EPG	Electronic Program Guide
ESM	End System Multicast
GMPLS	Generalized Multi Protocol Label Switching
HDTV	High Definition Television
HTTP	Hypertext Transfer Protocol
ICD	Intelligent Content Distribution
ID	Identifier
IGMP	Internet Group Multicast Protocol
IP	Internet Protocol
IPTV	Internet Protocol Television
ISP	Internet Service Provider
LSPP	Live Streaming over Peer-to-Peer Protocol
MI	Media Injector
MN	Master Node
MPEG	Moving Picture Experts Group
MSOMP	Multi-sender based Overlay Multicast Protocol
MSRRA	Multi-sender based Redundancy Retransmitting Protocol
NAT	Network Address Translation
PC	Personal Computer
PIM-DM	Protocol-Independent Multicast Dense Multicast
PIM-SM	Protocol-independent multicast Sparse Multicast
PVR	Personal Video Recorder
QoS	Quality of Service
RP	Rendezvous Point
RTT	Round Trip Time
SIM	Scalable Island Multicast
TCP	Transport Control Protocol
TV	Television
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VoD	Video on Demand
VoIP	Voice over Internet Protocol

Chapter 1

Introduction

Since 1969 when the first message was sent over the ARPANET (Advanced Research Projects Agency Network), a tremendous evolution of networking, based upon internetworking, has taken place; leading to the Internet as we know it today. The Internet's growth and wide adoption changed the world, especially the field of telecommunications. Traditional telecom services moved from traditional circuited switched architectures to packet switching, which together with a common Internet network protocol are two of the key ideas behind Internet. The increasingly wide-spread adoption of Voice over Internet Protocol (VoIP) has changed the traditional telecommunications industry, both in a way which few anticipated and at a rate that many have trouble adapting to.

Today, traditional broadcast television systems are changing from analog to digital transmission. However, this is happening against a background where the number of over the air viewers of programs has decreased, where young people are more interested in interactive gaming and other opportunities which were not previously available (hence they are spending less time watching traditional broadcast television), where increasing amounts of user generated content is competing with TV production studios and others for viewers attention, and video content has become just another media which is carried over IP networks. Part of this evolution in television is referred to as **I**nternet **P**rotocol **T**ele**V**ision (IPTV); in this approach digital video and audio content is transfered from a content distributor to viewers. IPTV has now become mainstream and many Internet Service Providers (ISP) are offering IPTV, often in conjunction with broadband Internet connectivity and Voice over IP – as a so-called "triple play" service (IP, telephony, and TV).

Extensive research has been conducted in order to achieve reliable multimedia transmission over Internet. The most common approach used, has been to utilize simple file transfer; however, this does not provide real-time or near-real-time distribution. Fortunately, many users are not interested in (near) real-time content and they have enthusiastically adopted file transfer based schemes as an alternative to physical transport of DVDs or other media to transfer video content. Meanwhile (near) real-time video content is being transmitted in current systems via IP multicast. Multicast transmission offers a one-to-many media delivery mechanism that transmits a single stream over each link of the network only once, creating copies only when the paths to the destinations split. However, this architecture suffers from many problems; of which scalability is the most significant. Another problem has been the resistance of ISPs to provide multicast in their infrastructures (even when they have routers which are multicast capable). As a result, another technology called Application Level Multicast (ALM) was proposed as a means to circumvent the need for having multicast enabled in the routers along each path. ALM moves the multicast service from the IP layer (routers) to the application layer (end users). This solution provides higher scalability and is easier to deploy because there is no need for changes in the network operators' infrastructure. In addition, users can decide to deploy ALM even if ISPs do not support IP layer multicast. However, ALM cannot efficiently address the requirements of *live* multimedia transmission. As bandwidth and latency are considered the most important metrics when it comes to live multimedia transmission, and the constraints that should be satisfied are very stringent (see section 2.1 where these constraints will be described); therefore we are interested in alternatives that can provide enough bandwidth while at the same time offer low latency.

In the late 1990's a new type of network architecture was introduced, called peer-to-peer. This model of communication differs from the traditional client-server model in that all peers are equal and provide both client and server functionality. Today, peer-to-peer systems are designed to be highly scalable, self-configurable, and self-sufficient. Because of these characteristics, peer-to-peer systems were considered as a possible solution to achieve reliable multimedia content distribution. Previously, the peer-to-peer architecture has been used primarily for file transfer applications. During the past four years, there have been several attempts to combine video streaming and peer-to-peer architectures (see section 2.3). Some of these attempts have been successful; but there still remains a number of open issues. For example, asymmetric link bandwidth has been common in the fixed broadband service offered to private users. This asymmetry means that data can flow in one direction at a much higher speed than in the other (at least when considering single links). For many ADSL subscribers this ratio might be four to one or even greater (with the downlink generally having much greater available bandwidth than the uplink). Thus, we have to find out how to exploit this difference between what a node attached to such a link can offer versus what it might demand. It should be noted that this asymmetry was often introduced purposely, as it was assumed that the private end user would want to consume much more than they would produce (while business could be charged a higher price for "symmetric DSL" service - as they might have significant outgoing traffic).

Additionally, when it comes to IPTV, there is a special characteristic that complicates the situation; the need for live transmission. Fulfilling the requirements introduced by this requirement, while avoiding buffer starvation at the receiver and avoiding other problems, is thought to be a very challenging task. Buffer starvation occurs when the buffer of the media player at the receiver, has no more data to feed the player. This generally happens because something went wrong during the transmission of the media stream, thus it occurs when no packets have arrived for a period of time longer than the playout time of the buffered contents (hence this buffer is often referred to as a **playout buffer**). The size of this buffer is a component in the maximum delay between when content arrives at the receiver and when it can be viewed/played, as this buffer stores the incoming packets until they are played. The buffer is also called a de-jitter playout buffer, since it also needs to buffer sufficient traffic that given the expected jitter, there is always content to be played out. Hence, its size is proportional to the maximum jitter which is to be expected and which is to be hidden.

We argue in this thesis that peer-to-peer systems offer an alternative to multicast and that this alternative is an efficient way to deal with live multimedia streaming over the Internet because of the system's capability to quickly adjust to a dynamically changing environment. We will revisit the proof of this claim in chapter 5. In this thesis, we propose an innovative architecture for IPTV distribution of (near) real-time content based on peer-to-peer systems. The resulting systems will be used to deliver live multimedia content to end users efficiently and reliably. We will examine this claim of efficiency and the claim for reliability with our measurements in chapter 4.

The thesis is organized as follows:

Chapter 2 The present IPTV concept and traditional architectures that have been used to deliver media content to end users are presented. We present the peer-to-peer architecture and try to justify why we consider it the most appropriate solution for live multimedia streaming. Current proposals, both academic and commercial, are presented and reviewed. Finally, the motivations that lead to the design of our proposed architecture are presented.

Chapter 3 The architecture of our system is described in detail. We present an in-depth analysis of the protocol and the packet scheduler, both from a theoretical and implementation point of view.

Chapter4 A number of experiments and their corresponding measurements are presented and a detailed description of the methodology and tools used are given.

Chapter 5 An evaluation of our system's performance in comparison to existing proposals (as previously presented in Chapter 2) is presented.

Chapter 6 Concludes our thesis and proposes future work that should be done.

We hope our solution will not only contribute to the widespread deployment of IPTV, but will constitute a reference point which is valuable to the reader as well.

Chapter 2

Background

In this chapter we present some background regarding what others have done in the field of IPTV. First, we describe the evolution of IPTV from initial attempts in 1994 to current systems, such as Joost [1]. Traditional architectures such as IP multicast and ALM will be elaborated as well. Next, peer-to-peer systems are presented. For each of the alternatives, their historical evolution, categorization, and their advantages and disadvantages will be discussed. The reasons for adopting a peer-to-peer architecture for live multimedia streaming via the Internet will be presented. Related work is reviewed and the most important and widely known systems will be described. Finally, we conclude our discussion and present the motivations that leads to the proposed new architecture for real-time media distribution for IPTV.

2.1 IPTV

***I**nternet **P**rotocol **T**elevision (IPTV) distributes digital video and audio content via the use of the Internet Protocol (IP)*. IPTV is a convergence of communication, computing, and content [2]. Content in the case of IPTV could be proprietary content that is being send from a content producer via a content distributor to a (specific) set of users; hence only these users should be able to view this content. This is comparable to the case of a broadcast or cable TV operator who distributes content only to their subscribers. IPTV can utilize two-way communication between the distributor and the users, such that the content which is transferred from the distributor to the user depends on users's preferences and their agreement(s) with the content distributor. Fig. 2.1 shows a complete view of an IPTV network.

IPTV has been available for some time via a number of broadband network operators. Until recently, however, it has mainly been offered in small networks or in parts of networks. It should be noted that in Sweden there is no longer any analog TV broadcasting, hence viewers have a choice of DVB-T, cable TV, satellite digital TV, or IPTV. In the next few years IPTV services will increase the traffic over broadband access networks as shown in the estimates in Fig. 2.2. According to Lawrence Harte [3], there are three key reasons why people and companies are adding or converting their existing broadcast television systems to television over data networks:

More channels IPTV channels are routed through data networks and this allows providers to offer an increased number of channels (almost without limit). While cable TV and satellite TV operators have a limit on the number of broadcast channels.

More control IPTV viewers can choose from multiple providers, rather than being restricted to local providers. The customer can add and remove features when they desire and they can setup their own personalized IPTV service through a web page.

More services The increasing extent of the user's control and the increase in number of channels drastically increases the variety of potential services and their type. Additionally, due to the two-way

communication that IPTV can offer, these services can be designed to satisfy a wide variety of users and hence a large portion of the users.

The alternative media used for distributing television nowadays are: (1) terrestrial digital radio network that broadcasts television over the air, (2) cable television (using coaxial cables and/or fibers), and (3) satellite broadcast. Although both cable and satellite television operate very similarly (as both are primarily "push" technologies"), IPTV is a different technology with regard to how the system distributes the video and audio content. IPTV represents a "push & pull" technology. This means that the operator transmits video content based on requests ("pull") from the users. A request is received by the IPTV operator, which "pushes" the audio-video stream to the user. Because a single audio-video stream flows only in response to a request, bandwidth consumption occurs only when there is a subscriber who wants to see and hear this specific content. In contrast, cable and satellite systems broadcast many channels at the same time, regardless of there are viewers for this content or not. The biggest difference from today's distribution of television is the sharp increase in control of content distribution by both the users and the distributors.

Although many existing networks have been built to support triple-play services, upgrading to support IPTV services (live multimedia streaming) often represents a major challenge. IPTV introduces the possibility of a new media experience that is not restricted only to passively viewing content, but rather allows end users to interactively watch and participate in (potentially) fully personalized interactive content. Distribution over an IP network makes it easy to deliver many new features, since not only traffic can be sent in both directions, but traffic only needs to be sent where there is a need to communicate and this communications can be scaled to the amount which is actually needed. The bidirectional communication character of IPTV gives end-users the ability to select what to see and when to see it; without needing to use any extra components - such as DVD recorders, VCRs, PVRs, etc. This two way communication enables IPTV to support many different services, such as those that Gilbert Held describes in [5]:

- Homeowner entertainment
- On-demand video
- Business TV to the desktop
- Distance learning
- Corporate communications
- Mobile phone television
- Video chat

To this list of applications we can add Electronic Program Guide (EPG), Personal Video Recorder (PVR), and features such as pause, fast forward, and rewind. Of all the aforementioned applications, homeowner entertainment seems likely to experience the most significant growth over the next few years. This is especially true with the end of traditional analog terrestrial broadcast television, as consumers are now forced to spend money to shift to some alternative media.

IPTV content can be viewed on a multimedia computer, a standard television using an adapter, or on a dedicated IPTV [3]. A multimedia computer is a data processing device that is capable of using and processing multiple forms of media and has access to Internet services. A multimedia computer can be a computer or a mobile telephone with multimedia and Internet capabilities. An IP Set Top Box is an electronic device that adapts IPTV content into a format that is accessible by the end user [3]. Using such set top boxes,

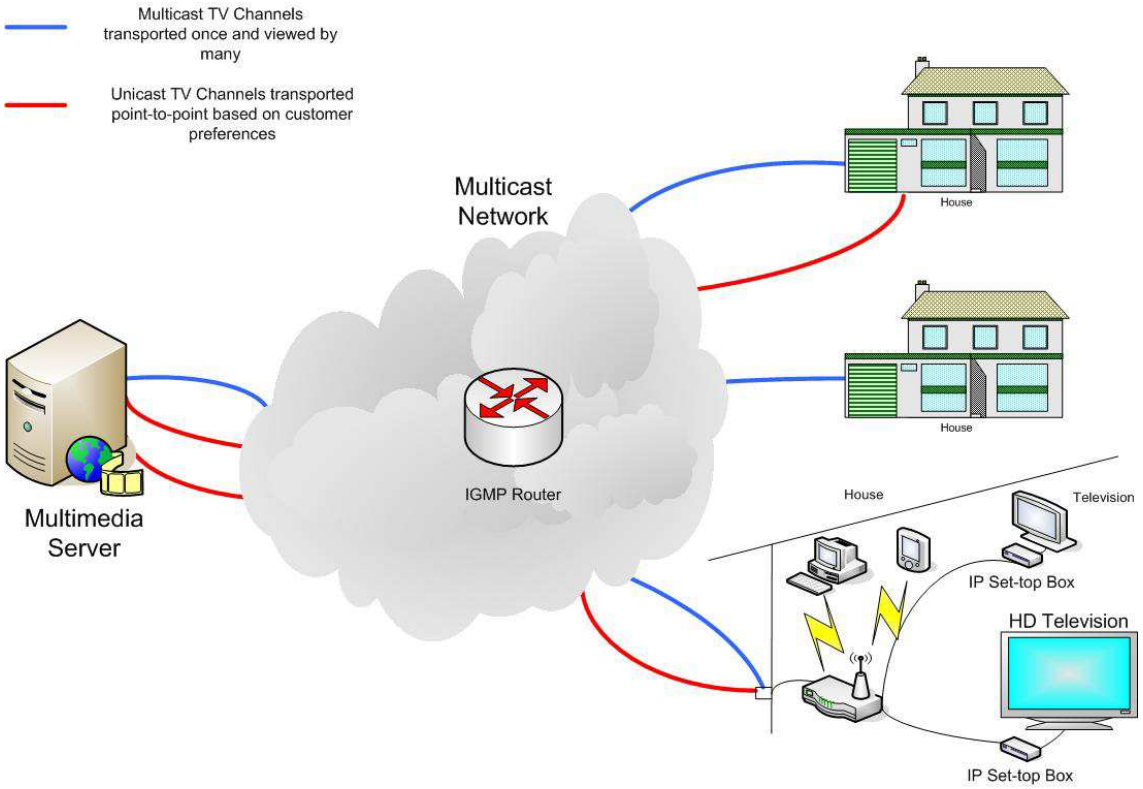


Figure 2.1: IPTV Network Overview.

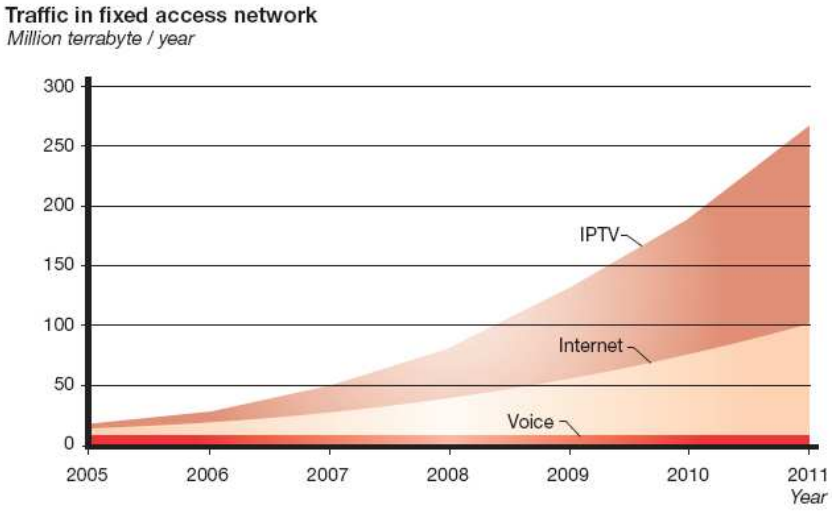


Figure 2.2: Traffic increase in broadband networks worldwide (Ericsson Strategic Forecast, under author's permission). [4]

it is possible to use standard televisions for viewing television programs that are sent over a data network. IPTVs are display devices that are specifically designed to receive and decode television content from IP networks without the need for adapter boxes or media gateways [3]. The major difference between IPTV and the alternative television broadcast mediums is that a very large variety of video content, including live streams, can be shown on any IP attached device which has the matching CODEC including TVs, PCs, and phones.

In order to be able to meet IPTV's requirements, a network must address the following issues:

Quality of Service(QoS) End users prefer to have continuous content playout without any stops or fluctuations in the quality (to the degree that this is feasible). This is one of the major challenges that IPTV must deal with.

Ability to separate the traffic Due to interactive and personalized content the network must be able to separate the traffic among the users efficiently. The different services also have different streams, which must be properly supported in order to be fully functional.

Security and billing The system must ensure that everyone receives the correct content. The billing system must also be designed to operate correctly and accurately.

Scalability The infrastructure should be able to operate despite updates, upgrades, or additions of services, that will add extra load to the system.

IPTV is a very demanding service and requires every node in the network to provide continuous bandwidth and IP control mechanisms in order to address the above issues. Streaming flows can be transmitted by using IP multicast or IP unicast. IPTV video content may be transmitted from a media server or a television gateway directly to a specific end user (unicast) or it may be multicast to hundreds or even thousands of end-users at the same time.

Multicast Multicast transmission refers to an one-to-many delivery mechanism that transmits a single stream over each link of the network only once, creating copies only when the paths to the destinations split. Fig. 2.3 depicts the IP multicast mechanism. There are two major types of multicasts [6]: (1) dense multicast (such as Protocol-Independent Multicast Dense Multicast, PIM-DM) and, sparse multicast (such as Protocol-Independent Multicast Sparse Multicast, PIM-SM). PIM-DM is designed for multicast LAN applications, while the PIM-SM is for wide area, inter-domain network multicasts where there may be few (or no users) in a given domain. In order to take advantage of multicasting the IPTV network must support **I**nternet **G**roup **M**ulticast **P**rotocol (IGMP), which is used by the end users or hosts (e.g. IP set top box) to join or leave multicast streams. For efficient multicasting, IGMP should be implemented throughout the network and as close as possible to the end-users [4]. IGMP is the control mechanism used to control the delivery of multicast traffic to interested and authorized users. For multicast services to operate, routers along the network path have to be multicast capable and their firewalls have to be configured properly to allow multicast packets. This is not always true or easy, since ensuring this currently requires many agreements between different companies. If the network being used for the IPTV service is controlled by a single company, then all the nodes within the network can be setup and controlled - hence easily realizing multicast routing (this will generally include support for IGMP at least near the leaves of the network). Networks capable of multicast transmissions is increasingly likely to be using lots of IP switches (which today are generally IGMPv3 capable). Depending on the architectural choices made by the network operator, IGMP control occurs in the Digital Subscriber Line Access Multiplexer (DSLAM), an aggregation switch, or at an edge router [7]. Thus IGMP should be supported for all of these choices.

Unicast Unicast transmission refers to the delivery of data to only one host attached to the network. Unicast is used when the media server wants to send streaming content to a single client. Unicast is not necessarily bandwidth efficient, but of course is supported throughout the Internet (unless explicitly blocked by firewalls, Network Address Translators (NATs), etc.). It is anticipated that a large fraction of services, driven by interactivity and personalization of the IPTV service, will be delivered as unicast traffic. As a consequence, according to Peter Arberg, et al. it will become necessary to distribute such content closer to the end users, in order to minimize the traffic load on the core network [4].

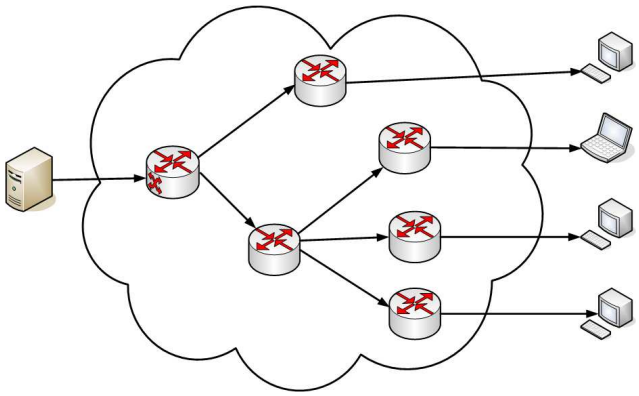


Figure 2.3: IP Multicast Architecture.

Multicast networks are often combined with Generalized Multi Protocol Label Switching (GMPLS). GMPLS is designed to support broadband and IPTV services, controlling all the layers, including network layer, and physical layer; often by manipulating optical network resources, with a separation of media data plane and control plane [8], [9].

Network capacity is strongly coupled to service penetration (as greater service penetration required increased network capacity), but it is also a function of the network's architecture. Currently, IP multicast has scalability problems when a large number of users and groups is reached, and it is not yet accepted globallly for being a solution to Internet-wide multicast [6]. According to Stuart Elby, vice president of network architecture and enterprise technology of Verizon Communications Inc. [10] nobody has ever built an IP multicast network on a large scale before and that is why pioneering companies - such as Verizon - in the IPTV field has not chosen to utilize a multicast architecture [11]. The necessity of multiple agreements between the different network companies in the case of multicast makes multicast less viable as an IPTV solution. In order to support all the customers' needs and meet their expectations the broadband network must examine alternative architectures beyond the existing multicast mechanisms, which as we said before has some scalability problems.

2.2 Peer-to-peer systems

Androutsellis-Theotokis and Spinellis define peer-to-peer systems as: "Peer-to-peer systems are distributed systems consisting of interconnected nodes able to self-organize into network topologies with the purpose of sharing resources such as content, CPU cycles, storage and bandwidth, capable of adapting to failures and accommodating transient populations of nodes while maintaining acceptable connectivity and performance, without requiring the intermediation or support of a global centralized server or authority." [12] This definition constituted our base to design our proposed architecture.

If we go back in time, one of the first peer-to-peer systems was the ARPANET. The ARPANET was a peer-to-peer network based upon interface message processors which self-organized, adapted to failures, and accommodated transient nodes (and node failure). Since then, peer-to-peer technology evolved tremendously. Today, peer-to-peer systems generate more than 60% of the total Internet traffic. The main reason for their wide adoption is that users want content that is available on specific network nodes. The location of the content can be discovered based upon logical overlay networks. A user can join a peer-to-peer network and begin exchanging information with other peers within minutes. File sharing was the "killer" application that boosted the adoption of peer-to-peer systems. Today, new areas for peer-to-peer networks are starting to appear. One of the most interesting, because of its commercial importance, is live multimedia streaming and especially IPTV. However, a lot of problems have to be overcome in order to achieve scalability, reliability, and user satisfaction. In order to be able to fully exploit the advantages and suppress the disadvantages of peer-to-peer systems, we will describe all the possible variations of them and present their main characteristics.

The first categorization of these peer-to-peer systems is made according to their degree of centralization. Here, we have identified two categories: pure and hybrid.

Pure peer-to-peer systems do not rely on any central entity for managing the network or routing traffic. Peers are equals and each of them provides the functionality of both server and client. Gnutella [13] is a typical pure peer-to-peer file sharing system. There is no central database that knows all of the files available on the Gnutella network. Instead, the machines on the network learn about the location of files using a distributed query approach. In Fig. 2.4, the procedure for finding a song in the Gnutella network is depicted.

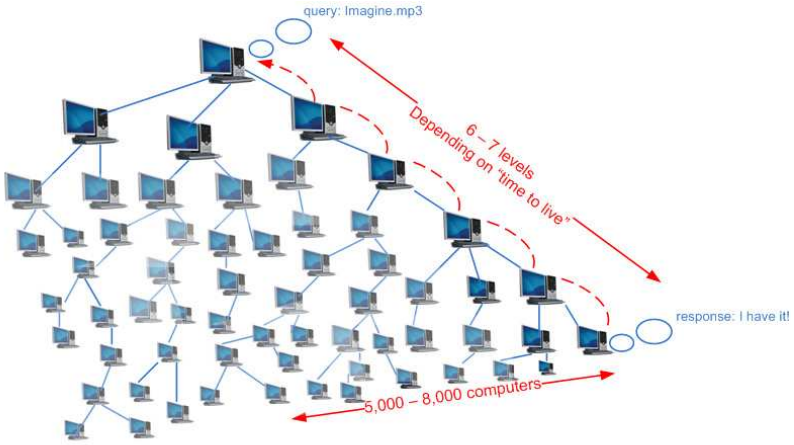
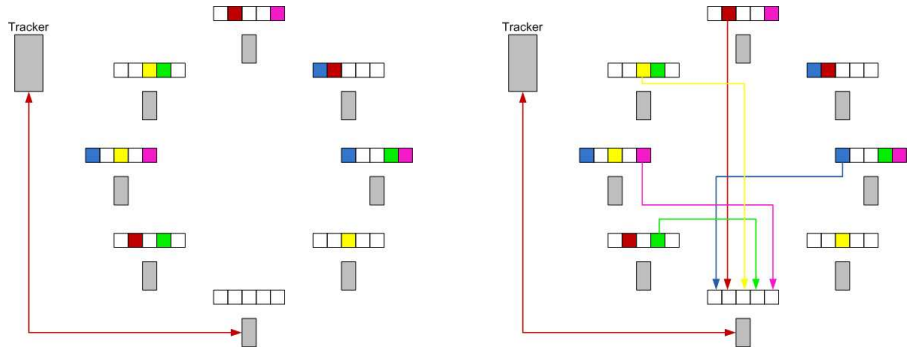


Figure 2.4: How a Gnutella client finds a song.

By merging the two worlds of centralized systems (client-server architecture) and pure peer-to-peer systems, *hybrid* approaches were introduced. Napster [14] and BitTorrent [15] are two of the most popular hybrid peer-to-peer systems. As mentioned in [15], "BitTorrent is a protocol for distributing files. It identifies content by URL and is designed to integrate seamlessly with the web. Its advantage over plain HTTP is that when multiple downloads of the same file happen concurrently, the downloaders upload to each other, making it possible for the file source to support very large numbers of downloaders with only a modest increase in its load". In this protocol, only information **about** the file is kept in a server called a *tracker*. Each user connects to the tracker, gets the appropriate meta-information (a file with .torrent extension), i.e., information about a file, and starts downloading it from the sources specified in the meta-information. An important point here is

that one can use a search engine to locate the .torrent file for a given content item, then download this content from one of many or from several of the many places which have this content. A graphical representation of this procedure is depicted in Fig. 2.5.



(a) Initial contact to the tracker. **(b)** Transfer of a file

Figure 2.5: BitTorrent functionality - based on fetching different parts of the file from potentially multiple copies of this part of the file.

A special category of hybrid peer-to-peer systems is illustrated by Kazaa [16]. Kazaa introduces a special category of nodes, called *SuperPeers*. SuperPeers contain some of the information that other nodes may not have. Thus, peers typically lookup information at SuperPeers if they cannot find it otherwise.

Peer-to-peer systems can also be categorized based on their structure as follows: structured and unstructured. In *structured* systems, peers create topologies of trees or graphs. The challenging task here is to create and maintain these topologies. On the other hand, once the structure is formed, discovery and downloading of a file is fast and reliable. However, problems arise when peers join and leave the system frequently because the structure should be updated. When a peer is expected to leave, tree grafting operations are performed, but when it leaves unexpectedly, the structure should be destroyed and built from the beginning. Fig. 2.6 presents tree structured peer-to-peer system.

On the other hand, in *unstructured* (often called mesh) approaches, overlay links are established arbitrarily. Here, each node contacts a subset of peers to obtain parts of a file. The problem in these architectures is the large overhead induced in the network in order for all peers to know about the existence and availability of the others. The reason to learn about a large number of other peers is to be able to choose the best of them to get the file from. A graphical representation of a mesh topology is depicted in Fig. 2.7.

As we mentioned earlier, peer-to-peer networks have a lot of interesting characteristics. First, they are

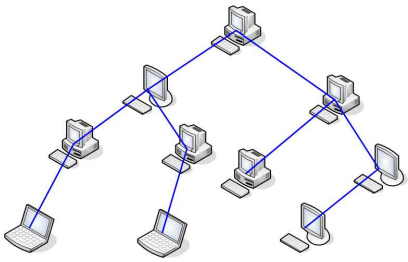


Figure 2.6: A tree structured peer-to-peer system.

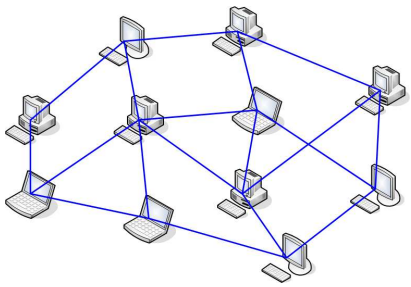


Figure 2.7: An unstructured peer-to-peer system.

self-configurable, self-adapted, and self-organized. Second, they provide an overlay network such that applications running on top of this network can rapidly and dynamically form communities of nodes which share the same interests. The formation of a community is driven by social networking. Third, there is high redundancy as there is no single point of failure. Of course, if there is only one copy of the content in the network, a risk still exists that it may become unavailable. However, this risk is quite unlikely to happen in a peer-to-peer network that contains multiple copies of the content. In addition, a possible bottleneck around a central server is avoided because of the network's distributed architecture. Finally, most peer-to-peer networks provide dynamic management of the available resources in the network since the availability of the resources scales with the number of active peers in the systems.

Extended research has been conducted both form the academic and the commercial sections to introduce the peer-to-peer concept into live IPTV distribution field. However, early implementations presented in the next section, do not manage to fully fulfill the requirements needed for an IPTV distribution architecture (see section 2.1). In our own research, presented in Chapter 3, we propose a new peer-to-peer architecture and show how it fulfills these requirements.

2.3 Related Work

In this section, we will present and describe a number of live streaming systems based on peer-to-peer architectures. We are going to restrict our description to only **streaming** systems since our system is planned for video streaming and therefore this is the case we are interested in. We are going to describe not only proposals originating from the academic community, but commercial ones as well. These approaches can be classified into two main categories: structured and unstructured. In structured architectures, nodes are organized into trees or graphs, while in unstructured architectures nodes form meshes in order to communicate.

2.3.1 Structured systems

As we mentioned above, in structured architectures the peers are organized into trees or graphs. Thus, the challenging tasks are:

Scalability The node join/departure procedure should affect the tree structure as little as possible, as significant changes in the topology may result in service interruption and decreased streaming quality.

Service interruption In the event of a node failure the system should be able to recover quickly in order to minimize service interruption (e.g. bad audio-video quality or lack of video altogether).

Start-up delay When a new node joins the system (i.e. a user switches on their output display), this device should be able to get the media stream quickly. This means that the viewer should be able to see the content of the stream within 1-2 seconds. Otherwise, he/she will not use the application. This restriction is even more important in the case of channel change in which the transition from the old channel to the new one should be almost instantaneous.

Efficiency Mechanisms used to manage the communication among nodes should not generate large overhead in the system. Although overhead is not a problem for the fixed (probably fiber based) infrastructure, it is very critical when it comes to Mobile IPTV. A mobile user that generates multimedia content and wants to broadcast it, should not be subjected to large overhead as a wireless link has limited capacity.

Peer Selection A common issue in both structured and unstructured systems is how to select the best available peers to receive the media stream. An appropriate solution should guarantee that a peer chooses the best available peers to receive the media stream from, but it provides this functionality with the least possible traffic overhead and minimum delay.

Media Stream Reception Method The method by which a peer will receive the media stream from the other peers is another common problem between structured and unstructured systems as well. Two methods are generally proposed: the pull method and the push method. In the pull method, when a peer needs a certain fragment, it requests it from another peer based on a specific algorithm. In the push method, on the other hand, a peer subscribes to get the fragments it needs to get from a neighbor peer based on a specific algorithm. When this peer receives fragments that match the subscription pattern of another peer, it will push them directly to it.

The following approaches try to address these issues mainly by using sophisticated algorithms for the management of the tree structure and heuristics in order to be able to locate peers that can transmit the media stream while attempting to fulfill certain QoS goals.

- **BulkTree**

  BulkTree is a proposed peer-to-peer architecture used for live media streaming [17]. It can be classified as a structured approach because it uses a tree structure for the organization of the nodes. The innovative idea underlying this proposal is the use of sets of nodes, called *Super-Nodes*, instead of single nodes in traditional tree-based systems. The main intelligence of the system lies in the algorithms used to create the Super-Node structure and maintain it, as simple nodes join or leave the network. The protocol used for the communication among Super-Nodes can be flooding or server based, which makes it very simple to implement. For communication between Super-Nodes and peers multicast is used. Apart from the basic algorithms proposed, some optimizations are considered as well, in order to increase the efficiency of the algorithms and provide faster media delivery to new peers. Through simulations of different scenarios, it is shown that organizing nodes into Super-Nodes increases the robustness of the system, reduces the alteration time of the structure, and decreases the average link delay time. Alteration time of the structure is defined as the time needed for the structure to reconstruct after a node arrival or departure.

- **Peer-to-peer video streaming framework with adaptive server playback rate**

  Luan, et al. [18] investigated the problem of asymmetric peer connections. Specifically, because the majority of users in a peer-to-peer network have asymmetric connections, a problem arises due to the lack of symmetry in bandwidth between demand and supply. This problem can cause decreased video quality as a peer may suffer from buffer starvation. To address this situation, a fully distributed system with two key features is introduced. First, a special link-level (given that links connect nodes in a tree structure) homogenous overlay network is formed. In this network, all the overlay links attempt to have an identical bandwidth. If this is possible and this bandwidth is sufficient, then video flowing through the overlay links will not encounter any bottlenecks, thus peers can achieve the guaranteed streaming rates. Second, the streaming server can adjust the video playback rate in an adaptive way by locally observing the peer streaming rate(s). Thus, peers can fully utilize their bandwidth in order to achieve the best video quality possible given their locally constrained bandwidth(s).

- **P2broadcast**

  P2broadcast [19] tries to improve three inefficiencies of *Application Level Multicasting (ALM)* architectures. In ALM the multicast service is provided by the application layer instead of the network

layer as in traditional IP multicasting. This offers compatibility with current IP networks and provides greater flexibility in customizing the network in order to achieve application-specific needs. For the interested reader more details about ALM can be found in references [1-13] in [19]. Most of the proposed ALM solutions use a bandwidth first scheme in order for a newcomer peer to find the most appropriate parent to get a copy of the media stream from. In order to achieve this, the overlay tree is traversed to estimate the available bandwidth to potential peers. The peer that is selected as parent is the one that offers the highest bandwidth.

In Fig. 2.8 the join process for the bandwidth first scheme is presented. First, the newcomer N sends a JOIN message to the server S in order to join the network (Fig. 2.8a). S and its child peers (P_1, P_2 and P_3) estimate available bandwidth to N (Fig. 2.8b). The measured available bandwidth is indicated by the numbers on the dash arrows. In this scenario, the peer P_2 can provide the greatest available bandwidth for N. Therefore, P_2 is the candidate parent peer in the next round. Next, P_2 and its child peers (P_6 and P_7) measure the available bandwidth to N (Fig. 2.8c). As we can see from the figure, P_2 still provides the greatest available bandwidth for N. Thus, P_2 sends a WELCOME message to N (Fig. 2.8d) and the newcomer N selects the peer P_2 as its parent peer. At this point, the join process is completed.

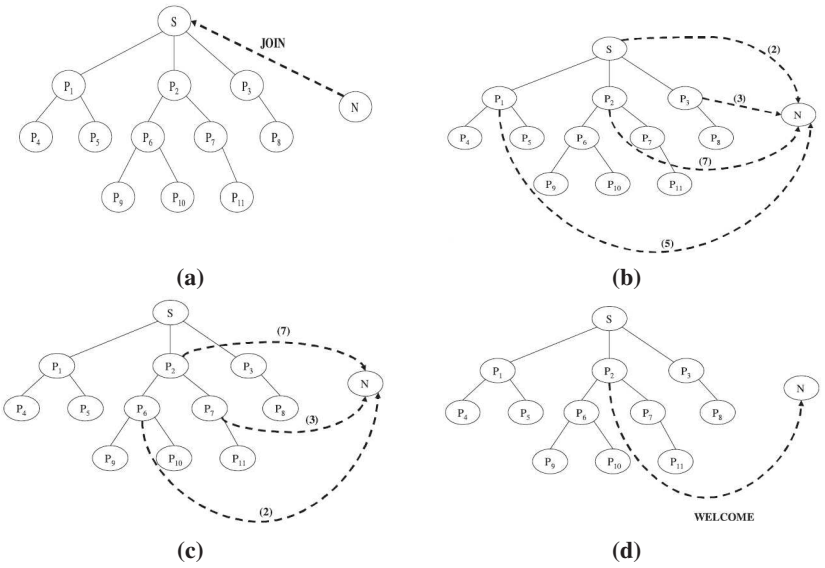


Figure 2.8: Join process of the bandwidth first scheme. [19]

Bandwidth first scheme has three main problems. First, because the overlay tree in a bandwidth first scheme can be very deep and the bandwidth estimation techniques are quite time consuming, a long start-up latency can occur as more peers join the network. Second, poor availability of the system can occur because of the nature of the algorithm used to traverse the overlay tree. The basic idea of this algorithm is to traverse only a part of the tree and check each potential parent peer to see if it has sufficient bandwidth to support the newcomer. If this is not the case, the new peer will be rejected. However, this algorithm does not traverse all the tree, and thus the possibility of peers existing (but un-discovered) that would have the necessary bandwidth to support the newcomer is quite high. Of course, only new peers joining the system are affected by this phenomenon. However, if we consider that when a user changes channel, it leaves the peer-to-peer system (of the old channel) and joins again (to the new channel), hence the rate of peers joining the system could be very high. Finally, there exists

a high probability of service interruption as peers may leave the network or fail at any time, causing service interruption to their "children".

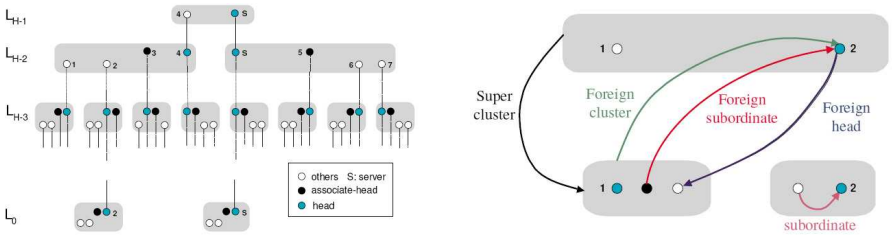
P2broadcast reduces the start-up latency and increases the system availability by organizing peers into a hierarchical cluster structure. This structure enables a newcomer to quickly identify potential parent peers by performing a "traceroute" from a few potential peers in the overlay network. Based on the round trip time (RTT) extracted from each "traceroute", the newcomer can predict which of the potential parent peers have adequate bandwidth to support it. As a consequence, time consuming procedures for bandwidth estimation are avoided and the start-up delay is reduced. Simulations showed that P2broadcast achieves about 66% reduction in start-up delay compared to bandwidth first scheme. Furthermore, P2broadcast tries to reduce the probability of service interruption by considering not only the available bandwidth, but the depth of the newcomer in the overlay network as well. A cost function of these two parameters is introduced in order to evaluate which peer is the most appropriate to become the parent for the newcomer. This optimization leads to the construction of a short-and-wide overlay tree in which the average depth of peers is shorter than in the pure bandwidth first scheme. Thus, the probability of service interruption can be reduced. Simulations suggest that this yields a 10% decrease compared to bandwidth first scheme.

As will be described in chapter 3, our proposal differs from P2broadcast as we utilize a mesh-based architecture and not a tree-structured one. Thus, we do not suffer from high start-up delays or poor availability as the newcomer peer always contacts one of the Master nodes to get all the information it needs for the rest of the peers. In addition, because of the design of the protocol if a node departs for any reason, then a new potential node to connect to is found immediately (see section 4.4) without the need to exchange messages or effort to discover routines.

- **ZigZag**

  ZigZag [20] is another peer-to-peer based approach for live streaming. It tries to alleviate three main problems that occur in large-scale streaming systems using multicast: "(1) the network bandwidth bottleneck at the media source; (2) the cost of deploying extra servers, which could be incurred in content distribution networks; and (3) the infeasibility of IP Multicast on the current Internet". In order to achieve this, ZigZag introduces two important entities: the *administrative organization* (Fig. 2.9) and the *multicast tree* (Fig. 2.10). The first represents the logical relationships among the peers and the latter represents the physical relationships among them. Specifically, the administrative organization consists of organizing receivers into a hierarchy of clusters and building a multicast tree on top of this hierarchy according to a set of specific rules, called C-rules. In each cluster there exist two logical entities: a head which is responsible for monitoring the memberships of the cluster and an associate-head responsible for transmitting the actual media content to cluster members. The introduction of these two entities in the architecture provides increased reliability and redundancy. If the associate-head fails, then the head of its children, if is still working, will help them reconnect to another parent peer quickly. On the other hand, if the head fails, it will not affect the transmission of the stream to the members of the cluster and further actions regarding failure recovery will take place in order to assign a new head.

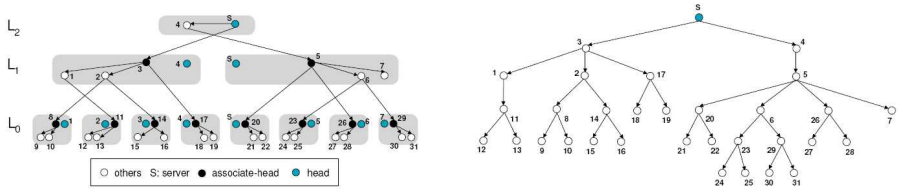
  Besides the two aforementioned mechanisms, ZigZag also introduces a control protocol to maintain the structure and coherence of the multicast tree. In this protocol, each node periodically exchanges control messages with the rest of the peers in its cluster, its children and its parent on the tree. Furthermore, several algorithms are proposed that should be executed when a new node joins the network or an existing node leaves it. Mechanisms to maintain a limited size of each cluster are introduced as well. These mechanisms are responsible for splitting a cluster if the number of peers inside it exceeds a certain number, or merging clusters that consist of too few peers.



(a) Administrative organization of peers.

(b) Relationships among clusters and peers.

Figure 2.9: Administrative organization of peers and relationships among them. [20]



(a) The multicast tree atop the administrative organization.

(b) The multicast tree in another perspective.

Figure 2.10: The multicast tree atop the administrative organization and its view in another perspective. [20]

Finally, a load balancing algorithm is proposed in order to achieve better quality of service. Based on this algorithm, service load is balanced by considering not only the number of children per peer but their bandwidth capacity as well. Specifically, it tries to equalize among peers a ratio computed using the fan-out degree and the bandwidth available to each peer.

A basic difference between our approach and ZigZag, as will be described in section 3.1, lies in the number of peers from which each peer receives the media stream. In ZigZag a peer receives the media stream only from its associate-head peer; while in our approach each peer receives parts of the media stream from different peers. In addition, in our approach we do not introduce either a tree structure or require multicast communication.

- **Scalable Island Multicast (SIM)**

  Jin, et al. propose a fully distributed protocol called *Scalable Island Multicast (SIM)* which integrates IP multicast and ALM [21]. The main idea behind this approach is that even though global multicast is not possible in today's Internet, there exist many local networks that are multicast capable. Such networks are known as multicast "islands". Therefore, ALM protocols should exploit these islands in order to increase their efficiency, as IP multicast in more efficient than ALM itself. In SIM, the communication among peers inside an island is done though IP multicast, while the communication among islands is unicast. The SIM protocol provides two main mechanisms: a mechanism for the construction of overlay trees and a mechanism to integrate IP multicast whenever is possible. The latter mechanism composes two main functionalities: formation of multicast groups and island detection. It is shown that SIM outperforms traditional ALM approaches such as Narada [22] and Overcast [23].

- **Narada and End System Multicast**

  Chu, et al. in [22] define *End System Multicast (ESM)* as a scheme in which "multicast related features, such as group membership, multicast routing and packet duplication, are implemented at end systems, assuming only unicast IP service". Such a system introduces duplicate packets on physical links and

larger end-to-end delays than traditional IP multicast. However, as we mentioned earlier in section 2.1, IP multicast has a lot of problems and it is not deployed widely in today's Internet. So, the performance of an ESM system is of interest, because such a system can be deployed on a large scale almost immediately because only unicast connections are needed. However, the ability of this scheme to construct good overlay structures based on limited topological information should be evaluated. In [22], all these issues are investigated though a proposed protocol called *Narada*.

Narada is a fully distributed, self-organizing, and self-improving protocol that constructs and maintains an overlay structure among participating end systems without requiring any native multicast support. It can also handle failures of end systems or dynamic changes in group membership (nodes joining and/or leaving the network). To achieve this, it uses information regarding the network path characteristics obtained by the end systems either though passive monitoring or active measurements. Based on this information, it continuously refines the logical network topology.

Fig. 2.11 illustrates the differences between IP multicast, unicast, and ESM. In Fig. 2.11a the physical topology of the network is presented. *A*, *B*, *C*, and *D* are end systems, and *R*1 and *R*2 are routers. The numbers depicted illustrate link delays. Let us assume that link *R*1-*R*2 represents a costly transcontinental link and that *A* wants to send a media stream to all the other nodes. In Fig. 2.11b unicast is presented the media stream is sent separately to all the other nodes. An IP multicast tree is depicted in Fig. 2.11c, while an overlay tree constructed using the ESM architecture is presented in Fig. 2.11d. Here, it is shown how the amount of information transmitted over the costly *R*1-*R*2 link is reduced compared to unicast while keeping the same efficiency with IP multicast without the need for additional configuration to routers.

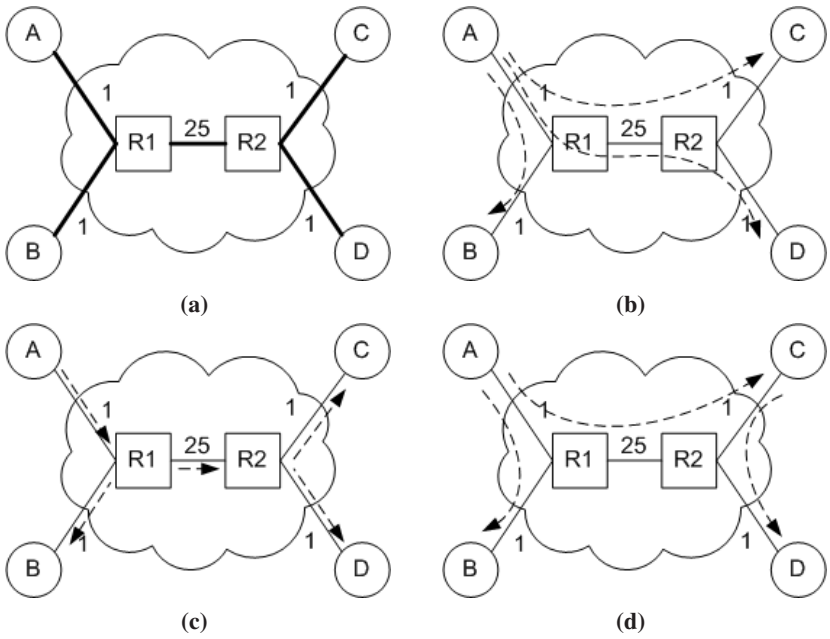


Figure 2.11: Example to illustrate unicast, IP multicast, and End System Multicast. [22]

In Narada, data is delivered through overlay spanning trees. In contrast to other approaches that construct these trees directly, Narada introduces a two-step process: First, it constructs a non-optimized graph called a *mesh*. Non-optimized in that each node is connected with more nodes than it really needs to be connected to. Meshes created by Narada have two important performance properties: (1) the path quality between any pair of nodes is comparable to the quality of the corresponding unicast path, and

(2) each peer has a limited number of neighbors in the mesh. The functionality of the Narada protocol can be categorized in terms of:

- group management,
- mesh performance,
- data delivery, and
- application-specific customizations.

Group management is responsible for maintaining the coherence of the the mesh when new peers join the network or leave it either on purpose or because of a failure. The challenging task here is to react to changes in the tree structure by informing the relevant peers as quickly as possible. Narada proposes specific algorithms for the cases of *member join, member leave and failure, and repairing mesh partitions*. In our review, we will not elaborate these algorithms, but a detailed description is given in [22]. However, we have to note two design flaws. First, in the procedure executed when a node joins a group, no bootstrap mechanism is proposed to get the list of group members. Chu, et al. assume that such a mechanism is application specific, but we argue that it is not and in order to achieve minimum delay, a suitable bootstrapping mechanism should be incorporated inside our proposed architecture (see section 3.1). In addition, they assume that when a node fails, it stays in this state and that all members of the group can detect this change in state. We argue that this assumption does not hold, therefore we carefully consider this situation in the design of our protocol (see section 3.2.3).

Mesh performance concerns the time and effort need when constructing optimal meshes. The key idea of such algorithms is the use of heuristics to determine the addition and dropping of links. Such heuristics should provide stability in the system and avoid partitioning.

Data delivery mechanisms provides all tools necessary to construct per-source shortest path spanning trees that will be subsequently used to deliver media to peers. Narada constructs these trees using a distance vector protocol on top of the mesh.

Finally, Narada offers *application-specific customizations*. The key idea behind this, is the use of TCP Friendly Rate Control protocol [24] as the underlying transport protocol on each overlay link in combination with the use of a routing protocol which is based on multiple metrics.

- **NICE**

NICE [25] is a scalable application layer multicast protocol. As Banerjee et al. state in *Scalable Application Layer Mutlicast*: "our goals for NICE were to develop an efficient, scalable, and distributed tree-building protocol which did not require any underlying topology information" [25].

The NICE hierarchy is created by building different levels (layers) with members being the end hosts as it is shown in Fig. 2.12. Layers are numbered sequentially with the lowest layer being layer zero. Each layer has a set of clusters created by the different nodes. Furthermore, each cluster has a cluster leader, which has the minimum maximum distance to all other hosts in the cluster. When a new host joins the multicast group, it has to join a cluster in layer zero. The newcomer contacts a special host, the Rendezvous Point (RP), that all members know of a-priori. Though, it is possible for the RP not to be part of the hierarchy, and the leader of the highest layer cluster to maintain a connection to the RP. The RP responds with a list of the hosts that belong to the highest layer of the hierarchy. The newcomer contacts all members in the highest layer to identify the member closest to itself. The member that is closest responds with a list of the hosts that belong to a layer lower than the previous. By iterating this procedure the newcomer node finds the lowest layer cluster that it should belong.

When a host wishes to gracefully depart the multicast group, it sends a Remove message to all clusters to which it is joined. If a cluster leader departs, then a new leader is chosen for each of the affected

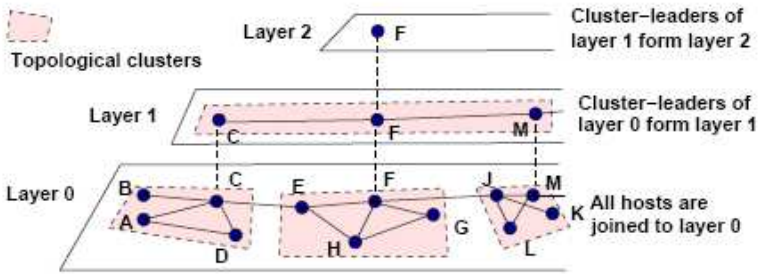


Figure 2.12: Hierarchical arrangement of hosts in NICE. The layers are logical entities overlaid on the same underlying physical network.

clusters. However, if a host fails without sending the Remove message all cluster peers will detect this through non-receipt of a periodic HeartBeat message. The HeartBeat message is sent by every member of a cluster to each of its cluster peers. The message contains the distance estimate of the host to each of the other peers. In the cluster leader's HeartBeat message, that is sent to all the other members, the complete up-to-date cluster membership is sent. A cluster leader periodically checks the size of its cluster, and appropriately splits or merges the cluster when it detects it is either too large or too small.

- **SplitStream**

  Castro, et al. describe their approach as: "SplitStream is an application-level multicast system for high-bandwidth data dissemination in cooperative environments." [26] The key idea underlying SplitStream is to stripe the data across a forest of multicast trees to balance the forwarding load across multiple paths and to tolerate faults. In this way SplitStream tries to distribute across all the peers the responsibility for forwarding data to other peers. As we know, this is not the case in conventional tree-based multicast systems in which only a few peers share the burden to duplicate and forward traffic to other peers. However, concentrating the responsibility for forwarding on only a small number of peers is not an acceptable behavior when dealing with peer-to-peer based systems. Fig. 2.13 illustrates the basic approach of SplitStream.

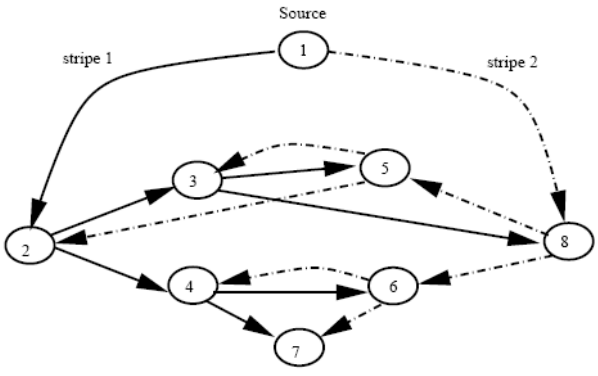


Figure 2.13: A simple example illustrating the basic approach of SplitStream. The original content is splits into two stripes. An independent multicast tree is constructed for each each stripe such that a peer is an interior node in one tree and a leaf in the other [26].

As stated above, the main idea of SplitSteam is to split the media content into several stripes and to

multicast each stripe using a separate tree. This task raises several problems when deciding how to construct this forest of trees, as it must be done in such a way that all the peers will finally get the full content, while there should be no duplicate transmissions and each peer need only forward as much traffic as it can handle. In addition, failure recovery schemes should be introduced in order to provide reliability and error resilience to the system.

In order to solve all the aforementioned problems, SplitStream incorporates algorithms used to build interior-node-disjoint trees, to limit nodes' degree, to locate the best available parent, and to maintain a solid tree structure in the event of node failure or a node leaving the tree. SplitStream runs on top of Pastry [27] and Scribe [28] which are used for tree-based application-level multicast. Pastry is a scalable, self-organizing, structured peer-to-peer overlay network, and Scribe is an application-level group communication system built upon Pastry.

Castro, et al. claim that "the overhead of forest construction and maintenance is modest and well balanced across nodes and network links, even with high churn. Multicasts using the forest do not load nodes beyond their bandwidth constraints and they distribute the load more evenly over the network links than application-level multicast systems that use a single tree." [26] Through simulations it is shown that SplitStream achieves better load balancing by using a significantly larger fraction of the links in the topology to multicast messages compared with the other systems. Specifically, it uses 98% of the links in the topology while IP multicast and centralized unicast use 43%, and Scribe uses 47%. In our approach, we achieve load balancing by using a sophisticated scheduling algorithm which distributes the requests among potential peers based on a ranking system (see section 3.4).

2.3.2 Unstructured systems

In this section, unstructured systems approaches will be presented. The key idea of these proposals is that nodes form meshes in order to communicate and receive the media stream. Because of the nature of mesh systems, fulfilling the stringent requirements for continues bandwidth, short end-to-end delays, and playback continuity is a challenging task. The following approaches propose solutions to these problems.

- **AnySee**

  AnySee[29] has been implemented as an Internet based live streaming system. It adopts an inter-overlay optimization scheme, in which resources can join multiple overlays. By introducing this optimization AnySee aims to improve the existing intra-overlay optimization that other approaches are using. First, peers construct an efficient mesh-based overlay by joining one after another the overlay. The mesh-based overlay manager is responsible for optimizing the overlay, finding the latest neighbors, and eliminating slow connections using algorithms like Location-aware Topology Matching [30]. It is important to let the mesh-based overlay match the underlying physical topology in order to achieve the best possible network performance (logical neighbors, eliminate slow connections, etc.).

  As illustrated in Fig. 2.14, the basic workflow of AnySee is as follows. A single overlay manager is responsible for peers joining/leaving procedures. It is mainly based on traditional intra-overlay optimization, such as Narada [22] and DONet [31]. All streaming paths are managed by the single overlay manager. Each peer maintains one active streaming path set and one back up steaming path set. If either of these two sets looses a number of paths below a threshold, then the inter-overlay optimization algorithm is run to recover the set by adding new paths. The key node manager offers an admission control policy that makes the resources utilization optimal. Finally, the buffer manager is responsible for receiving and transmitting media data among multiple nodes.

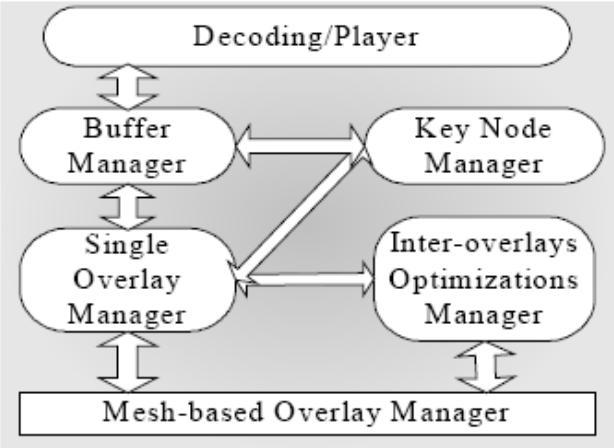


Figure 2.14: System Diagram of each AnySee node. [29]

What makes this system different from others is their inter-overlay optimization. Utilizing their algorithm each peer maintains one active streaming path set and one backup streaming path set. In parallel invalid paths are removed, with the backup set replacing them. This enables high resilience and minimizes the impact of a node failing or leaving. However, it requires additional traffic to create these backup sets, which might never be used. Fortunately, if the basic topology is relatively stable, then the computation of the backup set can proceed at a slow pace, thus reducing its impact both in terms of computation and communication.

- **MutualCast**

  Huang, et al. adapt their earlier MutualCast [32] for peer-to-peer streaming. They propose a scheme with a fully connected mesh that takes advantage of MutualCast and adapts to changing network topology and link traffic loads. Their system is designed to be able to adjust to and utilize multiple bit rate video content. This means that the faster all the peers can upload, the better the quality of the content that each node should receive (up to some limit). In order to achieve this they combine Optimal Rate Control, a control-theoretical framework for quality adaptation, with their MutualCast delivery mechanism. Every peer node with different capabilities distributes different amounts of content. Each block is assigned to one single node for redistribution. This node is responsible for duplicating the specific blocks and distributing it to any node, which asks for it. Of course, nodes with greater bandwidth available to them will be in charge of more blocks than less capable nodes. If the bandwidth of a node changes, MutualCast can easily adapt by assigning less or more (blocks) of the content to each specific node.

  The MutualCast communication scheme can be illustrated using an example shown in Fig. 2.15. The source node has content to distribute, which is chopped in 8 pieces. There are four more peers in the network. Among them only peer t_1, t_2, and t_3 are interested in getting these packets. Peer t_4 is not interested to receive any block but merely assists in the distribution. Taking block five as an example, it is first delivered to peer t_4 and then t_4 redistributes it to t_1, t_2, and t_3.

- **CoolStreaming/DONet**

  CoolStreaming/DONet [31] is a Data-driven Overlay Network for live media streaming. Introduced in 2004, it is one of the very first integrated systems to offer live media streaming via the Internet. DONet utilizes a mesh based protocol and its basic operations is the periodic exchange of data and data

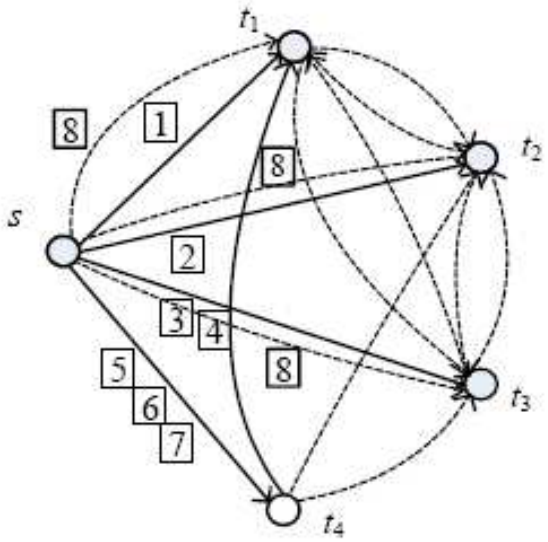


Figure 2.15: Example of MutualCast delivery. [32]

availability information between peers. As depicted in Fig. 2.16, the system has three basic modules: (1) the membership manager, which helps each node maintain a partial view of other overlay nodes and is based on a gossiping-based protocol similar to [33]; (2) a partnership manager, which is responsible for partnership matters among nodes; (3) a scheduler, which schedules the transmitted video data. Data transmission is basically based on the exchange of messages between peers that contain information about the availability of the segments of the video stream. The authors call this representation a Buffer Map. Each node continuously exchanges its Buffer Map with its partners. By keeping in mind the information from the Buffer Map and by applying a heuristic algorithm the scheduler manages the requests and the replies for the video stream segments. The heuristic algorithm first calculates the potential suppliers for each segment. A priority is given to the segments with the fewest potential suppliers and among them the one with the highest bandwidth and enough available time is selected. A request format is implemented with a Buffer Map-like bit sequence. The segments that are requested are marked in the this sequence, which is sent to the supplier. The origin node can also serve as a supplier and if needed can control its payload by advertising more conservative Buffer Maps.

When a node leaves gracefully it sends a departure message to all of the partners to inform them for its pending departure. If the departure is due to a crash, this can easily be detected after an idle time, when the partner that detects the failure it issues a departure message on behalf of the failed node. An example of the partnership in DONet is shown in Fig. 2.17.

- **GridMedia**

  GridMedia [34] is a single source peer-to-peer multicast architecture. Its functionality is built upon two basic operations: (1) multi-sender based overlay multicast protocol (MSOMP) and (2) multi-sender based redundancy retransmitting algorithm (MSRRA). MSOMP builds a mesh-based two layer overlay structure that provides multiple distinct paths from the root to each receiver peer. All the peers are self-organized into a two-layer structure. The lower layer is first created when these peers are grouped into clusters in terms of End-to-End Delay. Every cluster has an upper size limit and has one or several leaders. The leaders of all the clusters are combined to build the backbone of the overlay of the upper layer. Each header peer is responsible for propagating streaming data to the other peers

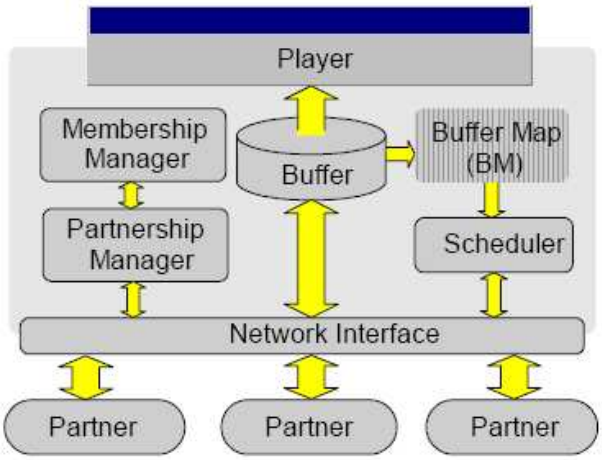


Figure 2.16: A generic diagram for a DONet node. [31]

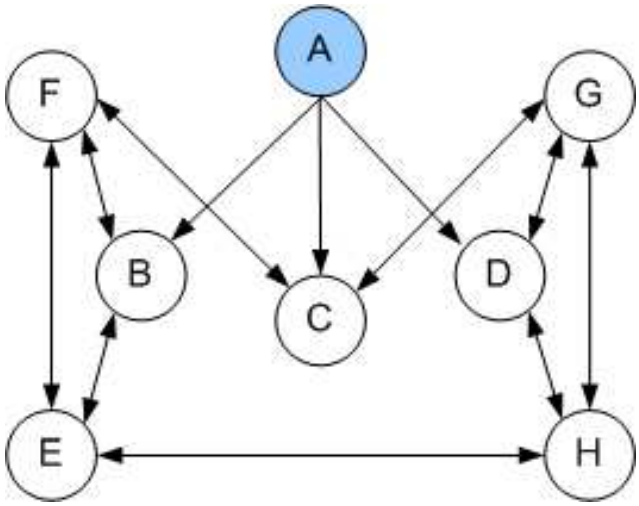


Figure 2.17: Illustration of the partnership in DONet. [31]

of the cluster and it has multiple parents to receive distinct streams simultaneously. When the leader departs a new leader will be elected. The more parents a host has and the more leaders each cluster has, the more independent paths exist from the root to a single node, therefore the more robust the overlay is. An example of the GridMedia architecture based on MSOMP is illustrated in Fig. 2.18.

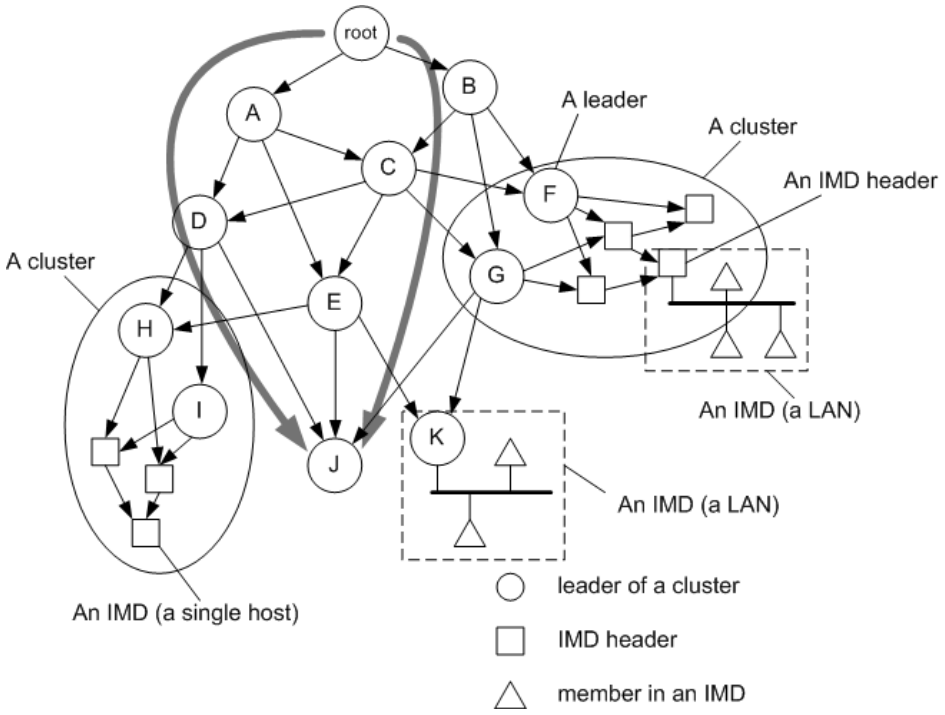


Figure 2.18: GridMedia architecture based on MSOMP. [34]

With MSRRA, each receiver peer obtains streaming packets simultaneously from multiple senders. The streaming content is split into chunks and each peer transmits some of them. As soon as there is congestion or failure in one link, the receiver waits for a time T and it sends a notify to other senders to transmit the lost packets. When the problematic sender recover, then it and continues to send the rest of the requested packets; after this the receiver notifies the other senders to stop transmitting packets that were requested from the recovered sender. Obviously, each sending peer should be provided with information about packets delivered by itself as well as packets sent by other senders.

- **Chainsaw**

Chainsaw [35] is a peer-to-peer pull-based system that does not rely on a rigid network structure. Instead it uses a simple request-response protocol to disseminate the chunks of data that the streaming content is divided into. The protocol is based on gossip-based protocols and BitTorrent. The source node, called a seed, generates packets with increasing sequence numbers. A simple peer can join a set of peers that are called neighbors. Each peer has a list of available data which each of its neighbors has. It also maintains a window of interest, which are the packets this peer is interested in acquiring at the current time. Every time a peer wants to request data from a neighbor it creates a list of desired packets and sends it to the neighboring peer. Peers keep track of requests from their neighbors and send the corresponding packets as the connecting link bandwidth allows. Whenever a peer receives a packet it sends a NOTIFY message to all of its neighbors. Since the seed node does not receive any packets it sends NOTIFY messages whenever it generates new packets.

Chainsaw is an attempt to build a peer-to-peer live streaming system that does not use tree algorithms, but rather uses gossip-based algorithms and was inspired by BitTorrent.

- **PeerStreaming**

  PeerStreaming [36] is a peer-to-peer media streaming system. A peer may act as client as well as server during a streaming session. The design philosophy behind this decision is to ensure that the peer is lightweight and that the peer-to-peer network is loosely coupled. The peer performs simple operations, and may elect to cache only part of the streaming media based on the peer's upload capability.

  The PeerStreaming operation as described by Li in [36] operates as follows; First, the client gets a list of peers that have the requested streaming media. The list can be obtained from a server, from a known peer, or by using a distributed hash table approach. It connects, then to each of these peers and obtains its availability vector, that shows the available data, i.e., that data from the stream which has been stored in the peer. It then retrieves the lengths of the media header and the media structure from one of the peers, calculates their data unit IDs and retrieves them from the peer cluster. Data units are fixed size data units of length L that PeerStreaming creates by breaking the media packet, the media header, and the media structure. Once the media header arrives, the client analyzes the media header, and setups the audio/video decoder DMOs (DirectX Media Object) and the rendering devices in the form of a Microsoft DirectShow filter graph (part of Microsoft's Platform SDK). PeerStreaming client is implemented via the Microsoft DirectShow framework and using DMOs is convenient. For the embedded coded media packets, the streaming bitrate is dynamically adjusted based on the available serving bandwidths and the status of the client queues. Periodically, the client updates the serving peer list, and connects to potential new serving peers.

2.3.3 Commercial systems

This section presents the most popular and well-known commercial solutions in the area of peer-to-peer live streaming. It describes not only their basic functionality, but their internal architecture as well. However, because of their commercial nature, the latter is not always feasible. In these cases we will use information derived through measurements and thorough analysis of traffic patterns generated by these applications.

- **SOPCast**

  SOPCast [37] is commercial peer-to-peer live streaming software that was started as a student project in China. Users can be streaming content consumers as well as producers. Consumers can choose their favorite channel and producers can register a channel and broadcast their streaming data. Since SOPCAst is a closed protocol we do not know much about it. However, Sentinelli, et al. in [38] and Shahzad, et al in [39] determined by experiments the following details about SOPCast:

  - It is a mesh-based protocol.
  - First a SOPCact node contacts a server from which it obtains a list of available channels. After choosing a channel the node receives a list of contacts-peers. When the session is established it exchanges control and data messages with is peers using a gossip-like protocol.
  - SOPCast utilizes UDP traffic for most of its functions.
  - No inter-overlay optimization is performed.
  - Each node buffer is approximately one minute of video.
  - End-to-End security is implemented with encrypted messages.
  - Silverston, et al. in [40] deduce that each SOPCast node switches periodically from one provider peer to another and it seems to always need more than one peer to get video content.

SOPCast was a success almost instantly and it can support more than 100,000 simultaneous users.

- **PPLive**

  PPLive [41] is a free IPTV application which divides video streams into chunks and distributes them via overlays of cooperative peers. The system consists of multiple overlays, with one overlay per channel. Peers are able to download and redistribute live television content from and to other peers. Nevertheless the PPLive network does not own the video content. It permits its users to broadcast their own content. According to [42] and [43] PPLive follows two basic application level protocols: (1) a gossip-based protocol for peer registration and harvesting of partners; and (2) peer-to-peer video distribution protocol.

  The procedure that is followed when a peer wants to join and watch a channel is as follows: (1) it sends out a request to the PPLive channel server that asks for the channel list; (2) after it selects the channel it requests for peer list with the peers that can send it streaming content for this specific channel; (3)it sends out probes to peers on the list to find active peers for its channel.

  Hei, et al. in [43] conclude the following about PPLive after experiments that they have conducted:

  - Peers utilize TCP for both signaling and video streaming.
  - Peers are downloading and uploading video segments from and to multiple peers. Typically a peer receives video from more than one peers at any give moment.
  - During a peer's lifetime it continuously changes its upload and download neighbors.

  With over 100,000 simultaneous users PPLive is considered one of the most popular IPTV application.

- **Joost**

  Joost [1] (also known as The Venice Project) is a peer-to-peer application originally created by the authors of Skype and KaZaA with the aim of delivering (standard) television-quality video on-demand over a peer-to-peer network. Joost operates only with streaming videos that have proper broadcast licence from their creators and it does not permits its users to broadcast their own contents. Thus, all the new content is distributed through Joost's own servers. Joost uses AVC H.264 codecs and is built on MozillaŠs XULRunner engine, so it easily providers cross platform support.

  The procedure that is followed when a new peer wants to watch Joost content is similar to the other related systems (i.e., similar to Skype and KaZaA). First, the client contacts a super-node, which handles control traffic only and direct clients to peers. These peers are renegotiated frequently. As Hall, et al. in [44] deduce from their experiments with Joost "the port number that was negotiated when first connecting to the network is cached and reused. Next, the peer attempts to reconnect to peers from which it has downloaded content previously." The signaling and control traffic between peers and Joost servers is encrypted. Hall et al. also show that the majority of video traffic (at least two-thirds of the content) comes from Joost infrastructure nodes. Their measurements also showed that Joost uses approximately 700 kbps down and 100 kbps up.

  Joost seems to be a partial peer-to-peer system since it does not use the network resources of all the peers efficiently (two-thirds of the traffic comes from Joost's servers [44]).

- **Octoshape**

  Octoshape provides solutions for large-scale live streaming. In [45], *GridCasting* technology is described. It is based on a peer-to-peer tree structure approach and tries to benefit from end users' idle bandwidth to reduce the bandwidth needed for the media source. The key idea is that the live stream transmitted from the media source is used to construct several data streams. The number of

these streams is equal to the number of end users in the system. Data streams should fulfill two very important requirements: (1) none of them should be identical to any other, and (2) combinations of some of them should fully reconstruct the initial media stream. Another innovative idea introduced in Octoshape technology is that when streaming the media content, information about peers joining the network is sent as metadata along with the stream. Thus, each peer builds a list (an addressbook) containing information about the other peers. These mechanisms in combination with algorithms for congestion control, failure recovery, and load balancing enable the development of a large-scale live streaming system.

- **RawFlow**

  RawFlow [46] is another commercial approach for live audio and video streaming through Internet. The key idea behind RawFlow is the *Intelligent Content Distribution (ICD)* technology which creates a seven-layer software-based multicast. The goal is to achieve high scalability in Internet broadcasting. ICD uses grid technology to distribute media streams to peers. The first peers that join the network, get the media stream directly from the media server. However, the next peers joining the network will get the media stream from the other peers and thus, the media server will not transmit any additional information. In order for this scheme to work, buffers in the peers are used to temporarily store parts of the stream. In addition, ICD introduces a monitoring mechanism used to evaluate the status of active connections. As stated in [46], "in case of quality reduction or loss of connection, ICD Client searches the grid for available resources while continuing to serve the client's media player from its buffer".

  In Fig. 2.19, the functionality of ICD is illustrated. Quoting from [46], the following actions are presented:

  1. Encoded content is sent to the media server.
  2. The media server stores and streams content to the end user.
  3. Each user receives a unique stream (unicasting). This results in cumulative bandwidth consumption equal to the number of users multiplied by the connection speed.
  4. With RawFlow, when a user clicks to get the content, a connection is made to a media server running RawFlow's ICD software. The software manages the peering network.
  5. The ICD "server" pulls a single stream from the media server.
  6. The ICD "server" streams the content to the user (A).
  7. The second user (B) joins the broadcast and receives the stream on A's redundant upstream. Multiple users can jointly contribute bandwidth so that every spare kilobit of bandwidth is used most efficiently.
  8. When there is insufficient bandwidth available on the network to support the broadcast, the ICD software instructs the media server to deliver a "booster" stream to the network (C).

2.4 Conclusions

In this chapter we tried to present all the aspects of IPTV. First, we presented the history and evolution of traditional IPTV. Next, we described peer-to-peer live streaming systems and analyzed their different categories. We presented their advantages and disadvantages and justified why they (each) seemed the most rational solution for live streaming through Internet. Then, the most typical approaches, both research and commercial, were presented. Through this analysis we tried to discover the problems and deficiencies in the existing proposals and define potential open issues. Thus, we concluded that even though extensive research has occurred, there are still open issues and technical challenges - as per [47] and [48] these open issues include:

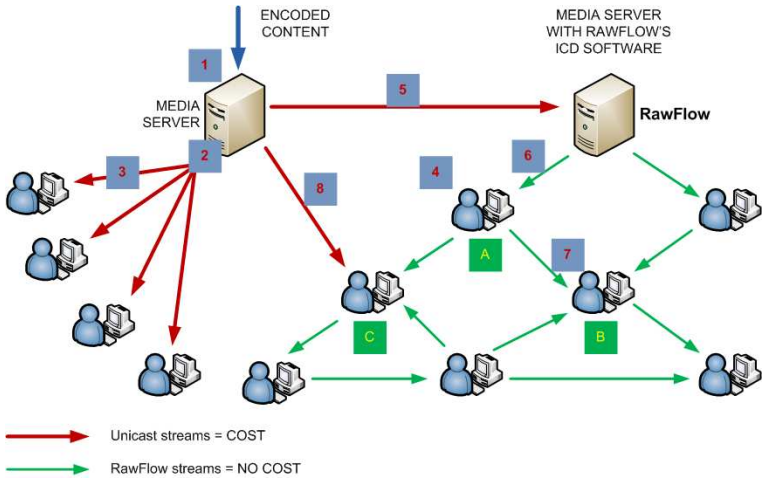


Figure 2.19: RawFlow Intelligent Content Distribution Model. [46]

- **Appropriate video coding scheme**

  Meddour, et al. state that in order to achieve reliable multimedia transmission, sophisticated video coding schemes should be used [47]. However, we argue that it is not necessary and a media agnostic architecture can provide reliable transmission as well. This is one of the basic assumptions we make in our proposal (see chapter 3).

- **Managing peer dynamics**

  Generally, in a peer-to-peer system, nodes join and/or leave the system unpredictably without notifying the other nodes. This behavior should be managed appropriately in order to achieve guaranteed video quality to end users. In our proposal, we introduce a mechanism in which each peer notifies other peers when it joins or leaves a group (see sections 3.2.2 and 3.2.3). In case of failure, a monitoring mechanism is responsible to detect failure and to notify the other peers (see section 3.2.5).

- **Peer heterogeneity and selection of best peers**

  Not all the peers in the network have the same capabilities in terms of bandwidth and availability. Most of them have asymmetric connections with various download and upload capacities. Thus, a peer-to-peer live streaming approach should consider this fact in order to suitably support end users. Our approach introduces a ranking algorithm to continuously evaluate every peer's capabilities and inform a newcomer of the best possible peers to connect to (see section 3.4.1).

- **Monitoring of network conditions**

  Because of the dynamic nature of peer-to-peer systems, network conditions may change very quickly and frequently during a multimedia session. Thus, a mechanism to monitor the network conditions and quickly adapt (reactively) should be used. In our proposal, this mechanism is a part of the communication protocol (see section 3.3.4).

- **Incentives for participating peers**

  A common problem in all peer-to-peer systems is the selfish user behavior in which a node does not share data after having downloaded it. Thus, an appropriate mechanism should be introduced to encourage users to utilize their upload bandwidth and transmit the media stream to others. In our proposal, we leave this issue as future work (see section 6.1).

- **Network coding**

  Network coding is an approach used in information theory to improve throughput in communication sessions. Recent work introduces the use of network coding in a peer-to-peer live streaming system [49]. However, because of the strict timing and bandwidth constraints in live streaming applications, extensive research should be conducted in order to find a system fulfilling these requirements and at the same time adjusted to peer-to-peer characteristics.

In the next chapter, our proposed architecture will be presented. There, we describe solutions for the problems stated above based on a fully distributed peer-to-peer system.

Chapter 3

Daedalus: A media agnostic peer-to-peer architecture for IPTV distribution

Daedalus is a peer-to-peer architecture designed to provide reliable media content delivery in a large scale network. It provides the network with different channels of content that can be either commercial television channels or channels created by individual users (user generated media content). Each peer of the network can join a channel and will start receiving the selected media stream. If a new peer joins the same channel, it will get the media stream not from the initial node that injects the content into the network, but from other peers that are receiving this channel at this point in time. Thus, a peer-to-peer based overlay network is created on demand.

An innovative idea introduced in Daedalus is a notification procedure to be followed by a peer that wants to leave the channel it is currently receiving. Therefore, before a peer leaves the channel, it sends a Part message (see section 3.2.3) to the peers that are currently in its peer table, in order to inform them that it is going to leave. Thus, these peers can perform the appropriate actions to find another peer to get the media stream from, before stop receiving it from this peer. In this way, we can avoid buffer starvation and abruptly decreased video quality. In the event of a peer failure, peers that had active communication with this peer (i.e. they used to get the media stream from this peer) must quickly discover this failure and try to quickly find another peer. Therefore it should be clear that the danger of a temporary buffer starvation and abruptly decreased video quality cannot be eliminated; but we believe that the probability of either occurring can be reduced to a suitably low value. The computation of this value is left for future work (see section 6.1).

The selection of the peers from which a peer will get a media stream is another challenging task, when designing a peer-to-peer system for live media distribution. In Daedalus, we introduce a priority scheme to provide a peer with information about the best available peers at any given point in time. For each peer in the peer table, a priority value is initially assigned, then based on the reliability and the efficiency in the communication with this peer, this value is dynamically increased or decreased (for the exact algorithm see section 3.4.1). Thus, when a newcomer joins the system and gets the list of peers from a peer based on a request/response mechanism, it will send Join messages to them based on their priority. The larger the priority, the better the peer is. After having started the exchange of information with these peers, the priority will fluctuate based on the communication characteristics.

As we mentioned in section 2.1, IP multicast is the most efficient method for multimedia content distribution, but it faces certain problems that prevent it from being widely adopted and implemented. However, a peer-to-peer system that is designed for live media streaming should be capable of exploiting the IP multicast infrastructure where possible in order to be more efficient. Daedalus adopts this functionality natively. Each peer in the system is identified by an IP address and a port number. If this IP address is a multicast address, then sending a fragment to this address will forward it to all the actual peers that belong to this multicast group. Thus, the transmission efficiency is increased.

Finally, as described in section 2.3.1, another open issue in live streaming peer-to-peer systems is how a peer will receive the media stream from the other peers. Two methods are generally proposed: the pull method and the push method. In the pull method, when a peer needs a certain fragment, it requests it from another peer based on a specific algorithm. The advantages of this approach are its simplicity and robustness. On the other hand, it introduces additional control traffic overhead in the network, thus decreasing the efficiency of bandwidth utilization and decreasing the total system throughput. Because of its simplicity, a sophisticated scheduler should be implemented in order to increase efficiency and allow near-synchronization between the peers. Note that the peers will not be exactly synchronized as it takes time for the information to propagate from one to another, but the difference in play out can be reduced to a small offset.

In the push method, on the other hand, a peer subscribes to get the fragments it needs to get from a neighbor peer based on a specific algorithm. When this peer receives fragments that match the subscription pattern of another peer, it will push them directly to it. This approach is more complicated, but it minimizes the delay and almost eliminates the additional control traffic overhead. Furthermore, a sophisticated and complex scheduler is not needed. However, the pull method is still necessary in order to recover from errors in transmission. That is, a fragment may be lost in the network. In this case, a request for this specific fragment is sent to the appropriate peer.

Based on the above considerations, we concluded that a hybrid scheme is the most appropriate solution for a peer-to-peer live streaming system. Initially, the pull method is used to get the initial media content. Then, the peer switches to the push method to get the rest of the media stream. If a fragment is missing, it is requested through the pull method. The Deadalus architecture is designed based on this hybrid scheme. However, the prototype utilizes only the pull method in combination with a sophisticated scheduler, thus the implementation of the full hybrid scheme is left for future work.

In the next section, the Daedalus architecture will be presented and the functionality of the proposed entities will be described.

3.1 Architectural Overview

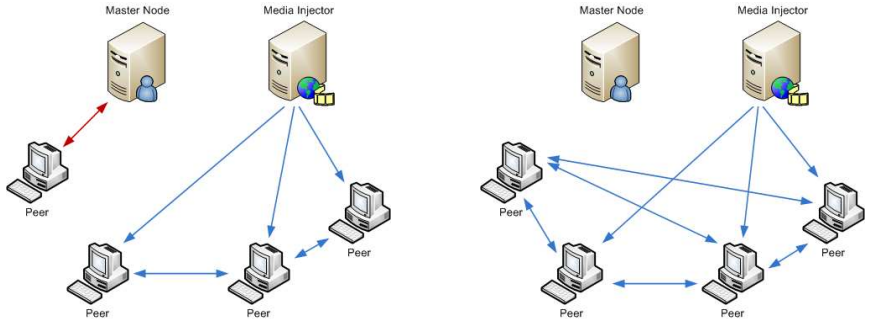
In this section, we are going to present the architectural overview of Daedalus and describe the functionality of the different entities. Daedalus's basic functionality is depicted in Fig. 3.1. Daedalus's is composed of the following three entities:

Master Node The Master Node provides fast bootstrapping for a newcomer (a newly arrived peer). It maintains information about all the peers within the logical peer-to-peer network. Thus, when a peer wants to join the system to receive a specific channel, it first contacts the Master Node to get the list of peers that are currently receiving that channel (Fig. 3.1a). Then, based receiving this information, the peer requests this channel from another peer and starts receiving the media stream (Fig. 3.1b). Notice that the existence of the Master Node is not necessary in an actual system based on the Daedalus architecture. More sophisticated mechanisms can be introduced in order to provide information for fast bootstrapping. However, for the purposes of our prototype, the concept of MN was adopted.

Media Injector The Media Injector is a peer that gets the media feed from the video decoder. It is responsible for spreading this feed to all the interested peers in the network. One of the main assumptions of our approach is that the architecture should be media agnostic. Thus, we assume that

the Media Injector gets the media feed from the video decoder through MPEG over UDP (MPEG/TS). This stream is then fragmented into pieces called *fragments* and is stored in a buffer. A 32 bit sequence number called a Fragment Identifier (FID) is assigned to each fragment. The minimum valid value for the FID is one and increases by one for every new fragment. The FID equal to zero is used for a specific case (see section 3.3.7). When the maximum value of the FID is reached, it swaps to one. It is upon the implementation the exact handling of the FID swap when it reaches its maximum value. The Media Injector is the initial source which provides the peers of the network with the media stream.

Peer A peer is simply a node that wants to receive a certain channel (generally this will be because there is a user of this node that wants to watch this specific channel). A peer first joins the channel by contacting the Master Node in order to get a list of the available peers currently receiving this channel. It then contacts these peers and starts getting the media stream which is temporarily stored in a local buffer. A process running in parallel in the peer, is responsible for forwarding the media content from this buffer to the video decoder. The details are presented in section 3.4.



(a) Initial contact to the Master Node.

(b) Media content streaming.

Figure 3.1: Daedalus functionality - media content distribution in a mesh-based way.

Fig. 3.2 presents a small scale Daedalus network. The Media Injector receives the live media stream using MPEG/TS over UDP and communicates with the rest of the peers running Daedalus instances using our Live Streaming over Peer-to-Peer Protocol (LSPP). Every peer runs an instance of the Daedalus module and communicates with the Daedalus network using LSPP. Additionally every peer contains a video decoder that receives and displays the MPEG/TS packets that Daedalus sends to it over UDP. The description of the Daedalus instance module is presented in section 3.2; while the specification of the LSSP protocol is presented in section 3.3.

3.2 Functional Overview

Daedalus consists of four basic modules. Fig. 3.3 depicts how a Daedalus peer works. Each module handles a different part of Daedalus's operation and communicates with the other modules. The modules are as follows:

Communication Module This module is responsible for communication between peers. It handles the creation and reception of messages according to our LSPP protocol. Received messages are processed and information contained in the messages feed the Buffer and Peer module. Messages created are

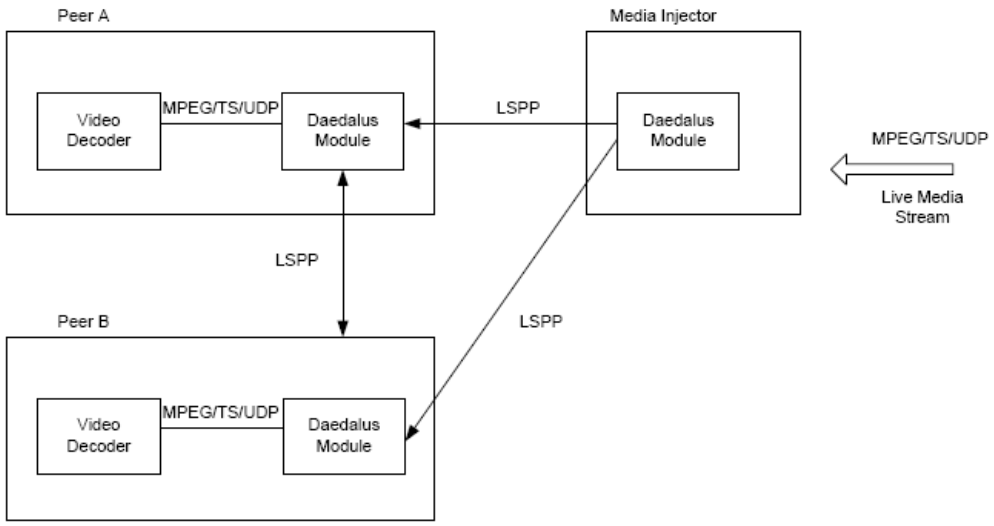


Figure 3.2: Daedalus architecture.

using data returned from Buffer and Peer module. The messages are created and processed based on our proposed LSPP protocol specification (see section 3.3).

Buffer Module In the Media Injector the buffer module requests and receives the requested fragments from the Communication module and spits the video stream into fragments and stores them locally to its buffer. The buffer module also contains the scheduler which is responsible for deciding when to make requests to other peers and when to forward fragments to the video decoder. In section 3.4, we will describe in detail the scheduler and the algorithms within it.

Peer Module The peer module manages the neighbor peers information. It provides the other modules with information about neighboring peers.

KeepAlive Module The KeepAlive module periodically sends message to all its (direct) peers to indicate that it is alive. The scope behind these messages is to keep track of the network topology. For example, if a peer crashes or a problem occurs in the network and the communication is stalled, it will stop sending KeepAlive messages to its neighbors and thus, they will delete it from their peer tables.

In the following subsections we describe the procedures that are followed by Daedalus architecture, beginning with the bootstrap procedures and continuing with the actions performed upon receipt of each LSSP message.

3.2.1 Start-up Procedures

As described in section 3.1, a peer must first contact the Master Node in order to get a list of peers that are currently on the same channel. After receiving this list, the peer sends Join messages to these peers in order to inform them that it wishes to join this specific channel and to request the initial media content. The steps followed in this procedure are depicted in Fig. 3.4 and described below:

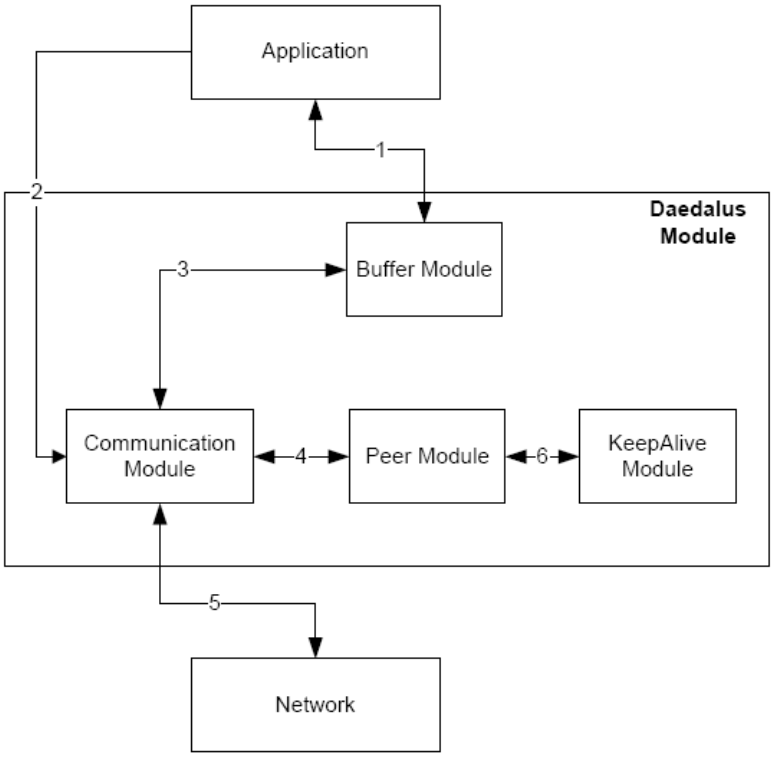


Figure 3.3: Architecture for a Daedalus peer.

1. The peer sends a Join message to the Master Node requesting a list of the peers that are currently registered to receive the channel that this peer wants to join. As we mentioned in the previous section, it is not necessary for the first Join to be sent to the Master Node .
2. Upon the reception of the KeepAlive message containing the list of peers from the Master Node , the peer sends new Join messages to the peers contained in the KeepAlive message. Using this message (Join), it requests the peer's peer table (these are the peers which will receive KeepAlive messages), information about the status of their buffer (Indication message), and initial media content (Payload message).
3. Upon the reception of the initial media content, the peer starts requesting for additional parts of the stream and starts sending periodic KeepAlive messages.

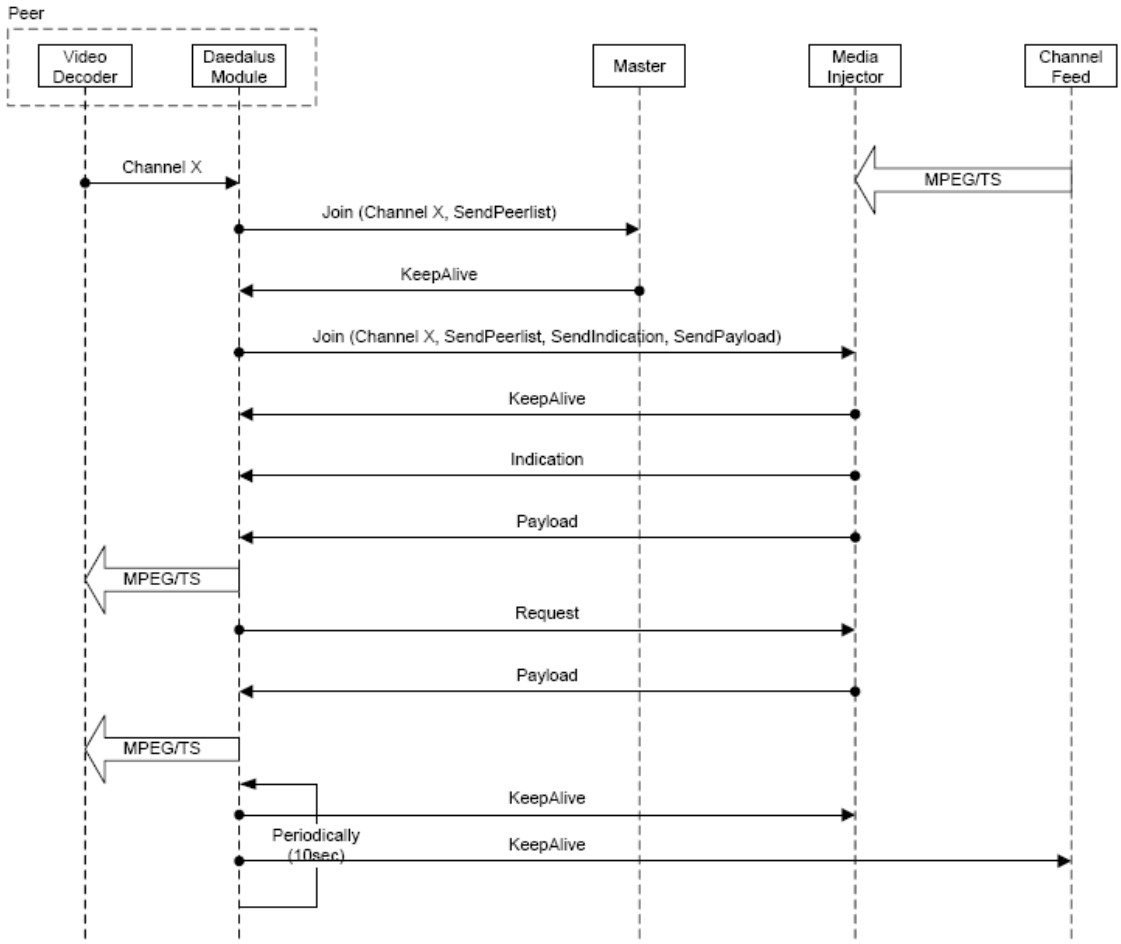


Figure 3.4: Message flowchart for the first peer joining the system.

The same procedure for a second peer joining the system is illustrated in Fig. 3.5. The difference from the procedure for a single peer is that in this case, requests are made both to the Media Injector and the first peer that has already joined the channel.

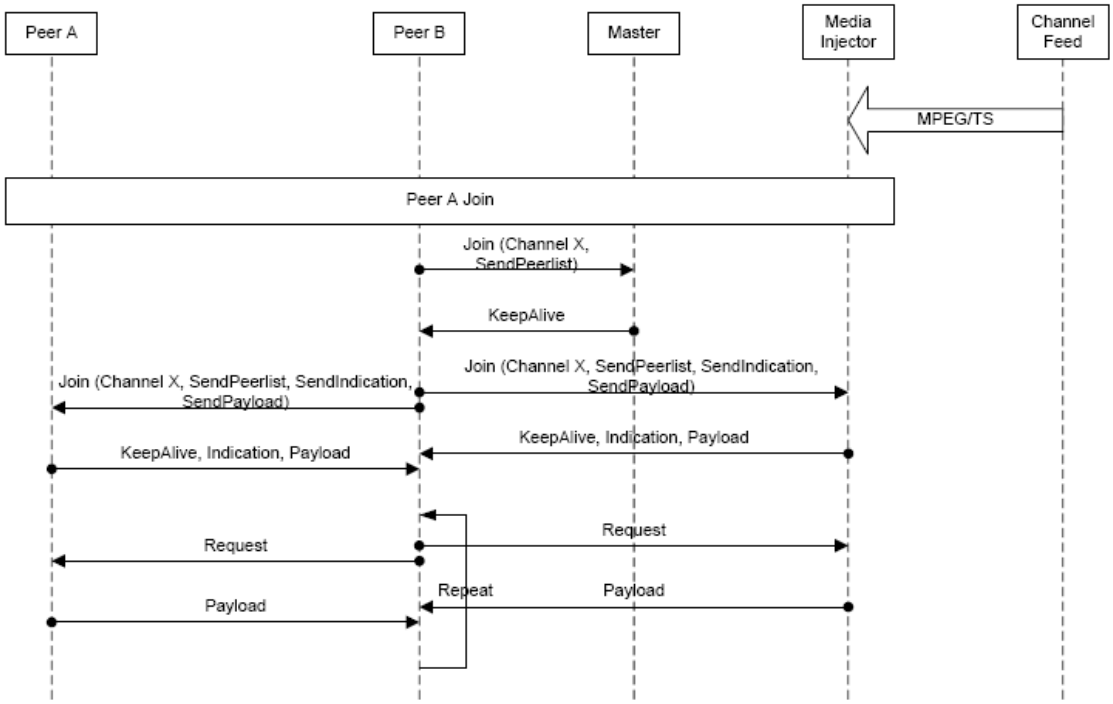


Figure 3.5: Message flowchart for the second peer joining the system.

When a peer wants to leave the system completely (e.g. to power off) or change channel, a Part message is sent to the Master Node and its live peers. Upon the reception of the Part message, these peers are informed that the peer leaves this channel and they remove the specific entry from their peer tables. This procedure is depicted in Fig. 3.6.

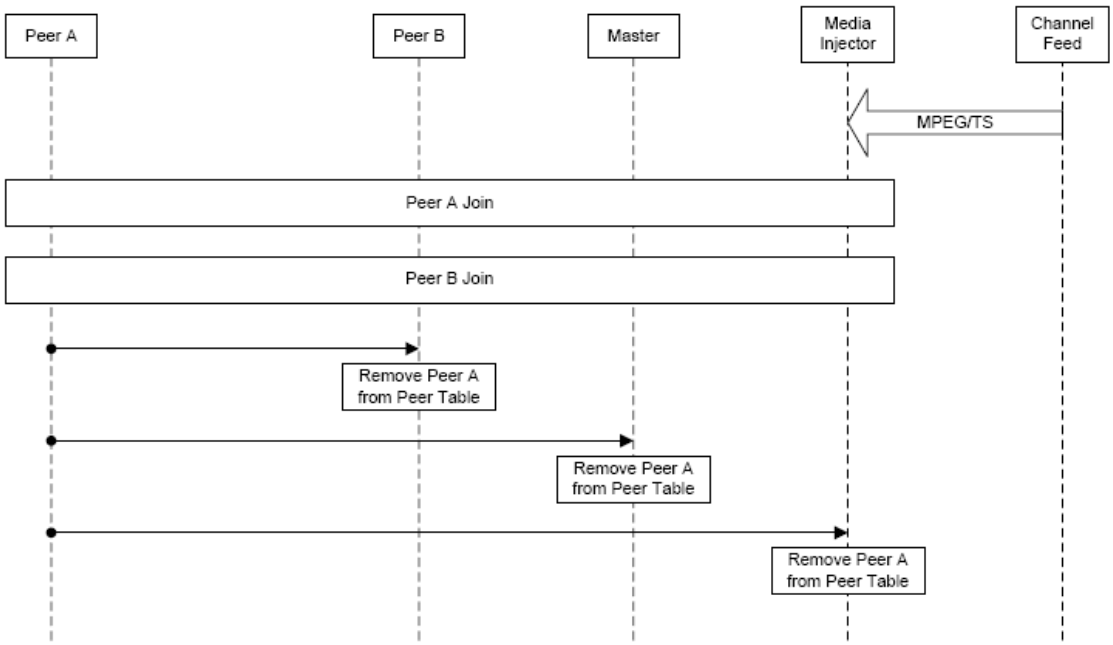


Figure 3.6: Message flowchart for a peer leaving the system or changing channel.

3.2.2 Join Process

In this section, the description of the process followed upon the reception of a Join message is presented. If the peer is the Master Node , it first checks whether the SendPeerList flag is on or not, and based on this it creates a Keepalive message with the list of peers of this channel. If the SendPayload flag is on, it creates a Payload message with the NOP flag on indicating that it does not have any payload, thus it does not ask again. If the peer is not the Master Node , it checks if the Join message is for its channel. If it is not, it creates and sends a Payload message with NotInChannel flag on indicating that it does not belong to this channel. If it is for its channel, then it checks the SendPeerList, the SendIndication, and the SendPayload flags. If any of them are on, it asks the Peer and the Buffer modules to return the list of peers, the buffer status, and a portion of the media content in its buffer respectively. Finally, it creates and sends a KeepAlive, an Indication, and a Payload message. This process is illustrated in Fig. 3.7.

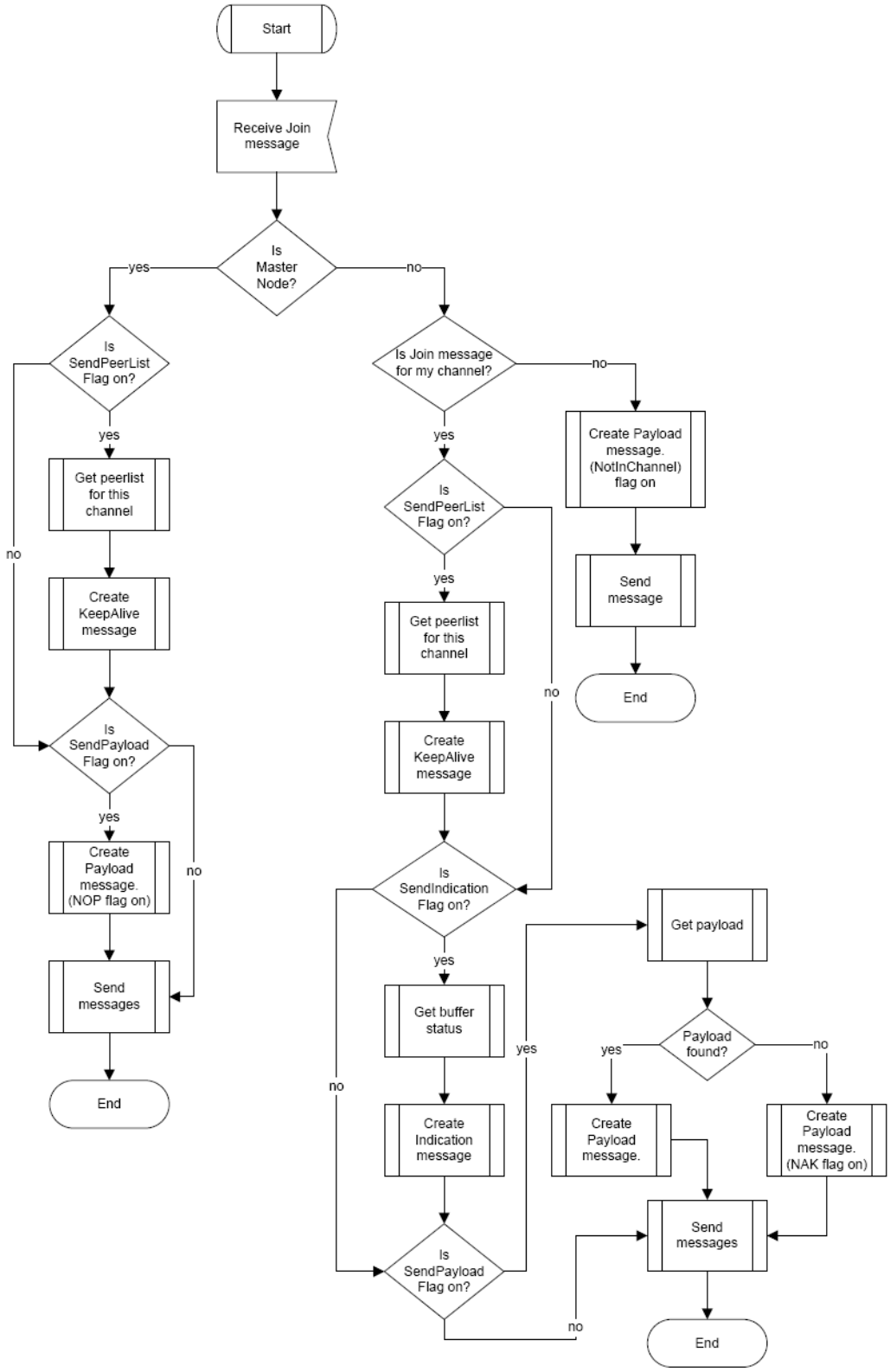


Figure 3.7: Message flowchart for a Join message.

3.2.3 Part Process

The Part Message is sent when a peer wants to leave a channel. If a peer receives such a message, it first checks if the channel of the Part message refers to its channel. If so, it deletes the peer who send it from its peer table. If it is a common peer, it checks also if the number of the live peers has dropped down than a certain number (*LIVE_PEERS_NUMBER*) and if it does, it sends a Join message to a new peer from its peer table. Fig. 3.8 recapitulates the above steps.

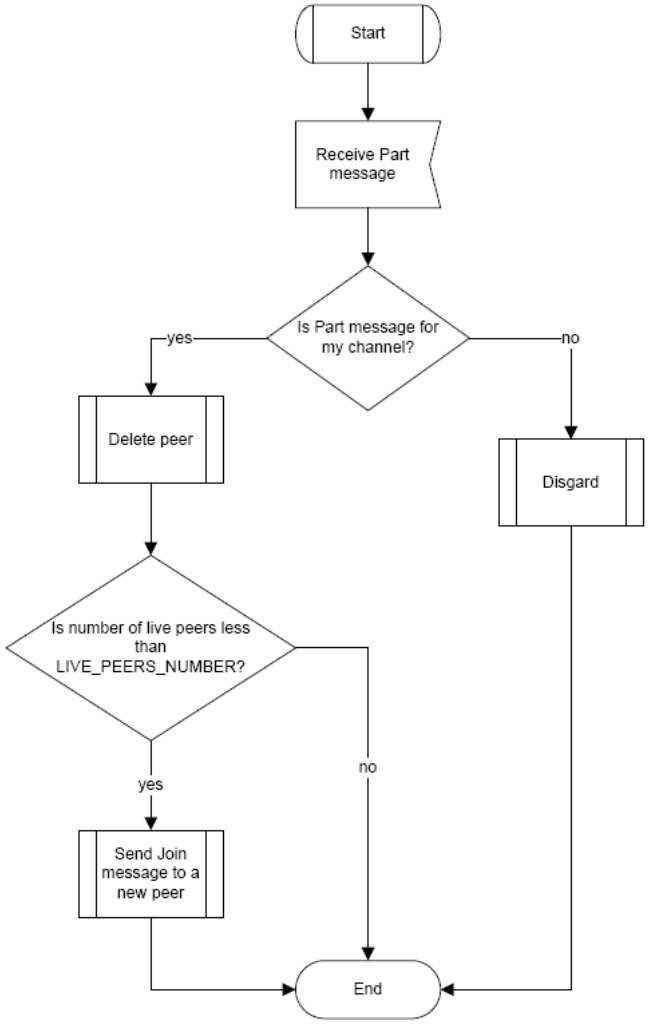


Figure 3.8: Message flowchart for a Part message.

3.2.4 Indication Process

Indication message is sent in order to inform peers about the status of buffer of the sendee peer. When a peer receives an Indication message, it follows the steps depicted in Fig. 3.9. First, it examines if the channel in the Indication message is the same with the channel it is currently on, and if it does, it stores the information about the buffer status of the sender peer. If the channel is different that its channel, the message is discarded. Information about the status a peer can be used by the scheduler to make better decisions about which fragments should be asked from which peers.

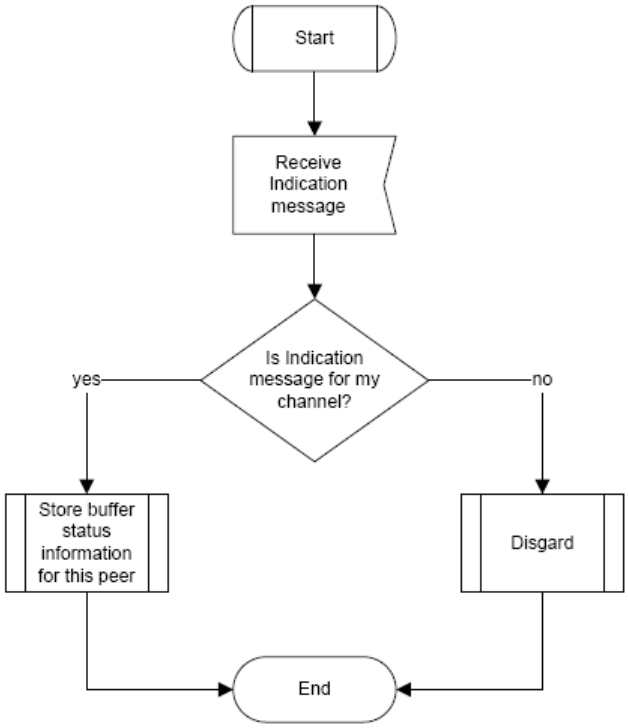


Figure 3.9: Message flowchart for a Indication message.

3.2.5 KeepAlive Process

In Fig. 3.10, the processing of a KeepAlive message is illustrated. If the message contains statistics, the peer extracts them and stores them to a file. If it contains a list of peers, the peer extracts this information and forwards it to the Peer module which updates the peer table. If the peer is a common peer and not a Master Node or a Media Injector , it also checks if the number of live peers, the peers that it asks for media content, is less than a certain value. In this case, it sends a Join message to a new peer from its peer table.

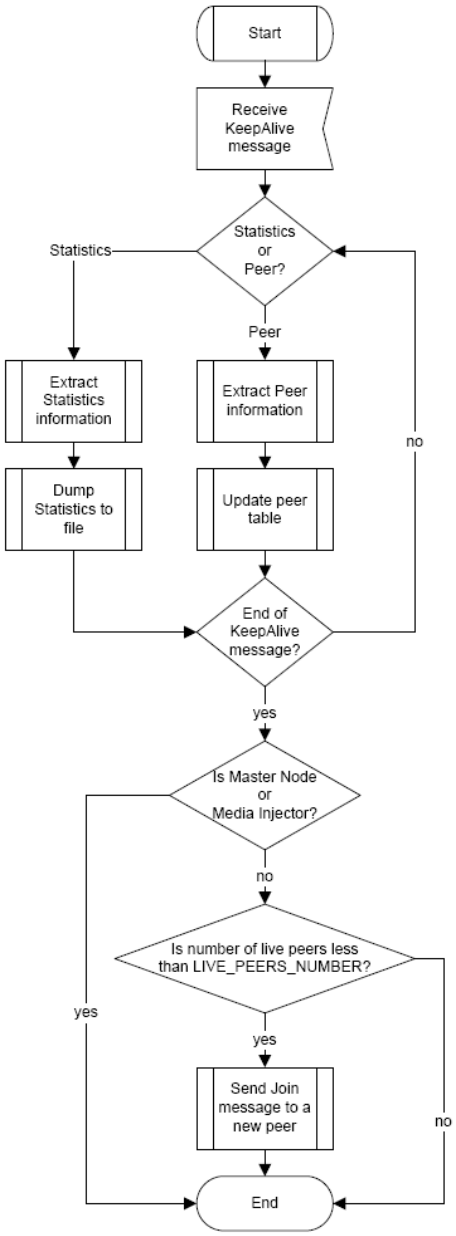


Figure 3.10: Message flowchart for a KeepAlive message.

3.2.6 Request Process

When a peer receives a Request message, it checks if the message is for its channel. If not, it sends back a Payload message with the NotInChannel flag on indicating that the request for the specific media content is not for its channel. If the request is for the correct channel, the Resynchronization flag is checked and if it is found on, the peer sends the initial payload it can send from its buffer within a Payload message. If the flag is not set, the peer searches in its buffer for the requested fragment and if it finds it, it creates and sends back a Payload message. Otherwise is creates and sends a Payload message with the NAK flag on indicating that it does not has the specific payload. This process is depicted in Fig. 3.11.

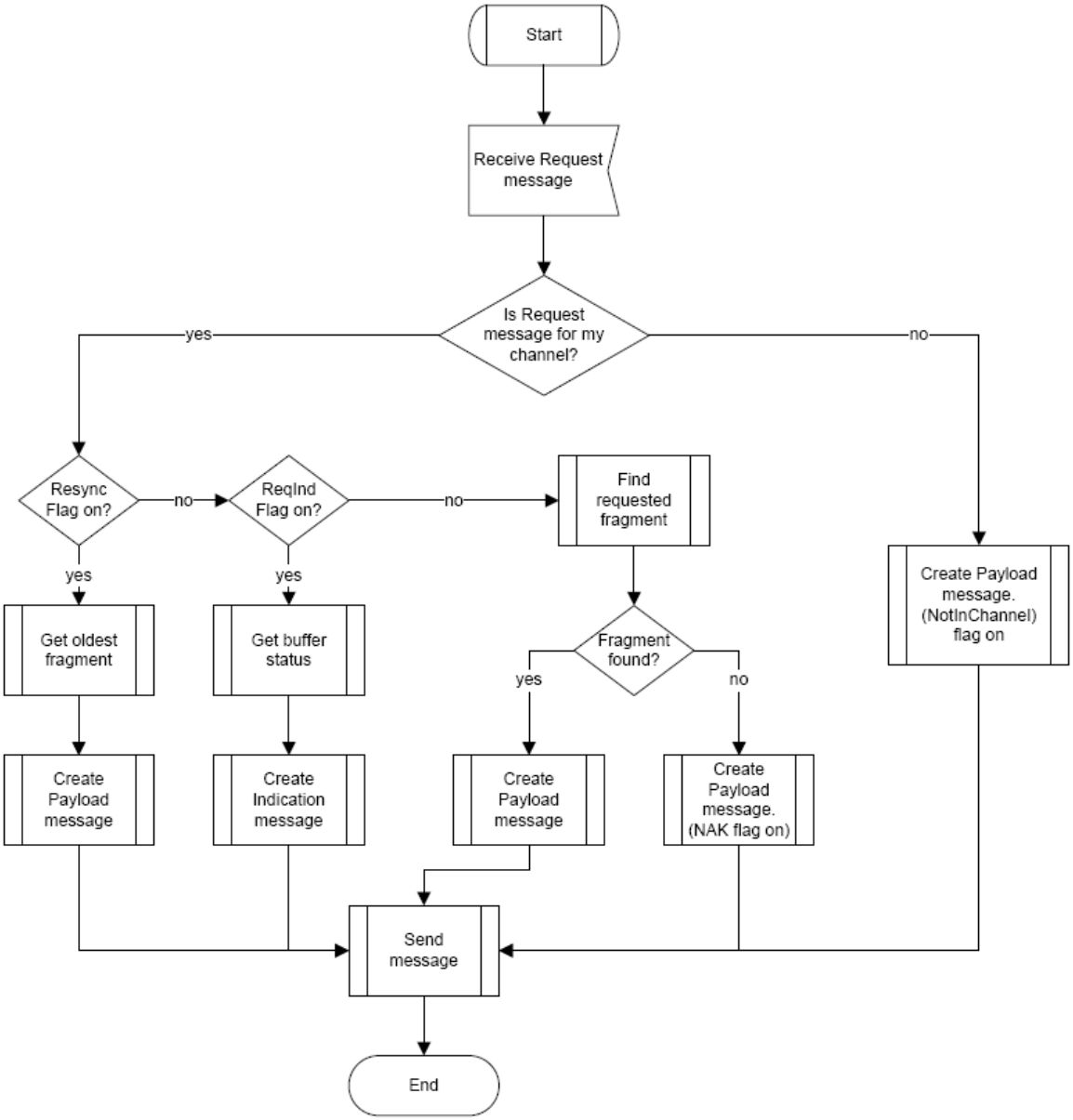


Figure 3.11: Message flowchart for a Request message.

3.2.7 Payload Process

Payload message is received as a response to a Request message. The procedure followed by each peer when it receives a Payload message is illustrated in Fig. 3.12. First, the peer checks if the message refers to its channel. If it does not, it discards the message. Otherwise, it parses the NOP, NAK, and HasPayload flags, that are currently used for our prototype. If the NOP flag is on, the sender peer is marked and the peer will not request any packet from it anymore. If the NAK flag is on, that means that the peer that we made the request does not have the specific fragment, so we update its priority and we ask the same packet from another peer. If the payload flag is enabled, we are searching if we have asked this payload and if we do we store the payload in our buffer and update the sender's peer priority. Otherwise we discard the message.

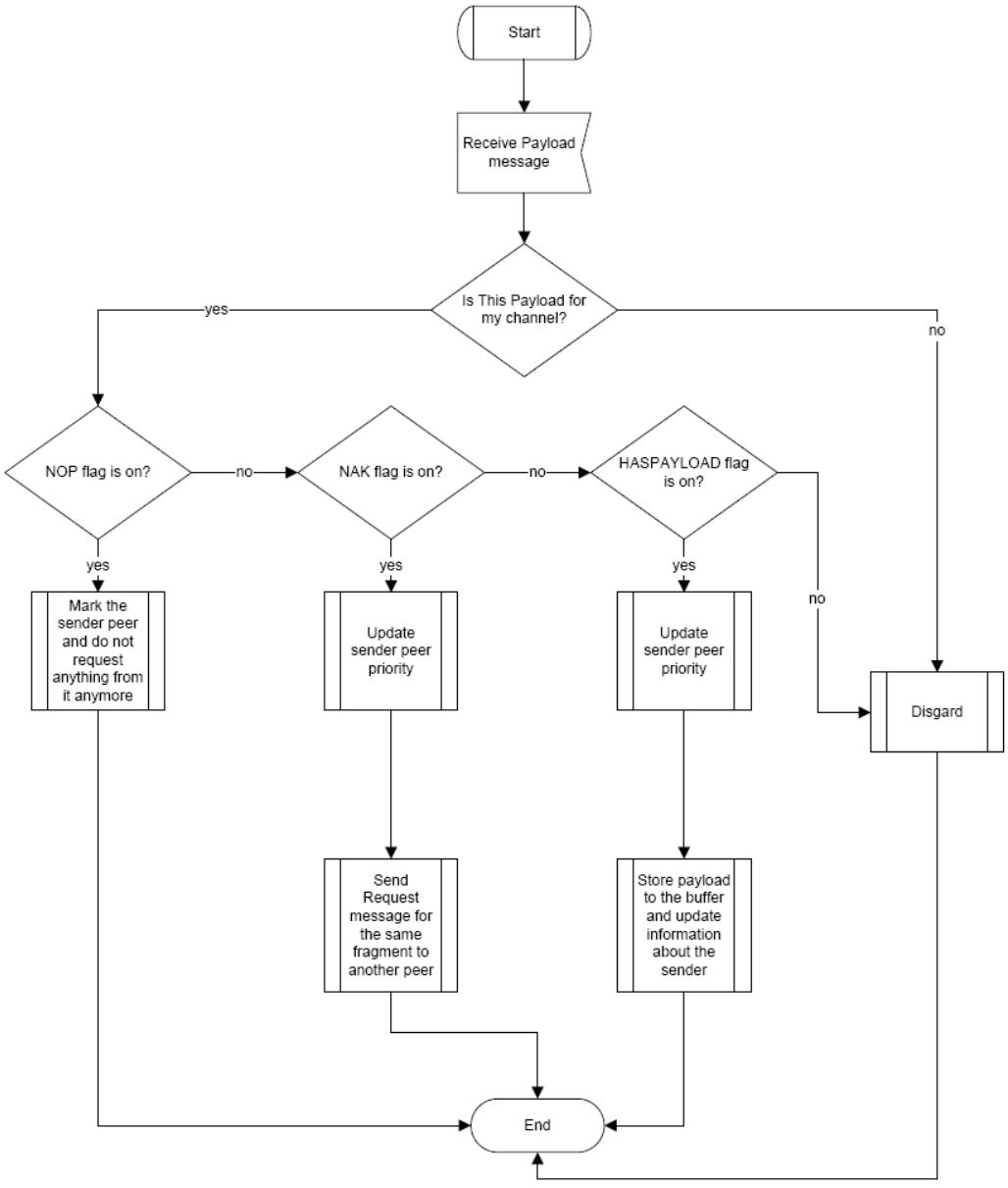


Figure 3.12: Message flowchart for a Payload message.

3.3 Protocol Specification

In this section, the specification of the protocol we have developed to provide the necessary functionality for the Daedalus architecture is presented. We call our protocol the Live Streaming over Peer-to-Peer Protocol (LSPP). The relation of LSPP to other protocols is depicted in Fig. 3.13. LSPP runs on top of UDP, thus requirement for reliable transmission should be incorporated into LSPP. Before we continue with the protocol specification, we will present a set of assumptions on which LSPP was designed.

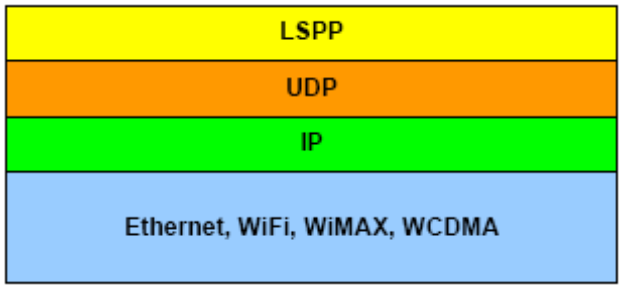


Figure 3.13: Relation of LSPP to other protocols.

- Each LSPP packet should be transmitted inside a single UDP datagram. This is necessary in order to avoid problems regarding fragmentation.
- Each LSPP packet may contain multiple messages. Different message types are defined in the LSPP. Carrying multiple messages per LSPP packet increases the efficiency of the protocol and decreases the overhead.
- Each peer is uniquely identified by a 128bit UUID (Client UUID) generated based on the method proposed by Leach, et al. [50]. Two peers that are behind a Network Address Translation (NAT) firewall, may have the same IP address and port when seen by the other peers. Thus, the IP-Port combination is not enough to differentiate such peers from each other.
- Each channel is uniquely identified by a 128bit UUID (Channel UUID) similar to a Client UUID. Any user that creates media content can reserve a channel and broadcast content via this channel. Thus, we expect that no channels have the same identifier.

3.3.1 LSPP Packet

The format of the LSPP packet is depicted in Fig. 3.14. A LSPP packet is used to transfer all the messages designed to provide both control information and media content transmission.

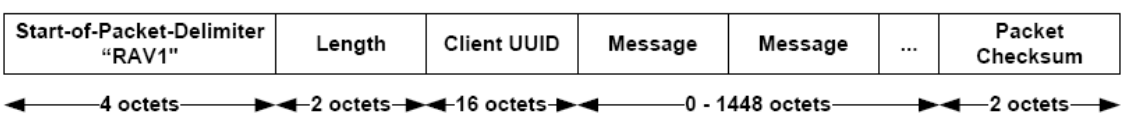


Figure 3.14: Format of a LSPP packet.

Start-of-Packet-Delimiter: 32 bits (4 octets)
It is used to distinguish a LSPP packet from packets from other protocols. For the prototype developed, the value used was 'RAV1'.

Length: 16 bits (2 octets)
Length is the length in octets of this LSPP packet including this header and the data. (This means the minimum value of the length field is twenty four.)

Client UUID: 128 bits (16 octets)
It holds the UUID of the peer sending this LSPP packet.

Message:
One or more of the possible messages described in the previous section - which define LSPP. Each message can be either a control message or a message transferring actual media content. Note that each message indicates its length or has a fixed length. The messages are:

- Join
- Part
- Indication
- KeepAlive
- Request
- Payload

Packet Checksum: 16 bits (2 octets)
It is a CRC computed on the whole packet. In the calculation of the checksum, the value of this field is set to zero. For the purposes of the prototype we implemented, no specific algorithm was used. The selection of the the algorithm to compute the checksum is left as future work.

In the next subsections we are going to present the message format for the above messages and describe their semantics as well.

3.3.2 Join Message

The Join message is sent from one peer to another peer when it wants to join a channel. First, a Join with only the SendPeerList flag on is sent to the Master Node in order to get a list of peers that are currently receiving the same channel as indicated in this Join message. The Master Node responds with a KeepAlive message containing information about the available peers. Upon the reception of the KeepAlive message, the peer sends Join messages to each of these peers in order to inform them that it has joined this channel and to indicate that it wants this channel's media content (all flags are on). The format of the Join message is depicted in Fig. 3.15.

Type: 8 bits (1 octet)
It defines the type of the message. For Join, it is set to the ASCII character value for 'J'.

Length: 16 bits (2 octets)
Length is the length in octets of the Join message.

Figure 3.15: Format of the Join message.

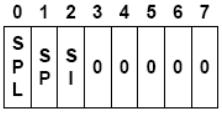


Figure 3.16: Format of Join message flags.

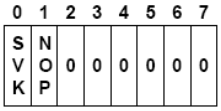


Figure 3.17: Format of Client Flags.

Flags: 8 bits (1 octet)
It contains various flags that define how the peer that will receive this Join message should respond. Its structure is depicted in Fig. 3.16.

SendPeerList Bit 0 is used to request from the peer receiving this Join message to send back a KeepAlive message with a list of peers that are in the same channel as the one in this Join message.

SendPayload Bit 1 is used to request from the peer receiving this Join message to send back media content. Upon the reception of a Join message with this flag on, a peer will get the oldest fragments it has in its buffer and send them to this peer inside Payload messages. This flag is used in order to quickly request fragments and fill the local buffer.

SendIndication Bit 2 is used to request from the peer receiving this Join message to send back an Indication message, which keeps information about the status of the buffer of this peer. This information can be used in order to design a sophisticated mechanism for making requests for fragments among the available peers (see section 6.1).

Spare Bits 3-7 are set to zero and left for future use.

Notice that bits 1 (SendPayload) and 2 (SendIndication), should never be on in a Join message sent to the Master Node , as the Master Node does not have any actual payload. If the Master Node receives a Join message with these flags on, it will respond with an empty Payload message with NOP flag on (see section 3.3.7).

Channel UUID: 128 bits (16 octets)
It holds the UUID of the channel the peer sending this Join message wants to join.

Client Flags: 8 bits (1 octet)
Various flags that define certain characteristics regarding the peer sending this Join message. This field is 8bit (1 octet) long, and its structure is depicted in Fig. 3.17.

SendVerboseKeepAlive Bit 0 is used to inform the target peer that when it sends periodic KeepAlive messages to this peer, it should be in verbose mode. That is, the KeepAlive message should include not only the default information (the list of peers) but extra one related with statistics (see section 3.3.5).

NoPayload Bit 1 shows that this peer will not have any payload; thus, do not make requests to it. The priority for this peer is equal to zero. For further information regarding the priority scheme, see section 3.4.1.

Spare Bits 2-7 are set to zero and left for future use.

3.3.3 Part Message

A Part message is sent from a peer to all the peers it has currently communication with in order to inform them that it leaves this channel. Upon the reception of a Part message, a peer removes from its peer table the peer whose Client UUID was in the Part message. The structure of the Part message is depicted in Fig. 3.18.

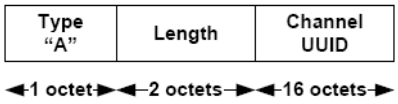


Figure 3.18: Format of the Part message.

Type: 8 bits (1 octet)
It defines the type of the message. For Part, it is set to the ASCII character value for 'A'.

Length: 16 bits (2 octets)
Length is the length in octets of the Part message.

Channel UUID: 128 bits (16 octets)
It holds the UUID of the channel the peer sending this Part message wants to leave.

3.3.4 Indication Message

The Indication message is introduced in order to inform peers each other about the status of their buffer. This information is used by the scheduler, which generates the requests, to distribute them in a sophisticated way among the available peers. In this way, the probability that a peer does not have the requested fragment is minimized. In addition, in the current implementation, this information is used to select the best possible peer to ask for a portion of the media stream, if one of the current live peers becomes inefficient. The structure of the Indication message is depicted in Fig. 3.19.

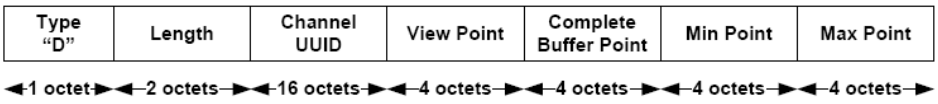


Figure 3.19: Format of the Indication message.

Type: 8 bits (1 octet)
It defines the type of the message. For Indication, it is set to the ASCII character value for 'D'.

Length: 16 bits (2 octets)
Length is the length in octets of the Indication message.

Channel UUID: 128 bits (16 octets)
It holds the UUID of the channel the peer is currently watching.

View Point: 32 bits (4 octets)
It shows the FID of the last fragment sent from the buffer of the peer to the video decoder.

Complete Buffer Point: 32 bits (4 octets)
It shows the FID up to which the buffer of the peer is completely full. There are no empty positions in the buffer up to this point.

Min Point: 32 bits (4 octets)
It shows the FID of the oldest fragment in the buffer of this peer.

Max Point: 32 bits (4 octets)
It shows the FID of the latest fragment the peer has in its buffer. There may be "holes" in the buffer up to this point.

3.3.5 KeepAlive Message

The KeepAlive message is sent to a peer in two cases. First, as a response to a Join message which had SendPeerList flag on. In this case, the KeepAlive includes only the peer list of this peer. Second, periodically every 10 seconds. In this case, it may include additional information related with various statistics of this peer. As described in section 3.3.2, Client Flags is one of the fields of a Join message. If the SendVerboseKeepAlive flag is on, then the periodic KeepAlive message sent to that peer will contain information about the statistics of this peer as well. The structure of the KeepAlive message is depicted in Fig. 3.20.

Figure 3.20: Format of the KeepAlive message.

Type: 8 bits (1 octet)
It defines the type of the message. For KeepAlive, it is set to the ASCII character value for 'K'.

Length: 16 bits (2 octets)
Length is the length in octets of the KeepAlive message.

Flags: 8 bits (1 octet)
It defines various flags regarding the KeepAlive message. Currently only the first two bits are used to define what kind of information follows in the KeepAlive message. The structure of this field is presented in Fig. 3.21. If the value of the first two bits is equal to one, then information regarding statistics follows. If this value is equal to two, then information about a peer in the peer table of this peer follows. Bits 3-7 are set to zero and left for future use.

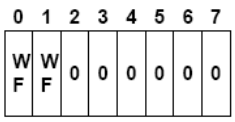


Figure 3.21: Format of the flags in the KeepAlive message.

Statistics:
The structure of information regarding statistics of a peer is depicted in Fig. 3.22.

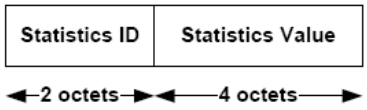


Figure 3.22: Format of the Statistics structure.

Statistics ID It is 16bit (2 octets) long and defines which type of statistics the value following in Statistics Value field refers to. For the purposes of the prototype, the following statistics types were defined:

- Number of sent fragments (1). It represents the number of fragments sent from this peer to the network.
- Number of received fragments (2). It represents the number of fragments received from this peer.
- Number of realtime fragments requested (3). It represents the number of fragments requested that had the realtime flag on.
- Uptime (4). It is equal to the time period this peer has joined the system in milliseconds.
- Number of buffer under runs (5). It represents the number of times a fragment that should be sent from the local buffer to the video decoder was missing.

Statistics Value It is 32bit (4 octets) long and holds the value of the statistics type specified in Statistics ID field.

Peer:
In this structure, information about a peer in the peer table of the peer sending the KeepAlive message is held. Its structure is depicted in Fig. 3.23.

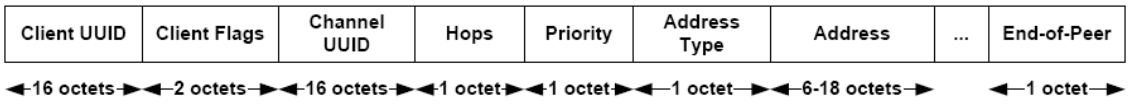


Figure 3.23: Format of the Peer structure.

Client UUID Client UUID is 128bit (16 octets) long and holds the UUID of this peer.

Client Flags Same field as the one described for the Join message (see section 3.3.2).

Channel UUID Channel UUID is 128bit (16 octets) long and holds the UUID of the channel this peer is watching.

Hops Hops is 8bit (1 octet) long and shows the number of hops this peer is far from the peer sending this KeepAlive message.

Priority Priority is 8bit (1 octet) long and shows the priority of this peer.

Address Type It defines if the following address is IPv4 or IPv6. It is 8bit (1 octet) long and if it is equal to four, an IPv4 address follows. If it is equal to six, then an IPv6 address follows. Each peer may have a list of IPv4 or IPv6 addresses.

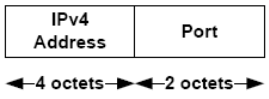


Figure 3.24: Format of the IPv4 address.

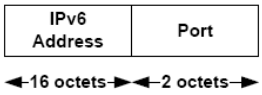


Figure 3.25: Format of the IPv6 address.

Address The Port shows the UDP port number the peer used for communication and it is 16bit (2 octets) long. The IP field can be either an IPv4 (32bit) (Fig. 3.24) or an IPv6 (128bit) (Fig. 3.25) address. In our prototype, only IPv4 addresses were considered.

End-of-Peer It is 8bit (1 octet) long and is used to define the end of the information regarding a certain peer. Only the bit zero is currently used. If it is equal to one, then the information about this peer ends here. Bits 2-7 are set to zero and left for further use.

End-of-KeepAlive: 8 bits (1 octet)
It defines the end of the KeepAlive message. Its value is set to zero.

3.3.6 Request Message

A Request message is used in order to request fragments from the other peers in the system. Its structure is presented in Fig. 3.26.

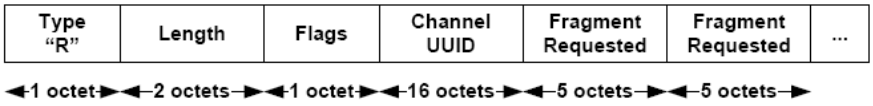


Figure 3.26: Format of the Request message.

Type: 8 bits (1 octet)
It defines the type of the message. For Request, it is set to the ASCII character value for 'R'.

Length: 16 bits (2 octets)
Length is the length in octets of the Request message.

Flags: 8 bits (1 octet)
It provides information about this request. Its structure is depicted in Fig. 3.27.

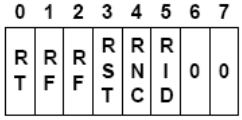


Figure 3.27: Format of the flags in the Request message.

Realtime Bit 0 is used to define if this is a realtime request or not. A request with this flag on is sent in case the fragment requested is necessary to be fetched as soon as possible; otherwise the media stream will be corrupted.

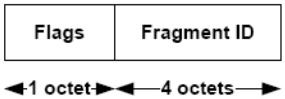


Figure 3.28: Format of the simple request.

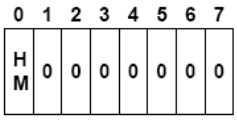


Figure 3.29: Format of the flags in the simple request format.

Request Format Bits 1-2 are used to define the format of the request. Currently, two formats are considered. The simple request, is presented in Fig. 3.28 and it is used to ask for a specific fragment. The extended request has not be defined for the prototype, but it is designed in order to provide more sophisticated ways to request for fragments.

Reset Bit 3, is used to reset any previous format regarding the way requests are made. Is is used only in combination with the extended request format.

Resynchronization Bit 4, is used in order to inform another peer that resynchronization is needed. Resynchronization is performed when the synchronization with this peer is lost (i.e. either we ask to fast and the peer does not have these fragments or we ask to slow and it has already remove them from its buffer). Resynchronization is detected when a peer sends a large number of consecutive Payload messages with NAK flag on.

RequestIndication Bit 5, is used to request from a peer to send an Indication message. This functionality is introduced in order to enable a peer decide which of the available peers in its peer table is suitable to start getting a part of the media content. Remind that an Indication message contains information about the buffer status of the peer that sends it.

Spare Bits 6-7 are set to zero and left for future use.

Channel UUID: 128 bits (16 octets)
It shows the UUID of the channel the peer is currently on.

Fragment Requested:
It provides information about the requested fragment (in the simple request format). Its structure is depicted in Fig. 3.28.

Flags: 8 bits (1 octet)
It provides information about the specific fragment requested. Its structure is depicted in Fig. 3.29.

HasMoreFragments Bit 0 is used to denote if more requests for fragments will follow or not. If 1, more requests follow, otherwise it is set to 0.

Spare Bits 1-7 are set to zero and left for further use.

Fragment ID: 32 bits (4 octets)
It holds the ID of the requested fragment.

3.3.7 Payload Message

The Payload message is used to transfer the actual media content requested from a peer. One Payload message can either have a fragment or not. Its structure is depicted in Fig. 3.30.

Figure 3.30: Format of the Payload message.

Type: 8bits (1 octet)
It defines the type of the message. For Payload, it is set to the ASCII character value for 'P'.

Length: 16bits (2 octets)
Length is the length in octets of the Payload message.

Channel UUID: 128 bits (16 octets)
It shows in which channel the fragment belongs to.

Fragment ID: 32 bits (4 octets)
It is used to determine which fragment it is. The minimum valid value for the FID is one and increases by one for every new fragment. The FID can be zero only in one case. If a peer receives either a Join message with the SendPayload flag enabled or a Request message with the Resynchronization flag enabled, and it does not have any fragments in its buffer, then it will send back a Payload message with FID equals to zero and the NAK flag enabled. This informs the original peer that it should find another peer to get the media stream. Notice, that if the FID reaches its largest value ($2^{32} - 1$), then it wraps-around to one. Thus, the zero value is used only in the aforementioned case.

Peer-to-Peer Fragment Flags: 8 bits (1 octet)
It is used to provide information about the fragment which FID is in this Payload message. Its structure is depicted in Fig. 3.31.

NAK Bit 0 is used to inform the peer that requested this fragment that the peer sending the message does not have it in its buffer.

NOP Bit 1 is used to inform a peer that this peer will not have any media content. If the MN, for example, receives a request for media content, it will respond with a Payload message with NOP flag on.

InChannel Bit 2 is used to inform a peer that this peer is in the same channel.

Realtime Bit 3 is used to inform a peer that this Payload message is a response to a request message with Realtime flag on. This is used to achieve a minimum QoS in order to prevent buffer starvation or decreased quality in the decoded video.

HasPayload Bit 4 shows if in this Payload message actual media content is contained.

Spare Bits 5-7 are spare and set to zero.

Persistent Fragment Flags: 8 bits (1 octet)
It is used to provide information regarding the media content itself. Its structure is depicted in Fig. 3.32.

Encrypted Bit 0 shows that this fragment is encrypted and decryption before forwarded to the video decoder should be take place.

KeyFrame Bit 1 is used to show that this fragment is an I-frame and should be the first fragment sent to the video decoder in order to achieve increased picture quality.

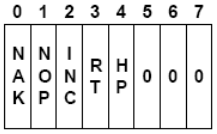


Figure 3.31: Format of the Peer-to-Peer Fragment Flags.

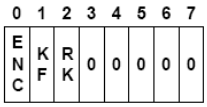


Figure 3.32: Format of the Persistent Fragment Flags.

Rekey Bit 2 is used to define that the encryption key for this and the following fragments has changed, and the new key should be acquired.

Spare Bit 3-7 is spare and set to zero.

Hops To Live: 8 bits (1 octet)
This field indicates the maximum time the fragment is allowed to remain in the peer-to-peer system. If this field contains the value zero, then the fragment must be destroyed. This field is modified in payload message processing. The time is measured in number of peers that received this specific fragment. Each peer that receives this fragment, it decreases the value of this field by one. The intention is to force peers that are very deep in the topology to find other peers nearer to the MI to get the media content. However, a modification to this approach is not to destroy the fragment when the value of this field reaches zero, but keep it in the buffer and send it only to peers that request it with Realtime flag on. Thus, we minimize the probability of decreased media quality in a peer lying deep in the network topology.

Time Difference: 32 bits (4 octets)
It shows the difference in time since the first fragment was received. In order to be resilient to delays introduced from the network, we designed a mechanism that recovers this situation. Specifically, when the MI starts fragmenting the media stream it gets from the decoder, it keeps a timestamp of the first fragment. Then, for all the following fragments, it get timestamp when it gets them from the decoder, subtracts from the first reference value and the difference is set in the Time Difference field for this fragment. In this way, we can be aware of the relative time we should send this fragment in the receiver.

3.4 Scheduler Description

The scheduler is one of the basic modules in our system. Practically, it is responsible to maintain the buffer in a stable status, that is it will always be able to feed the video decoder with a continuous data stream. Furthermore, it should be able to share with other peers as many fragments in its buffer as possible. That means that the scheduler should guarantee that there always exists a minimum number of fragments ready to be sent to the video decoder. If we have enough such fragments, the scheduler stops making requests until this number decreases up to a specific threshold. The concept behind this temporal pause is to save bandwidth and decrease the load of the whole system, since we are not sending new Request messages and receiving Payload messages anymore.

Every live peer is assigned a specific priority when it joins the system. Then, this priority is adjusted according to the peer's behavior (see section 3.4.1). The distribution of the Request messages among the live peers is done based on the priority of each peer. The greater priority a peer has, the greater number of requests he is assigned.

The scheduler is also responsible for feeding the video decoder with the media stream on time and without any unwanted pauses. In order to achieve scheduler's functionalities, we introduce an attribute for each fragment in the buffer called *Time Difference*. Time Difference indicates when a fragment has to be sent to the video decoder. Furthermore, every fragment has one more attribute called *state*. The state can take five possible values and based on them, the scheduler decides how to handle each fragment.

3.4.1 Priority Scheme

In order to be able to select the best available peers to get the media stream, a priority scheme has been introduced. Priority is used to monitor live peers behavior and is adjusted based on the peer's performance. Specifically, a peer's priority shows the peer's capability to transmit portion of the media stream in terms of the peer's load and the network conditions. It can be seen as a metric for each peer. Based on the priority of each peer, the appropriate number of requests is assigned to it. Then, the round-robin algorithm presented in section 3.4.4 decides which requests should be sent to which peer. The priority scheme is the following:

- The minimum priority for live peer in the live peer table is zero.
- The maximum priority is set to 128.
- When adding a new peer in our peer table, its priority is set to 64 (default value).
- The fluctuation of this priority follows the following rules:
  - Upon the reception of a Payload message, the priority is increased by one.
  - Upon the reception of a Payload message with NAK flag on, the priority is decreased by two.
  - Upon the reception of a Payload message with NOP flag on, the priority is set to zero.
  - Upon a timeout for a requested fragment, the priority is decreased by five.

The weighted algorithm that will be described in section 3.4.3 is using the priorities of the peers and calculates how many fragments each live peer will be requested.

3.4.2 Fragment States

Each fragment stored in the buffer has identified by its state. The state can take five different values that indicate the status of the fragment. Based on the status of the fragments, the scheduler decides rather to request for more packets, re-request a packet or send a packet to the video decoder. The five different states are defined as follows:

EMPTY This state indicates that this is just an empty place in the buffer and the scheduler has not used this buffer position yet.

READY_TO_SEND When a fragment has this state means that the peer have asked for this packet and it has received it. It will be forwarded it to the video decoder when its time comes.

ASK_FOR Indexes in the buffer with ASK_FOR state is packets that we have sent to the application and now we can ask for new fragment and save it to this particular index.

SCHEDULED Fragments with SCHEDULED state are the packets that we have asked for and we are waiting to receive. If we do not receive them on time we are changing to the next state.

RESCHEDULED This state indicates that we have asked the current fragment and we have not received it yet and so we are asking for it again but from a different live peer.

For every state though we have sub-occasions that take part in the decisions the scheduler makes. The scheduler scans the buffer every a specific time interval (see section 3.5). If the state of the buffer index is either EMPTY or ASK_FOR we are asking for a new fragment. However, if the difference between the *Time Difference* attribute of the last received fragment and of the last sent fragment to the video decoder is bigger than a threshold ,then we temporarily pause the requests. The reason is that we have already enough requested fragments. For the current being, this threshold is not dynamic but hard coded with a fixed value.

If the state is READY_TO_SEND the scheduler checks the *Time Difference* of the fragment, that indicates if it is time to send the fragment to the video decoder. If it is early, the fragment is not forwarded to the video decoder and the procedure continues with the next index in the buffer. On the contrary, if it is late and a fragment with greater FID has been already sent to the video decoder, the current fragment is skipped and its state is changed to ASK_FOR.

If a fragment is in the SCHEDULED state, it means that the peer has asked for this fragment and it has not yet been received. After one cycle that the buffer module parses the buffer, if this fragment is still in state SCHEDULED, then its state changes to RE_SCHEDULED. Here, we introduce a variable that shows how many times should we wait until we request the fragment again from another peer. If this variable is greater than a specific threshold, a timeout occurs. In this case, we adjust the priority of the peer that triggered the timeout and we ask the fragment from a new peer. If the FID of the fragment is smaller than the FID of the last sent fragment to the video decoder, it means that we do not have to request it again because we have already passed that fragment. So, we change its state to ASK_FOR and continue with the next fragment in the buffer.

3.4.3 Weighted Algorithm

As we have already mentioned, we are using a weighted algorithm to calculate the number of fragments that we will request from each live peer based on their priorities. The algorithm uses the priorities of the live peers to distribute the requested fragments among them. First, the algorithm checks if we have only one live peer and if we do, it requests all the fragments from this peer. Every time we are requesting from each of the live peers at least one fragment in order to give the chance to everyone to increase its priority. For each live peer, we calculate a first approximation for the number of fragments that we will request from it. Then, we sum up all these approximations and since they might not be always equal with the number of the total fragment requests, we divide this sum by the number of the total fragment requests. In the next step, we use the result from this division to normalize the number of fragments requests for each peer. After the normalization, we round down to the largest integer. At the end there might be some fragments left, that are not assigned to any live peer. Thus, we assign the remaining fragments to the different live peers until there are no fragments left. Next, the proposed algorithm is depicted in pseudo-code format.

An example of our algorithm is shown in Table 3.1. There are three live peers and the number of fragments we have to request is 90. The first and second peer have 128 priority and the third one has 65. So after applying the algorithm the first and the second peers will be requested 36 fragments and the third 18.

Algorithm 1 Pseudo-code representation of the weighted algorithm

N: number of live peers
M: number of fragments to be requested
A: array each element of which stores the number of fragments to be requested from each live peer
P: array that holds the priorities of each peer

if $N = 1$ **then**
    $A[0] \leftarrow M$
    Return
end if
if $M > N$ **then**
    $M \leftarrow M - N$
else
    for $i = 0$ to $i < N$ **do**
        if $M > 0$ **then**
            $A[i] \leftarrow 1$
            $M \leftarrow M - 1$
        else
            $A[i] \leftarrow 0$
        end if
    end for
end if
$sum \leftarrow 0$
for $i = 0$ to $i < N$ **do**
    $d \leftarrow P[i] \div 128$
    $A[i] \leftarrow d \times M$
    $sum \leftarrow sum + A[i]$
end for
$w \leftarrow M \div sum$
for $i = 0$ to $i < N$ **do**
    if $M > N$ **then**
        $A[i] \leftarrow \lfloor w \times A[i] \rfloor + 1$
    else
        $A[i] \leftarrow \lfloor w \times A[i] \rfloor$
    end if
end for
$nsum \leftarrow 0$
for $i = 0$ to $i < N$ **do**
    $nsum \leftarrow nsum + A[i]$
end for
if $nsum < M + N$ **then**
    for $i = 0$ to $i < N$ **do**
        $A[i] \leftarrow A[i] + 1$
        $nsum \leftarrow nsum + 1$
        if $nsum \geq M + N$ **then**
            Break
        end if
    end for
end if

Table 3.1: Example of the weighted algorithm.

		Peer 1	Peer 2	Peer 3	
Number Of Fragments = 90	Priority	128	128	65	
New Number Of Fragments = 87					
	Div	1	1	0.5078	
	1st Approximation	87	87	44.1786	
					Sum = 218.1786
Weight = 0.3987					
	Round Down	35	35	18	
					Nsum = 88
	Add Remaining Fragments	1	1	0	
	Final Fragments per peer	36	36	18	

3.4.4 Round Robin

Based on the weighted algorithm described in the above section, the number of fragments that should be requested from each live peer is calculated. In this section, we describe how these requests are distributed among the live peers. In order to increase the robustness of the system against packet losses that follow a burst mode, we implemented a round robin algorithm. An example of the functionality of this algorithm is presented in Fig. 3.33.

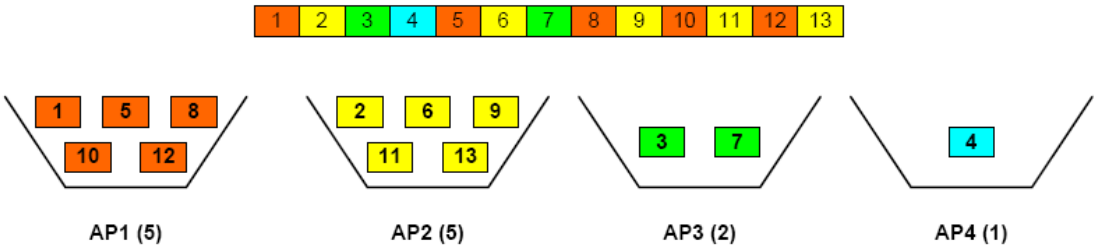


Figure 3.33: Example of a round-robin algorithm in the distribution of requests.

Each bucket represents a live peer. The number in brackets below each bucket represents the number of fragments to be requested from each live peer. This number is calculated based on the algorithm described in the above section. From the figure, we can see that we request from each peer a non-continuous set of fragments. The reason is to avoid events of buffer starvation due to burst packet losses that may happen in the network.

3.5 Implementation Overview

For the development of the Daedalus prototype, we chose to use the C++ programming language. The reason was to minimize CPU load and memory footprint, while at the same time having the option to program based on either procedural or object oriented concept. In addition, portability to a variety of hardware platforms was another factor that lead us to this choice. Specifically, we used the open source GNU Compiler Collection

[51] (the GNU C++ compiler for development and GNU Debugger for debugging purposes). The Linux Operating System [52] was chosen as the default environment for the development and the testing of our prototype. A variety of hardware platforms were used during this process (including both 32 bit and 64 bit architectures). Finally, the Subversion version control system [53] was adopted in order to achieve better collaboration.

The application is based on a hybrid system; both event-driven and time-driven. It is event-driven because upon an event (e.g. packet reception), a predefined mechanism is executed and performs the appropriate actions. It is time-driven because at certain time instances messages are sent from a peer to its neighbors (e.g. periodic KeepAlive messages). In addition, multithreaded programming was introduced in order to better manage the different modules described in section 3.2. Debugging was another factor that encouraged the use of treads. A realtime application similar to our prototype cannot be extensively debugged without the use of threads because of the huge amount of data exchanged in short time intervals among peers. The following threads were implemented:

Communication Thread The communication thread starts in the *INIT* state in which it initializes the socket each peer binds to. Afterwards, it sends the first Join message to the Master Node in order to get a list of peers that currently receive the channel this peer wants to join. Upon the reception of the KeepAlive message from the Master Node , the state changes to *OPERATIONAL* and the tread loops for ever waiting for message from other peers. Upon the reception of a message, the necessary processing for the integrity of the message takes place and the appropriate actions are performed. Then it waits for the next packet and so on (event driven model). If no message is received for a specific time interval, the state of the thread changes from *OPERATIONAL* to *INIT* in order to achieve communication with other peers.

Scheduler Thread The scheduler thread performs two very critical operations: it forwards the fragments stored locally in the buffer to the video decoder and performs requests to get new fragments. It is time-driven with a sleep period of 50 milliseconds. In a commercial implementation this sleep period should be adapted dynamically based on the traffic conditions of the network and the multimedia stream characteristics. However, for the purposes of our prototype, a fixed value optimized for a specific stream characteristics was introduced. The whole buffer is parsed at each working cycle (not sleeping period) and based on the state of each fragment (see section 3.4) the appropriate action is performed. In order to follow the rate in the Media Injector , a maximum number of requested fragments was defined for each cycle. In a commercial implementation, this value should be dynamically adapted based on the traffic conditions, the load of the peer, and the characteristics of the multimedia stream. For the purpose of our prototype a specific algorithm was adopted and was optimized for the characteristics of the media stream we used for testing. The formula that defines the maximum number of requests per cycle is presented in Eq. 3.1.

$$max\_reqs = 25 * live\_peers\_num * reqs\_per\_peer \tag{3.1}$$

Parameter *live_peers_num* represents the number of peers that this pees gets part of the media stream, and *reqs_per_peer* represents an additional number of requests for each peer. For the prototype this number was set to five. Also we have to mention that for the purposes of the prototype we have set the maximum number of live peers to four.

The playout rate to the video decoder of a peer follows the rate of the video decoder in the Media Injector by exploiting the *Time Difference* field of the Payload message described in section 3.3.7.

The state diagram of the scheduler module in our implementation is depicted in Fig. 3.34. Three states are defined in the diagram:

- *BUF_INIT:* It the initial state of the buffer in which the peer sends Join messages to the peers in its peer table in order to start receiving the media content. When the peer receives the first Payload message and marks the peer that sent this message as live, the state of the scheduler changes to *BUF_NORMAL.*
- *BUF_NORMAL:* It the operational state of the scheduler. While in this state, it asks for fragments based on the algorithm described in section 3.4, and sends media content to the video decoder as well (as described above). While parsing the buffer, if the difference between the *Time difference* field of a *RESCHEDULED* fragment and the last received fragment is greater than a specific threshold (*RESYNC_THRESHOLD*), then the state of the scheduler changes to *BUF_RESYNCING*.
- *BUF_RESYNCING:* Being in this state means that for some reason we lost the synchronization with the peers we used to get the media content and we need to resynchronize. Thus, Request messages with the RESYNC flag on (see section 3.3.6) are sent to live peers of this peer. Upon the reception of a Payload message, the state changes again to *BUF_NORMAL* as resynchronization is achieved. If there are no live peers, then the state changes to *BUF_INIT* in order to find new live peers.

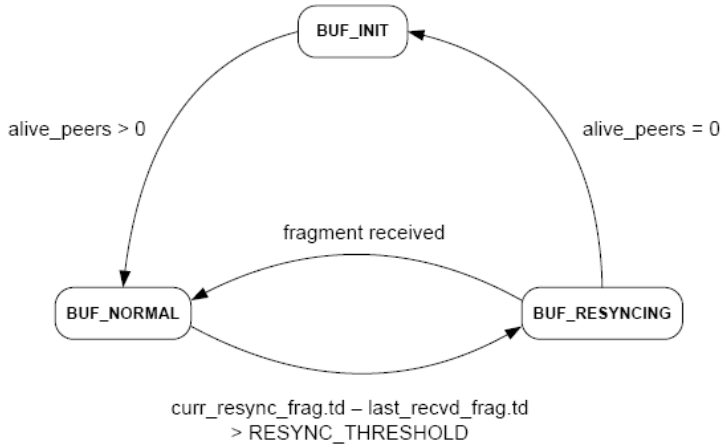


Figure 3.34: State diagram of the scheduler implementation.

KeepAlive Thread The KeepAlive thread sends every 10 seconds a KeepAlive message to neighboring peers that are at hop zero from this peer (peers that have direct communication with this peer). In addition, if a peer has not sent any message to this peer for 25 seconds, this peer is removed from the peer table.

Monitor Thread This thread was implemented in order to continuously monitor the status of the application and be able to observe any kind of strange behavior. Moreover, statistics about the traffic exchange among the peers were presented in the screen.

3.6 Conclusions

In this chapter our proposed architecture was presented. We described the architectural components of the proposed network and presented the specification of the protocol introduced to manage this communication. In order to provide a better understanding of Daedalus's functionality, some message flowcharts were presented. We also described our proposed scheduler which is the core component of the architecture as currently only a simple request format is adopted (compared to an extended request format in which push methods are implemented). Finally, a brief description of the implementation design was presented. In the next chapter, the measurements we conducted are presented. The simulation setup is presented as well.

Chapter 4

Measurements

In order to evaluate the Daedalus architecture, we designed and conducted a set of measurements. Some of these measurements were designed to evaluate Daedalus's efficiency in comparison to other existing solutions for peer-to-peer live streaming; while others examine the implications in terms of network behavior due to Daedalus's functionality.

In this chapter, we describe the measurements performed and present the setup for each experiment. During the measurements we setup fake network environments in order to purposely introduce specific delays and loss rates for each of the links. We used a network tool called *tc* that is included in the *Iproute2* package [54]; along with *Netem* [55], which provides a Network Emulation functionality for testing protocols by emulating the properties of wide area networks. Iproute2 is a collection of utilities for controlling TCP/IP networking and traffic control in Linux. Furthermore, based on the results we generate graphs that make it easy to comprehend and understand the results.

Before we continue with the description of the measurements performed, we should mention that our prototype was optimize to work based on a specific type of media streams. The reason is the time constraints we had n order to finish our thesis work. Thus, we used an MPEG-2 variable bit rate stream with $1.5 Mbps$ rate.

4.1 Join Overhead

This join overhead metric represents the number of messages required to get a list of peers that are capable of transmitting the media stream. In our proposal, this metric is strictly a function of the number of live peers. Specifically, as we mentioned in section 3.5, each peer receives the media stream from a specific number of peers called live peers. Thus, based on the procedure that is followed by a peer in order to join the system, the *Join Overhead* is:

$$JO = 2 + ((max\_live\_peers - live\_peers) \times 2) \times 4 \tag{4.1}$$

This formula is better comprehended if we consider the join procedure illustrated in Fig. 4.1. First, the peer sends a Join message to the Master Node in order to get the list of peers that are currently receiving this channel. The Master Node responds with a KeepAlive message. This procedure results in the first factor (2) in Eq. 4.1. Next, the peer will send Join messages to a number of peers that is equal to $(max\_alive\_peers - alive\_peers) \times 2$. Where max_alive_peers represents the maximum number of live peers (in our prototype it is equal to 4), and $alive\_peers$ represents the number of current live peers of this peer. Notice, that when a peer tries to join the system for the first time, this value is equal to zero. Each peer that receives a Join

message, will respond with a KeepAlive, an Indication, and a Payload message (see section 3.3.2). Thus, the initial Join message plus the three messages that sent a response to it, result in the last factor (4) of Eq. 4.1.

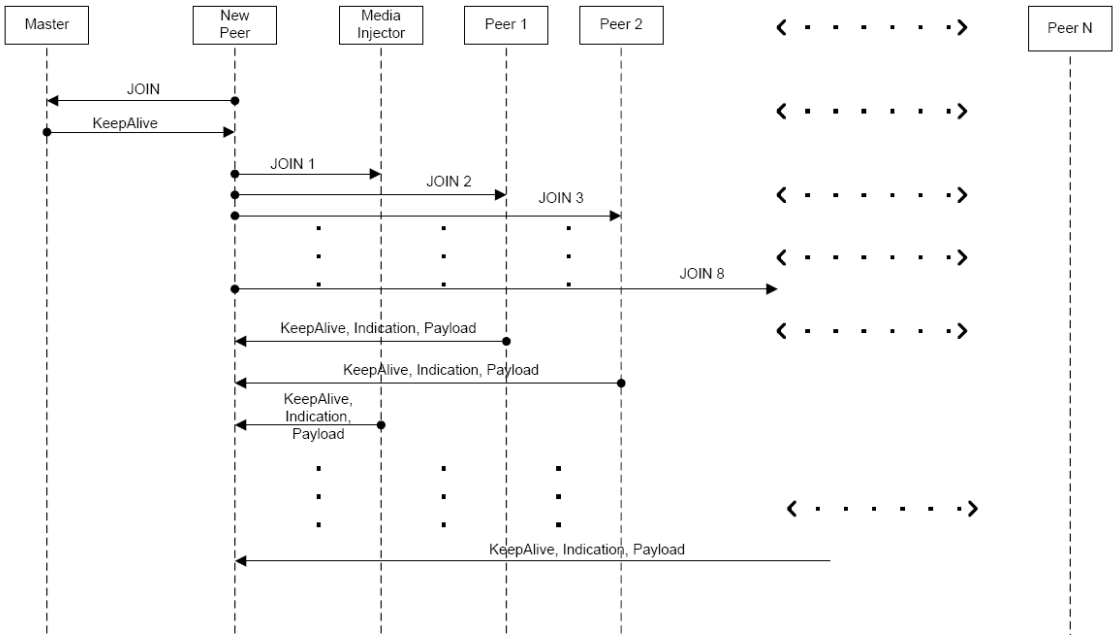


Figure 4.1: Join procedure message flowchart.

4.2 Control Overhead

The *Control Overhead* metric represents the percentage of the total traffic produced by the Daedalus network that transfers control information needed for peer discovery, maintenance of the status of the network and so on; but not actual payload. This metric shows the efficiency of the proposed solution in terms of bandwidth utilization, as it describes what portion of the bandwidth is **not** used to transfer actual media content.

In order to evaluate this metric, we used counters in each peer in order to store the count of the number of bytes for all the different messages exchanged. Then, we run the application with 20 peers in different network environments. Based on the measurements performed, this number of peers is adequate to generate a reliable estimate and extendable to larger network topologies. The time period for this measurement was set to 10 minutes - thus enabling the system to reach a steady state.

Our first measurement was conducted under almost ideal network conditions using machines that were in the same local network. Based on the results, we generate a graph that shows the amount of the control traffic and the actual multimedia content traffic. These results are depicted in Fig. 4.2. This shows that the control overhead of our protocol is roughly 2%. Furthermore, Fig. 4.3 presents a more detailed analysis of the control traffic. Specifically, it shows the percentage of the control traffic generated by each type of control message. As one can see, the majority of the control traffic is generated by Request messages. Notice, that this type of control traffic can be almost completely eliminated if we change the method by which a peer receives the media stream (see chapter 3).

Next, we performed the same measurement, but we introduced a link delay equal to 60*msec*. Thus, the total

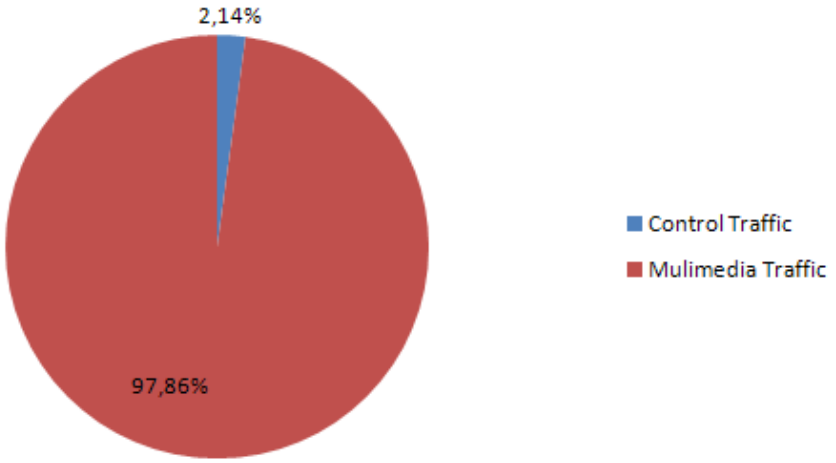


Figure 4.2: Control overhead under ideal network conditions.

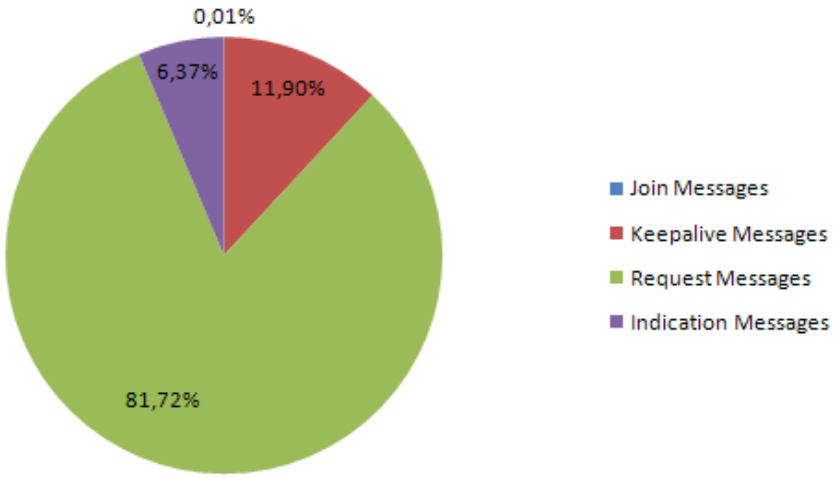


Figure 4.3: Analysis of the control overhead under ideal network conditions.

RTT between two peers was 120*msec*. Notice, that this delay was deterministically set for **all** links between any two peers. This value is the maximum value of RTT that our prototype can handle based on the current implementation and the characteristics of the media stream we used for the measurements. The results are depicted in Fig. 4.4 and show that the delay caused a slight increase in the control overhead. Compared with the control overhead under ideal network conditions, we observe an increase of roughly 10%. This increase is caused by the retransmissions that are triggered because of timeouts. Specifically, as we mentioned above, the largest portion of the control overhead is because of Request messages. When we introduced delay in the network, many requested fragments needed to be re-requested because of timeouts (they did not arrive when expected). Thus, the number of Request messages increased and cause this increase in the control overhead. The exact percent of the increase is closely related with the characteristics of the media stream and the characteristics of the delay we introduced in the network.

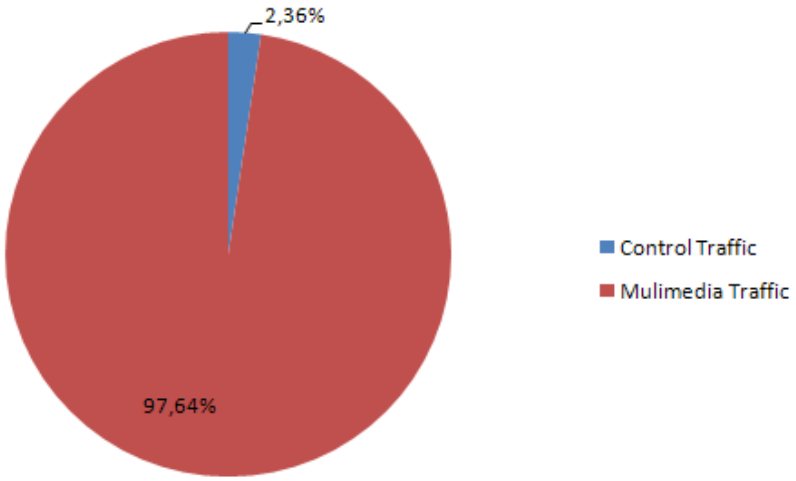


Figure 4.4: Control overhead under network conditions with delay.

We also measured the control overhead measurement when a 6% packet loss is introduced on every link. Thus, every message that is sent has a 6% possibility of being lost. Again, this packet loss probability was set for all links between any two peers. The results are shown in Fig. 4.5 and similar to the former case (delay), we also see an increase in the control overhead as compared to either the 0 delay or the case of 60*msec* (one-way) link delay.

The observed increase of control overhead for both delay and loss cases can be justified by the retransmissions of the lost or delayed messages. Before performing these measurements, we expected an increase in the control overhead because some packets would be lost or delayed, timeouts would be triggered, and new requests for these packets would be generated. The increase in the overhead though, is small and indicates the good behavior of our protocol.

4.3 Initial Delay

Initial Delay is defined as the time needed to start receiving the media stream from the network and send it to the video decoder. This is a very critical metric when it comes to live streaming systems because it is highly correlated with the end user's experience. As presented in section 2.3, most of the existing proposals suffer

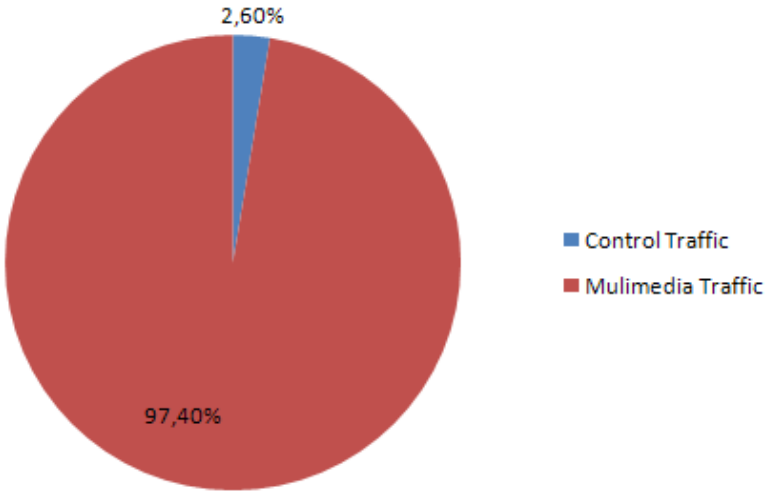


Figure 4.5: Control overhead under network conditions with packet loss.

from long start-up delays. In our proposal, we tried to minimize this time interval, as we consider it one of the most important factor for users' satisfaction. We first measure the initial delay under almost ideal network conditions in a local network. The results are shown in Fig. 4.6 and present the initial delay of each of the 19 peers that we started one after another (with the time between startup being 1 minute). We waited one minute after the start of each peer in order for the network to become stable; so as not to have errors in the measurements due to inter-peer startup effects. However, after performing the same measurement without waiting interval between starting consecutive peers, the results showed similar behavior (these results are shown in Fig. 4.7). In Fig. 4.6, we can see that the initial delay after seventeen peers converges to a value of approximately 650*msec*, which is more than acceptable for a live streaming system. Recall, that this value corresponds to the time needed between when a user presses the button to watch a specific channel until full video quality appears on the user's screen.

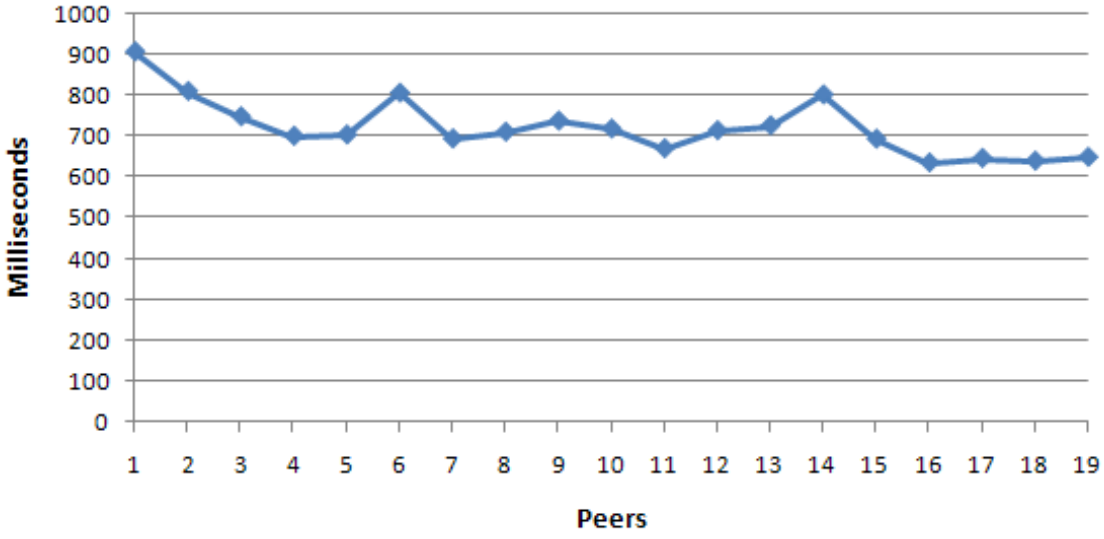


Figure 4.6: Initial delay under ideal network conditions. Peers started with time difference 60sec.

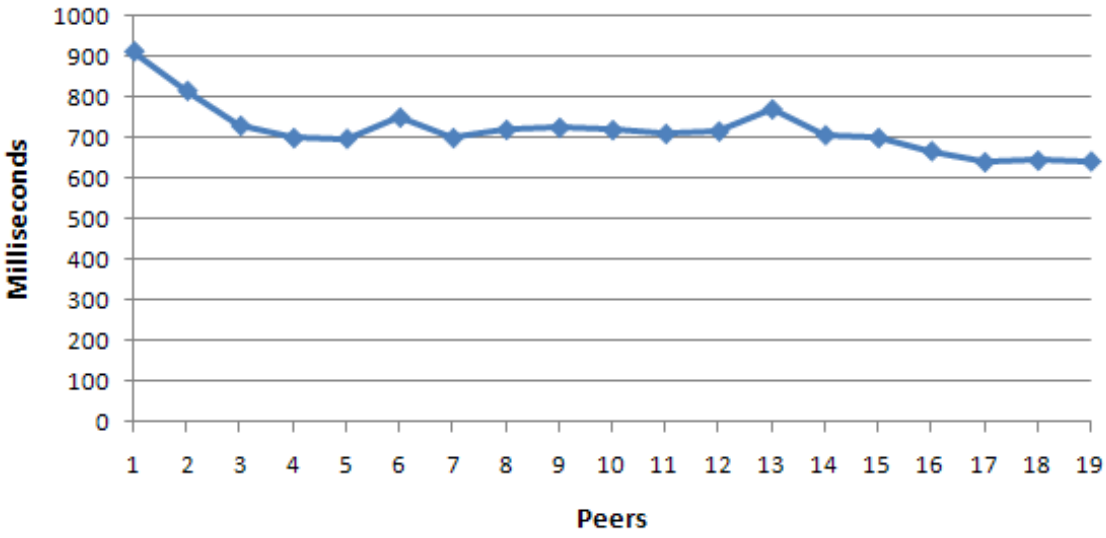


Figure 4.7: Initial delay under ideal network conditions. Peers started with almost no time difference.

Next, we performed the same measurement in a network that suffers from 60*msec* (one-way) delay on every link. Fig. 4.8 presents the results obtained under these conditions. Here, we can see that the initial delay has increased by approximately 350*msec* compared to the former case. Here, the RTT is around 120*msec*, a reader might wonder why an increase in the initial delay results in a much higher initial delay than the case above. However, recall that in order to have adequate image quality when the video is presented on the screen, we have to have buffered a certain number of fragments in our local buffer. In order to receive this number of fragments we have to send an additional message with requests except the first Join message to the master and the second message of the Join to the live peers and the request for payload. So these three messages and its requests add a $360 \times (3 * 120)$*msec* delay, which is the additional delay we measured. As it takes each fragment longer to get to the local device and it takes longer for the peers to know that this device is interested in these fragments, this add significantly to the delay. As one can observe in the figure, there is an increase of the delay in the range of peers from 6 to 13. The reason is that the system is not in steady state yet, and this increase is the result. When the system reaches a steady state (after the 14^{th} peer), the initial delay converges to a value.

Finally, we performed the same measurement after introducing 6% packet loss on every link. The results are depicted in Fig. 4.9. We can see that in steady state (after 15 peers) the system performs similar to the case in which there was no delay or packet loss. However, when there are only a few peers in the system, the initial delay is much higher. The reason is that when the number of peers in the network increases, the probability that a packet will be lost when multiple peers send it, decreases.

Notice also that the reason that our graphs do not continue to decrease is that we have set a limit on the number of live peers to 4. So if we increase this limit we will notice a further decrease of the initial delay. To sum up, it is obvious that network delays and packet loss affect the initial delay. Nonetheless, its value stays low and is always under 1.3*sec* (for the range of numbers of peers, delay, and loss values which we have tested).

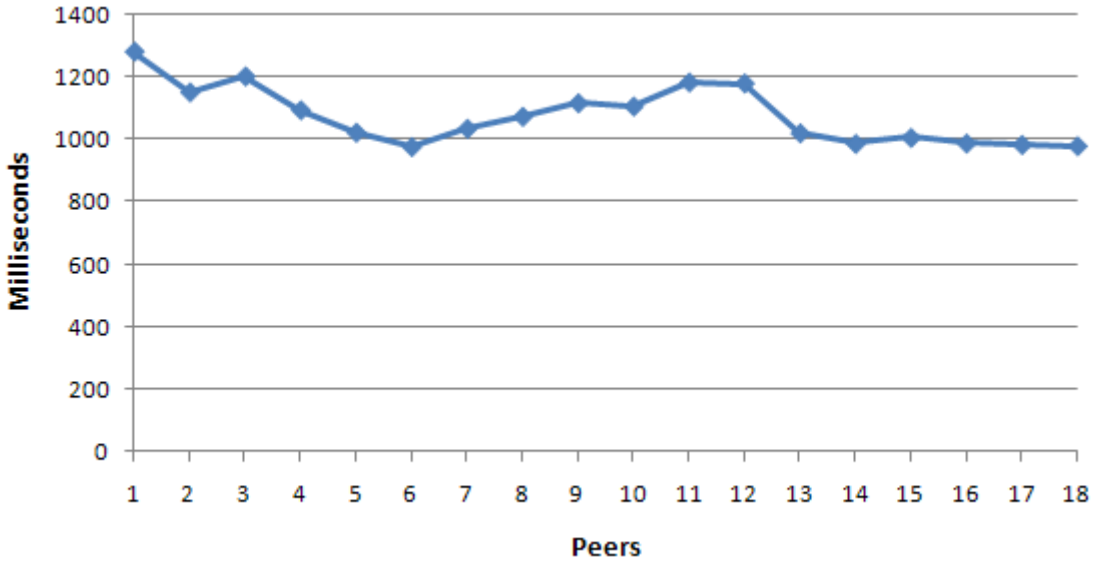


Figure 4.8: Initial delay under network conditions with delay.

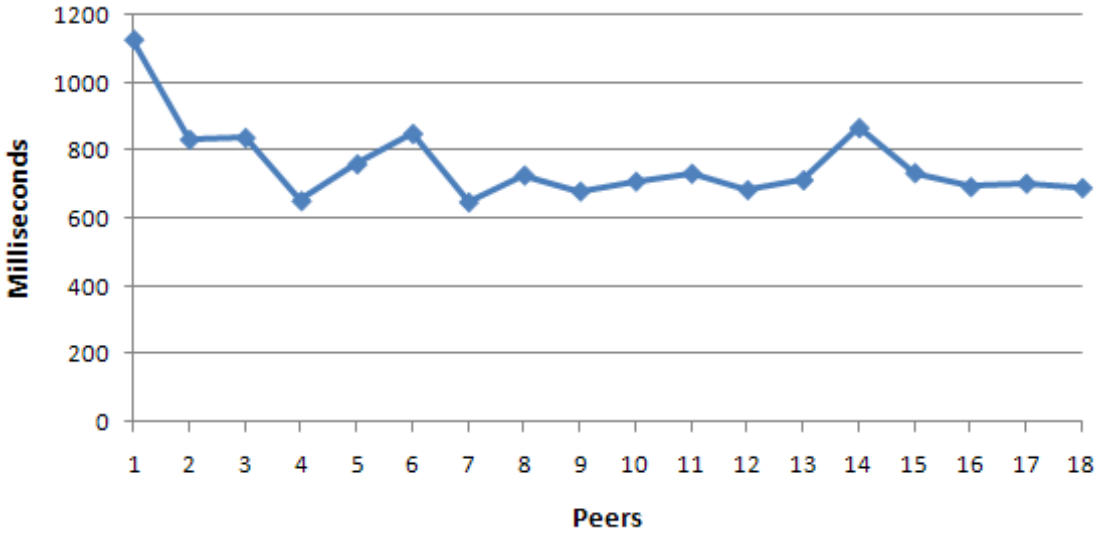


Figure 4.9: Initial delay under network conditions with packet loss.

4.4 Transition Delay

The time period needed to find and contact a new peer in order to get a portion of the media stream because a live peer is not efficient anymore is called the *Transition Delay*. The transition delay consists of the time that a peer waits until it decides that one or more of its live peers are inefficient (let us define it as a *waiting time*), plus the time from the Join message it sends to a new peer until it begins to receive payload from it. The second time interval is relevant when there are delays in the network. Following our measurements we found out that it takes between 2.5*msec* and 122.5*msec* following the Join message until we begin to get the request payloads from the new peer. The first value (2.5*msec*) corresponds to the case that there are no delays in the network, and only processing delays occur. The second value (122.5*msec*) corresponds to the case of 60*msec* delay per link, which adds 120*msec* RTT to the processing time hence giving exactly 122.5ms.

However, in order to find out what is the optimal value for the transition delay we had to measure the behavior of another factor while changing the transition delay. We decided that this factor should be the control overhead since is directly affected and could give us a hint about what is the optimal value for transition delay. As we mentioned earlier, the transition delay consists of two time intervals. Since the second time interval depends only on the network delays, therefore we decided to experiment with the first time interval, the waiting time. Based on our experience with this prototype over several months, we concluded that the value of the waiting time should range between 300*msec* to 800*msec*. The lower limit (300*msec*) guarantees that the peer is actually inefficient (as opposed to simply being due to a long RTT), and that we do not consider temporal changes in the network. The upper limit (800*msec*) was chosen in such a way that we avoid problems with the continuous supply of the video stream because one of the live peers is not appropriate anymore. As we can see in Fig. 4.10, the difference in the control overhead (as a percentage of all traffic) for the various values of the waiting time is small. So we choose not to wait too long and to replace an alive peer when the delay exceeds the 300*msec*. Consequently, we set the optimal value of the waiting time to 300*msec*, thus the transition delay is around 302.5*msec* to 422.5*msec*.

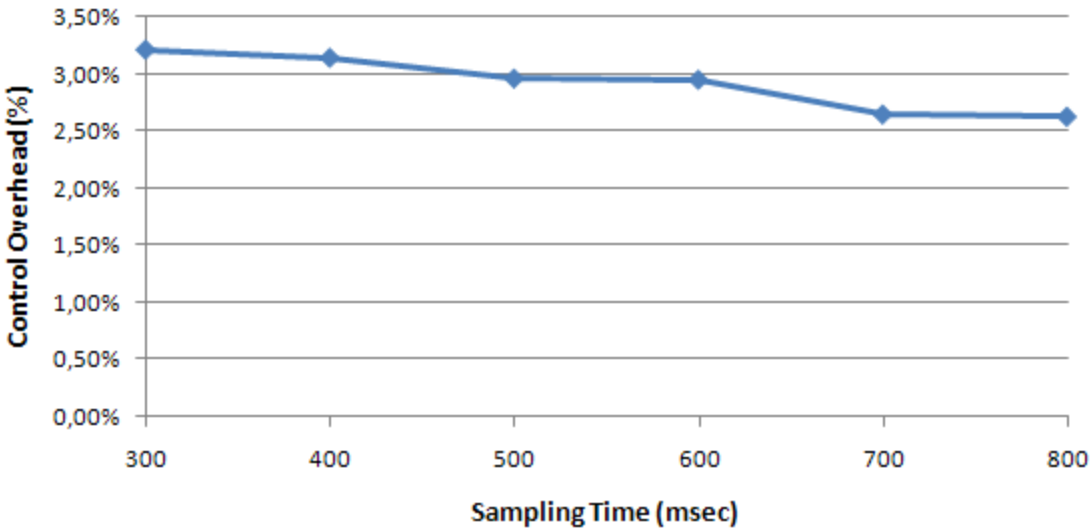


Figure 4.10: Control Overhead as a function of different values of the transition delay.

4.5 Optimal Change Point

The *Optimal Change Point* is defined as the point in time that a peer should try to find another peer to get a portion of the media stream because one of current peers cannot fulfill the required QoS. In theory, this metric depends on the network conditions (e.g. delay or packet loss) and the peer's load (e.g. how fast the peer responds). However, in our case, the use of a priority scheme (see section 3.4.1) provides a mechanism to continuously monitor the status of the connection with a specific peer. Thus, when its priority drops below a certain threshold for a continuous time interval, a procedure to find a better peer is triggered. Then, the peer asks the peers in its peer table to send it their buffer map through an Indication message. Based on this information, when a peer is ahead of this peer for a specific time interval, it is chosen to be a live peer for this peer.

Consequently, in the Daedalus architecture, the problem is to find the optimal priority threshold point in order to trigger the procedure to find a new better peer, in combination with the time shift that a peer is ahead of the peer that looks for more live peers. A strict selection of these values will cause an unnecessary trigger and generate unnecessary additional control overhead in the network. On the other hand, a loose selection of these values will possibly result in decreased video quality. An optimal solution would be to adapt these values based on the media stream characteristics. However, for the purposes of our prototype, optimum values were estimated based on measurements performed considering a specific media stream. For the first parameter, based on our experience during the whole time period of the prototype development, we chose to set its value equal to 32. The reason is that if a live peer's priority is below this value, then this peer is unstable and it is not reliable to get the media stream. Thus, the optimal change point metric finally turns out to be the time interval that a potential is ahead of the peer that needs to find more live peers.

For the purposes of our measurements we introduce a *connectivity factor* that indicates how good the sharing ratio is between the peers. This connectivity factor is produced by dividing the sum of the live peers in the channel by the theoretically maximum sum of the live peers of all the peers in the channel. In order to change a live peer we are comparing the time difference of the last fragment sent to the video decoder of the peer with the last received fragment from the candidate peer. If this difference is lower than a threshold, we exchange the live peer with the candidate peer. By changing this threshold, we can find the optimal change point that will maximize the connectivity factor. Fig. 4.11 depicts the results of our measurements and indicates that the optimal point to choose set a peer as live is when it is 100 and 200 msec ahead.

4.6 Daedalus Gain Factor over Unicast

The Daedalus gain factor over unicast is the percentage gain in terms of content bytes delivered per unit time by using Daedalus versus using a unicast media streaming from a single source. Where a factor of 0% would be the same performance as a unicast media streaming and a value of 100% would be twice the data per unit time vs. unicast media streaming from a single source. Fig. 4.13 shows how the Daedalus gain factor over unicast increases as a function of the number of peers. As someone can see after a certain point the gain becomes stable at a value of around $250-300\%$. Fig. 4.12 shows how the Media Injector's load increases as new peers are added to the system. We show the load for three cases; first for our prototype, second for our theoretical solution, and last for the unicast solution. It is obvious that even with our prototype solution the decrease of the load in comparison with the unicast is huge.

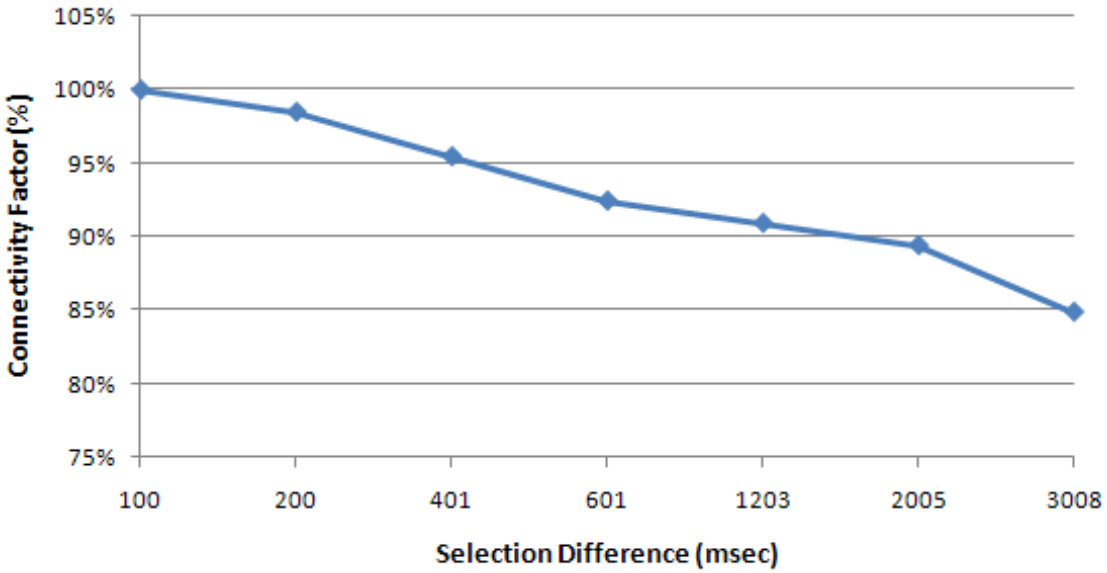


Figure 4.11: Connectivity Factor based on the Optimal Change Point decision.

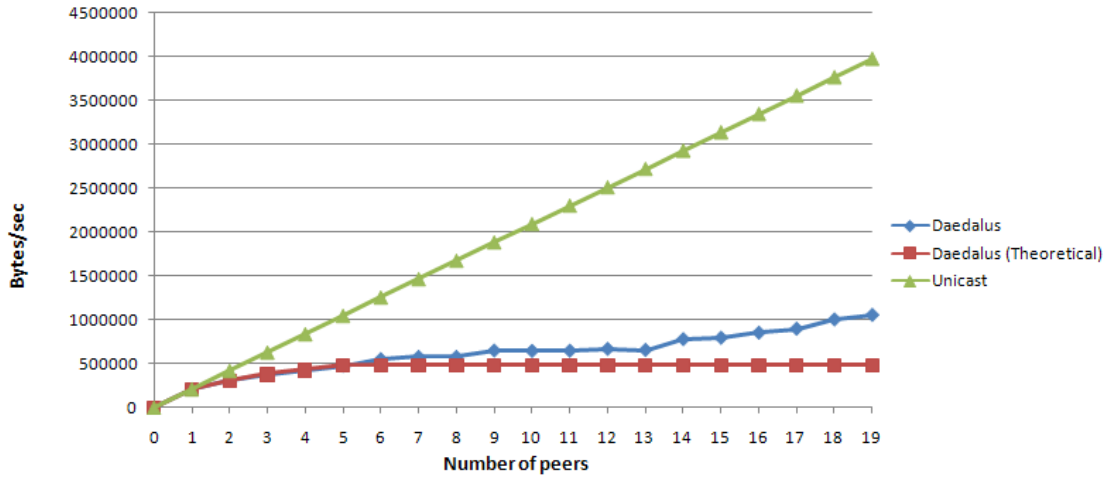


Figure 4.12: Media Injector output traffic in comparison with unicast.

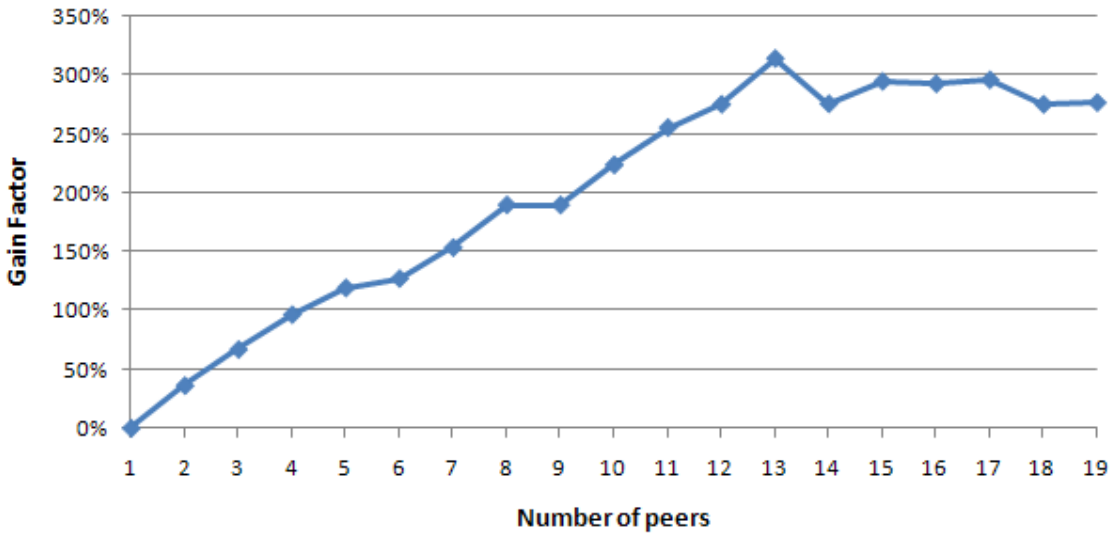


Figure 4.13: Daedalus gain factor over unicast.

4.7 Summary

In order to visualize the network overlay created by Daedalus, we built a module that collects information about our current network topology and sends it to a web server. We used PHP [56] to create a server that listens to the specific port that we are sending the data. The server collects the data and saves it to a file that we use later to construct two graphs with the help of Graphviz [57]. The first graph shows the network topology real time and is updated every five seconds. The second one presents the output bandwidth gain of the media injector.

Fig. 4.14 is a screenshot that shows a network with three peers. The blue polygon represents the Master Node , the green polygon represents the Media Injector , and the circle boxes represents the peers. The dotted lines shows that there is direct communication between the nodes and it is sent KeepAlive messages (control traffic). The normal lines show the exchange of actual media content.

Fig. 4.15 is also a screenshot random in time that presents the actual output traffic of the Media Injector in comparison with a unicast solution. The green part of the graph shows the real output traffic while the red one depicts how the output traffic would be if we were using unicast. In this example with four active peers, the actual output traffic is 1.5$Mbps$, and the traffic using unicast reaches almost 6$Mbps$.

In Table 4.1 we are presenting a summary of our measurements giving a description for each of them. The first column present the type of the measurement, the second shows the unit that we are using for our results, the third describes the measurement, and the last one indicates the importance of the measurement.

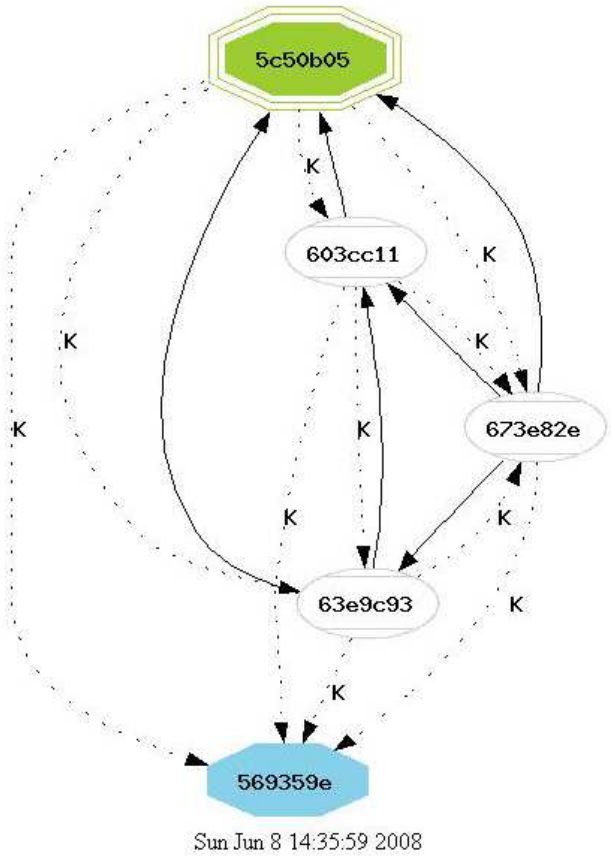


Figure 4.14: Network topology.

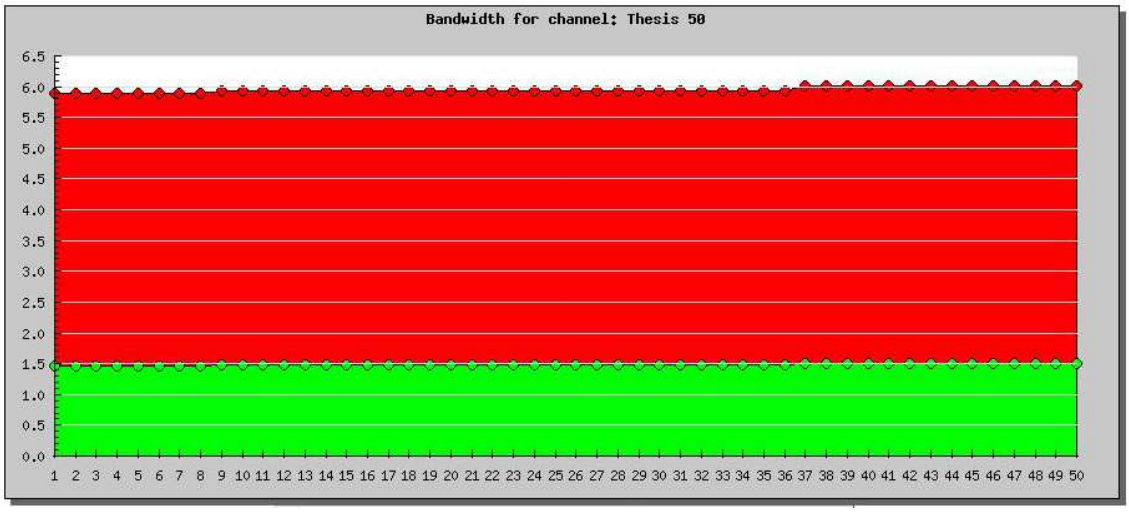


Figure 4.15: Media Injector output traffic in comparison with unicast solution.

Table 4.1: Summary of evaluation metrics

Factor	Units	Description	Importance
Join Overhead	Pure Number	Number of messages needed to start receiving the media stream	Low
Control Overhead	Percentage (%)	Percentage of the total traffic produced by the Daedalus network that transfers not actual payload but control information	Medium
Initial Delay	Milliseconds (msec)	The time needed to start receiving the media stream from the network and start sending it to the video decoder	High
Transition Delay	Milliseconds (msec)	The time period needed to find and contact a new peer in order to get portion of the media stream because a live peer is inefficient	High
Optimal Change Point	Milliseconds (msec)	The time interval that a potential is ahead of the peer that needs to find more live peers	High
Gain Factor	Percentage (%)	The percentage gain in terms of content bytes delivered per unit time by using Daedalus versus using unicast media streaming from a single source	High

Chapter 5

Evaluation

The evaluation of Daedalus architecture is presented in this chapter. Here we analyze and discuss our measurements results and show why our solution introduces something new in comparison with existing solutions.

We will start our discussion by presenting how our system performs for the challenging tasks mentioned in section 2.3.1. As depicted in Fig. 4.10, the transition delay ranges between 302.5*msec* and 422.5*msec*. Thus, if a peer becomes inefficient, a new peer will be found quick enough in order to avoid buffer starvation and decreased video quality. The same applies in the case that a node fails or leaves the system, because the system will react in the same way. It will be able to recover quickly and the user's experience will not be affected, since there will be sufficient live peers to provide the media stream. Therefore, the scalability and the service interruption constraints are satisfied.

Furthermore, the start-up delay task is low, as we presented in Figs. 4.6, 4.8, and 4.9. Specifically, the initial delay does not exceed 1300*msec* even in the cases that we have loss and delays in the network. We have to mention that none of the existing solutions that we know has succeeded in having such a small initial delay. Some of the solutions we covered in section 2.3 had 10*sec* of initial delay.

Based upon the measurements of the control overhead in section 4.2, we observe that even in cases of loss and delays, the control overhead is less than 2.6% of the total network traffic. According to the task efficiency (see section 2.3.1) our solution is even feasible for the Mobile IPTV, since Daedalus's overhead does not add considerable load in a wireless link, and high bandwidth utilization in terms of pure multimedia traffic transmission can be achieved. Recall, that the Request messages constitute the largest portion of the control overhead. Thus, if a hybrid (pull and push) or a pure push method is used in order to transfer the media stream, then the control overhead will be almost eliminated.

In addition, our priority scheme and the algorithm that defines how the priority values fluctuate based on a peer's behavior in combination with the distribution of the requests per peer, makes the peer selection procedure efficient. Moreover, if a live peer does not behave correctly and efficiently, our monitoring mechanism will choose the next appropriate peer to ask for fragments. The hybrid method introduced in Daedalus for a peer to receive the media stream, may further decrease the control overhead and the total network load, thus greatly increasing our system's efficiency compared to the most solutions that use either pull or push.

Moreover, as we saw in Fig. 4.12, 4.15, our solution provides a huge decrease in the load on the Media Injector in comparison to a unicast solution. Notice, that in a full Daedalus implementation, the expected gain will be much higher that the gain we obtained based on the current prototype. Fig. 4.13 shows that the Daedalus gain factor over unicast increases as the number of peers in the network increases. The reason is that after an sufficient number of peers exist, new peers that join the system will get the media stream not from the Media Injector , but from other existing peers. This behavior is depicted in Fig. 4.12, in the curve "Daedalus (theoretical)". There it is shown that after a certain number of peers have joined the system, the Media

Injector output traffic remains constant and equal to five times the traffic of one unicast flow, independently of the number of peers that continue to join the system. Thus, as the number of peers increases, the gain factor increases as well.

Finally, comparing with the IP multicast architecture, Daedalus does not require any specific configuration in the network or any additional installation. Daedalus is fully compatible with the existing networks and operation systems.

Chapter 6

Conclusions

In this master thesis, a new approach for live multimedia streaming based on peer-to-peer was presented. Daedalus is a mesh-based peer-to-peer architecture which provides the possibility to efficiently transmit live multimedia content.

In Chapter 1, we gave a brief historical overview of the IPTV evolution, and introduced the reader into the problem we are going to investigate.

In Chapter 2, we first described the related background regarding IPTV and peer-to-peer systems. We presented the current solutions regarding IPTV and the limitations they impose to the network configuration. Next, we presented the concept behind peer-to-peer systems, the historical evolution, and the different types. We also described their advantages and disadvantages, and showed why peer-to-peer systems constitute a promising solution for live IPTV distribution. Then, we presented the most interesting and popular solutions for live multimedia streaming based on peer-to-peer architecture, both academic and commercial ones. We concluded the chapter with a summary of open issues regarding peer-to-peer systems for live multimedia streaming.

In Chapter 3, we described our proposed solution called Daedalus. First, we presented the overall architecture and the functionality of the entities that compose it. Then, we gave the functional overview of the LSPP protocol and its specification. Next, a detailed description of the packet scheduler introduced in Daedalus was given, and the functionality of its core components was discussed. Finally, an implementation overview of the prototype we developed in order to evaluate Daedalus was presented.

In Chapter 4, we discussed the measurements we performed in order to evaluate our proposal. We first presented the setup of the environment we used. Then, we presented the metrics we chose to measure and the results we obtained from these measurements. A discussion on these results and their importance was also presented.

In Chapter 5, we compared our solution with the existing ones based on the results we obtained from our measurements. We discussed Daedalus advantages and disadvantages compared to the existing solutions, and gave hints to further boost Daedalus performance and efficiency.

To sum up, Daedalus is a sophisticated architecture for live IPTV distribution. It introduces a variety of mechanisms to exploit most of the advantages of peer-to-peer systems while at the same time tackling all the problems they introduce when it comes to live multimedia streaming. In addition, Daedalus provides the capability to exploit the existing IPTV infrastructure, such as IP multicast, in order to further boost its performance when multicast is available.

Notice, that not all of the potential capabilities of Daedalus are investigated in this report. The prototype we implemented and tested does not cover all the functionality Daedalus can provide. The reason is that Daedalus, as a complete architecture, is complicated enough, and a full implementation and extended testing

could not be conducted in the context of a master thesis report. In the following section, we present open issues regarding Daedalus and potential enhancements that will further increase its performance.

6.1 Future work

In this thesis, in order to evaluate our proposed solution, we implemented a prototype application. However, because of time constraints, a full deployment of the whole architecture was not possible. Thus, some parts should be re-implemented or should be extended in order to provide the full functionality.

First, as we have mentioned in the introduction of chapter 4, our prototype is optimized to function for a specific type of media stream. However, this will not be the case for a real application. Thus, all the thresholds that are now statically set to specific values that were obtained by observation, should be calculated on the fly based on algorithms that take into consideration the media stream's characteristics and the network conditions.

In addition, the LSPP protocol needs to be extended in order to support the additional functionality which is needed to compute these settings dynamically. Moreover, options in the existing specification that are not yet implemented (e.g. Realtime flag in Request, Payload messages, Resynchronization flag in Request message, and so on) should be implemented, as they extend the functionality of Daedalus and increase its efficiency. The LSPP protocol should also be verified using any verification tool, such as SDL, in order to extensively test its correctness. Finally, redundancy mechanisms need to be introduced in order to increase the reliability of the information transmission.

As far as the scheduler is concerned, information contained in the Indication message (see section 3.3.4) should be exploited in order to improve the decision making procedure when it comes to decide which fragment should be requested from each live peer. In addition, different methods to get the media stream from other peers should be implemented, besides the existing pull method. We expect that the adoption of a hybrid (combination of pull and push methods) will greatly increase the performance the system.

Furthermore, further investigation should be performed in order to exploit the characteristics of the media stream so as to enhance Daedalus's efficiency. For example, in the case of an MPEG-4 media stream, the loss of an I-frame causes the following P and B frames to be useless. Thus, fragments that contain such information should be discarded, and a fragment with the next I-frame should be requested. Such a mechanism increases the efficiency of the bandwidth utilization and should improve the experience of the end user.

Moreover, most of the existing solutions suffer from firewall and NAT traversal problems. Daedalus capability to tackle this problem is not investigated in the current report. However, this aspect of the architecture should be further investigated and extensions in the LSPP protocol should be introduced.

Finally, extensive measurements and testing is needed in order to further investigate Daedalus's performance. For example, the probability of temporary buffer starvation should be estimated through extensive simulations.

References

[1] Niklas Zennstrom and Janus Friis. Joost. http://www.joost.com/ (last visited: 2008-01-30).

[2] Ramesh Jain. I Want My IPTV. *IEEE MultiMedia*, 12(3):95–96, 2005.

[3] Lawrence Harte. *IPTV Basics, Technology, Operation and Services*. Althos Publishing, 2007.

[4] Peter Arberg, Torborn Cagenius, Olle V. Tidblad, Mats Ullerstig, and Phil Winterbottom. Network Infrastructure For IPTV. Technical Report 3, Ericsson, 2007.

[5] Gilbert Held. *Understanding IPTV*. Auerbach Publications, 2007.

[6] Radia Perlman. *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols (2nd Edition)*. Addison-Wesley Professional Computing Series, 1999.

[7] Jason Eisner. Introduction to IGMP for IPTV Networks. White paper, Juniper Networks, Inc., October 2007. Available online (12 pages).

[8] Carolyn Wales, Sukun Kim, David Leuenberger, William Watts, and Ori Weinroth. IPTV - The Revolution is Here. http://www.eecs.berkeley.edu/~binetude/course/eng298a_2/IPTV.pdf (last visited: 2008-01-30).

[9] Benjamin Alfonsi. I Want My IPTV: Internet Protocol Television Predicted a Winner. *IEEE Distributed Systems Online*, 6(2), 2005.

[10] Verizon Communications. Verizon. http://www.verizon.com/ (last visited: 2008-03-04).

[11] Light Reading. Verizon's Elby: IPTV Could Take Years. http://www.lightreading.com/document.asp?doc_id=85708 (last visited: 2008-03-04).

[12] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A Survey of Peer-to-Peer Content Distribution Technologies. *ACMCS*, 36(4):335–371, December 2004.

[13] Justin Frankel and Tom Pepper. Gnutella. http://www.gnutella.com/ (last visited: 2008-01-26).

[14] Napster LLC. Napster Free. http://free.napster.com/ (last visited: 2008-01-28).

[15] Bram Cohen. BitTorrent. http://www.bittorrent.org (last visited: 2008-01-28).

[16] Sharman Networks. Kazaa. http://www.kazaa.com (last visited: 2008-01-26).

[17] An Gong, Gui-Guang Ding, Qiong-Hai Dai, and Chuang LinDOI. BulkTree: An overlay network architecture for live media streaming. *Journal of Zhejiang University - Science A*, 7(0):125–130, 2006.

[18] Hao Luan, Kin-Wah Kwong, Zhe Huang, and Danny H. K. Tsang. Peer-to-Peer Live Streaming towards Best Video Quality. In *CCNC*, pages 458–463. IEEE, January 2008.

[19] De-Kai Liu and Ren-Hung Hwang. P2broadcast: a hierarchical clustering live video streaming system for P2P networks. *Int. J. Communication Systems*, 19(6):619–637, 2006.

[20] Duc A. Tran, Kien A. Hua, and Tai T. Do. A peer-to-peer architecture for media streaming. *IEEE Journal on Selected Areas in Communications*, 22(1):121–133, 2004.

[21] Xing Jin, Kan-Leung Cheng, and S.-H. Gary Chan. SIM: Scalable Island Multicast for Peer-to-Peer Media Streaming. In *ICME*, pages 913–916. IEEE, 2006.

[22] Yang-Hua Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast. In *SIGMETRICS*, pages 1–12, 2000.

[23] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James O'Toole Jr. Overcast: Reliable Multicasting with an Overlay Network. In *OSDI*, pages 197–212, 2000.

[24] M. Handley, S. Floyd, J. Padhye, and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification, January 2003. RFC 3448.

[25] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *SIGCOMM*, pages 205–217. ACM, 2002.

[26] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: High-Bandwidth Content Distribution in Cooperative Environments. In *International Workshop on Peer-to-Peer Systems (IPTPS), LNCS*, volume 2, 2003.

[27] Antony Rowstron and Peter Druschel. Pastry: scalable, decentraized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, November 2001.

[28] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 20(8):1489–1499, 2002.

[29] Xiaofei Liao, Hai Jin, Yunhao Liu, Lionel M. Ni, and Dafu Deng. AnySee: Peer-to-Peer Live Streaming. In *INFOCOM*. IEEE, 2006.

[30] Yunhao Liu, Li Xiao, Xiaomei Liu, L.M. Ni, and Xiaodong Zhang. Location Awareness in Unstructured Peer-to-Peer Systems. *IEEETPDS: IEEE Transactions on Parallel and Distributed Systems*, 16, 2005.

[31] Xinyan Zhang, Jiangchuan Liu, Bo Li, and Tak-Shing Peter Yum. CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming. In *INFOCOM*, pages 2102–2111. IEEE, 2005.

[32] Cheng Huang, Philip A. Chou, Jin Li, and Cha Zhang. Adaptive peer-to-peer streaming with MutualCast. *Journal of Zhejiang University - Science A*, V7(5):737–748, 2006.

[33] Yang-Hua Chu, Aditya Ganjam, T. S. Eugene Ng, Sanjay G. Rao, Kunwadee Sripanidkulchai, Jibin Zhan, and Hui Zhang. Early Experience with an Internet Broadcast System Based on Overlay Multicast. In *USENIX Annual Technical Conference, General Track*, pages 155–170. USENIX, 2004.

[34] Meng Zhang, Yun Tang, Li Zhao, Jian-Guang Luo, and Shi-Qiang Yang. Gridmedia: A Multi-Sender Based Peer-to-Peer Multicast System for Video Streaming. In *ICME*, pages 614–617. IEEE, 2005.

[35] Vinay S. Pai, Kapil Kumar, Karthik Tamilmani, Vinay Sambamurthy, and Alexander E. Mohr. Chainsaw: Eliminating Trees from Overlay Multicast. In Miguel Castro and Robbert van Renesse, editors, *IPTPS*, volume 3640 of *Lecture Notes in Computer Science*, pages 127–140. Springer, 2005.

[36] Jin Li. PeerStreaming: A Practical Receiver-Driven Peer-to-Peer Media Streaming System. Technical Report MSR-TR-2004-101, Microsoft Research (MSR), September 2004.

[37] Sopcast Team. Sopcast. http://www.sopcast.com (last visited: 2008-01-28).

[38] Alexandro Sentinelli, Gustavo Marfia, Mario Gerla, Leonard Kleinrock, and Saurabh Tewari. Will IPTV ride the peer-to-peer stream? *Communications Magazine, IEEE*, 45(6):86–92, 2007.

[39] Anket Mathur Shahzad Ali and Hui Zhang. Measurement of Commercial Peer-To-Peer Live Video Streaming. In *Workshop in Recent Advances in Peer-to-Peer Streaming*. ICST, 2006.

[40] Thomas Silverston and Olivier Fourmaux. P2P IPTV Measurement: A Comparison Study. *CoRR*, abs/cs/0610133, 2006.

[41] HuaZhong University of Science and Technology. PPLive. http://www.pplive.com/ (last visited: 2008-01-30).

[42] L. Vu, I. Gupta, J. Liang, and K. Nahrstedt. Mapping the PPLive network: Studying the impacts of media streaming on P2P overlays. Technical Report UIUCDCS-R-2006-275, Department of Computer Science, University of Illinois at Urbana-Champaign, August 2006.

[43] X. Hei, C. Liang, J. Liang, Y. Liu, and K. W. Ross. Insights into PPLive: A measurement study of a large-scale P2P IPTV system. In *IPTV Workshop*. International World Wide Web Conference, 2006.

[44] Yensy James Hall, Patrick Piemonte, and Matt Weyant. Joost: A Measurement Study.

[45] Stephen Alstrup and Theis Rauhe. Introducing Octoshape - a new technology for large-scale streaming over the Internet. Technical Report TREV-303, European Broadcasting Union (EBU), July 2005.

[46] Ingjerd Straand Jevnaker. RawFlow - using P2P to create virtual 'multicasts'. Technical Report TREV-308, European Broadcasting Union (EBU), October 2006.

[47] Djamal-Eddine Meddour, Mubashar Mushtaq, and Toufik Ahmed. Open Issues in P2P Multimedia Streaming. In *IEEE ICC 2006 Workshop on Multimedia Communications Workshop (MultiCom)*, June 2006.

[48] Jiangchuan Liu, Sanjay G. Rao, Bo Li, and Hui Zhang. Opportunities and Challenges of Peer-to-Peer Internet Video Broadcast. To appear in the Proceedings of the IEEE.

[49] Mea Wang and Baochun Li. R2: Random Push with Random Network Coding in Live Peer-to-Peer Streaming. *IEEE Journal on Selected Areas in Communications*, 25(9):1655–1666, December 2007.

[50] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace, July 2005. RFC 4122.

[51] Free Software Foundation. GNU Compiler Collection. http://gcc.gnu.org/ (last visited: 2008-05-07).

[52] Linux Online Inc. Linux Online! http://www.linux.org/ (last visited: 2008-05-07).

[53] CollabNet Inc. Subversion. http://subversion.tigris.org/ (last visited: 2008-05-07).

[54] Iproute2. Iproute2. http://www.linuxfoundation.org/en/Net:Iproute2 (last visited: 2008-06-02).

[55] Netem. Network Emulation. http://www.linuxfoundation.org/en/Net:Netem (last visited: 2008-06-02).

[56] PHP. Hypertext Preprocessor. http://www.php.net/ (last visited: 2008-06-08).

[57] Graphviz. Graph Visualization Software. http://www.graphviz.org/ (last visited: 2008-06-08).

COS/CCS 2008-11

www.kth.se