Eindhoven University of Technology

MASTER

Analysing and improving Iphion collaborative IPTV

Poelstra, M.L.

Award date:
2008

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Eindhoven University of Technology
Department of Electrical Engineering
Signal Processing Systems

TU/e technische universiteit eindhoven

Analysing and improving Iphion Collaborative IPTV

Confidential until September 2009

by M.L. Poelstra

Master's Project Report
Project period: July 2007 – June 2008
Report number: 2008.012

Commissioned by:
Prof.dr.ir. P.H.N. de With
J.P.P.A. Saman
Ir. W.J. Withagen

Supervisors:
Prof.dr.ir. P.H.N. de With (TU/e)
J.P.P.A. Saman (Iphion B.V.)

The Department of Electrical Engineering of the Eindhoven University of Technology accepts no responsibility for the contents of M.Sc. theses or practical training reports

Contents

1 Introduction **1**
- 1.1 Master's project introduction . . . 1
- 1.2 Structure of this report . . . 1
- 1.3 System requirements . . . 2
- 1.4 Why Peer-to-Peer for IPTV? . . . 3
- 1.5 Iphion collaborative IPTV . . . 5
- 1.6 Problem statement . . . 8

2 Analysis **9**
- 2.1 Design considerations . . . 9
- 2.2 Existing proposals . . . 18
- 2.3 Research summary and conceptual application analysis . . . 18
- 2.4 Iphion implementation-specific analysis . . . 19
- 2.5 Typical cable modem behaviour . . . 20
- 2.6 Simulator . . . 21
- 2.7 Iphion application on cable modem links . . . 26

3 Generic improvements **29**
- 3.1 Streamer . . . 29
- 3.2 Architecture . . . 32
- 3.3 Locality . . . 35
- 3.4 Packet scheduling . . . 37

4 Congestion control improvements **40**
- 4.1 Requirements . . . 40
- 4.2 Algorithm choice . . . 41
- 4.3 Introduction to TFRC . . . 42
- 4.4 TFRC performance . . . 43
- 4.5 Improving TFRC . . . 47
- 4.6 The bandwidth-delay product . . . 54

5 Conclusions and recommendations **58**
- 5.1 Conclusions . . . 58
- 5.2 Recommendations . . . 60

A Existing P2P algorithms **62**
- A.1 Single tree . . . 62
- A.2 Multiple tree . . . 63
- A.3 Hybrid . . . 65
- A.4 Unstructured . . . 67
- A.5 Structured . . . 68

References **71**

1 Introduction

1.1 Master's project introduction

The Internet is rapidly evolving and with increasing bandwidth, new applications come within reach. One very promising possibility (with an estimated marketplace of several billions of Euros in upcoming years) is to watch television over the Internet. Appealing as this seems, several technical hurdles have to be taken before consumers can actually enjoy a high-quality Internet Protocol Television (IPTV) experience. The huge amount of data (bandwidth) involved in streaming live television to thousands of users simultaneously, combined with the very stringent deadlines on the delivery of this data, make IPTV an interesting technical challenge.

Most current implementations of streaming video over the Internet make use of separate (IP unicast) sessions to each and every connected user. This requires an enormous amount of resources on the broadcast server(s). One way to alleviate this is to use IP multicast, but unfortunately, this is not practical because this protocol is not supported by most Internet routers, and because of the so-called 'feedback implosion' on the server. As client computers have an increasing amount of uplink bandwidth, it makes sense to use this bandwidth to help distribute the data. This approach, called Peer-to-Peer (P2P) networking, has become very popular for filesharing nowadays.

Using P2P technology for live IPTV is a non-trivial matter because of the rather dynamic nature of the network: nodes join and leave all the time, sometimes leaving even without notice (e.g. in case of a computer crash or network outage). Additionally, the highly asymmetric up- and downlinks of the clients make finding an optimal distribution of bandwidth complex. Moreover, consumers have very high expectations of and demands on visual quality, continuous delivery and zapping speed. These aspects call for specialised algorithms to make IPTV a reality.

The startup company Iphion aims at bringing easy-to-use IPTV to the homes, ultimately delivering IPTV as users expect it to be: just as easy to use as 'conventional' television currently is, but with an increased visual quality and the added benefits of Internet technology, such as interactive program guides. The most important advantages of IPTV are the potential low costs for the user, flexible channel access schemes coupled to an excellent visual quality, and efficient bandwidth usage for broadcasters and ISPs.

1.2 Structure of this report

This first chapter introduces the project, its requirements, objectives and problem statement, and briefly describes the state of the Iphion application when this project was started, and at its first (small-scale) release.

The next chapter starts with the results of research into existing P2P algorithms, to continue with a first analysis of Iphion's application Version 1.0. Then, the potentially large queue size of cable/xDSL links is examined, leading to an additional analysis of the application performance on typical network links of Iphion customers. These results were obtained using a custom-made simulator, which is also described in the chapter.

Chapter 3 then describes improvements made to the application itself and/or implemented in the simulator, based on the analyses of Chapter 2.

Because of the impact of large queue sizes on congestion control, and the importance of congestion control for the Iphion application, a separate chapter (Chapter 4) is dedicated to further investigations and experiments on the initial choice for TCP-Friendly Rate Control (TFRC). The chapter shows that although TFRC is generally considered a good congestion-control algorithm for multimedia streaming,

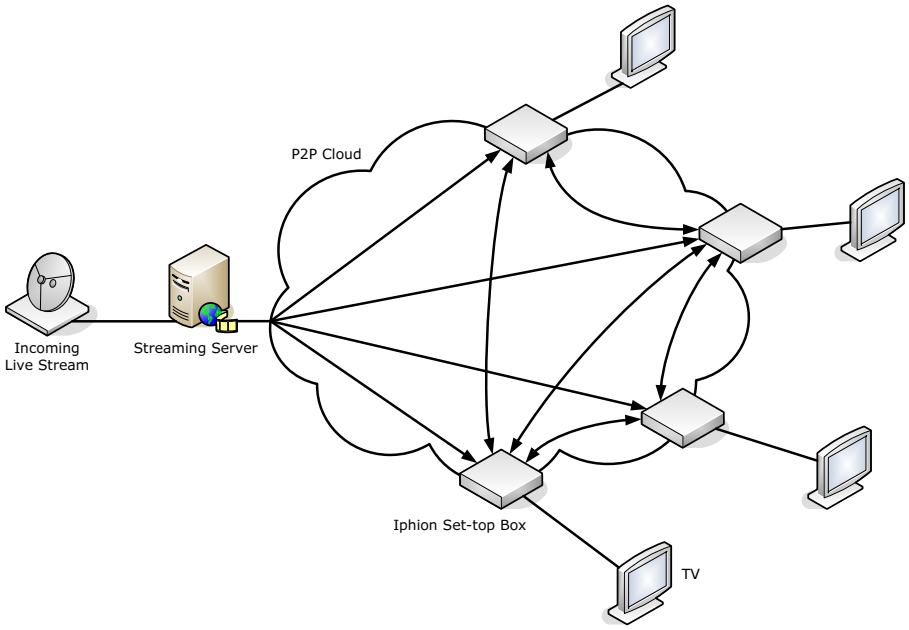


Figure 1.1: General overview of the Iphion network. Note that the streaming server will likely be located at the broadcasting company itself (without the need to pick the signal from a satellite).

this no longer holds for typical cable/xDSL networks. The last section therefore proposes the rather surprising use of (a specific version of) TCP's congestion control.

The last chapter concludes this project, and contains recommendations resulting from previous chapters.

The appendix contains a summary of existing P2P algorithms, ranging from the very first to very recent designs, and provides a better understanding of typical pitfalls and considerations for P2P protocol design.

1.3 System requirements

The following requirements on the application are defined:

- **Reliable transmission**: when nodes fail, other nodes must still maintain a steady playback rate.
- **Efficiency**: make optimal use of the limited upstream bandwidth of users, and minimise inter-ISP (so-called "egress") bandwidth.
- **Scalability**: the algorithm must be scalable to thousands of users.
- **Short latency**: the time between broadcasting a program (e.g. a soccer match) and the time of reception should be as short as possible.
- **Fast network joins**: zapping must be as fast as possible (ideally sub-second).
- **Security**: nodes should be verified and message integrity should be guaranteed.
- **Connectivity**: measures must be taken to ensure good connectivity between all hosts, even behind NATs or firewalls.

The Iphion application is targeted at streaming SD-quality television (for the first version(s) at least) and will be using the H.264 codec. This level of quality can be achieved with a bitrate of around 1 Mbit/s

(approximately 100 kB/s). Additionally, the application is targeted at low-end set-top boxes, which means CPU and memory requirements of the application must be kept at a minimum. Most research efforts do optimise for these requirements in terms of scalability to large P2P networks, but assume an implementation of their protocol on a 'normal' desktop computer, which sometimes (especially in case it utilises network coding or video recoding) renders these approaches unusable for the Iphion application.

Because the aggregate upload bandwidth of peers will not suffice to support the network by itself, too much inter-ISP bandwidth is not desirable, and because of commercial considerations, dedicated relay servers will be placed at strategic points in the network. These relay servers can (and should) be used to quickly bootstrap new peers, top-off the 'missing' bandwidth, and aid in building and maintaining the network topology. Additionally, there will be servers to provide additional services like authentication, peer verification, etc.

1.4 Why Peer-to-Peer for IPTV?

The previous sections touched on some of the difficulties of using P2P (peer-to-peer) technology. Why then go through all the trouble to use this technology, when live streaming over the Internet is already possible?

This section quickly introduces the general methods of streaming (TV) data over the Internet, and explains that P2P is the practical answer to the unavailability of efficient lower-level infrastucture.

1.4.1 IPTV vs. Internet TV

First of all, there seems to be some confusion caused by the different definitions of the term IPTV (which, in its broadest sense just means "TV over IP"), so it is useful to create clarity first.

The abbreviation **IPTV** is mostly used to denote the distribution of TV data in a controlled network environment, meaning a network that is owned by the telecommunication company (TelCo) that also owns the "last mile". One can think of the Video-on-Demand (VoD) services of a cable company. 'Normal' IP unicast is used for VoD, as there is no significant overlap between users' VoD sessions. For live streaming, IP multicast is mostly used in this case, which is possible because all involved routers are owned by the TelCo, and as such can be upgraded/configured to support multicast.

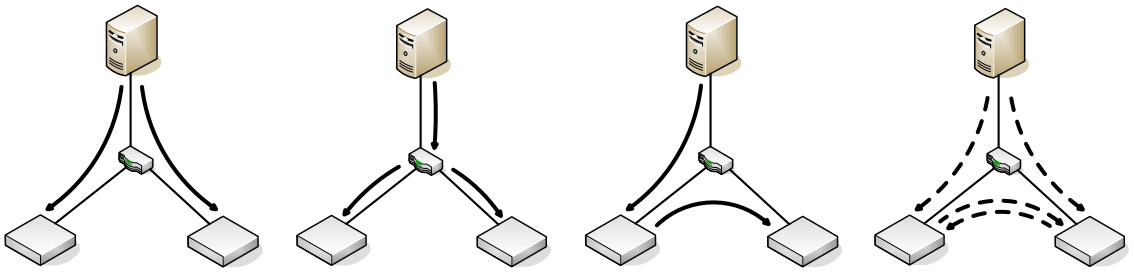
By using a controlled network, tight guarantees on quality-of-service (QoS) can be made, e.g. by prioritising IPTV and VoIP packets using the Differentiated Services (DS) field of IP headers [57] (this field obsoletes the Type of Service (TOS) field). Because of the possibility of using multicast on this type of network, P2P technology does not have to be used.

Internet TV on the other hand is used to denote streaming of TV-like material over the Open Internet. This means that IP multicast cannot be used, as most Internet routers do not implement this technology. This is due to both technical, but also political matters [26]. As unicast connections are used instead, the available bandwidth is limited. The use of uncontrollable networks makes tight guarantees on QoS hard, and videos will usually have QCIF to CIF resolution (176×144 to 352×288) with low bitrates. Examples in this area are YouTube, but also uitzendinggemist.nl.

P2P technology can be used to make this approach scalable again, even with high-quality content. However, as both the terms P2P and Internet TV are often associated with inferior quality, illegality (due to filesharing), etc., Iphion will use the commercially more attractive terms Collaborative or Peer-assisted IPTV for its more flexible and reliable technology.

1.4.2 Internet video streaming approaches

The Internet was designed to support different means of transporting data. One of the most well-known protocols is the unicast TCP/IP protocol, but provisions for efficiently sending data from one source to many destination have been made. Unfortunately, the latter suffers from several issues, to which several solutions are possible.



(a) IP unicast (b) IP multicast (c) Early application layer multicast (d) Modern application layer multicast (P2P)

Figure 1.2: Schematic representation of the 3 different ways of distributing data to end nodes. The dashed arrows indicate disjoint subsets of the full stream.

1.4.2.1 IP unicast

IP unicast is used for sending data from one source to one receiver (one-to-one communication). Probably the most well-known usage of this type of communication is that between a webbrowser and -server. Note that in this case one server can communicate to many clients, but the data to each client is sent independently (Figure 1.2(a)).

This immediately shows the main reason why this technology does not scale well for streaming TV: bandwidth usage (especially at the server) increases linearly with the number of connected users.

1.4.2.2 IP multicast

IP multicast [24] [25] was devised to alleviate this problem, and can be used for one-to-many and many-to-many communication. Every user interested in the data stream can connect to the multicast group of a stream using the IGMP protocol [25] and supporting routers along the network path will automatically take care of 'splitting' data to different parts of the network. As data for every stream is sent just once over a link, irrespective of the number of connected users, this approach seems the ultimate solution for video streaming (Figure 1.2(b)).

However, on the Open Internet, many (technical and political) problems prevent the widespread implementation of multicast on backbone routers. To name a few: "feedback implosion" due to nodes re-requesting lost packets, per-group state in routers, scalability issues with address allocation, cost sharing between ISPs, etc. For an overview of these problems, see [26].

Some of the problems can partly be solved. For instance, to solve occasional packet loss without requiring a feedback channel one can use Forward Error Correction (see Section 2.1.13). In the case of a controlled network environment (such as within the network of one ISP), some problems are not an issue anymore, so despite the mentioned potential technical problems, IP multicast is widely used for streaming TV in such networks. Additionally, because of the direct control over routers in the network, strict policies can be made to guarantee high QoS for multicast data (e.g. in combination with QoS flags in packets [57], bandwidth reservation, etc.), thereby making the technology both highly efficient and robust.

1.4.2.3 Application layer multicast

To overcome the lack of IP multicast on the Open Internet, a different strategy has to be taken: it is possible to implement multicast on a higher network layer (e.g. TCP or UDP), by forming an overlay network on top of a unicast network. This technique is called Application Layer Multicast (ALM) and can be implemented using two main principles: dedicated proxy servers and peer-to-peer technology, the latter of which is discussed in the next subsection.

The best known example of dedicated proxies is the MBone [28], devised by the Internet Engineering Task Force (IETF) in 1992. It creates an overlay network on top of standard unicast connections, using

dedicated servers in the network to tunnel 'regular' multicast traffic on the local network to an MBone server on another multicast-capable network. Standard IGMP is still used to connect to multicast groups on the MBone, which makes it transparent to IP multicast applications.

Note that ALM will always be less efficient than IP multicast, for example in terms of bandwidth usage: many (physical) links will carry data at least more than once. This of course also adds to the delay. However, it can (and should) be more efficient than IP unicast, as data is (hopefully) copied closer to the endpoints, offloading the link of the central server (Figure 1.2(c)).

1.4.3 P2P multicast overlays

As the MBone still requires dedicated proxy servers and multicast enabled networks, the logical next move is to take the overlay network one step further: to the end hosts themselves. This is called Peer-to-Peer (P2P) communication, and the (multicast) overlay is now formed directly by IP-unicast connections between the applications at the end hosts, called peers (Figure 1.2(d)). Although this concept is not hard to grasp, creating a reliable multicast transmission using P2P technology is far from trivial, considering the unreliable nature and strong heterogenity of the end systems (joining and leaving at will, sometimes even failing without notice, having upload and download bandwidths differing by multiple orders of magnitude, etc.).

One of the first and most widely known Peer-to-Peer multicast systems is Narada, which was published first in 2000 [20]. This early system directly resembled IP multicast, using a single tree to distribute the data. Although such a technology works well with dedicated servers, more robust approaches should be used in the P2P case. Since Narada, a vast amount of research has been (and still is) invested in improving many aspects of P2P networks.

Section 2.1 and Appendix A discuss the main considerations and proposals resulting from a literature study in this field, and provide the necessary knowledge to further discuss the Iphion application.

1.5 Iphion collaborative IPTV

For this project, the most relevant part of Iphion's IPTV solution is of course its P2P network. This network will be formed by several servers, and the client nodes called Iphion players. Figure 1.3 shows how the Iphion players communicate with each other and relay servers. The solid lines in the figure indicate the main flows of communication (being stream data, partnership negotiations, etc.). It can be seen that the content provider delivers its data to the streaming server (which could be located at or near that provider, or elsewhere), which in turn distributes the stream to the relay servers. The relay servers each manage a (possible overlapping) part of the network, and will likely be located inside the networks of ISPs. Additionally, there will be servers offering additional services like cryptographic key management, a rendez-vous point to assign new peers to their 'local' relay servers, providing EPG data, etc.

However, as the figure's caption suggests, this figure does not represent Iphion's current network. Although much work has already been done, major concepts such as using relay servers have not been implemented yet, and on a smaller scale important issues in the P2P algorithm (such as problems with the current packet scheduling under moderately high round-trip times) would prevent large-scale deployment of the application (discussed in Sections 2.4 and 2.7).

The following subsections briefly describe the state of the initial and 1.0-version of the application and the already substantial improvements so far.

1.5.1 Original application

The following points shortly illustrate the status of the Iphion application as it was at the dawn of this Master's Project, before a release of the software was made (internally called Version 0.1):

- The gossip approach of CoolStreaming/DONet (see Appendix A.4) was used to build the P2P overlay structure: this method is relatively easy to implement, and is robust to node churn,

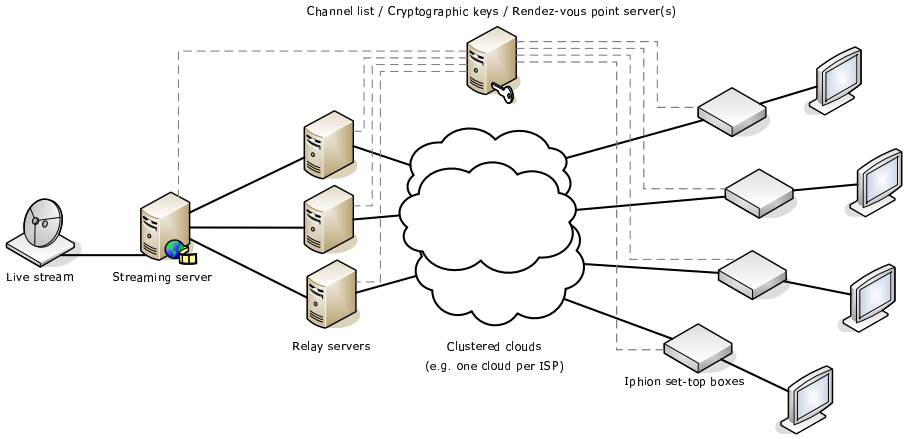


Figure 1.3: Schematic representation of future Iphion network

without depending on central servers for maintaining the structure of the network (apart from the streaming server itself of course). The main disadvantage appeared to be large end-to-end delays between the streaming server and peers, as nodes did not attempt to find peers with 'better' services (e.g. shorter end-to-end delay, higher bandwidth and/or smaller network distance). Additionally, the bandwidth overhead of sending gossip messages was found to be too big.

- Because of having multiple parents, data is pulled (as opposed to pushed) from neighbours: peers therefore communicate buffermaps and requestmaps with each other.
- There was initial support for NAT detection and traversal, including the concept of "supernodes" to communicate with peers behind a NAT box. The streaming server was always used as the supernode.
- No protection against intrusion of the network (the so-called "crypto code") had been created at the time.
- No congestion control was implemented and in fact, all data requested by another peer was sent in one large (UDP) burst. Although this did work over a local 100 Mbit LAN, it failed when a 10 Mbit hub was connected between testing computers (even with a stream bandwidth of less than 1 Mbit).
- The buffer had a fixed size of 300 packets, and was advanced (moved forward) by looking at the head pointers of other peers. Data 'falling off' the tail of the buffer was then transferred to the media player. Therefore, advancements had to be rather smooth, which does not match the not-completely-constant bitrate of the videostreams, so that an additional buffer in the media player of at least 2 seconds was needed. Additionally, 300 packets appeared to be too few for proper delivery of high bitrate ($\geq$1 Mbit) streams.
- The stream was produced from a file, instead of from a live input.

1.5.2 Current application

Many improvements have already been made during this project by the developers of Iphion itself. The changes most relevant in the context of this project are summarised in the following list, which reflects the state of the software at its initial (1.0) release.

- Although the pull-based data exchange remains, the gossiping algorithm was removed in favour of a server-based rank assignment: every new peer is placed in the highest possible rank, based

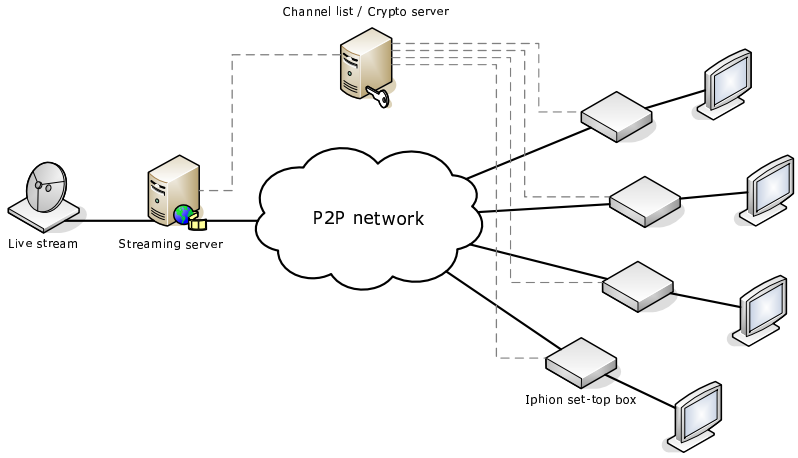


Figure 1.4: Iphion network structure at release 1.0.

on the amount of available bandwidth in every rank, which is estimated by the uplink speeds advertised by peers, and the current amount of peers in the rank. Every peer then tries to receive data from higher-rank peers (the streaming server has the highest rank, and relay servers do not exist yet).

- Requested data is no longer sent in one burst, but sent in an interval of 100 ms. Because requestmaps are sent every 150 ms, there is about 50 ms of 'silence' between the data sends. To overcome the problem of excessive packet loss when using the application over slower links, some means to limit the requested amount of data was implemented. However, when a peer experiences data shortage (most notably during its initial pre-buffering), it effectively ignores this rate control and sends out large requestmaps. This behaviour is called the "mean-to-network mode" and combined with the implemented packet scheduling renders P2P communication inoperable on cable modem links (see Section 2.7).
- A cryptographic-signature system was added to prevent non-authorised clients to intrude the network. Note that this system does not encrypt the contents of the packets, but protects the network by only accepting messages originating from nodes that obtained a valid 'login key' from the server. The media stream itself is not yet protected.
- The size of the buffer can now be assigned upon joining a media channel, and is chosen to match stream characteristics.
- Sending data to the media player is now done by a separate pointer, instead of the tail pointer of the buffer. This new playback pointer allows to follow the instantaneous rate of the videostream, thereby considerably reducing the buffering needed in the media player (currently 400 ms, instead of more than 2 seconds). This principle (and its additional improvements) is described in Section 3.1.
- Initial experiments with live streaming were performed (video-encoder settings were a major concern here).

The 1.0-version software was deployed to a small group of beta testers, including some with cable/ADSL links. It already works quite well, but due to the small amount of peers there is not much peer-to-peer traffic yet. Although the concept of relay servers has been planned since the earliest versions, the current network is still directly supported by the streaming server itself and partnerships are, apart from the rank assignment, still fully random (i.e. there is no locality awareness).

As will be shown in Section 2.3, the concepts of the application and protocol are good, but do need improvements. Development after Version 1.0 then started to include recommendations from this project such as the new architecture (Section 3.2), improved packet scheduling (Section 3.4), etc.

1.6 Problem statement

The objective of this Master's Project is to analyse and improve upon the P2P protocol and algorithms of the existing initial Iphion application. The following sub-problems are defined:

- **Literature review** of state-of-the-art (streaming) P2P algorithms is needed to obtain a thorough understanding of fundamental choices for creating a fast, reliable and efficient network.
- **Analysis of Iphion's existing application**
  - based on **design considerations** resulting from the literature review allows to judge on general concepts,
  - and analysis of its **specific implementation** will highlight major inefficiencies.
- **Simulation** is needed to investigate original and future network behaviour, with a strong focus on realistic simulation of typical residential access networks (most notably the occurrence of extremely large queuing delays).
- **Improvements** must be designed, implemented and verified, most notably removal of the large media decoder buffer, a redesign of the internal application architecture, a fast and efficient method of improving on network locality and 'large-latency-resistant' packet scheduling.
- **Congestion control** must be improved as well,
  - however the popular choice for **TFRC** will appear to cause **excessive delays** on typical residential networks
  - to which an **improved version** must be proposed,
  - leading to a **fundamental re-evaluation** of the control principle in favour of FreeBSD's TCP algorithm.

2 Analysis

In order to design an improved peer-to-peer (P2P) algorithm, many aspects have to be taken into account. This chapter (and Appendix A) first introduces the most important issues to consider when creating a P2P protocol, resulting from literature study of (streaming) P2P protocols.

Then, an analysis of the current Iphion application is made, based on these considerations. It will be clear that a complete redesign of the existing protocol is not necessary, but significant improvements are needed.

Because the application will often be used on cable/ADSL links and the queues on those links are rumoured to be large, this has been investigated in Section 2.5. The consequences of this simple investigation later appeared larger than expected: the packet-requesting algorithm, and the lack of congestion control in the current Iphion application render the application highly inefficient on such links. This efficiency is described in Section 2.7, based on results from a custom-made simulator which is introduced in Section 2.6.

2.1 Design considerations

P2P technology can be used for a wide variety of tasks, e.g. filesharing, video conferencing, redundant and distributed archiving, messaging, etc. This section describes the subset of P2P problems that will be encountered when streaming live TV over the Internet (although other P2P areas have many of these in common). This includes important choices for overlay topology structure, and system aspects like packet scheduling and buffering.

2.1.1 Central vs. Distributed

As P2P networks can become very large (100k+ nodes), global knowledge of the network should not be maintained by every peer. Three main approaches to overcome this can be discerned:

- **Central server(s)**: Some implementations use one or more central servers to store the global state. A centralised solution provides for efficient algorithms to place new peers in the network and makes maintaining tree structures trivial in most cases. A disadvantage is that it has limited scalability, usually up to a few hundreds or thousands of nodes.
- **Super nodes**: Other implementations use "super nodes" to control a part of the network. These nodes can be normal peers that happen to have good connectivity, stability, etc. but can also be dedicated nodes inserted in the network by the application network maintainers if needed. They generally maintain a medium-sized list (e.g. 1000 peers) of peers. Normal peers are assigned one super node, and super nodes can communicate with each other.
- **Distributed**: Then there are fully distributed algorithms, which are usually created to have good scalability, but are harder to design, have more control overhead (messages) and are slower (for joining, repairing and optimising) than a centralised solution.

Generally, all peers maintain a list ("peer list") with a subset of all nodes in the system (typically 20-100). This list is usually a uniform partial view of the network (i.e. random-ish). Many algorithms then create partnerships for the actual exchange of data with even a subset of the peer list, called the "partner list".

The Iphion application will be a hybrid between centralised and distributed: a node will first contact a central server to obtain a list of relay servers. The relay servers are dedicated nodes placed at

strategic locations (e.g. inside ISP networks) and maintain a localised view of the network, aiding in more efficient bandwidth usage inside ISPs and faster channel-switching in clients, see also Section 3.3.

2.1.2 Trees vs. Meshes

The partners can be chosen in many ways, for example at random, or to form a tree. In fact, this choice has a great impact on the rest of the application: in case the system creates a tree-like topology (i.e. each peer has one 'parent') data can be pushed down the tree without the need for explicit determination of disjoint data sets. If nodes can have multiple parents (as in the case of a random mesh, or even structured graphs) some means to prevent duplicate packets from being sent must be implemented.

The first application-layer multicast algorithms use a single tree to distribute the data (see Appendix A.1). This tree is then used for delivering both control messages and data. A single tree has the advantages that it is fairly simple to construct (especially in the case of centralised assignment) and data can simply be forwarded to all children of a node without the need for set-reconciliation algorithms (see Section 2.1.4). However, it has a number of distinct disadvantages:

- Failure of one single node immediately leads to the starvation of the complete subtree below that node, so quick repair is necessary.
- Every node in the tree has to be able to supply the full videostream to each of its children. This is impossible in the realistic scenario of highly asymmetric links, like most cable and xDSL users have.
- In contrast to the internal nodes, leafs of the tree cannot contribute any data back to other nodes. Even with binary trees, already half of the nodes is a leaf node.

The next generation of protocols therefore started to use multiple trees. The original videostream is then split into disjoint sets, which are sent down their own tree. Some protocols simply splice the data, other proposals use Multiple Description Coding (MDC) (SplitStream [12], CoopNet [59]) or e.g. the Information Dispersal Algorithm (IDA) (Trickle [34]) to split it. (See Appendix A.2.)

When data is simply spliced and a tree breaks, the tree still has to be repaired very quickly, but less missing data has to be recovered. IDA allows to fully recover the stream, even when a (configurable) number of slices is missing. In case of MDC, when not all slices are received, the quality of the video gracefully degrades (e.g. appearing a bit more blurry) without suffering from playback discontinuities. MDC requires special video encoders and decoders, which are not readily available in (cheap) hardware decoding chips. It is also possible to create a tree for distribution of control messages, sometimes even a part of the videostream, and distribute the bulk of the data through 'cross links' between nodes in the tree (e.g. Bullet [44]).

To overcome the problem of unreliable nodes in trees, it is possible to replace one single node by a cluster of nodes (BulkTree [32]), or to only 'promote' nodes to interior nodes after they have 'proven' to be reliable enough (mTreebone [77]), see Appendix A.3.

A different approach is to not enforce any structure on the nodes, and create random meshes using e.g. gossipping algorithms to distribute the data. Although this is very simple to implement, it has the huge disadvantage of wasting large amounts of bandwidth on duplicate packets. A direct improvement to this approach is therefore to use gossipping for just maintaining the network overview, and building partnerships with a subset of this view (CoolStreaming/DONet [85], Appendix A.4).

Random meshes allow every node to supply as much bandwidth as they can and are highly robust to node failures, but they have the disadvantage of introducing larger delays, and are very inefficient in terms of localising data exchanges. Some algorithms therefore create some structure in the mesh to improve on delays and locality, while still guaranteeing sufficient connectivity in the mesh (Dagster [58], DagStream [48], Cyclon/Vicinity [75], PULSE [61], Appendix A.5). Others use combinations of trees or even normal multicast and meshes (HON [86], HyMoNet [14]).

No real consensus has yet been reached as to which approach (trees or mesh) is better with respect to streaming video, as both have fundamental trade-offs between delay and robustness (in case of data

push, because of the buffer needed to cope with tree failures) or delay and control overhead (in case of data push, because of e.g. buffermaps). However, it seems that (structured) meshes are receiving more research attention lately, often using advanced network coding techniques, effectively enabling data push even on multi-parent structures.

The Iphion protocol currently uses a random mesh, with many concepts (such as buffermaps) based on CoolStreaming/DONet (see Appendix A.4), but will use a more localised mesh in the future (Section 3.3).

2.1.3 Buffering

Most (also non-P2P) video players use quite substantial buffering when playing content over the open Internet. Buffers are mainly used to keep a continuous flow of videodata to the decoder, to cope with temporary network slowness and variances in the bitrate. Additionally, they can be used to allow packets to arrive out-of-order, most notably for the data pull strategies, and/or because of packet re-ordering (when e.g. UDP is used as the transport mechanism).

Especially determining a suitable initial buffer size is a delicate matter: a large (initial) buffer causes a longer startup delay, and potential loss of resources when a user zaps to a different channel. Additionally, a larger load will be put on existing peers and relay servers while a new peer tries to fill its buffer as quickly as possible. A too small buffer on the other hand can cause an immediate buffer underrun when starting playback, as videostreams are usually not of constant bitrate (see also Figure 3.1 in Section 3.1), and the buffer must be able to sustain these 'surges'.

Additionally, the choice of the initial playback position is crucial, as it can not be changed during playback, but greatly determines the amount of packet trading opportunities with other peers.

2.1.4 Packet scheduling

When a data pull strategy is used, nodes need to determine which packets to send to each other. The usual approach is that a node advertises which packets it already obtained to other nodes, which can then decide which packet to request from which node. This advertising and requesting is performed using buffermaps or sometimes Bloom filters (see Bullet in Appendix A.3).

Buffermaps typically contain some sequence numbers (e.g. the current position of the head and tail of the buffer, and sometimes the playback position of the media player) and a set of bits. If a bit is set, this means the corresponding data packet is available in the buffer (or requested).

Given a set of buffermaps of other peers, a node must determine what packets to ask from each peer. One strategy is the random one. This potentially leads to problems as some packets might become very scarce, and thus risk missing their playback deadline (see Chainsaw in Appendix A.4). According to CoolStreaming [85] and BitTorrent [22], a "rarest-first" scheduling is important to guarantee an even packet distribution in the network to prevent this scarcity. However, this does not take the playback deadline into account, so some packets might still be scheduled too late.

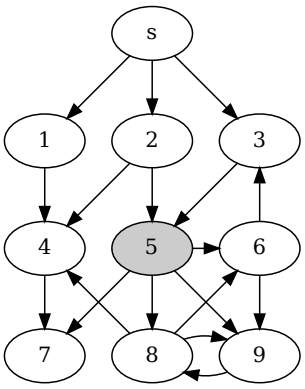
A simple solution seems to be to just prioritise the packets near the playback pointer, but this makes it hard for nodes to obtain 'innovative' packets that can be used to send to other peers (e.g. to improve their 'cooperation score'). This innovation is needed for incentives (see Section 2.1.9) and is supposed to be responsible for the success of BitTorrent (however, also consider [41], that shows some flaws in the reasoning of the original BitTorrent protocol).

The BiToS algorithm [74] contains a nice approach to this problem by modifying the BitTorrent scheduling to balance between prioritising near-deadline packets and innovation packets. An improvement to the CoolStreaming scheduling is given in [84].

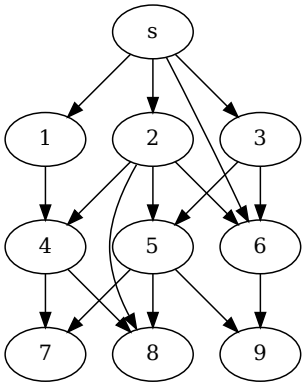
Implementation specific problems appear to cause even greater problems in the Iphion application, see Section 2.7.

2.1.5 Clustering

The expected usage of the application will be that many users will be watching the same (small) set of channels, and that they will most likely also be close together in terms of network distance. This



(a) If node 5 fails, nodes 6, 8 and 9 are isolated



(b) Directed Acyclic Graph (DAG) with every node (except first level) having at least 2 parents

Figure 2.1: Example of weakly connected cluster (left), and clustering with at least one backup path from every node (right).

means there could be a large gain by taking this locality into account. The benefits are obvious: by using nearby neighbours, (local) bandwidth can be higher, latencies shorter, packet losses and delay jitter smaller, inter-ISP ("egress") bandwidth demands will be lower, etc.

However, the disadvantage is not so obvious: great care must be taken to keep the mesh sufficiently connected, i.e. there must be enough distinct paths from any client to the source to prevent clusters in the network from becoming isolated if one node fails. This problem worsens as the number of partners is decreased, which would be beneficial in case the number of close neighbours is small.

This means that measures must be taken to guarantee good connectivity, for example by ensuring that multiple disjoint paths are available to the source. Good algorithms are available to achieve this, see for example DagStream (Appendix A.5), which generates a structure as illustrated in Figure 2.1.

2.1.6 Locality

Given the desire to connect to nearby peers, it should still be decided how to find nearby peers and many different approaches can be taken. Obvious means include using the round-trip times (RTTs) or measuring [78] (or inferring [16]) bandwidth between nodes. Additionally, nodes could look at the quality-of-service another node could provide [8].

More advanced approaches are measuring latencies to known landmark hosts to determine a position vector (e.g. GNP [55]) or using RTTs between peers to distributedly compute an x-dimensional (usually, $x = 2$, 3 or 4) position vector (e.g. Vivaldi [23]). Vivaldi also introduces the idea of a height value, additional to e.g. a 2D position, which allows easier modelling of the last mile latency that xDSL users experience, and which would be hard to include in e.g. a 'normal' 3D position.

A disadvantage of using latencies to estimate network distance, is that it involves actively probing other nodes, which increases overhead and takes time.

Another disadvantage of using RTTs is that many of the clients of Iphion will be behind cable and xDSL modems. These modems tend to have long queues, which can significantly increase RTTs, especially when links get saturated (see Section 2.5). This means that e.g. a cable modem user could easily mistake a distant university campus node for being closer than another cable modem user in the same area. Although having shorter RTTs might be advantageous from the user's point of view, it may lead to a large increase of egress bandwidth.

It is therefore desirable to use 'passive', non-RTT-based techniques to estimate distance.

One of these techniques is to use an IP-to-location database, but this usually requires the database to be stored on the nodes, and the coverage of these databases is often not sufficient (missing IP ranges) and subject to changes. Another solution is to make use of the fact that nearby hosts often

share routers along paths to the source. In [81] an algorithm is proposed that traces the route from a client to some landmark node(s), and saving that information in a central database. The central server can then determine nearby nodes by comparing the list of routers of a newly connecting node with those in the database.

It is evident from the previous observations, that knowledge about the local network would greatly help to take more accurate decisions. Proactive Provider Participation for P2P (P4P) [79] proposes that ISPs should set up servers in their networks that allow peers to learn about the network topology and status. For the Iphion application, another approach will be used, which is based on a combination of information about the provider networks, and a distance metric based on comparing IP addresses using the XOR metric. As P4P will likely not be available soon, Iphion's relay servers will be responsible for supplying the network information. See Section 3.3 for a simple alternative solution.

2.1.7 Node churn

Due to the nature of P2P systems peers can join and leave the network at will (as opposed to routers, which can be regarded as much more stable in this respect). This continuous process is called "node dynamics" or "node churn", and is one of the most prominent reasons why tree/push-based approaches are challenging to get right.

Nodes may leave the network in a controlled way, by having the user nicely shut down their set-top box, but they may also leave the network by a (temporary) failure of the (wireless) network connection, or simply crash. In the case of set-top boxes the most important cause of unannounced leaves will probably be network outages.

See also Section 2.1.16.

2.1.8 NAT / Firewalls

Because of the limited address space of IPv4, a concept called Network Address Translation (NAT) was introduced. This allows many computers to share the same public IP address, each of them having their own private IP address. The "NAT box" (router) will automatically translate the addresses of packets going from the client to the server, and back again. This approach works good for 'normal' tasks like web browsing, but proves to be a real challenge in P2P networks: it is no longer possible to directly reach a node behind a NAT box from any other node on the Internet. Several tricks exist to try to 'punch holes' through the NAT box, of which STUN [66] is a very well-known example. The approach is to exploit properties of certain types of NATs (full cone, restricted cone, port restricted cone, symmetric), but unfortunately, it is still impossible to build direct connections between some types of NATs (most notably between two symmetric NATs). Some other combinations do allow communication by first sending a special connection message via another (publicly reachable) node, usually called a "super node". It must be noted that "NAT punching" works best with UDP, and that NAT boxes usually prevent other IP protocols than a few well-known ones (e.g. UDP and TCP) from being usable. This may be unfortunate, as e.g. DCCP [43] might be a very good alternative to UDP because it provides congestion control and still allows unreliable transport (see also Sections 2.1.10 and 2.1.11).

It is important to take the (in)connectivity between nodes into account when selecting peers to communicate with, next to the already discussed locality and isolation prevention criteria. Note that NAT support is present in the Iphion application since Version 0.1.

2.1.9 Incentives

Studies on the Gnutella protocol [1] revealed that a significant amount of users do not contribute to the network by uploading files, and just download: 70% does not upload anything, and nearly 50% of all responses are returned by the top 1% of sharing hosts.

This is called "free-riding" or "free-loading" and has a great impact on the available bandwidth and QoS. It may seem harder to 'cheat' the network this way when using a set-top box, especially since it is not easy for an end user to change its internal algorithms or install additional software, but in practice

it is very well possible to constrain the bandwidth used by the set-top box on a smart home router, or simply by using the systems on a low-end Internet connection, or e.g. running a filesharing program on a PC in the same house.

If enough high-bandwidth nodes are connected to the system (including repeaters in the ISP networks), this will usually not be a problem, but when resources get scarce, it will be fair to prioritise the well-contributing nodes. As with any ecosystem, a good way to achieve this is by providing an incentive to cooperate in the network. One possibility is to allow 'good' nodes to 'climb up' to the source, and/or to give those nodes more bandwidth in times of shortages.

It is shown in [8] that considering priority can greatly improve both the QoS of a single host and that of the complete system. The paper includes log traces of several flash crowds, and shows how several different prioritisation strategies would have performed, if they were implemented at the time of the flash crowd.

Probably the most popular incentive system is that of "tit-for-tat", as used in e.g. BitTorrent [22]. The advantage of such a system is that it just requires state to be kept about the (usually low) number of data-exchange partners, it does not need any central authority to verify the other party is not cheating (as the node can directly verify the performance) and is very easy to implement (although care must be taken to implement it correctly! See e.g. [41] for some flaws in the BitTorrent reasoning). In literature, the problem is often also known as the "Prisoners dilemma".

In SWIFT [70] the authors propose an algorithm that uses the notion of Risk Takers and Paranoid Traders to allow for fair trading, while providing a way for new peers to start using the network, but preventing free riders to use "white washing": constantly adopting a new identity, trying to make use of the bandwidth provided to allow bootstrapping new peers. Although the system is targeted at VoD in this paper, the authors suggest in another paper ([60]) that it can be applied to a streaming environment as well.

Another paper [35] describes a way to let each peer calculate a score that can be used by other peers to prioritise packets. The score will be higher if a peer contributes more, but too much contribution will hurt its own performance (because the uplink saturates) and this fact is incorporated in this score as well. The paper also shows that taking incentives into account can greatly decrease the variation in QoS of all peers and increases the mean QoS.

The EigenTrust algorithm [42] can be used if a completely decentralised solution is needed to allow peers to judge upon each other in terms of e.g. trustworthiness. The authors show that it is highly resilient to all kinds of attacks to the system. Note that such a distributed algorithm might not be needed for the Iphion protocol, as relay servers in the network can offer a much more lightweight alternative and even enable administrative control over this information.

Note that due to the nature of live streams there will be an inherent 'downwards flow' (from nodes near the data source to 'leafs' in the network) of data, and thus any reward will flow upwards. Some means must therefore be provided to return this flow back to the leafs: lower nodes should be able to reward higher nodes, most likely by an 'external' incentives system (i.e. additional to an immediate tit-for-tat mechanism between similarly ranked nodes). One can think of an extra score to obtain higher priority at relay servers, or even points to download extra themes/skins for the user interface, etc.

2.1.10 Transfer protocol

Most applications on the Internet exchange data using the Transport Control Protocol (TCP) [3]. It was created to allow reliable connections even in the case of packet losses, by measuring and communicating these losses between nodes, and retransmitting packets if needed. As TCP ensures reliable and in-order delivery, lost packets will have to be retransmitted before other packets can be delivered to the application. TCP buffers up all consecutive packets that arrive out-of-order, until the lost packet is received. In a live streaming environment, this means it can hold up the stream, waiting for a packet that may arrive too late to decode in time, and thereby also thrashing the usefulness of the held up packets.

To prevent this, most streaming applications use the User Datagram Protocol (UDP) [62]. It is an unreliable protocol, which means there is no guarantee a packet will arrive at the destination, and no

promises are made for in-order delivery. Usually, some extra (application layer) protocol is used on top of UDP (like RTP) to provide sequence numbering, timestamping, etc.

The Iphion application uses UDP to transfer its data, and implements a custom protocol on top of that to e.g. re-order packets in its buffer and re-request missing packets.

2.1.11 Congestion control

Although UDP is clearly a better choice for live videostreaming than TCP, it lacks a built-in congestion control mechanism. To be able to maximally use the available bandwidth in the network, without causing excessive packet loss and queuing delays, some means to determine the best send rate must be implemented on top of UDP.

Routers and switches on the Internet drop packets if their link cannot sustain the current rate (i.e., when the link becomes congested), so it is logical to implement congestion control by measuring the amount of packets that are lost at the receiver, sending summaries about this back to the sender, and having the sender adapt the send rate accordingly.

Designing such a congestion-control algorithm is a highly non-trivial task, as many factors have to be taken into account. For example, although missing packets may seem easy to detect using sequence numbers, out-of-order delivery of a packet could cause this packet to be reported missing, whereas it will be delivered shortly thereafter. Additionally, traffic has to compete with other traffic on the network, preferably in a fair way. In general this is called TCP-friendliness, as it usually involves using similar back-off techniques as those used in TCP, enabling TCP flows on the same network to obtain their fair share of bandwidth. A good description of why congestion control is crucial for any Internet application is given in [29].

Note that just limiting the amount of requests at the receiver side based on a custom heuristic on data actually arriving is *not* sufficient! First, this does not take fairness to other data flows into account, and second there is a great risk of so-called 'congestion collapse' causing the network to effectively grind to a halt. Version 1.0 of Iphion's application in fact exhibits this behaviour (see Section 2.7).

A well-known congestion-control algorithm that is specifically designed for multimedia streaming that is used in practice is TCP-Friendly Rate Control (TFRC) [36]. Besides being fair to competing TCP flows, the algorithm tries to maintain a more stable rate than TCP does, which unfortunately also implies a slower response to a sudden increase in bandwidth availability. This algorithm is used in many streaming applications, because of the usually (near) constant bitrate of the streams. It seemed a good idea to use this algorithm for Iphion as well, but as will be shown in Section 4.4.3, the algorithm does not quite behave as expected when used on typical cable modem links.

2.1.12 Load balancing

The subject of load balancing is closely related to both incentives and congestion control, and interacts with them. There are subtle differences though: incentives are about inspiring people to share, congestion control is about preventing unnecessary packet loss and being fair to other network users, and load balancing is about finding a way to optimally share the bandwidth between nodes, thereby satisfying the constraints of, and using information and principles of incentives and congestion control.

Many algorithms are devised to create optimal and efficient bandwidth allocation. Indeed, it is a fundamental problem in graph theory, often referred to as "minimum cost", "minimum cut" or "maximum flow" problems. For general graphs, this problem is NP-complete, which means approximating heuristics will have to be used. Often these algorithms require full knowledge of the network, which is unfeasible in a P2P setting.

Some algorithms are available that try to solve the problem in a distributed fashion. This usually includes transforming the problem to a Linear Programming (LP) problem [17]. Although a hefty bit of math seems to be involved here, the software implementations appear to turn out a lot less complex, and achieve fast convergence to near-optimal solutions.

Examples of such systems using (micro-)payment schemes or (micro-)auctions can be found in [47], [37] and [46]. These approaches all focus on the P2P scenario. Other, more lightweight approaches (in terms of implementational headaches) include [56].

2.1.13 Packet retransmissions vs. Error correction

Several types of errors can occur when sending data over the Internet, such as bit errors or complete packet losses. Bit errors do not have to be considered in the Iphion application, because UDP (and TCP) packets generally contain checksums and the operating system will discard packets that fail this check. In effect, bit errors are 'converted' to packet loss. Note that there exists an alternative to UDP, called UDP Lite [45], that allows the checksum to cover just a part of a packet (e.g. just the header), thereby passing the payload unmodified to the application even in the case of bit errors. A disadvantage of UDP Lite is that it uses a different IP (Version 4) protocol number than UDP, which prevents the protocol from traversing UDP Lite-unaware NAT boxes.

The main type of error to consider therefore, is packet loss. This can be random, or of bursty nature (in case of failing wireless links for example), but can also happen as a normal part of congestion control (see Section 2.1.11). To combat packet loss, two main approaches can be taken: packet retransmissions and Forward Error Correction (FEC).

Packet retransmissions can be handled by the IP protocol itself (such as TCP) or be inherent behaviour of another part of the application (such as the packet scheduling in data pull approaches). This requires that a feedback channel exists from receiver to sender (which is problematic for one-to-many communication because of feedback implosion).

In the case of FEC, no feedback about missing packets is needed, because the encoded packets contain extra information to decode the original data even when some packets are not received. Of course, this introduces overhead, and there is a limit on the amount of packets that can be lost. Examples of standard FEC algorithms are the Information Dispersal Algorithm (IDA) [63], Tornado Codes [10] and the recently invented Raptor Codes [69]. Implementing FEC in an application can always be done by 'encapsulating' the original datastream with FEC protection (but beware of special assumptions about TS packets having specific headers/timestamps etc.), or it can be integrated into the protocol/algorithm. A good example of the latter is SplitStream [12], which uses multiple trees to transmit different 'slices' of the protected stream, and subscribes to more trees when more packet loss occurs in the network. When enough slices arrive (i.e. enough trees deliver the data) the original stream can be decoded.

Note that FEC is probably less useful for data pull, because it can often be solved by sending a requestmap earlier. FEC provides a delay/data overhead trade-off in case of data pull, and a continuity/data overhead trade-off for data push.

2.1.14 Performance measures

To comment on the performance of P2P algorithms, several measures are used. For example, Narada introduced physical link stress and relative delay penalty (or link stretch) [20, 19]. Application layer multicast implies that peers not only receive data, but also send data. The number of times the same data is sent over one link determines the link stress. So, with IP multicast this number is simply 1, but in the case of a binary tree overlay this number will be 1 or 3 (per link), depending on whether a peer is a leaf or internal node. Link stretch is defined as the increase in path length (in hops) divided by the shortest (IP multicast) path that would be possible. Another measure is the amount of overhead for maintaining the overlay, usually expressed relative to the amount of video data.

Some papers (e.g. [50, 83]) mention the so-called playback continuity, which is defined as the ratio of packets that were not available at decode time to the total amount of packets needed by the player.

Additionally, simulations and real world implementations express values in (milli-)seconds, like startup delay, end-to-end latency, tree repair delay, etc. Papers containing good comparisons between several existing protocols include [72, 77].

2.1.15 Simulation

Few proposals are actually implemented and deployed in public, the most famous examples probably being BitTorrent and CoolStreaming. Although a real world implementation provides valuable data,

it is not feasible in most cases, and special scenarios like massive node failure, very high or very low churn rates are hard to (re)create.

Thus, some researches chose to simulate or emulate their proposals. It is possible to use standard P2P simulator frameworks like PeerSim [40] , use a topology generator (e.g. BRITE [53] or GT-ITM [11]) and/or network simulator (like NS-2 [51]) or emulator (like Dummynet). However, most papers seem to implement a custom simulator (or nothing at all), because the existing simulators are found to be too detailed or too generic to illustrate the problem a specific algorithm tries to solve. For a recent overview of used simulators including their properties, see [54].

It is also possible to implement the application on PlanetLab [21], which is a global network of nodes (mostly at universities). It provides a good environment to perform real world testing, with the ability to (remotely) control the clients. Its scale is however limited to a few hundred nodes, which usually is not enough to hit the limits in terms of scalability of an algorithm. Also, the nodes tend to be highly loaded by other (simultaneous) experiments, and their network links are probably not very representative for those of the Iphion clients (which will mainly be ADSL and cable users).

See Section 2.6 for this projects custom-made event-based simulator.

2.1.16 User behaviour

Although some papers use a uniform distribution for joins and arrivals of clients, it has been shown that a heavy-tailed (Pareto) distribution is more appropriate. This fact can be used to create more stable overlays, see mTreebone in Appendix A.3. According to [68], a realistic number for the node churn in P2P systems is between 0.1% and 1%, and catastrophic churn could be up to 30%. Other papers about measurements of public implementations include [38] and [2], both covering the proprietary PPLive system. Note that in the Iphion scenario, many users could be watching the same channel (e.g. a popular soccer match) and massively zap away when commercials start. This qualifies as highly catastrophic behaviour, but should still be handled gracefully by the network!

Tolerable zapping delays seem to differ per application: users watching analog television experience sub-second zapping delays, satellite users have delays of a few seconds, and in the CoolStreaming [85] case users experienced delays of around 30 seconds. These numbers differ by two orders of magnitude! According to [50] there seemed to be no correspondence between the popularity of a program and the zapping time, which makes it hard to state definitive numbers for tolerable delays, although Iphion ultimately desires sub-second delays.

2.1.17 Network coding

When downloading data (be it a file or a stream), every peer has to obtain the same set of packets that the source produced. This is the problem of set reconciliation. One way to solve it is to constantly exchange so-called buffermaps between peers, so a peer can determine which neighbour has a desired packet. This creates a direct coupling between the bandwidth allocation problem, and the set-reconciliation problem: nodes can only request packets from the nodes that actually have those packets, instead of being 'free' to select nodes that can provide more bandwidth (or more reliable delivery, etc.).

One approach to combat this coupling is to use network coding. Network coding is a type of information coding, where nodes in the network actively re-encode the passing packets, in contrast to 'just' encoding the packets at the source. A very promising technique called Random Network Coding [39] works by creating random linear combinations of incoming packets (usually in GF(8) or GF(16)) and just passing these combinations on to the 'child' nodes. To decode the data, the received packets are gathered in a buffer, and after enough packets (slightly more than the number of source packets) have been received, a Gaussian Elimination process can be used to solve the linear equations these packets in fact describe. Optionally, a standard Forward Error Correction (FEC) code can be used to protect the inner data against the remaining missing packets (which can occur because of linear dependencies between the coded packets).

Note that it is not necessary to decode the source data in order to create new outgoing packets, linear combinations are directly created from the incoming packets. This is very useful to achieve

a short end-to-end latency. However, in practical implementations [18] the packets are grouped in "generations", "segments" or "slices" (of about 100 packets).

Random network coding provides some significant benefits: two peers can simply agree upon a certain amount of packets per second, and no further communication about what packets to actually send is needed. By eliminating the set-reconciliation problem, a few round-trip times (RTTs) can be saved because a peer no longer has to wait for a buffermap to appear, and there is no need to send a requestmap back to specifically ask for certain packets. Additionally, there is no need for "rarest-first" scheduling anymore, as the data is completely randomly distributed over the linear combinations. The risk of creating very rare packets is therefore greatly reduced.

The big disadvantage is that Random Network Coding is very computationally expensive, as many multiplications between full packets have to be performed especially during decoding (Gaussian Elimination). Although there are very efficient implementations of this (especially in GF(8)), this still means many rotation and XOR operations are necessary. It is therefore probably not feasible to use this on a low-end set-top box (which Iphion will use).

It is also possible to use "Raptor Codes" [69] (an advanced version of "Digital Fountain Codes") which are much simpler in terms of CPU usage. They can be used in a network coding setting, but have the disadvantage that source packets have to be decoded before new packets can be encoded. Additionally, the technique is highly patented (by well over 100 patents) and thus requires a license.

Although much research effort is dedicated to (random) network coding approaches, it will probably just stay a nice dream for some time, particularly for the Iphion application.

2.2 Existing proposals

A summary of existing streaming P2P algorithms can be found in Appendix A, providing better insight in typical P2P pitfalls and their solutions. For example, it contains references to algorithms for achieving good locality while still maintaining some randomness for stability (Cyclon/Vicinity), some numbers on real flash crowds (CoopNet), the reality of the danger of random packet scheduling (Chainsaw), the appearance of 'stable nodes' (mTreebone), and a hybrid data push/pull strategy (GridMedia, HyMoNet) which might be useful to consider for Iphion, etc.

2.3 Research summary and conceptual application analysis

Relevant design considerations and technical terms are now defined, allowing a first analysis of Version 1.0 of Iphion's application.

- The mesh-based structure of Iphion's network enables optimal usage of available bandwidth in the network and is robust to node churn. Although having multiple parents requires set reconciliation, this is quite efficiently solved with buffermaps and offers more fine-grained control of data flows, which is relevant to locality and incentive considerations.
- The buffer size remains fixed during both initial buffering and when playing. It is better to use a small buffer after zapping, and gradually expand that buffer if the network allows or requires it.
- Packet scheduling occurs fully random, but a rarest-first and/or incentives-based approach is preferable. Additionally, the playback deadline of packets and estimated time to retrieve a packet from a certain parent could be included in the scheduler.
- Although clustering according to network locality is one of the goals of Iphion, no support for this (nor for relay servers) has been implemented yet. See Section 3.3 for a possible solution.
- NAT / Firewall support is implemented, but did not yet allow nodes behind the same NAT (with likely very good connectivity) to directly communicate. See also Section 3.3.
- The transfer protocol is UDP, which is a good choice for live streaming because of its 'non-blocking' (i.e. non-in-order and non-reliable) delivery and good NAT-punching possibilities.

- Some congestion control is implemented, but as shown in Section 2.7 needs to be improved significantly. This task appeared to be trivial at first by 'simply' implementing the TFRC algorithm, but as explained in Chapter 4 is rather complex instead.
- No Forward Error Correction (FEC) is implemented, but given the pull-based approach of Iphion, this should not be necessary either.
- Incentives can encourage users to contribute more resources and increase the overall quality-of-service of the network, but because of the 'downwards flow' of a streaming system, this requires some external score to be defined next to the more simple 'tit-for-tat' incentives. No provisions have been made for this yet (and some of the other issues have a higher priority!).

In general, the central ideas of Iphion are very good: the mesh-based approach, the (future) availability of relay servers for efficient (local) bandwidth distribution, improving zapping and helping out in 'emergency situations'. The application already works in real life, even through NAT, and provides initial security protection.

However, according to this first analysis, congestion control and locality are important issues to be solved.

2.4 Iphion implementation-specific analysis

Additional points of consideration, specific to Iphion's Version 1.0 implementation are:

- Streaming packets from the buffer to the media player was directly coupled to the tail of the buffer, which complicates the buffer-advancement code but most importantly requires a large (additional) buffer in the media decoder (see Section 3.1).
- New requestmaps are generated every 150 ms and packets are expected to arrive within this time. Section 2.7 shows that this approach will not work well on the typical Internet links that Iphion will encounter.
- A cryptographic-signature system is implemented which enables peers to verify that received packets indeed originate from the indicated sender. However, it does not protect against protecting the network in case an Iphion set-top box is hacked: the stream data itself is not checked for authenticity. Once an Iphion box is hacked, it is fairly easy to completely 'crash' the network, and even insert a malicious stream.
- Although the Iphion application can be turned into a 'relay mode', in practice this just disables the output to the media decoder. All administrative tasks, such as maintaining the network overlay, but also distributing the stream to the network is still handled by one central server.
- Nodes with increasingly lower ranks will have increasingly higher absolute playback delay, because they need to wait for the data to be downloaded and advertised by their parents first. In the current implementation, this delay (the "rank difference") is fixed (48 packets by default). It may be wise to make this difference adaptive to the actual state of the network (e.g. based on the RTTs between partnering peers).
- Because of a limitation in the current bandwidth administration of partnerships, it is not possible to create partnerships between equally ranked nodes. The streaming server (and later, relay servers) will be able to support a substantial amount of peers in the first rank, which will thus not be able to contribute any bandwidth to each other. It is a good idea to remove this limitation from the implementation, as it will allow the bandwidth of leaf nodes in the network to be used.
- Partnerships between nodes are currently bi-directional. Ironically, bi-directional links are mostly useful for 'cross links' between equally ranked nodes, which are currently not possible. Because being a lower rank node implies having a higher absolute delay, the average flow of data will

always be from higher to lower ranks. Thus, it is logical to make partnerships uni-directional, also saving some bandwidth (as buffermaps no longer need to be sent from lower rank nodes to higher rank ones) and simplifying the internal architecture. When cross links become possible, nodes can simply set up two uni-directional links.

- The rank-select algorithm assigns ranks to new nodes based on the amount of upload bandwidth a newly joining peer advertises, and a peer is assigned a rank for the full duration of its stay in the channel. No reassignments of existing nodes are performed when nodes join or leave the network, possibly resulting in suboptimal rank assignment of both existing and new nodes.

- The rank-select algorithm is currently performed by the central streaming server, but should ultimately be distributed over decentralised relay servers in the network.

- There is an unnecessarily tight coupling between the scheduling of packets, and the rate-control algorithm. This means requestmaps have to be computed and sent fairly often, which is a relatively expensive operation (a.o. involving comparing buffermaps of all partners). Decoupling also opens up possibilities for easier integration of incentive-based data exchange and simplifies the internal architecture.

This second analysis shows that (additional to congestion control and locality) the system architecture, packet scheduling and packet streaming need substantial improvements as well. These issues will therefore be the topics of the remainder of this report.

2.5 Typical cable modem behaviour

Some experiments have been performed to investigate the behaviour of typical cable or ADSL modem links. The reason is the rumoured large queues in these modems, which could cause delays of several seconds and can thereby seriously affect the performance of the Iphion network. Although the remainder of this report will generally refer to this type of links as cable links, the results equally apply to xDSL links as well.

As will be clear from the measurements, queue sizes can indeed cause RTTs to a e.g. a relay server of 9 seconds, which means a maximum RTT of 18 seconds (!) can arise between two of these nodes. Considering the desired sub-second zapping delays and a buffer size of just a few seconds of the Iphion application, this will make data exchange (even with the relay server) impossible.

Note that these extremely large latencies usually do not manifest themselves when using TCP connections (e.g. when uploading a large file through FTP), which is explained in Section 4.6.

2.5.1 Test setup

A simple experiment was designed to determine the size of the (uplink) queue in a typical cable modem. As there is no direct access to the internals of the cable modem, the size must be inferred from measuring the increase in RTT caused by packets waiting in the queue.

To be sure that just the uplink is saturated, the experiments have been performed from a machine in the home network, such that the uplink (bottleneck) is traversed before the downlink is traversed, see Figure 2.2. To gradually fill the queue, the "ping"-command of Linux (Ubuntu 7.10) was used, with the following commandline:

```
ping -n -s 1440 -i 0.001 -c 3000 ping_host
```

where `ping_host` is a node with very good Internet connectivity (100 Mbit). This command sends (large) packets at a speed faster than the uplink can handle. On a 'standard' queue (a so-called "drop-tail queue"), the measured RTT will gradually increase until the queue is full, after which the RTT remains nearly constant and packet loss occurs.

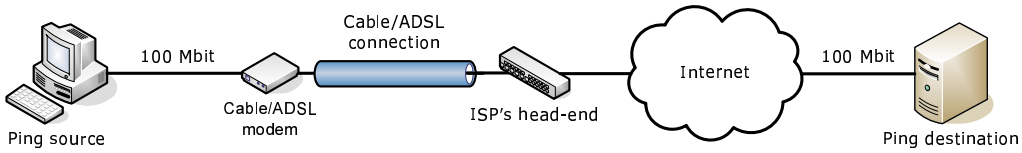


Figure 2.2: Schematic representation of setup to determine queue size in cable/ADSL modem. Note that because the ping source is located at the residential side of the modem, the replies of the ping destination can not saturate the downlink, if the uplink is the bottleneck.

2.5.2 Queue sizes and round-trip times (RTTs)

The results of these experiments on three different links are shown in Figure 2.3, which indeed shows the initial increase in latency, and the flattened curve when the queue is saturated, indicating a drop-tail queue. From these results, the queue size can be computed as:

$$n_{queue} = (t_{max} - t_{link}) \cdot b_{link}/s_{packet},$$

with n_{queue} the size of the queue in number of packets, t_{max} the maximum RTT in seconds, t_{link} the RTT of the idle link in seconds, b_{link} the speed of the bottleneck link in Bytes/s, and s_{packet} the size of the packets in Bytes.

To determine whether the size of the queue is limited by a number of Bytes or a number of packets, the same experiment was repeated using 720 Byte ping packets, which yielded similar results, but with a maximum RTT of half that of the 1440 Bytes packets. This indicates a limit based on number of packets.

From Figure 2.3, it can be found that the maximum RTT of around 8.9 s on the 256 kbps link and the 2.2 s RTT on the 1 Mbps link both correspond to a buffer size of approximately 200 packets. The queue of the 768 kbps line (maximum RTT of 0.9 s) is considerably smaller: 50 packets. Note that even this fairly fast link, with a moderate queue size still causes an RTT of almost one second, which is much larger than what Version 1.0 of the Iphion application can handle (see Section 2.7).

Although this experiment has been performed on just a very small number of links, a comprehensive investigation was found later in [27], indicating typical maximum (mostly drop-tail) queuing delays of several seconds, in accordance with the results of this section.

2.5.3 Relevance

The sizes of queues in typical cable and ADSL modems can be very large compared to the rule of thumb for queue sizes. This rule states it should be one or a few times "the bandwidth-delay product" of the link or flow, which is hard to define (even with typical RTTs varying from e.g. 10 ms to 100 ms and flow speeds from 1 kB/s to 100 kB/s, already covering a range of 10^3).

Many Internet protocol simulations use this rule of thumb, often with Random Early Detection (RED) as the queue model instead of a drop-tail queue (RED will generally lead to lower long-term RTTs). From the simple experiments in this section this is proven unrealistic for cable links: it is not just the speed and asymmetry of these links that makes them special!

Because the Iphion application will mainly be used on these types of links and has tight constraints on latency, it is important to include these facts in the design of the application, for both locality considerations (which are often RTT-based) and congestion-control algorithms.

2.6 Simulator

At first, it was assumed that the 'popular' choice for TFRC would adequately solve the lack of a proper congestion control, so focus shifted to another major missing issue: adding the concepts of relay servers and locality awareness into the protocol, while indeed taking the possibly large RTTs of the links into account.

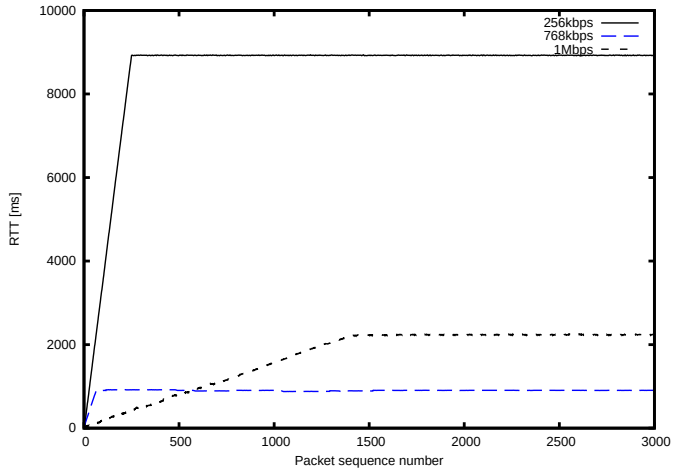


Figure 2.3: RTT measurements when gradually saturating modem queues using large ping packets. Note that when the curve flattens, packets get lost. Steepness of the first part of the plot depends on link speed and the speed at which packets are generated by "ping" (which is less than 1000 packets/s in practice due to operating system scheduling granularity). The final RTT is determined by link speed and queue size.

To test whether new ideas would actually work, it must be possible to instantiate many Iphion players, connected through different types and speeds of networks.

One way to do this would be to emulate those networks, and run 'real' instances of the Iphion application on it. Setting up this many instances with e.g. FreeBSD's network emulator Dummynet is very complex, and likely requires a multi-server setup. Additionally, some means to gather information about all nodes and network links must be developed. Even if all this is done, it is still practically impossible to obtain repeatable experiments, or to conveniently pause the system, change some parameters or e.g. inspect the contents of a network packet. Combined with the fact that the 1.0-version of the application did not allow for easy modification of its protocol and internal structure, this led to the conclusion that emulation was not the right tool for this job.

Most P2P projects do not have a real implementation at all, and instead simulate their proposal. Although a number of simulation frameworks exist, most of these simulators are still targeted at simulating certain aspects of algorithms. Some researchers focus on the low-level network aspects of peer-to-peer applications, for which NS-2 [51] is a suitable tool. Others want to investigate the scalability of their proposal, and therefore mainly model the complex underlying behaviour on a very high-level, so that thousands of nodes can be simulated efficiently.

A review of the state of current simulators [54] reveals that, despite the availability of existing simulators, most researchers still build their own simulation tools which can then be better suited to the specific problem at hand. Because of the limited scope of the initial problem (mainly improving locality, respecting the properties of cable links) and the obvious unavailability of the Iphion protocol in any existing simulator, a custom event-based simulator was developed.

It must be said that the project turned into quite some more than what was initially conceived (largely because of the later desire to also investigate TFRC's behaviour and more accurately study the effects of different buffering stategies) and has been an instructive experience in many ways...

2.6.1 Structure

Figure 2.5 depicts the most relevant simulator classes, ordered by three layers:

1. The simulator framework itself, providing for runtime inspectable objects, an event queue, a

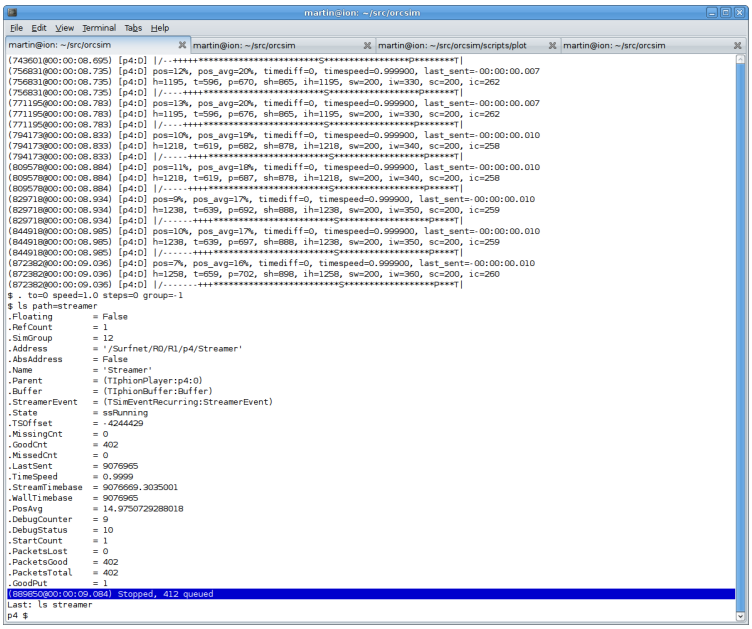


Figure 2.4: Screenshot of simulator, when running the default Iphion network for some seconds.

command system, etc.

2. The network simulation objects, which are derived from the base SimObjects and simulate the network links, routers, queues, sockets, etc.

3. The implementation of the Iphion application and protocol, consisting of streaming server, relay server and Player objects.

2.6.2 Framework

The simulator is event-based (as opposed to model-based) which allows to e.g. compute the behaviour of individual packets. An example execution flow of a network packet travelling from one node to another is depicted in Figure 2.6, which represents the execution of four events.

The framework of the simulator is built on SimObjects (see the first layer of Figure 2.5). These objects have methods and properties that allow them to be 'interrogated' while the simulator is running, and make it possible to view and change their behaviour even during simulations. Almost every other object in the application is derived from TSimObject or TSimCollection (which can hold other SimObjects).

The driving force of the simulator is the Simulator object, which maintains the simulator time and a list of SimEvents. SimEvents are used to create timers, simulate the latency of a link, etc. and are implemented with a microsecond precision timestamp at which its callback function should be executed. The event-based structure makes it possible to start and pause the simulation at any time, or run faster or slower than real-time if processing power is sufficient (the network of Figure 3.5 runs nearly real-time on a 2-GHz Intel Core2 Duo E4400).

The simulator is controlled by the CommandExecuter object, which provides a set of built-in commands to control the behaviour of the simulator, browse around from one simulator object to the other, view and set their properties. It is also possible to export the state of all objects to a file for visualisation using Graphviz tools (e.g. Figure 3.5). To create plots of e.g. bandwidth, queue sizes and number of playing nodes, one can define 'jobs' that log those characteristics to trace files for processing with gnuplot (e.g. Figure 4.3). The CommandExecuter also allows to define custom functions that combine existing commands.

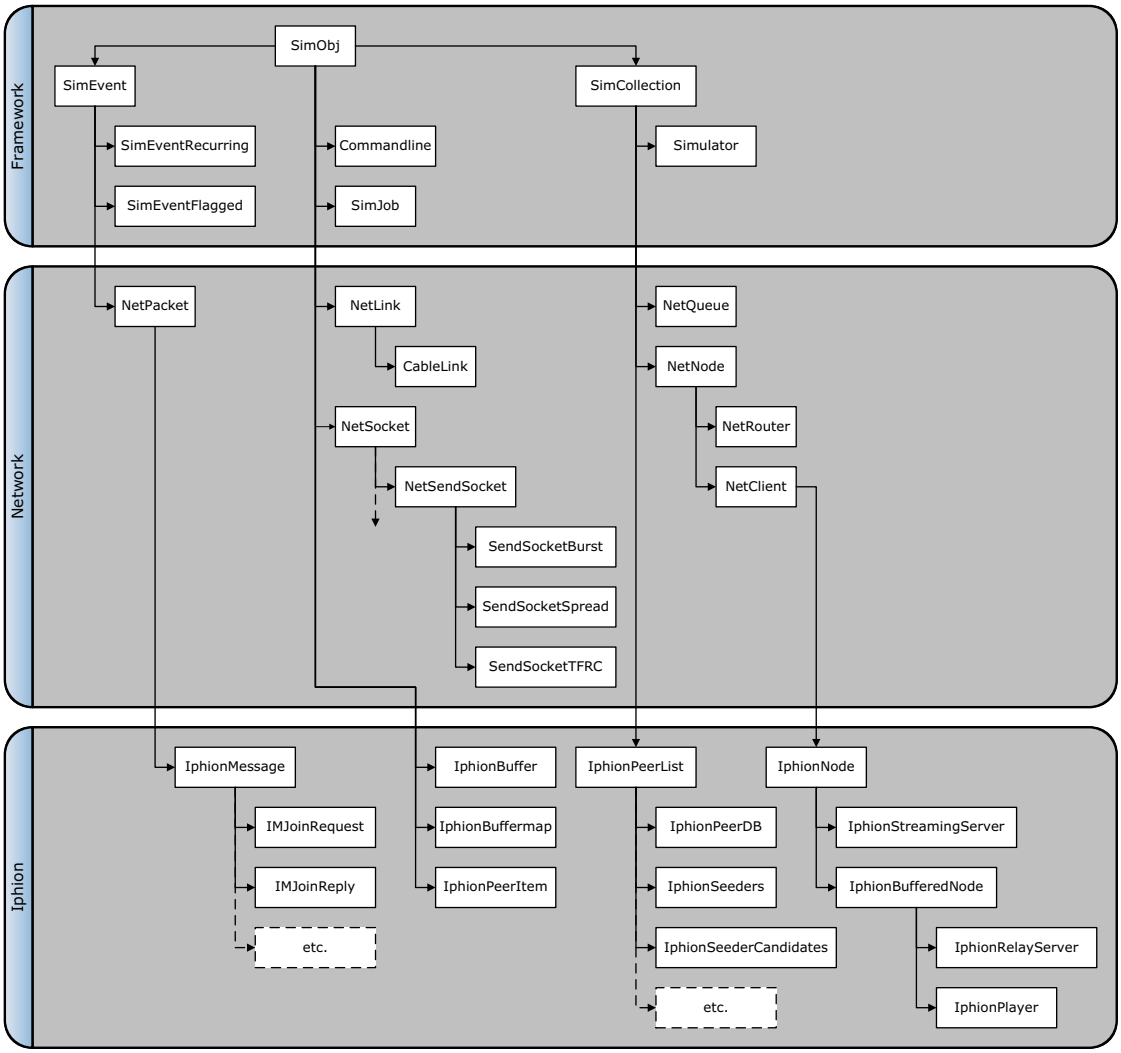


Figure 2.5: Class inheritance diagram of the most relevant simulator classes, grouped by functional layer.

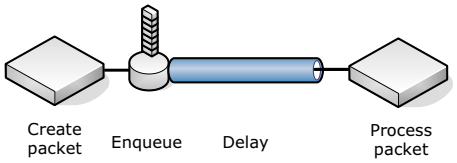


Figure 2.6: Example flow of execution of a network packet, travelling from one node to another.

Figure 2.4 shows a screenshot of the simulator, after running for a few seconds and examining the status of the buffer of one of the Iphion players ("p4").

2.6.3 Network simulation

The second layer in Figure 2.5 consists of objects that form the basis for network simulation, such as network links with their queues, and a basic packet class from which all other packet types are derived. Figure 3.5(a) shows a typical simulated network topology, containing 135 Iphion players connected in a hierarchical structure, divided over three different ISP networks, where each network can have different link speeds, latencies and queue sizes.

Implementation details The simulated network consists of the root node (called "Amsix", after the Amsterdam Internet Exchange), and zero or more other nodes connected in a tree structure. Thus, every node (except the root) has exactly one parent, and zero or more sub-nodes. There can be several types of nodes: the basic NetRouters and NetNodes, and the Iphion specific IphionStreamingServer, IphionRelayServer and IphionPlayers. Usually, one defines a network with NetRouters as the internal nodes of the tree, and any of the other types as the leafs.

The NetRouters implement hierarchical routing, which means that as long as the address of the current router is not a prefix of the destination address of a packet, the packet is sent to the parent router, until it reaches a common ancestor. From there, the packet is routed down the tree again to the final destination. Example addresses are "/Surfnet/R0/R1/p4" and "/Surfnet/R0/R3/p18". Note that although 'IP addresses' can be assigned to nodes, they are not (yet) used for routing packets, but are useful for testing IP-address-based locality algorithms (Section 3.3).

A pair of nodes is connected by two NetLinks: the "uplink" to the parent, and the "downlink" to the child. Every link has a number of configurable properties like latency, bandwidth, queue size, etc.

Cable and ADSL links can also be configured, by setting the bandwidth limit (usually called "speed cap") of such a link. Based on measurements and literature, this is implemented as a "leaky bucket", allowing (very) short bursts of data to pass at the physical (higher) link speed, limiting remaining data to the configured limit. This means that RTT measurements of short bursts of small packets on an otherwise idle link will indicate a high link speed, but as the speed cap kicks in, the increase in RTT due to packet queuing starts to show the expected differences based on the speed cap.

A few different configurations of link types and speeds are pre-defined in the simulator, allowing to simulate backbone links (which are dimensioned such that they support practically unlimited flows, but do introduce some latency because of their length), campus links (typical 100 Mbit short distance) and 'slow' and 'fast' cable/xDSL links, i.e. high speed physical links (e.g. 20 Mbit), with speed caps (e.g. 2 Mbit) and relatively large queues (200 packets by default).

Nodes can exchange packets, which are sent as UDP-like datagrams. Different types of packets can be created by simply extending the NetPacket base class, and adding custom properties (or even methods) to them. As packets are not serialised to a Byte stream for sending, one must also set the Size property in the constructor of the derived packet class, to allow the NetLinks to accurately simulate queuing delays etc. Note that it is usually not necessary to set up sockets on both ends of a 'connection': a node simply constructs a new instance of a packet class, sets the destination address on the packet, and calls the Send() method of itself. This means that these packets are sent without any rate limits: if the queue on a bottleneck link is saturated, the new packet is simply dropped (with drop-tail queues currently being the only implemented queuing policy).

In order to simulate different types of congestion control, there is a pair of base classes NetSendSocket and NetRecvSocket from which currently 3 different algorithms are descended: one to send all packets in one burst at once (Iphion's application Version 0.1), one to send all packets in a fixed interval (Version 1.0), and TFRC (see Chapter 4). These sockets should not be confused with 'conventional' network sockets, as they work quite different. For example, the sockets use 'callbacks' to request new packets from a node when needed, which enables nodes to quickly respond to newly arriving requestmaps, instead of enqueuing a batch of packets on reception of a requestmap. This principle may prove useful for implementation in the real Iphion application as well.

2.6.4 Iphion network implementation

The third layer in Figure 2.5 shows main objects involved in simulation of Iphion's network. In general, the implementation in the simulator accurately reflects the implementation of the 'real' Iphion application, with much of the original protocol being similar (e.g. the video packets, requestmaps, etc.). Important concepts such as the packet-scheduling algorithm have been 'translated' to the simulator as well to allow fair comparisons. Its behaviour was empirically confirmed to be consistent with that of its real counterpart.

There are also some differences: Iphion's application has had quite a lot of improvements since its original creation, some of which did not fit well to its original architecture, gradually leading to sometimes unnecessarily complex and interwoven code. Analysis of the (sub-)tasks of the application has led to a new, simpler and more natural architecture, which has been implemented in the simulator. These architectural improvements are explained in Section 3.2, and do not change the behaviour of the protocol itself.

Because the simulator has been designed to incorporate relay servers and locality into the protocol, some changes have been made to the protocol as well. These changes are described in Section 3.3. The original rank-assignment algorithm has not been implemented, but to still be able to emulate its behaviour (see the next section), it is possible to manually assign partnership relations.

Other notable differences are the absence of crypto and NAT detection and traversal algorithms, but this does not influence the locality and congestion-control experiments.

The 'video stream' generated by the streaming server can either generate a perfectly constant bitrate stream of any number of packets per second (each containing 7 MPEG TS packets), or a variable-rate stream which tries to mimmick variations in instantaneous bitrates. The default speed of the video stream is set to 100 packets per second, corresponding to approximately 1 Mbit, which is the targeted bitrate for Iphion.

2.7 Iphion application on cable modem links

Although the simulator was originally intended for locality experiments, its ability to accurately simulate cable modem links and detailed protocol behaviour allows for investigating congestion control as well.

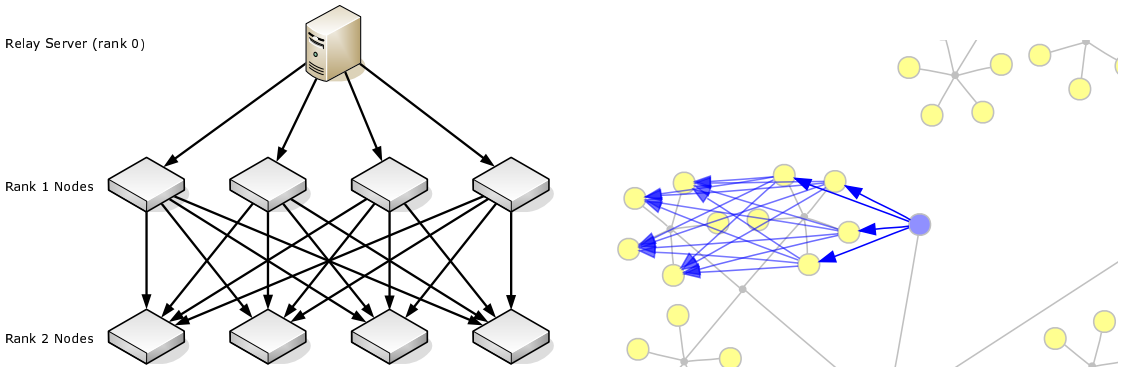
Given the problematic queues of typical cable/xDSL connections and the potential issues mentioned in Section 2.4, the original Iphion application is expected to perform badly on this type of links.

2.7.1 Test setup

To test this behaviour, the setup of Figure 2.7 was loaded in the simulator. It shows a relay server providing data to four rank-1 nodes, which in turn deliver data to four rank-2 nodes, as would be assigned by the rank-select algorithm mentioned in Section 1.5.2. Note that in general, the relay server will support many more rank-1 nodes before nodes will be assigned to rank 2, but as the behaviour of just one node of every rank will be examined it does not matter much how many other nodes will be in the same rank.

All network links except those of the player nodes are modeled as 10 Gbit links, to exclude any bottleneck effects inside the ISP network itself. The up- and downlinks of the players are 2 Mbit and 20 Mbit respectively, with an uplink queue size of 100 packets. Note that this configuration simulates a very fast link already, with a relatively moderate queue size. Given the video bitrate of 1 Mbit (completely CBR, for easy visualisation), this network should easily be able to support the players. Idle-link RTTs of players to the relay server are 13 ms, and approximately 20 ms between players.

First, the rank-1 nodes are enabled at $t = 6$ seconds (the first few seconds are used to let the relay server buffer from the streaming server). The players are allowed to settle for 10 seconds. Then, at $t = 16$ seconds, the nodes in rank 2 are enabled as well, and the simulation is executed for another 10 seconds.



(a) Logical representation of the test setup

(b) Physical representation of the test setup, see also Figure 3.5(a)

Figure 2.7: Setup to investigate behaviour of original Iphion application on typical cable connections. The images provide two conceptual views on the same topology.

2.7.2 Results

Figure 2.8 shows the results of this simulation. Note how initially the bandwidth usage is higher than the stream rate because of pre-buffering, but then remains steady for a while. When the rank-2 nodes join the network, the amount of packets they request quickly saturates the uplink queue of the rank-1 nodes, yielding large RTTs (450 ms, although reality has shown that much larger RTTs could occur as well!). This in turn leads to two effects:

1. The rank-1 nodes start to re-request packets over and over again, because they expect packets to arrive within 150 ms, which now takes approximately 470 ms. The nodes thus request all packets four times (which in fact all arrive due to the high speed of the downlink). Note that in the desired case, the line of the downlink usage should remain nearly constant after the pre-buffering stage, even when the rank-2 nodes join.

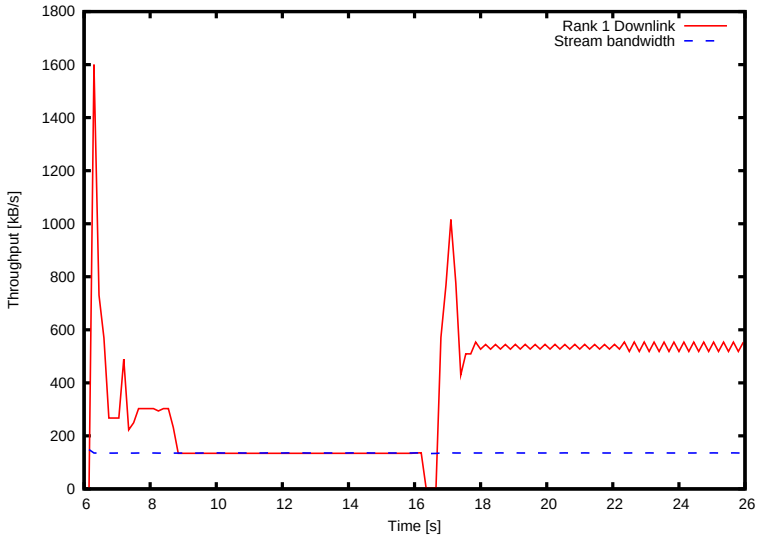
2. Because there is no limit on the amount of packets sent to the rank-2 nodes, the uplink becomes fully saturated. The huge amount of packet loss, combined with the high RTT is responsible for keeping the rank-2 node requesting packets repeatedly as well, thereby flooding the uplink because of the lack of congestion control, etc.

This presents a typical example of congestion collapse: a relatively stable situation in which much data is transmitted, but a small part actually arrives.

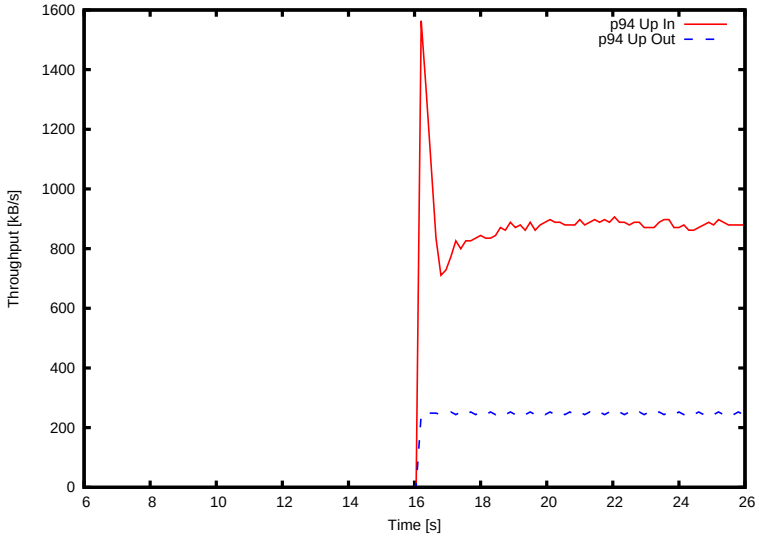
The problem would be less severe if even one of the issues would be solved: implementing congestion control could prevent the RTT from becoming too high, whereas smarter packet requests could prevent the continuous link overloading, so that finally the queue would reach the empty level again.

However, both approaches would need to be implemented to cope with both the scenario of large RTTs (e.g. due to other data flows), and prevention of (even temporarily) flooding the uplink queue (e.g. when new nodes join and as will be seen later also during steady state).

Section 3.4 describes a smarter packet scheduling solution for the first problem, and Chapter 4 is dedicated to finding a suitable congestion-control algorithm, solving the second problem.



(a) Stream rate versus download rate



(b) Bandwidth on both sides of the uplink. The difference between the two lines causes excessive packet losses.

Figure 2.8: Bandwidth usage of one of the rank-1 nodes. Note the unnecessary increase in download bandwidth, and significant overload of the uplink after the rank-2 nodes join the network at $t = 16$ s.

3 Generic improvements

The previous chapter introduced fundamental concepts of P2P applications and analysed some of the remaining issues of Version 1.0 of the Iphion application. Then, using the custom-made simulator and tests on some typical cable/ADSL links, showed possible problems of very large queuing delays for the existing Iphion application.

This chapter first describes an improvement (the "Streamer") to the Iphion application, which has been integrated into the 1.0-version. Then, a new view on the system architecture of the application is presented, which was first implemented in the simulator, and is now being implemented by Iphion for the upcoming 1.1-version. Next, the initial work on locality improvement is described, which is largely implemented in the simulator and positively received by Iphion developers. Finally, a solution for the inefficient packet requesting mentioned in Section 2.7 is presented.

3.1 Streamer

The first improvement to the Iphion application was the so-called "streamer". This not only was a good way to get acquainted with the application, but also solved two issues at once: the 2 to 4 seconds of additional buffering in the media player could now be reduced to just 400 ms, and the buffer-advancement algorithm was now decoupled from media playback, paving the way for simpler code and better response to network changes.

The streamer became part of the 1.0-release of the Iphion application.

3.1.1 Coupled pointers

To keep all nodes in the network synchronised to the stream of the server, peers compare their buffer's head pointer with those of their parents. As the buffer has a certain maximum size (configured when joining a channel), the tail of the buffer then maintains a fixed offset to the head and packets 'fall' out of the buffer at the tail.

In earlier implementations of the Iphion application, these packets were sent to the media player. In case of a completely constant bitrate, this would not seem a problem, but streams can have quite a substantial amount of variation in bitrate (due to differing encoding complexity of scenes, but also due to different bitrates for I, P and B frames), which means that the speed at which the media player is processing the data (at the tail) will be different from the speed at which packets are sent out at the server (which is tracked by the head).

Figure 3.1 shows an illustrative example of the possible divergence between the pointers. From this figure it is clear that the media player then needs a substantial amount of additional buffering to absorb the differences in (instantaneous) speed of the tail (which is directly coupled to the speed of the head because of the fixed buffer size) and the speed at which the media player needs the packets.

The tight coupling between buffer advancement and media playback has two disadvantages:

1. Large additional buffering in the media decoder
2. Too tight constraints on buffer advancement (which must be smooth enough for the decoder to follow)

3.1.2 Decoupling pointers

Therefore, an extra pointer called the "streaming pointer" or "playback pointer" was created. This pointer uses timestamps in video packets to determine the right moment to stream packets to the

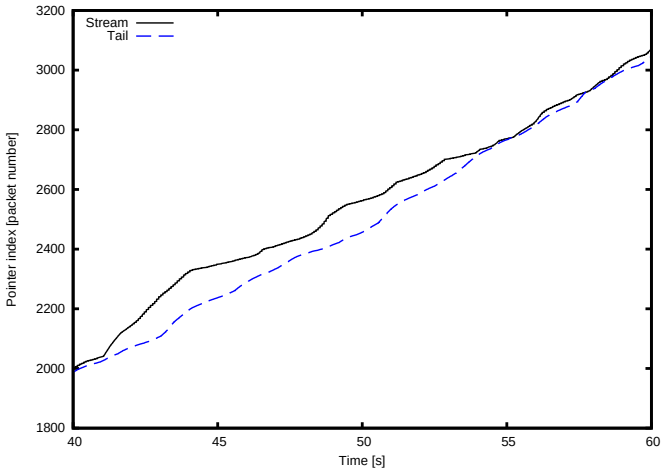


Figure 3.1: Instantaneous speeds of playback and tail pointer for one of the test streams, after implementing the Streamer. Note that the divergence between the pointers is up to approximately 150 packets for this stream. This would have required an additional buffer of at least 3 seconds in the media decoder, if just the tail pointer would be used to stream packets to the decoder.

media decoder, and can 'freely' wander inside the buffer (Figure 3.2). The buffer-advancement code is now relieved of the smoothness constraints, but care must be taken to not advance the tail beyond the playback pointer.

The buffer needed in the decoder could now be reduced from 2–4 seconds down to 400 ms. Note that this buffer can not be removed completely yet, mainly because of the current implementation of stamping packets at the server end (which was already implemented but unused in the 0.1-version). This introduces some jitter: in case of 'streaming' from a file, packets are grouped based on the presence of a PCR stamp, not their decoding timestamps. In case of live streaming, packets arrive in small batches due to caching behaviour in the transcoder and operating system, which causes all packets in one batch to be stamped with the same time. These issues were improved by using UDP for live stream input (instead of HTTP), but still partly remain.

3.1.3 Synchronisation

In a (unicast) streaming scenario, packets are sent over the Internet by the server, and received with a certain absolute delay by the media player. This absolute delay has jitter and the instantaneous stream bitrate varies, which is both handled by the buffer in the player. The 'mean' speed of arrival of the video packets determines the mean speed at which the player should consume its stream, and it will adapt its internal clock (a PLL) to this speed, possibly e.g. resampling audio as needed.

Before the streamer was implemented, the speed at which packets were sent to the media player was determined by the speed of the tail of the buffer, which was directly coupled to the speed of the head, which in turn was coupled to the speed at which new packets became available at the server. Therefore, the media player would correctly follow the 'mean' speed of the server.

With the streamer, this is no longer the case as the streamer has its own clock, and compares the timestamps in the UDP packets to this clock. If this clock is not somehow synchronised to the clock of the streaming server, the media player will slowly diverge from other players in the network. To prevent this, the streamer clock can be adjusted to run slightly (e.g. 100 PPM) faster or slower than the operating system clock. This adjustment will also cause the media player to sense the new arrival speed and adapt to this.

To determine whether the clock should run faster or slower, the initial implementation (having

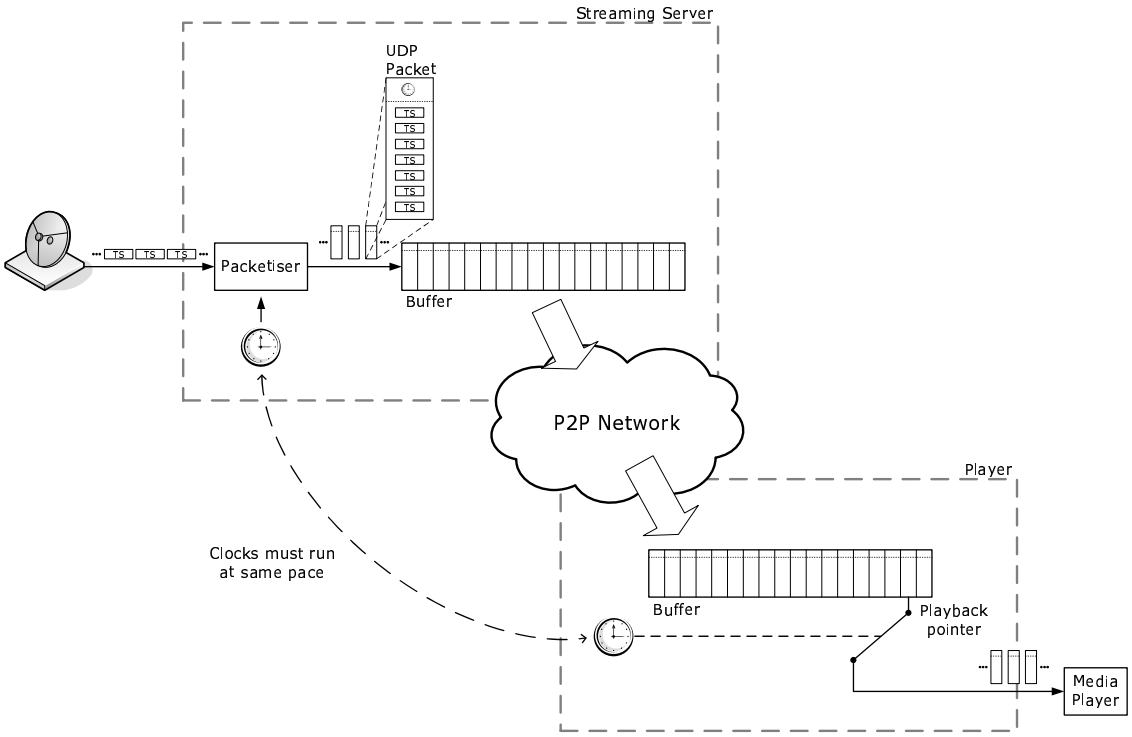


Figure 3.2: Packetising and stream synchronisation.

Algorithm 3.1 Streamer position averaging, to compensate for the `BufferTail` not being advancable beyond the `PlaybackPointer`

```
InstantPos := (PlaybackPointer - BufferTail) / buffer size;
if InstantPos < AveragePos then
  Alpha := 0.01
else
  Alpha := 0.001;
AveragePos := Alpha*Pos + (1-Alpha)*AveragePos;
```

minimal impact on existing other code) was to try to keep the playback pointer at a long-term average position of approximately 10% of the buffer's tail. This 10% offers sufficient decoupling between buffer tail and playback pointer, while still keeping most part of the buffer available for coping with temporary network problems etc. Note that the instantaneous position of the playback pointer can actually vary from 0% (i.e. at the tail) to several seconds ahead of the tail.

This approach works well in practice, but requires the position averaging operation to be asymmetric: the tail is never allowed to be advanced beyond the playback pointer, which means the instantaneous position of the playback pointer in the buffer can never be smaller than 0%. This would lead to the averaged position to be too high, causing the streamer clock to be slowed down, thereby further limiting the tail from advancing at the correct speed, etc. The averaging operation is therefore performed as shown in Algorithm 3.1.

As long as `AveragePos` remains within certain limits, the clock is set to run at its default speed, otherwise it is set to run faster or slower by 100 PPM. The values of `Alpha` and the speed change were empirically chosen to achieve proper following of server speed, and unnoticeable adjustment of the media player (i.e. not introducing visible or audible effects, e.g. because of audio resampling).

Note that the clock synchronisation could also be solved by using a mechanism like NTP, but the

'independent' tuning of the streamer clock is more flexible, in that it also allows to deliberately move to a different offset relative to the server's playback pointer. This may be useful for adapting to changing network dynamics: if more high-speed nodes join the network, it might be possible to (slowly) decrease the absolute playback delay.

Whereas a system like NTP is not required for keeping the video playback up-to-speed with the video source, it is still desirable to also synchronise the operating system's clocks with such a mechanism, as 'absolute' time references are used to validate expiration of cryptographic certificates, correct display of the Electronic Program Guide (EPG), etc.

For the next version (Version 1.1) of the Iphion application, it is recommended (and accepted) to further improve the synchronisation by directly comparing the average position of the playback pointer to those of its parents. This breaks the mentioned circular dependency between the playback pointer and the tail, and removes the need for the asymmetric `Alpha` parameter (see also Section 3.2.4).

3.1.4 Additional enhancements

Next to improving zapping delay by 2–4 seconds, the streamer implementation improves on the original application as follows:

- providing a better means to decide on when to start sending packets to the media decoder (i.e. not just when the buffer starts 'sliding', but when enough packets near the desired playback position have been downloaded)
- providing better detection and handling of discontinuities in the video stream (i.e. missing packets): when too many packets are missing (not necessarily contiguously), the streamer is stopped, and a suitable new playback position is determined to restart playback
- the buffer can be advanced more aggressively, allowing to quickly respond to availability of new data (e.g. after temporary network problems)
- in the original application, when the difference between the head of one peer and that of others (possibly including that of the video source) became too large, the head was simply reset to a new, 'better' position. This immediately caused a discontinuity at the media decoder (because the tail was used to stream packets to it). Additionally, the implementation of the pointer relocation code was such that it cleared the current buffer completely, whereas usually, the amount of relocation was much smaller than the buffer size. The new code keeps this valuable data, but because the buffer is allowed to advance more aggressively, pointer relocations are rarely needed anymore (except in case of longer network failures etc.).

3.2 Architecture

As stated in Section 2.6, the simulator uses a different internal architecture than Version 1.0 of Iphion's application. Although this does not change protocol behaviour, it can greatly simplify existing code and new algorithm development, and paves the way for solving remaining implementational issues mentioned in Section 2.4.

3.2.1 Layers vs. blocks

The internal structure of Version 1.0 is shown in Figure 3.3 showing its layered approach.

Although layers work well for specifying network protocols, it does not fit well to (sub-)tasks the application has to fulfill. For example, the network, peer, and scheduling layers all need to maintain state about other peers, and sometimes a layer needs information about a peer that in fact belongs to another layer. To still keep a clean seperation between layers, this was originally solved by giving each layer its own list of peers, which had to be kept in sync with the list of peers of other layers. The NAT and crypto code also need information about other peers, and were therefore implemented in the peer layer, but conceptually belong to the network layer.

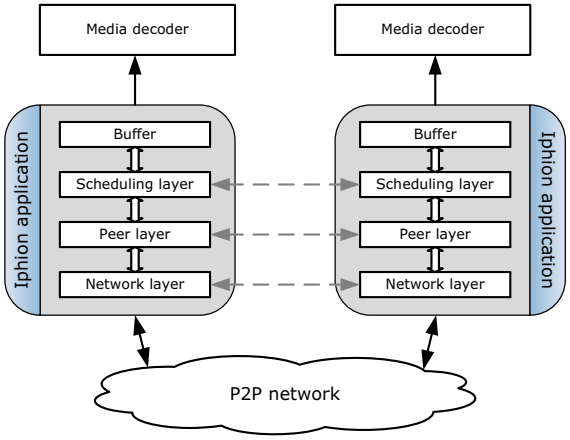


Figure 3.3: Visualisation of internal structure of the Iphion application at released Version 1.0. Note the layered approach, with 'virtual' communication channels.

Additionally, the scheduling layer performs tasks that have few dependencies on each other: code for downloading data, for sharing data, and for streaming to the media player is very distinct, with the buffer being their main connecting component.

Figure 3.4 depicts the new block-based architecture that resulted from these insights, clearly showing 3 main and seperate tasks (blocks) of an Iphion player: downloading, sharing and streaming, all connected through the central buffer. The downloading and sharing tasks both share the same lists of peers, which are also used by network code (for NAT traversal and cryptography).

The new architecture formed the basis of implementing the Iphion application in the simulator (Section 2.6), and is being adopted for the current Iphion application. Figure 3.4 was deliberately drawn 'layer-like', to demonstrate the relatively easy transition from the original architecture in Figure 3.3: essentially, the scheduling layer has been split up in *download*, *share* and *streamer* blocks, the peer layer was removed and its main functionality split between *peer database*, and *seed discovery*. The crypto code, which was scattered throughout the peer layer (and therefore very bug prone), is now a consistent 'sub layer' in the *network* block, as is NAT code (although these have not been drawn in the figure as the simulator does not have NAT and crypto functionality).

3.2.2 Partnerships

As noted in Section 2.4, partnerships in Version 1.0 are bi-directional, but as they can not be used for connections between nodes of the same rank, this connection is mostly used in one direction in practice. To solve this, and to allow algorithms of downloading and sharing to be decoupled, the simulator uses uni-directional partnerships. The terms for endpoints of such a connection are *seeder* and *leecher*: a seeder sends buffermaps and video data (if requested) to a leecher, a leecher sends requestmaps to a seeder.

3.2.3 Peer lists

The new architecture puts buffer and peer database (called *PeerDB*) in a central position, see Figure 3.4. Instead of keeping several copies of information about the same peer throughout the application, the *PeerDB* maintains one central record per peer (a *PeerItem*), containing information useful for all blocks in the application. This includes its (hierarchical simulator-)address, (virtual) IP address, estimated RTT, send/receive rates to that peer, list memberships, sent and received buffer-, request- and sentmaps, timestamps of the latest transmission and reception of every packet type, and timestamps of latest joins and leaves of lists.

To further simplify development of the P2P algorithm, the concept of *PeerLists* was introduced.

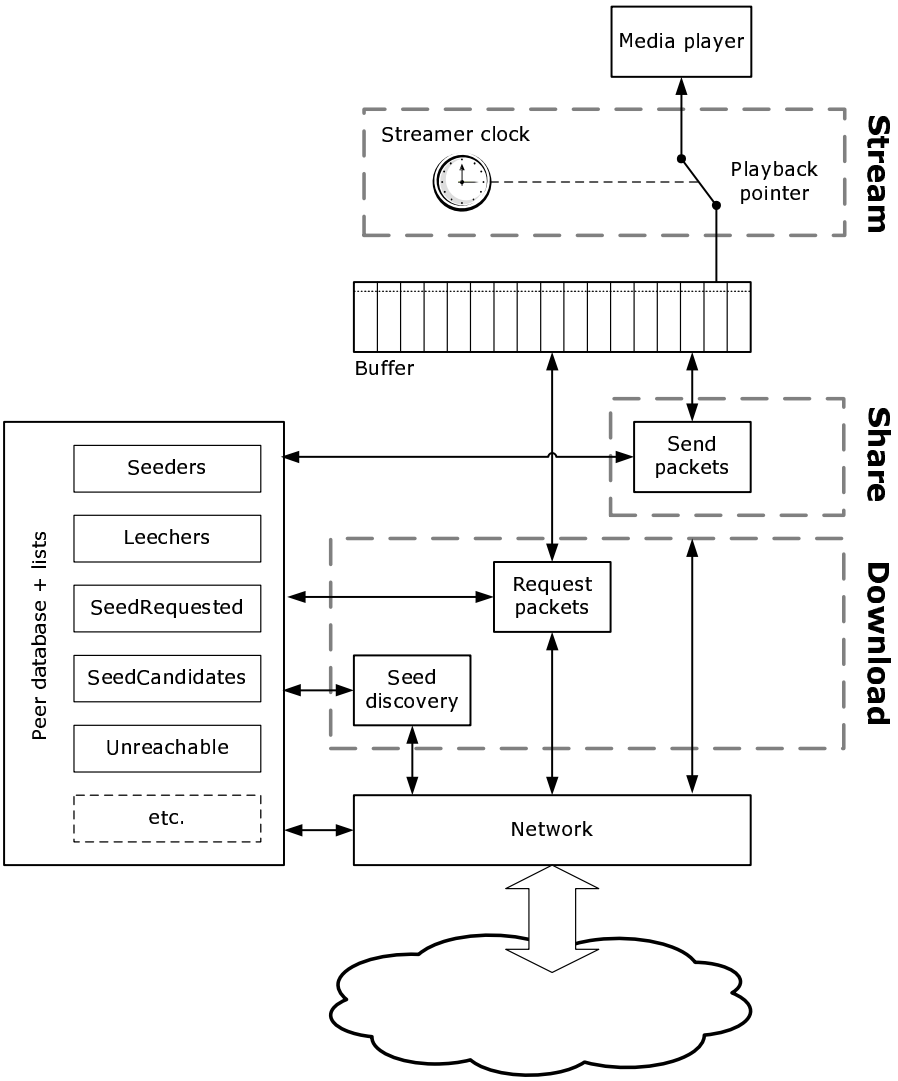


Figure 3.4: New architecture, as implemented in the simulator. Note the *peer database* and *buffer* as central connecting components, and the now separate functionality of *downloading*, *sharing* (uploading) and *streaming*. Compare this figure with the 'old' layered architecture in (Figure 3.3). Also note the concept of lists in / views on the peer database.

PeerLists contain a (reference counted) subset of *PeerItems* in the *PeerDB* (which is in fact implemented as a special type of *PeerList*), and provide properties and methods to automatically add or remove items from *PeerLists* based on incoming and outgoing packet types, timeouts of reception of certain packets and membership of other lists. For example, this is used to automatically add items to the *SeedRequested* list when a *SeedRequest* packet is sent and remove items from the *SeedRequested* list after reception of a reply, or a time-out of the request. Membership of this list is then used in the *SeedCandidates* view to prevent sending a request to a peer that is a member of *Seeder*, *SeedRequested*, *SeedRefused*, *Streamer* and/or *Unreachable* lists, thus keeping code to select the best candidate among remaining items clean and readable (for example, the sort function to find the best *SeedCandidate* is repeatedly given just two already valid candidates, and could simply return which candidate has better locality). Events can be defined to respond to membership changes, which is used to e.g. instantiate and destroy the socket endpoints for congestion-controlled flows upon entering and leaving the *Seeders* and *Leechers* lists.

Thus, this structure provides easy and automatic handling of elementary peer management, reducing the amount of complex if-statements scattered throughout all subalgorithms.

3.2.4 Buffer advancement

The buffer of Version 1.0 is smoothly advanced by comparing its head to that of other peers, and the playback pointer is synchronised to the tail (which has a fixed offset to the head). As stated when discussing the Streamer (Section 3.1), this leads to an awkward synchronisation algorithm.

In fact, the playback pointers of peers should be synchronised with eachother and with that of the relay server(s). Thus, the algorithm that now determines the position of the head (incorporating rank differences, and in the future probably RTTs etc. as well) should be moved to the Streamer.

Additionally, the simulator uses two 'heads': one is the original head of the buffer, which is now simply advanced when new packets arrive: there is no use in advancing the buffer prematurely, as this only discards packets at the tail that might still be useful for other peers. The new head is called "request head", and is set to the sequence number of the newest video packet advertised by any of the peers (but limited to stay within buffer size of the playback pointer).

The result is that the buffer is now only advanced when it is needed, thereby keeping older packets available longer, and even allowing to quickly advance the buffer after it has been held back by the playback pointer (remember that the tail is never allowed to be advanced beyond the playback pointer).

3.3 Locality

One of the desires of Iphion is to place relay servers (also called "repeaters") at strategic locations in the network, preferably inside ISP networks. In combination with a locality-aware protocol, this can greatly reduce the amount of inter-ISP traffic, but above all quality-of-service (QoS) of the players.

Figure 3.5(a) shows an example of a network consisting of three ISP networks, each with one relay server and 45 players (clients). In addition (not included in the picture), there is a streaming server attached to the root of the network, which provides video data to the relay servers, and is the central rendez-vous point (RVP) for all players. (In the simulator, the RVP and streaming server are implemented as one node, but this is not necessary in a real implementation.)

The depicted network can be instantiated in the simulator, with each ISP network having different bandwidth settings (e.g. 100 Mbit, 20480/2048 kbit, 1024/256 kbit end-host links). The links inside ISP networks are modeled as 10-Gbit links, with latencies of several milliseconds in order to simulate different locations across the country.

As explained in Section 2.1.6, some means to infer proximity of nodes without actively probing for e.g. latency is desirable. The proposed solution, as implemented in the simulator, is to direct nodes to their 'local' relay server inside the ISP network, based on their IP address, and supply them with a list of IP subnets in order to detect which nodes are a member of the same ISP.

The protocol works as follows:

1. When a node wants to join a channel network, it contacts the central RVP for that channel.

2. The RVP uses the (external) IP address of the node to find one or more suitable relay server(s), and returns a list of IP ranges that belong to the ISP the user is connecting from, if this information is known. Otherwise, it just returns an empty IP-range list.

3. The node contacts its relay server(s) to obtain a list of suitable peers to partner with, and additionally immediately starts requesting the video stream from its relay(s).

4. The relay server returns a list of possible peers, which may partly contain random results (for robustness) and results sorted by 'network distance' to the requesting node.

5. The node can now use IP addresses in the peer list combined with the subnet list to decide on the best local candidates to connect to.

Calculation of network distance is performed in two steps:

1. A rough classification is made into "Same NAT", "Same Subnet", "Same ISP" and "World", in order of increasing distance. The "Same NAT" class is not used in the simulator (because it lacks any NAT simulation), but could be implemented by comparing the external IP addresses of two nodes if the NAT type of that node indicates it is indeed behind a NAT. The "Same Subnet" and "Same ISP" classes are detected by comparing IP addresses with each of the entries in the subnet list. Iff two IP addresses appear to be a subset of the same subnet, the "Same Subnet" class is assigned, and if both IP addresses match any item in the subnet list the "Same ISP" class is assigned. The "World" class is used for nodes that appear to be outside of the current ISP's network, or in case no subnet list was returned by the RVP.

2. Within each class, a further refinement is made by an Exclusive-OR operation on the pair of IP addresses. The result of this operation is limited to a (configurable) maximum 'distance', based on the observation that although small differences in IP addresses usually correspond to small network distance, a large difference in IP addresses not necessarily means a large network distance. Therefore, if the IP distance is larger than e.g. 2^{10} (thus using the 10 last bits, out of 32), the distance will be clipped to this value.

Note that this scheme does not provide one simple 'distance value' per peer (because the IP distance of differing classes can not be directly compared), but instead allows comparisons: is node A closer to node B, than node C is to B?

The advantages of this approach are:

- Nodes can directly estimate network distance to candidate seeders, without first having to probe for e.g. latency (which is not necessarily a good indicator in any case). Note that probing takes time, but also does not scale well (many candidates will have to be examined).

- It is possible to compare distances between any two nodes, especially if those nodes reside within the same ISP network. This property is very useful for computing peer lists to send to other nodes.

- It is very simple to implement.

- It does not require any extra packets to be sent, and only increases the size of the reply to the channel join request by a small amount to transfer a list of ISP subnets.

- It scales well, because the RVP does not have to maintain any state about joining nodes, and can therefore simply be implemented as multiple physical machines, and the amount of information to store about ISP networks is small and proportional to the amount of relay servers in the network.

- It has the possibility for Iphion to direct groups of nodes to certain relay servers, which can, but do not have to, be located inside ISP networks.

Algorithm 3.2 Pseudocode of packet scheduling at receiver that prevents resending in-flight packets

```
BM := GetAllMissingPackets(); // returns a bitmap
BM := BM and (not PrevBM); // clear all previously requested bits
PrevBM := BM;
DivideOverSeedersAndSend(BM);
```

Algorithm 3.3 Corresponding sender side of Algorithm 3.2

```
// add new requests to existing requestmap
// (this can also happen during the
//  execution of the for-loop)
RM := RM or NewlyReceivedRequestmap;
// the sending loop
foreach b in 0..length(RM)-1 do
  if RM[b] = true then
  begin
    SendPacket(b);
    ClearBit(RM,b);
  end;
```

Work on a SeederCandidate-selection algorithm had just been started, when it was decided that the congestion control needs of the application were going to be of greater importance. Therefore, detailed simulation results and a more robust implementation that would e.g. also prevent isolated clusters, are not available.

However, the graphs in Figure 3.5 already show some promising preliminary results on the simulated topology (a): first typical partnering of most P2P applications (b), and the clustering effect of applying ISP selection (c), which is further optimised by incorporating the IP distance metric (d) as well. (Note: the clustering shown here may still suffer from forming isolated clusters, as mentioned in Section 2.1.5).

3.4 Packet scheduling

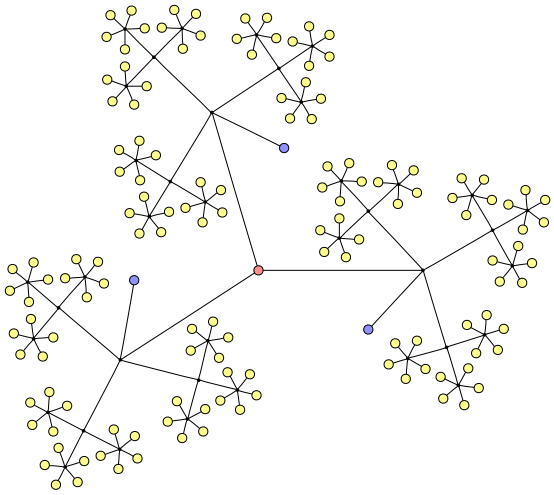
To prevent resending in-flight packets as described in Section 2.7, two general solutions are possible: let senders remember what packets have been sent already and 'subtract' these from incoming requestmaps, or let receivers remember what packets they have requested in previous rounds and only request new ones (which are then OR'ed with already received requests in the sender).

The first approach is implemented in the simulator, whereas the second is implemented in the real application (after Version 1.0), because it more easily fits the existing code. The functionality of both approaches is almost the same, with the first being more robust to lost requestmaps and the second yielding smaller requestmaps (after compressing). The latter approach will be discussed here, and is shown in pseudocode in algorithms 3.2 and 3.3.

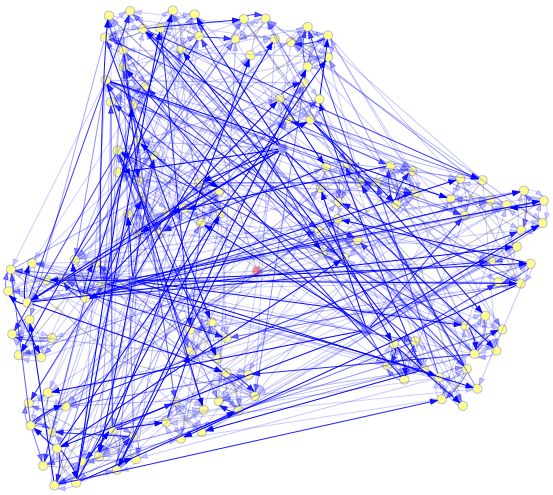
Although in-flight packets will now be sent once, this means that lost packets cannot be requested again. To prevent this, the currently implemented algorithm in the Iphion application does not remember requestmaps indefinately (as it should), but only the last three sent requestmaps. Note that this only prevents resending up to a limited RTT, but simply increasing the amount of remembered requestmaps makes re-requesting lost packets proportionally more slow.

Thus, a better way of detecting lost packets must be implemented, allowing to use the very simple 'infinite' requestmap memory (which still slides, as the buffer does, so is of fixed size) and simply resetting bits in it whenever a lost packet is detected.

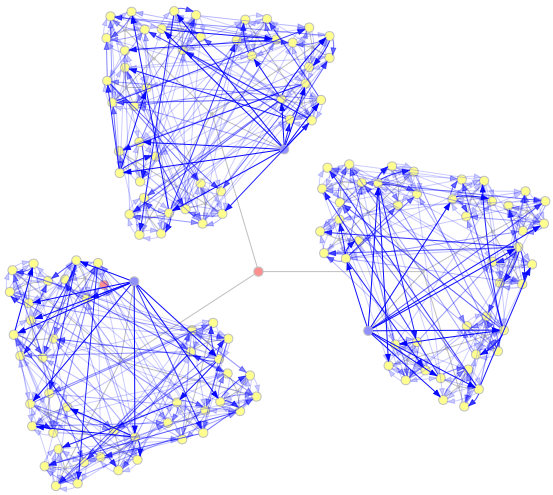
Detection of lost packets is not trivial, and requires quite some administration/communication, or some receiver assumptions about expected send order. The latter approach is proposed (but not yet



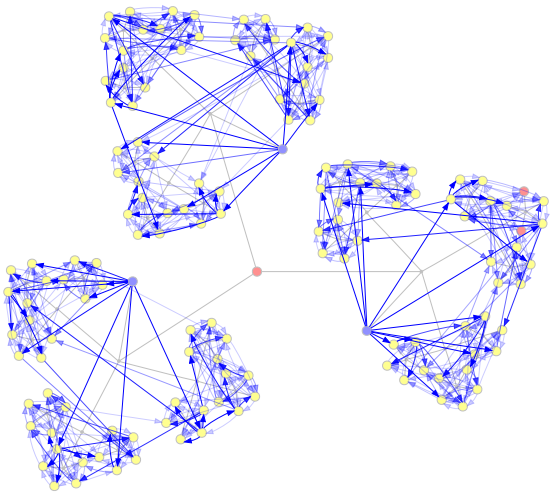
(a) Simulated Iphion network with 135 players. Note the three ISP networks, each with a relay server. The streaming server has not been depicted, but is connected to the root node.



(b) Locality algorithm disabled



(c) Relay assignment based on ISP, network distance still disabled



(d) Relay assignment and IP distance enabled

Figure 3.5: The effect of applying the proposed locality algorithm on seeding relations between peers. Partnerships are shown as arrow-headed lines overlayed on the underlying physical network (a). The opacity of the line denotes the amount of video data flowing over that connection, with fully opaque meaning one or more times the bandwidth of the video stream. Note that (b) corresponds to most P2P networks. (d) might be susceptible to forming isolated clusters, so the ultimate solution will likely be between (c) and (d).

implemented in simulator nor real application):

- The receiver assigns each requestmap (computed using the previous algorithm) a unique number, and remembers the last few requestmaps.
- The sender echoes this number back with each sent video packet.
- Assuming that the sender sends requested packets in stream order, the receiver can now (using its requestmap memory) determine the expected packet-arrival order.

Note that packets can be re-ordered in the network, so the receiver should not immediately react to a seemingly missing packet. Also, although the sender will send packets chronologically while processing one requestmap, the receiver could request packets in non-chronological order (e.g. to recover from packet loss).

This algorithm prevents unnecessary resending of packets due to (arbitrarily) high RTTs (from 300% to 0% packet duplication in case of Section 2.7), and still allows to re-request missing packets (and to detect them quickly). The quick detection of missing packets also handles lost requestmaps. Note that the complexity of the algorithm is mainly at the receiver side, which is beneficial for implementation in relay servers.

4 Congestion control improvements

Section 2.7 showed that packet scheduling and congestion control should be improved before the application is capable of (efficient) P2P communication on typical cable/xDSL networks. Packet scheduling is discussed in Section 3.4, and this chapter describes the results of finding a suitable congestion-control algorithm.

First, the requirements of Iphion's congestion control are defined, leading to the selection of an algorithm (TFRC) in Section 4.2. This algorithm is introduced in Section 4.3, after which its performance is investigated in Section 4.4. It will become clear that TFRC does not satisfy Iphion's requirements, and in fact exhibits similar queuing delays as Iphion's existing rate control.

Section 4.5 then describes attempts to improve upon TFRC, by limiting its maximum RTT. Although the modifications do improve on this aspect, they are probably only useful for the Iphion application, and require three parameters to be configured.

However, the modifications expose an underlying, more fundamental problem in TFRC, which leads to the recommendation of not using TFRC at all, but the congestion-control part of FreeBSD's TCP implementation as the algorithm of choice for Iphion (and possibly other UDP-based applications as well). Although extensive testing of this algorithm was not possible due to time constraints, a proof-of-concept simulation is included in the last section, Section 4.6.

4.1 Requirements

Before incorporating congestion control in Iphion's application, it is useful to first define when to use and when not to use congestion control. Next, two types of data flows are discovered, which lead to different sets of requirements on congestion control.

4.1.1 When to use congestion control

Congestion control is only needed for flows that carry a significant amount of data, which means that most packet types do not need to be congestion controlled. In general, it safe to assume that only the video packets need to be congestion controlled, and all other packets can be sent directly as they occupy a negligible amount of bandwidth compared to the video flows.

Even when only video packets will be congestion controlled, two distinct types of flows can be recognised: Relay-to-Peer flows and Peer-to-Peer flows, each with different characteristics and requirements.

In the case of Relay-to-Peer flows, the bottleneck will likely be the receiving peer's downlink, but sending data down this link as fast as possible is beneficial for that peer as it allows e.g. faster zapping.

With Peer-to-Peer flows, the bottleneck is likely going to be the uplink of the sending peer, which does not have any direct gain by sending more data (apart from e.g. earning a higher incentive score, Section 2.1.9).

4.1.2 Generic requirements

Generic requirements for a congestion-control algorithm are the ability to saturate the bottleneck link if possible, while still remaining responsive both in terms of latency (i.e. without unneccessarily filling queues) and reaction to congestion. It should also achieve link saturation as fast as possible, without causing excessive packet losses or delays.

Note that especially the latency requirements will later appear to be less obvious than one might think.

4.1.3 Iphion specific requirements

Example In order to derive the general principles for Iphion's congestion control, consider the following use-case:

User Adam is watching television via the Iphion player. Bob is also watching the same channel, but in a different home. Suppose Adam's Player is the P2P parent of Bob's Player. Now, as long as no-one else in Adam's home is trying to use the Internet, it is no problem to use all of Adam's uplink to send data to Bob. However, when Alice, also in Adam's home, wants to upload her photo album, it should be possible for her to do so, maybe even suppressing the flow from Adam to Bob, as Adam nor Alice really care about Bob's TV. On the other hand, suppose that Alice starts to download large amounts of data. This time, Alice's flow could cause Adam's stream to be disrupted if it were to compete fair with the other flows on the downlink. Although zapping may become slower on a busy downlink, watching TV should always remain possible to prevent people from complaining to Iphion.

Peer-to-Peer flows From this example, it is clear that sending data to other peers has different requirements than receiving data, but Peer-to-Peer flows are used for both.

Because it is practically impossible to detect whether this traffic competes at the sending or receiving side, and the sending side is likely to be the bottleneck, it is recommended to make this type of flow 'as friendly as possible' to competing traffic, allowing users to use their uplinks if desired.

Especially these uplinks can cause the large queuing delays of Section 2.5, which makes them the most challenging type of congestion-controlled flow.

Relay-to-Peer flows In order to guarantee continuous stream playback, at least one time the stream bitrate must be received, and likely somewhat more to compensate for packet loss and to keep the buffer sufficiently filled at all times.

It is therefore recommended to allow peers to download this amount of bandwidth from the relay server *without* any form of congestion control, as it is assumed that the downlink of the user has already been verified to be able to support at least this speed. This way, watching TV should always be possible: competing flows (like webbrowsing, filesharing, etc.) will be forced to only use the remaining bandwidth.

To speed up pre-buffering after zapping, peers can request additional bandwidth from their relay server (and other peers), which then *will* use congestion control.

The advantages of this approach are:

1. Users are always capable of watching TV, even if other users are using the same downlink.
2. Because (friendly) congestion control is used for receiving additional bandwidth, users are encouraged to offload their links as much as possible in order to achieve shorter zapping delays.
3. Sending data without congestion control has lower computational complexity, which is a useful feature for relay servers: it would be feasible to only use 'idle' CPU power for congestion-controlled flows, thereby allowing as many peers as possible to at least watch TV, albeit with longer zapping delays during peak times.

Note that it may be possible to use the same congestion-control algorithm for both the Peer-to-Peer and (congestion-controlled part of the) Relay-to-Peer flows, although especially the latter could benefit from modifications to further speed up zapping by e.g. having the slowstart phase already start at a 'known possible' high rate. In fact, this was one of the ideas that were planned to be investigated, but due to difficulties with the TFRC congestion-control algorithm (see the following sections), this has not yet been done.

4.2 Algorithm choice

Now that Iphion's congestion control needs are defined, an algorithm can be selected. As stated, the most challenging case is that of Peer-to-Peer flows on slow uplinks with large queues, so focus on

performance evaluation will be to adequately support such links.

Research on congestion-control algorithms consistently shows that design of such algorithms and modifications thereof is very complex, because of the wide variety of circumstances under which the algorithm should perform. Therefore, it is wise to chose an existing, well-known algorithm.

Many different congestion-control algorithms exist today, of which the congestion control of TCP is probably the most well-known. Note that TCP's reliable and in-order delivery is not suited for live video streaming, but its congestion-control principles might still be relevant. Also note that nowadays there is no such thing as "the TCP congestion control" anymore: many different variants have been proposed since its inception in the 1980's, some still following original principles, others with completely different approaches (e.g. TCP Tahoe vs. TCP Vegas). It is argued however, that the congestion control of TCP is not suitable for multimedia streaming because of its aggresive behaviour, which means that it indeed reacts faster to newly available bandwidth, but also quickly responds to congestion. The latter means that e.g. a video stream could suddenly be faced with a substantial reduction in amount of incoming data, which may lead to a buffer underflow.

To help remedy this, *TCP-Friendly Rate Control* (TFRC) was devised: an equation-based congestion-control algorithm, aimed at providing more smooth response to congestion. It enables the application to quickly take appropriate counter measures to the apparent reduction in available bandwidth, such as switching to a lower bitrate video/audio encoding (e.g. for teleconferencing applications), or connecting to more/different peers. An additional advantage of TFRC is its asymmetric structure: most of the computational complexity is at the receiver, which is beneficial when one relay server sends data to many players. The fact that TFRC requires feedback to be sent just once per RTT instead of for every packet (like with TCP) also helps to improve on scalability.

Because TFRC is specifically designed for multimedia, this algorithm is incorporated in a wide variety of multimedia applications, including some of the existing P2P streaming proposals (see Appendix A). A substantial amount of research has been performed on fairness towards competing TCP flows, performance on high bandwidth links, etc. Most of this work is done by simulation (e.g. with NS-2), or emulation (e.g. with Dummynet), and usually indicates promising results.

The 'famousness' of TFRC, recommendation by the Internet Engineering Taskforce (IETF), and its apparent applicability to multimedia streaming led to choosing TFRC as a candidate for Iphion's application.

4.3 Introduction to TFRC

This section briefly introduces TFRC's algorithm, as explained in RFC 3448 [36].

TFRC (TCP-Friendly Rate Control) is a so-called equation-based algorithm (as opposed to AIMD: Additive Increase Multiplicative Decrease). At its core is the "TCP rate equation", given by:

$$X = \frac{s}{R \cdot \sqrt{2 \cdot b \cdot p/3} + t_{RTO} \cdot 3 \cdot \sqrt{3 \cdot b \cdot p/8} \cdot p \cdot (1 + 32 \cdot p^2)}.$$

This equation computes the throughput (X) that a TCP flow would have, given the (constant) segment size (s) of the packets, the current RTT (R) and loss event rate (p), TCP's retransmission timeout (t_{RTO}) and the number of packets acknowledged by a single TCP acknowledgement (b). It can be simplified by taking $t_{RTO} = 4 \cdot R$ and $b = 1$, leading to:

$$X = \frac{s}{R \cdot \left(\sqrt{2 \cdot p/3} + 12 \cdot \sqrt{3 \cdot p/8} \cdot p \cdot (1 + 32 \cdot p^2)\right)}.$$

During TFRC's congestion avoidance phase (the 'steady state'), this equation is used to limit the send rate of the TFRC flow, based on what a TCP flow would have under the same conditions. Clearly, the loss rate p is the dominant control parameter in this equation.

The TFRC sender adds a sequence number, the sender's timestamp of transmission of the packet and the sender's estimate of the RTT to the data packets. The latest received timestamp is echoed by the receiver upon sending feedback, which allows the sender to estimate the RTT R.

The receiver also includes the current receive rate and the computed loss event rate in its feedback. To compute the loss event rate, the receiver first detects missing packets (allowing slight reordering of packets). The lost packets are then grouped, such that all losses that occur within one RTT of the first loss will all form one loss event (as they are likely caused by one queue overflow). The 'distance' (in sequence numbers) between such loss events determines the so-called "lossless intervals".

A default of 8 of such intervals is remembered, and a weighted average (with weights 1, 1, 1, 1, 0.8, 0.6, 0.4, 0.2) is applied to these intervals. This yields a "mean interval", of which the reciprocal is the loss rate p. Note that after registering a new loss interval, there will be a number of packets that are not lost. This constitutes the "current interval". The mean interval then includes the current interval and 7 of the previous intervals (or just the 8 older intervals, if including the current interval would yield a higher loss rate). See Figure 4.3(c) for an illustrative example of the interaction between the current and mean intervals (note that the mean interval sometimes stays constant, when not including the current interval).

Slowstart (for faster, usually exponential startup) is implemented by doubling the transmit rate every RTT, until the receiver detects the first loss. At that point, the loss event history is still empty, so the receiver needs to 'invent' a suitable value for the length of that first interval. The receiver tries to find a loss rate p, given R and using the current receive rate as X, by 'inverting' the TCP rate equation. The loss history is then initialised with one interval, with length $1/p$. Upon receiving feedback containing this non-zero loss rate, the slowstart phase is ended and the sender's rate equation should now return a value close to the apparent maximum receive rate so far.

An optional, but very useful addition to TFRC (also defined in the original RFC) is an RTT-based oscillation prevention. Based on the assumption that an increase in RTT is likely caused by queuing, the instantaneous send rate can be modified:

$$X_{inst} = X \cdot \frac{R_{sqmean}}{\sqrt{R}}$$

where R_{sqmean} is a moving average of the square root of the RTT R. For example, if the current RTT is twice as large as the 'nominal' RTT of the link, the send rate is reduced by a factor of 0.7.

4.4 TFRC performance

In view of the results of Section 2.5, it is useful to first test the behaviour of TFRC in an isolated case (i.e. not integrated in the Iphion application). TFRC was implemented in the simulator of this project, but because the results seemed worse than what was expected, the implementation has also been verified against the NS-2 simulator [51], which was used during the development of TFRC and many other Internet algorithms.

4.4.1 Test setup

To test the implementation of TFRC, the typical "dumbbell" setup of Figure 4.1 is used. The same type of topology is used in many protocol tests, including TFRC tests in NS-2. To be able to compare the results of both simulators, the same configuration for all links and queues has been used in both simulators, based on NS-2 test scripts that were used to produce the graphs in TFRC's announcement paper [30].

Each half of the bottleneck link has a speed of 15 Mbit and 20 ms latency, with drop-tail queues of 250 packets. The 100 Mbit links to the source nodes have latencies of 2 ms, the destination nodes have no latencies. The simulation script in NS-2 used an aritificial loss generator by default, to visualise the impact of several amounts of loss percentages on the loss intervals. This loss generator has been disabled to allow the queues to generate loss instead.

Both implementations use the oscillation-prevention algorithm as specified in the RFC [36], as this significantly improves TFRC's performance (compare figures 4.3 and 4.4).

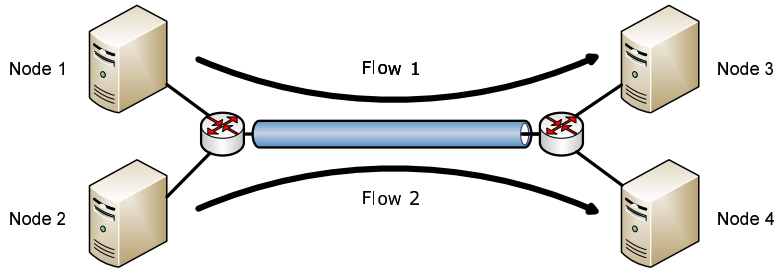
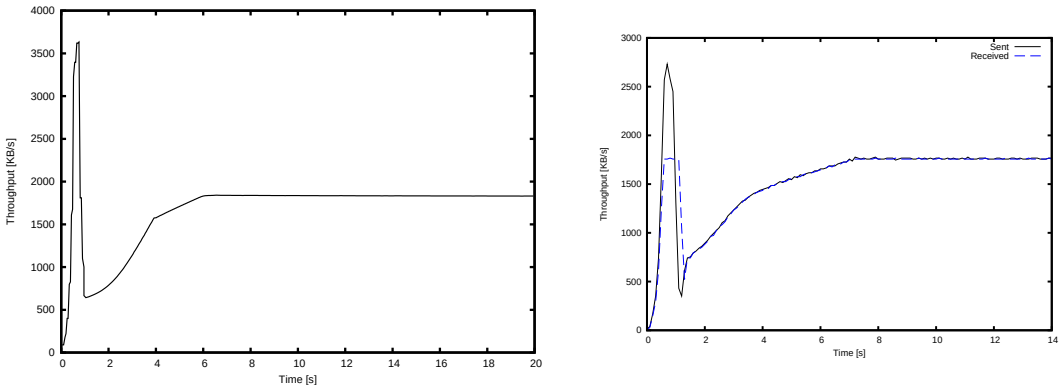


Figure 4.1: A typical "dumbbell topology" to test implementation of TFRC. The link between the routers is configured to be the bottleneck. One flow is created from Node 1 to Node 3, and optionally one from Node 2 to Node 4.



(a) NS-2

(b) Iphion-application simulator

Figure 4.2: Send rate as simulated by the two simulators, on the topology of Figure 4.1. Note the large initial slowstart burst, and subsequent 'restart' of the TFRC algorithm.

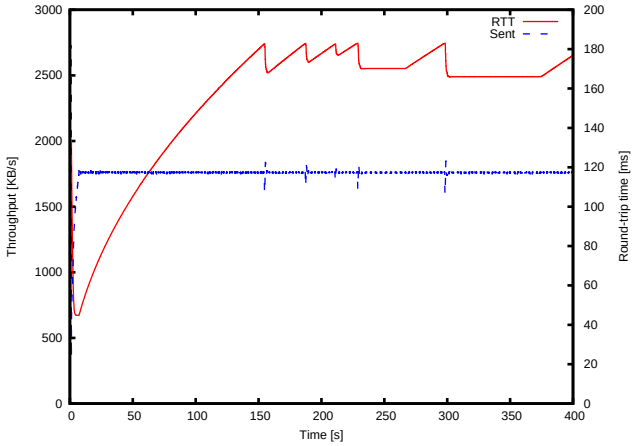
Other optional enhancements and experimental 'tweaks' have not been implemented in this project's simulator (and disabled in NS-2) because they were undocumented, did not visually impact NS-2 simulation results and/or are not implemented in DCCP's TFRC implementation (and are thus apparently irrelevant to a practical implementation). There was one (undocumented) parameter ("slow_increase_") in NS-2 which did result in different simulation results, but this has also been disabled as it should apparently improve on slowstart behaviour but in contrast just caused a strange extra throughput 'dip' in the initial slowstart burst.

4.4.2 Comparison between NS-2 and custom simulator

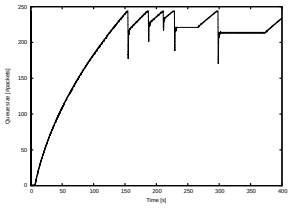
First, the behaviour of the send rate of both simulators is shown in Figure 4.2.

It is evident that both implementations exhibit the aggressive slowstart behaviour, but finally achieve link speed without significant oscillations in send rate. The small deviations between the simulations are caused by subtle differences in the definition of link speeds, which mainly leads to a different initialisation of the first few loss intervals.

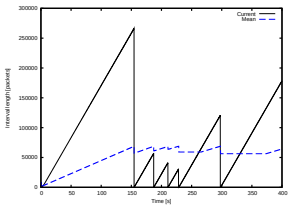
It is interesting to run the simulation for a longer time, to see the effect of loss on the TFRC loss intervals, queue utilisation and RTTs. From Figure 4.3, it can be seen that although the send rate already stabilises after a few seconds, the queue is far from saturated at that point, so no packet loss occurs. The already excellent control of the send rate in this stage is caused by the (optional) enhancement of TFRC which takes variations in RTT into account based on the observation that increasing RTTs are usually caused by queuing delays. This enhancement is recommended for situations with a low level of statistical multiplexing, which is the case with e.g. one flow on a the bottleneck. Because



(a) Sending rate (left axis) and RTT (right axis)



(b) Queue usage



(c) Current (most recent) and mean (weighted average) TFRC loss intervals

Figure 4.3: Results of extending the simulation of Figure 4.2. Note the already excellent control of the send rate even in the absence of losses, and the final saturation of the queue causing the loss-based rate adaptation.

of the significant improvement in smoothness and efficiency and the simplicity of the implementation, it might be wise to 'mandate' the enhancement for every TFRC implementation, instead of making it optional.

Note that the queue utilisation slowly increases until it is saturated, after which loss occurs. This marks the beginning of the documented behaviour of TFRC: rate control based on packet loss.

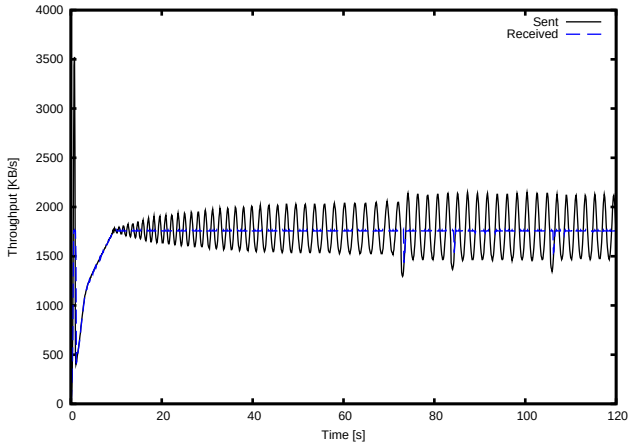
For completeness, Figure 4.4 illustrates the effect of disabling the oscillation prevention, clearly showing the highly oscillatory of both the send rate and queue utilisation. Note that in some cases the queue is actually fully empty, causing a slight drop in receive rate. The same highly oscillatory behaviour has been observed when testing an (old) 'real world' TFRC implementation (which did not yet include the oscillation prevention) on a real cable modem connection.

4.4.3 TFRC on cable modem links

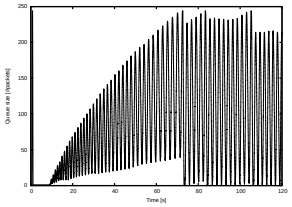
To test the performance of 'standard' TFRC on cable modem links (or more specifically: slow links with relatively large drop-tail queues), the same topology was used, but the bottleneck link was configured to have a speed of 32 kB/s from source to destination, and 100 kB/s on the return path (which is only used for the (small) feedback packets in this case). Queue sizes of both links were 200 packets, latencies were still set to 20 ms.

Note that especially the choice of the relatively large queue size is 'special': all examined research material uses 'conventional' choices for the buffer sizes, based on the rule-of-thumb of taking the bandwidth-delay product of the expected flow to set the queue size. This leads to skewed results, which do not correspond to reality in case of cable modem links.

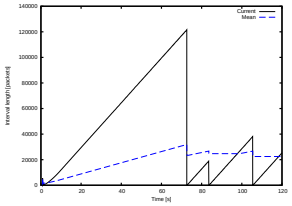
The results of this simulation are given in Figure 4.5, from which it can be seen that the slowstart phase lasts for 25 seconds, reaches an RTT of 4.5 seconds, never drops below 1 second and finally (after one hour, when the algorithm experiences its next loss event) even reaches more than 7 seconds. Note that this maximum delay depends on the packet size (1000 Bytes in this case), and will be larger when using larger packets, because the buffer is limited by a number of packets instead of a number of Bytes.



(a) Send and receive rates

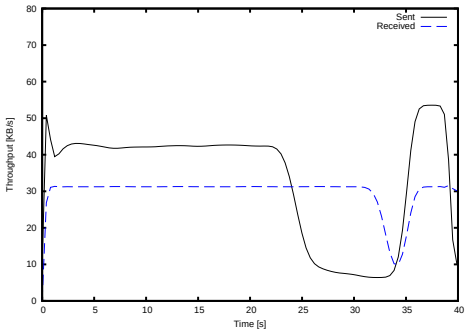


(b) Queue usage

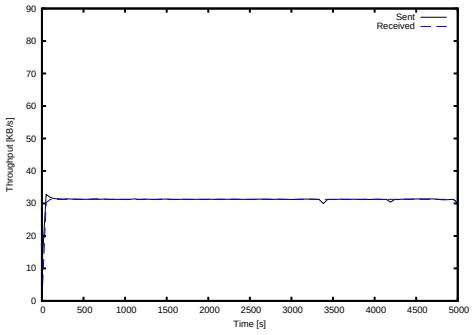


(c) Current and mean TFRC loss intervals

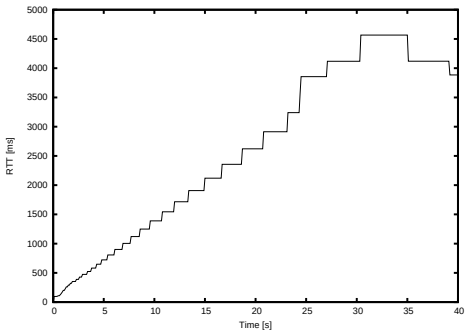
Figure 4.4: Behaviour of TFRC when the RTT-based oscillation prevention has been disabled.



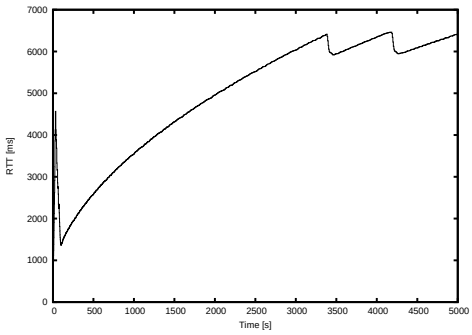
(a) Send/receive rates during first 40 seconds



(b) Send/receive rates of extended (>1.5 hour) simulation



(c) RTT (as measured by TFRC) during first 40 seconds



(d) RTT during extended (>1.5 hour) simulation

Figure 4.5: Standard TFRC behaviour on typical slow cable modem link. Note the slowstart phase lasts for 30 seconds, and the queue finally becomes fully saturated (after one hour), causing very large RTTs.

	Fast link	Slow link
Slowstart duration [s]	0.95	23.47
Slowstart loss [#packets]	132 (9.9%)	79 (8.3%)
Idle-link RTT [ms]	44	44
Minimum RTT [ms]	45	1361
Maximum RTT [ms]	183	6462

Table 4.1: Summary of important characteristics of standard TFRC on fast and slow links

4.4.4 Results

Two important results can be noted from these experiments:

1. Slowstart behaviour saturates the queue completely, causing much packet loss. This loss covers multiple RTTs (as measured by the receiver), yielding multiple small intervals. On the fast link, this causes the algorithm to restart bandwidth exploration almost from scratch, and it might have been better to just skip the slowstart phase in this case. In fact, this behaviour is not much different from the mean-to-network mode of Iphion's application (which also fills the queue completely).

2. Although the congestion avoidance phase of TFRC performs as specified, it is shown that the bottleneck queue eventually saturates completely again. For the cable link, this yields unnecessarily large RTTs.

Especially the large RTTs are problematic considering Iphion's desire for sub-second zapping delays. Apparently, 'just' using congestion control does not necessarily improve on RTTs and because of the long slowstart phase on the slow link, a considerable amount of packet loss occurs as well.

This leads to the intermediate conclusion that, contrary to popular belief, TFRC as-is might not be the obvious candidate for Iphion's congestion-control algorithm. In the following sections, some modifications to TFRC are proposed and simulated to improve TFRC's behaviour on these 'challenging' links.

Note that Iphion's usage of TFRC can be very different from that of other's: if TFRC is used for e.g. Internet telephony applications, it is likely that a flow will be limited by the amount of available data (also during slowstart, as voice data is just starting to become available when a connection is established), instead of the speed of a user's link. Although one could argue the same for the Iphion case (the stream bitrate should be lower than the user's downlink allows), this does not hold: during initial pre-buffering there will already be enough data to (temporarily) saturate a user's downlink, but especially a user's uplink is easily saturated when it uploads data to other peers.

4.5 Improving TFRC

One of the requirements of Iphion's congestion control is that it should try to keep responsive, both during slowstart and congestion avoidance (the 'steady state'). As has been shown in the previous section, TFRC does not live up to this requirement if the connection is not data-limited (i.e. has more data to send than the bottleneck link can handle). To keep relay servers available for newly joining peers, handling sudden network problems and save on bandwidth costs, it is wise to maximally utilise the available Peer-to-Peer bandwidth, and only resort to Relay-to-Peer bandwidth if Peer-to-Peer bandwidth is not sufficient. This means that uplinks of peers are likely to be fully saturated at all times, making queue saturation of TFRC a serious issue.

Because of the apparent advantages of using TFRC as the congestion control for Iphion, means to improve its default behaviour have been investigated. The results of these efforts are presented in the following subsections.

4.5.1 Constraints on algorithm modifications

Designing, but also modifying congestion-control algorithms is a highly complex task. The reason is the wide variety of circumstances that these algorithms generally meet: link speeds varying from tens of kilobytes to tens of megabytes per second, idle-link RTTs ranging from 0.1 ms to tens of milliseconds, queuing delays from zero to several seconds, zero or more cross flows, which may be TCP-like, or even unresponsive UDP. Additionally, link conditions might change, not only due to changing cross traffic, but also e.g. wireless links changing speeds.

Given these challenges, it is hard to guarantee that a modification to TFRC that improves one situation, will not have detrimental effects in other situations. An important constraint on the modification is therefore that the algorithm should only become 'more friendly' to its environment. Note that this choice means that the modified TFRC might not be an all-round solution anymore: in case the algorithm becomes too friendly in a certain scenario, its throughput might be fully suppressed. For Iphion however, the (non-congestion-controlled part of the) relay servers still guarantee proper operation in this case, and Iphion flows suppressing each other would not decrease the global bandwidth efficiency of the network.

4.5.2 Modifying TFRC's loss history

Basically two main strategies then remain for implementing such modifications: one is to create an extra limitation on the send rate, the other is to influence the loss event history.

For the first approach, one could think of inserting an extra equation into the system, and taking the minimum of the send rates given by the two already existing equations (which are based on equivalent TCP rate and on twice the receive rate) and the new equation, which could be based on e.g. latency. This approach means that the 'normal' mechanism of TFRC is effectively disabled while the modified system is in effect. This in turn leads to a receiver not experiencing losses anymore, which could lead to overly aggressive behaviour when e.g. the link suddenly becomes congested. (Remember that the algorithm is mainly used to protect a user's uplink, and 'too much' friendliness can be considered a feature.)

For the second approach, it is possible to introduce artificial losses into the loss history, which leads to a higher loss rate fed back to the sender, decreasing the send rate. This approach has the advantage that it will create a loss history that is representative for the current state of the link, making the algorithm more responsive to sudden congestion.

Because of the desire to make Peer-to-Peer flows more friendly (maybe even only using idle bandwidth), the second method was chosen for the TFRC modifications.

4.5.3 RTT limiting TFRC

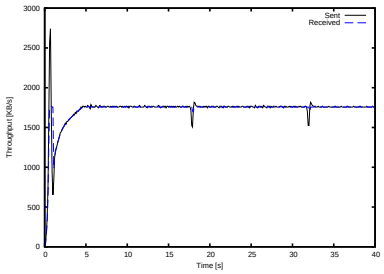
Although TFRC's large throughput dip after slowstart is not ideal, the main problem is its tendency to completely fill queues, causing large queuing delays (during both slowstart and congestion avoidance). Therefore, focus will be on limiting these delays.

As determined in the previous subsection, modification of TFRC will be done by introducing artificial loss events into TFRC's loss history. Note that this is a receiver-based approach, which does not require modifications to the protocol itself. Also, the artificial loss still allows the received packet to be used and can be compared to marking packets with ECN (Early Congestion Notification): the packet is not actually discarded, but should just be considered that way by the congestion-control algorithm.

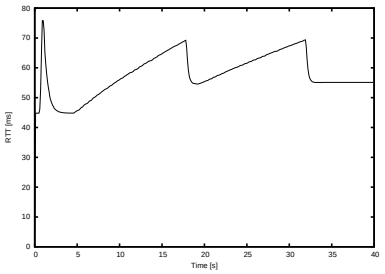
Although in fact the queuing delay itself should be minimised, it is hard to determine this delay: one would think it is possible to simply subtract the idle-link RTT (often called the "base RTT") from the current RTT, but computing the base RTT cannot be done reliably as e.g. new competing flows will not 'see' the link in its idle state, and the idle-link RTT can change due to route changes, wireless links changing speeds, etc. For example, simulation showed that a new flow will initialise its base RTT with the current RTT of the link (which was the 'real' idle-link RTT plus the configured maximum queuing delay of the first flow), and simply assumes that no queuing delay is yet present, thus sending more

Algorithm 4.1 Simply limiting the maximum RTT that TFRC is allowed to use, by introducing artificial packet loss in the loss history before real losses would occur (when the queue overflows).

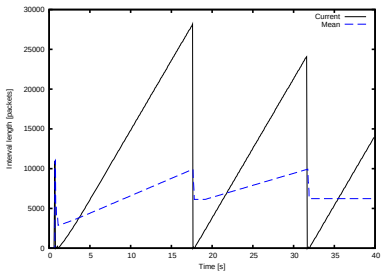
```
if CurrentRTT > MaxRTT then
  MarkThisPacketAsLost();
```



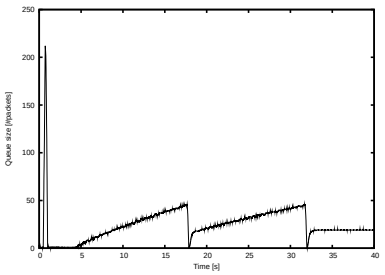
(a) Send and receive rates



(b) Round-trip time (as measured by TFRC)



(c) TFRC loss intervals



(d) Queue usage

Figure 4.6: Simple RTT limiting of TFRC on a fast link by introducing artificial losses in the history when the RTT exceeds 70 ms.

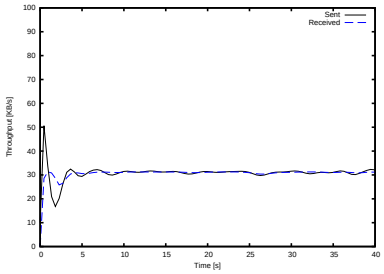
data up to its `BaseRTT+MaxqueuingDelay` and never discovering the 'real' idle-link RTT. Note that concerns on this principle of limiting queuing delays is also preventing wide acceptance of TCP Vegas.

Therefore, the choice was made to simply mark a packet as lost as soon as the RTT rises above a certain configurable limit (Algorithm 4.1). This method already works surprisingly well, as can be seen from the plots in figures 4.6 and 4.7, where the RTT limit has been set to 70 ms on the fast link (the link from the NS-2 simulations, 1800 kB/s, 20 ms, 250 packets queue), and 160 ms on the slow link (32 kB/s, 20 ms, 200 packets queue). Note that this simple condition will actually mark many packets in a row as long as the RTT stays above the limit, but usually only the first event actually causes a new entry in the loss history, with the following marks still being within one RTT of the first.

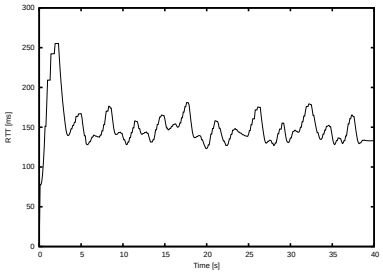
Note the queue usage, which clearly does not reach its limit anymore. The 'glitch wave' of the send rate when a loss event is generated can be explained by the response of the oscillation-prevention algorithm: first, the loss rate increases so the send rate decreases. Then, the RTT decreases because the queue drains, which causes the multiplication factor in the oscillation prevention to increase, causing the 'overshoot'.

4.5.4 Improving slowstart

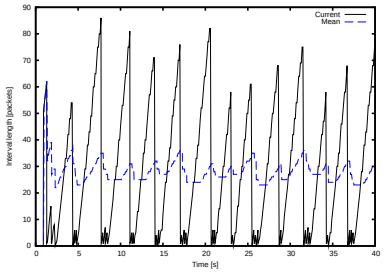
From these plots, it is clear that the slowstart behaviour is still not optimal (note the large throughput decrease), which can partly be explained from a closer inspection of the loss intervals during the first



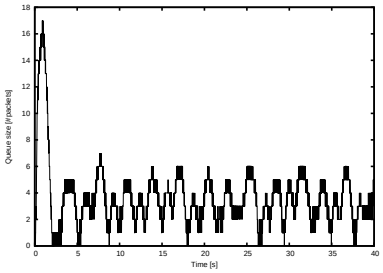
(a) Send and receive rates



(b) Round-trip time (as measured by TFRC)



(c) TFRC loss intervals



(d) Queue usage

Figure 4.7: Simple RTT limiting of TFRC on a slow link by introducing artificial losses in the history when the RTT exceeds 160ms (compare these results to Figure 4.5).

few seconds: in Figure 4.6, the mean interval decreases multiple times after the first loss, because all data buffered in the queue by the slowstart burst takes multiple RTTs to drain, which causes the RTT marking to extend to multiple loss intervals. The same problem can occur when e.g. a constant bitrate UDP flow is started, which can cause a sudden step in the RTT lasting for multiple RTTs. Additionally, the previous algorithm causes irregular loss intervals: a very long interval followed by a number of very short intervals, etc (see the loss intervals of Figure 4.7).

To prevent these repeated penalties, an extra condition was added to the RTT limit. This condition is based on the length of the mean interval, such that the RTT limiting is disabled for a longer number of packets if the mean interval is large, and vice-versa (Algorithm 4.2).

In this algorithm, `alpha` determines the response of the algorithm to sudden changes in RTT. Obviously, if `alpha` is zero, just the RTT limit remains. If `alpha`$\geq$1, the induced losses can never cause the mean interval length to decrease, so the RTT can neither decrease. The value of `alpha` is not very critical: experiments showed that any value between around 0.2 and 0.9 gives fairly good results. The value should be chosen as small as possible to have sufficiently fast response to increasing

Algorithm 4.2 Improved RTT limiting by inhibiting losses to prevent repeated loss events after large RTT changes (most notably during slowstart)

```
if (CurrentRTT > MaxRTT) and
   (CurrentSequenceNumber > MinNextSequenceNumber) then
begin
  MarkThisPacketAsLost();
  MinNextSequenceNumber := CurrentSequenceNumber + alpha*MeanInterval;
end;
```

RTTs, without generating too much repetitive losses, especially during slowstart. The value `alpha=0.5` was chosen as the default for its stabilising effect on slowstart, with still fairly good response to RTT increases.

Note that RTT limiting plays an important role in improving slowstart behaviour: not only does it prevent packet losses when the queue finally overflows, it also prevents the RTT from deviating too much from its final value, which is used to initialise the loss history:

A closer inspection of the first few seconds of the send rate reveals that in the original TFRC, the main cause of the dropped send rate is indeed the very large loss rate caused by repeated loss events spanning multiple RTTs. However, with an experimental algorithm that just prevented these extra losses, the send rate still showed a similar drop after slowstart. This was caused by the TCP rate equation limiting the send rate, but as loss rate should now correspond to link speed at the time of marking the first packet (by the initialisation of the loss history), this rate should have been close to the maximum receive rate. It appeared that the problem was caused by a relatively large difference in RTT as used by receiver and sender during slowstart. This RTT is measured by the sender based on receiver feedback, and then sent to the receiver again, which introduces lag in these measurements. At the time of marking the first packet in the receiver, the receiver uses its (relatively low) estimate of the RTT when inverting the rate equation initialising the loss history. When the sender then receives this loss rate and ends slowstart, the RTT measurements will still indicate increasing RTTs, causing the computed rate to be lower than what the receiver had intended.

With some combinations of link speeds, queue sizes, link RTTs, RTT limits etc. (most notably on cable-like links) the RTT limiter 'activates' before the queue is actually saturated. In that case, slowstart behaviour is fairly optimal. Sometimes, even small changes in the link configurations cause the queue to become saturated before the limiter activates, causing the 'after slowstartdip' again. This partly illustrates that great care must be taken when interpreting congestion-control results, sometimes apparently small changes to parameters can have great consequences for algorithm behaviour. Therefore, during development of the algorithms, several different configuration settings have been simulated as well (but are not discussed when irrelevant).

4.5.5 Multiple flows

TFRC's behaviour was now sufficiently improved (considering Iphion's delay requirements). However, these improvements only work correctly if flows sharing a bottleneck link have (nearly) the same idle-link RTT.

Clearly, the flow with the shortest idle-link RTT will take up all available bandwidth, leaving all other flows at an almost zero rate. But even flows having exactly the same idle-link RTT suffer from very slow startup, as slowstart is effectively disabled right from the start, and the RTT limiter limits the new flow not only because of its own increase in send rate, but also because of the 'exploring' competing flows.

To solve these problems, a simple addition to the RTT limiter was made: by giving relatively slow flows some extra 'RTT room' (the `RTTMargin`), the new (slower) flows will be limited more than existing (faster) flows, until both reach approximately the same speed (in case of equal idle-link RTTs). This already solves the slow startup problems. With a careful choice of the `RTTMargin`, the problem of differing idle-link RTTs can also (partly) be alleviated, as it is assumed that in the Iphion scenario hosts will likely be relatively close, with idle-link RTTs differing by at most a few tens of milliseconds. As long as differences stay within the `RTTMargin`, all flows will receive a part of the available bandwidth, relative to their RTT difference ratio's.

This leads to the code as given in Algorithm 4.3, which simply assigns every flow with a receive rate below `ExpectedRate` a proportionate fraction of the `RTTMargin`. On links (much) faster than `ExpectedRate`, the flow with the smallest idle RTT will receive the bulk of available bandwidth, but other flows will at least receive a fraction of `ExpectedRate`, and as `ExpectedRate` should be set to the video bitrate (see the next subsection), a single flow will become data-limited to below this rate after pre-buffering anyway. Additionally, the exact distribution of bandwidth between peers does not influence the amount of data that the relay servers have to provide.

Algorithm 4.3 Adding the `RTTMargin` to the RTT limiter to allow flows with differing RTTs to co-exists, and improve the startup behaviour of competing flows.

```
ExtraRTT := RTTMargin * Max(0, 1 - CurrentReceiveRate/ExpectedRate );
CurrentMaxRTT := MaxRTT + ExtraRTT;
if (CurrentRTT > CurrentMaxRTT) and
   (CurrentSequenceNumber > MinNextSequenceNumber) then
begin
  MarkThisPacketAsLost();
  MinNextSequenceNumber := CurrentSequenceNumber + alpha*MeanInterval;
end;
```

	Default value
`alpha`	0.5
`MaxRTT` [ms]	150
`RTTMargin` [ms]	70
`ExpectedRate` [kB/s]	100

Table 4.2: Suggested default values for improved TFRC parameters

4.5.6 Choosing the parameters

The modifications to TFRC as presented in this section require three main parameters to be chosen: `MaxRTT`, `RTTMargin` and `ExpectedRate` (leaving `alpha` at its default of 0.5), as summarised in table 4.2.

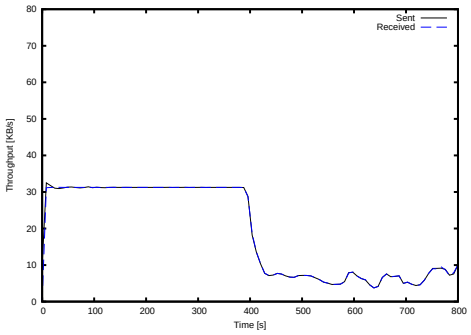
It is important that all peers use the same settings to allow flows to compete fairly. This means that the settings must be defined such that they apply to a broad range of link characteristics. Note that `MaxRTT` and `RTTMargin` depend mainly on characteristics of the links, whereas `ExpectedRate` depends on both link characteristics and stream bitrate: it is evident that a higher stream bitrate suggest a higher `ExpectedRate`, but if the 'minimum' supported uplink is just a small fraction of that speed, it may be better to set `ExpectedRate` to just a few times the speed of such slow links.

`MaxRTT` should be chosen small enough to have low RTTs in general and to try to improve slowstart on fast links, but should not be chosen too low, as slow cable links easily 'generate' RTTs of more than 100 ms even with just a few packets in their queue. Simulations suggest that a value of at least 150 ms should be used.

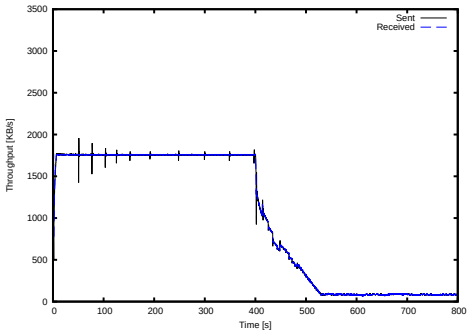
`RTTMargin` must be chosen small enough to limit the maximum RTT (`CurrentMaxRTT`), but large enough to allow new flows to quickly reach their share of bandwidth, and to partly compensate for differences in idle-link RTTs. A value of 80 ms was used in simulations, which should be large enough to cover typical RTT differences between peers, given the desire to communicate with nearby peers.

`ExpectedRate` was set to 100 kB/s to match the targeted 1 Mbit/s (VBR) H.264 stream bitrate.

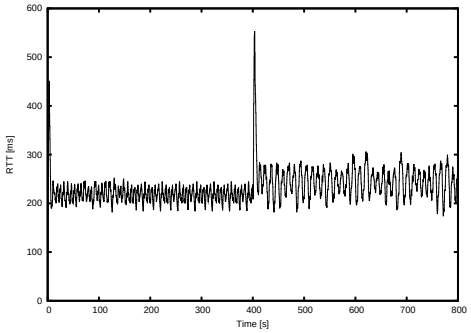
Simulation results using these settings are shown in Figure 4.8, where a second TFRC flow was turned on at $t = 400$ s, and the first flow having a 20 ms higher RTT. As can be seen, the flow still gets a portion of `ExpectedRate` on both fast and slow links. The fast link scenario may not seem realistic in this case, because the flow will become data-limited quite soon after pre-buffering, in which case both flows can simply transmit at the stream bitrate without saturating the link. However, when the connection is shared by many of these flows, the right part of the figure (still receiving a part of `ExpectedRate`) does apply.



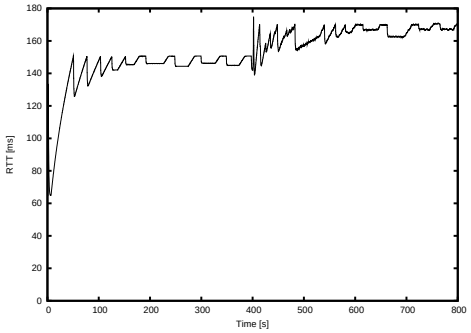
(a) Bandwidth of first flow on slow link



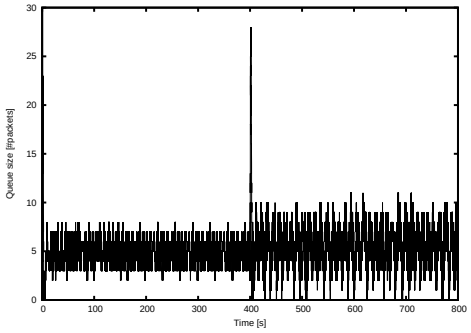
(b) Bandwidth of first flow on fast link



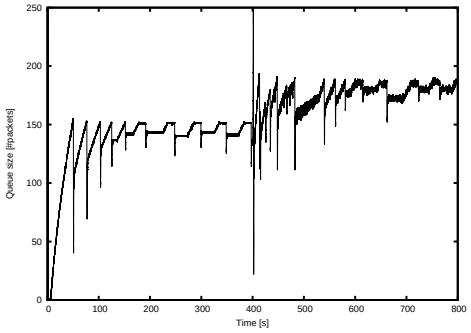
(c) RTT as measured by first flow on slow link



(d) RTT as measured by first flow on fast link



(e) Queue utilisation of slow link



(f) Queue utilisation of fast link

Figure 4.8: Results of RTT limiting TFRC. Note that when the second flow (with a 20 ms lower RTT) is added, the first flow is limited to less than its fair share, but still receives a flow relative to its RTT difference and the ExpectedRate. Also note that the queue of the fast link is loaded much more than what would be necessary to keep the link saturated. Compare these results to Figure 4.9.

	Original	Improved
Slowstart duration [s]	23.47	1.56
Slowstart loss [#packets]	79 (8.3%)	0 (0.0%)
Idle-link RTT [ms]	44	44
Minimum RTT [ms]	1361	200
Maximum RTT [ms]	6462	450

Table 4.3: Summary of important characteristics of original and improved TFRC on slow links (practically no change on fast links)

4.5.7 Remarks on RTT limiting

When choosing TFRC, it was assumed to be a drop-in solution for Iphion's congestion-control needs. Even after witnessing the first results, hope remained of 'fixing' observed deficiencies.

The given modifications do in fact improve on TFRC's behaviour in terms of fitness for Iphion: RTTs are limited, and the possible starvation of flows relative to competing non-Iphion traffic can actually be considered a feature.

However, during development of these improvements it became clear that the equation-based design of TFRC does not match well to what proved to be an important principle: minimising queuing delay (while still achieving link saturation).

The RTT of links will only be limited by queues sizes on such links or the configured maximum RTT, and this RTT is likely to be (much) higher than what is necessary. Combining this with having to tune three parameters, the improvements seem to be of use mainly if no better alternative is available. The next section discusses such an alternative.

4.6 The bandwidth-delay product

As stated in the previous subsection, it became clear that the fundamental concept of TFRC does not match well to the desire for short RTTs (next to efficient bandwidth usage).

In essence, the number of 'in-flight' packets in a network should be controlled such that further increase of that number would not increase the receive rate, but decreasing it would decrease the receive rate. In network parlance this optimal number of in-flight packets is called the bandwidth-delay product (BDP). Although this might seem to be a trivial multiplication of 'the RTT' and 'the bandwidth', especially 'the RTT' is rather undefined here. Should the idle-link RTT be taken (and if so, how to derive it?), should the current estimated RTT of the link be used, or the difference between the two (which could be a measure for the queuing delay)?

The number of in-flight packets is a natural concept in window-based congestion-control algorithms, such as that of TCP, where the window directly indicates the number of unacknowledged packets in a network. This leads to a re-evaluation of the initial choice for TFRC: is TCP's congestion control (not its reliable and in-order delivery) the right choice afterall?

A short study of a TCP flow on a real cable-modem link quickly revealed that that flow did not exhibit the typical sawtooth behaviour of TCP, nor did it saturate the queue (but did saturate the link). This behaviour stands in surprising contrast to what 'the usual textbooks' explain about TCP's algorithm. Additional research showed that modern (after around 2000) TCP implementations use quite different approaches to compute their window size, and now also incorporate queuing delays (albeit it indirectly sometimes) in their reasoning.

Having the desired concept clear (limiting in-flight packets to the BDP), two algorithms were quickly discovered: TCP Vegas , and FreeBSD's TCP . The first is a direct proposal to determine the BDP as described in 4.5.3, but although it has been implemented (but disabled by default) in the Linux kernel for a few years, it did not become popular.

FreeBSD's TCP implementation also tries to estimate the optimal BDP to limit the amount of in-flight packets and as the code has been enabled and in use for many years (since approximately 2002) with much attention from a large user community, its quality and performance can be considered

Algorithm 4.4 Window calculation at the sender, for the proof-of-concept BDP limiter

```
WindowSample := EchoedSequenceNumber - SequenceNumberToBeSent;
Window := 0.9*WindowSample + 0.1*Window;
```

Algorithm 4.5 Proof-of-concept BDP limiting in TFRC based on FreeBSD's TCP implementation

```
PacketsPerSecond := CurrentReceiveRate/SegmentSize;
Delay := (BaseRTT+CurrentRTT)/2;
MaxWindow := PacketsPerSecond * Delay + 2;
if (CurrentWindow > MaxWindow) and
   (CurrentSequenceNumber > MinNextSequenceNumber) then
begin
  MarkThisPacketAsLost();
  MinNextSequenceNumber := CurrentSequenceNumber + alpha*MeanInterval;
end;
```

proven. Additionally, by coincidence, the quick TCP experiments mentioned earlier were performed on FreeBSD, and indeed showed a remarkable performance: smoothly achieving link speed without large RTTs.

Although additional investigation will be necessary (e.g. comparing Linux's new TCP CUBIC to FreeBSD's TCP), it is expected that both will yield the desired performance for Iphion. The direct application of the BDP concept in FreeBSD's implementation and its deceivingly simple computation, lead to recommending this algorithm for Iphion's application.

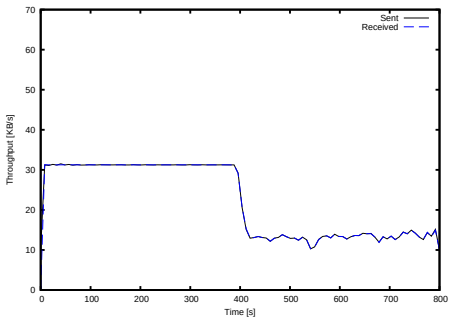
Due to time constraints however, an implementation of (the congestion-control part of) FreeBSD's TCP was not feasible. To still be able to comment on its potential, the BDP principle was implemented in the simulator's TFRC implementation using the existing artificial-loss technique, but as stated, the BDP principle is rather incompatible with TFRC's rate-equation principle: the BDP code will limit the send rate much earlier than TFRC would normally do, because queues are no longer saturated, and no losses will occur. This effectively renders TFRC inoperable. Additionally, BDP uses the number of in-flight packets, which is not available in TFRC. The following subsection should therefore only be considered as a proof-of-concept.

4.6.1 Proof of concept BDP in TFRC

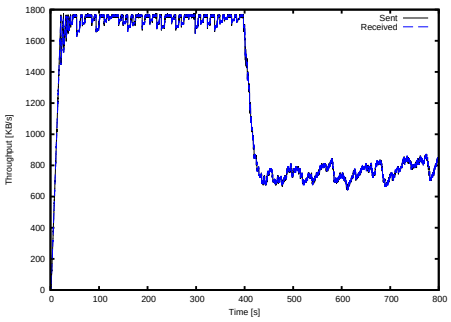
The BDP algorithm needs to know the number of in-flight packets. This was implemented by adding an extra field to the receiver feedback, such that the receiver not only echoes the latest timestamp, but also the sequence number of the latest received packet. Upon reception of such feedback, the sender then computes the window according to Algorithm 4.4, which is sent to the receiver by means of an additional field in the data packets.

The receiver now compares the received window with its computed BDP (from which the formula is directly taken from FreeBSD's implementation), and induces a virtual loss if necessary, see Algorithm 4.5.

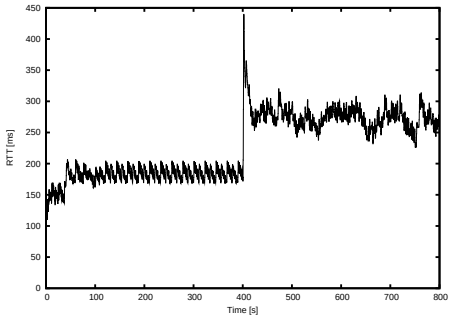
In this algorithm, the `BaseRTT` is measured by the receiver as the minimum RTT value ever observed on the flow, `SegmentSize` is the average size of the data packets (1000 Bytes in all experiments) and `alpha` is again set to 0.5. Note that the `Delay` is computed as the average of the `BaseRTT` and the `CurrentRTT`: using just the `BaseRTT` (or even `CurrentRTT` minus `BaseRTT`) is problematic due to the unreliability of determining the `BaseRTT`, using just the `CurrentRTT` would mean a positive gain of the control loop (i.e. instability), while the averag yields a negative gain in the control loop and is at the same time fairly robust to changes in the `BaseRTT`. The addition of 2 packets to the `MaxWindow` is needed to keep at least one packet in the queue, and to allow the window to grow in case more bandwidth becomes available.



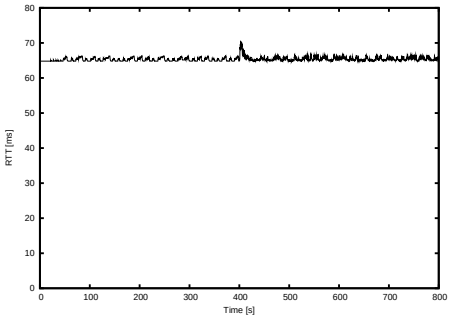
(a) Bandwidth of first flow on slow link



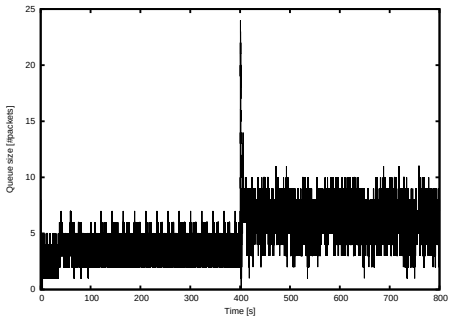
(b) Bandwidth of first flow on fast link



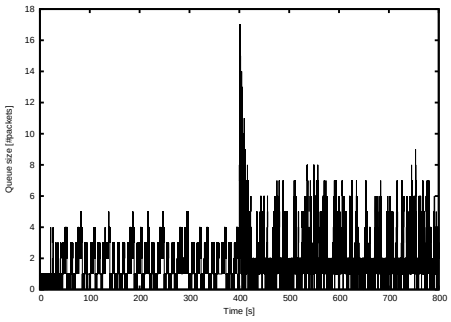
(c) RTT of first flow on slow link



(d) RTT of first flow on fast link



(e) Queue utilisation of first flow on slow link



(f) Queue utilisation of first flow on fast link

Figure 4.9: Proof-of-concept results of bandwidth-delay product limiting of TFRC. Compare these results to Figure 4.8.

	Original slow	RTT limited	BDP limited	Original fast	RTT	BDP
Slowstart duration [s]	23.47	1.56	n.a.	0.95	0.95	n.a.
Slowstart loss [#packets]	79 (8.3%)	0 (0.0%)	n.a.	132 (9.9%)	132 (9.9%)	n.a.
Idle-link RTT [ms]	44	44	44	44	44	44
Minimum RTT [ms]	1361	200	134	45	45	44
Maximum RTT [ms]	6462	450	215	183	183	46
Fairness to second flow	50%	30%	50%	50%	5%	50%

Table 4.4: Comparison between relevant properties of original, RTT limited and (proof-of-concept) BDP limited TFRC on slow and fast links, as defined in Section 4.5.3. Note that slowstart is disfunctional in this proof-of-concept BDP limiter, but should yield results at least comparable to the RTT limited scenario in a 'real' implementation.

The simulation results using this principle are shown in Figure 4.9, from which it is clear that the BDP principle is capable of achieving link saturation without queue saturation. Note that the artificial losses generated by the limiter cause a sawtooth-like behaviour, which will not be present in a 'normal' implementation. The same holds for the slowstart behaviour, which is now completely disabled mainly because of the (too) simple computation of the window and the receiver-based implementation.

5 Conclusions and recommendations

The project delivered results in various directions, including general design considerations, analyses of the existing application and typical networks, actual improvements to the application, implemented proposals in a custom-made simulator, and not yet tested but likely useful ideas for further improvements and investigation.

The first section presents the project's conclusions, followed by further recommendations in the next section.

5.1 Conclusions

When this project commenced, Iphion was already working on an initial version of a streaming P2P application. This version did not live up to Iphion's desires: efficiency and absolute latencies proved problematic.

The objective of this project was therefore to evaluate the existing application based on literature review and implementation-specific analysis. Improvements to various parts of the application were to be designed and verified, for which a simulator was needed. Special attention was paid to solving congestion-control issues, given potentially excessive queing delays in cable/xDSL networks.

Literature review and conceptual analysis Research into state-of-the-art (streaming) P2P algorithms enabled a thorough evaluation of fundamental concepts in Iphion's application. Important conclusions from this work are that mesh-based data delivery is indeed the best solution to optimally make use of available resources in the network, that congestion control should be implemented to prevent congestion collapse, that locality should be taken into account, while preventing weakly connected clusters from forming, that incentives should be implemented to provide a better overall quality-of-service, that FEC (Forward Error Correction) will not be necessary, and that packet scheduling should be a hybrid between rarest-first and near-deadline prioritised scheduling. Additionally, the summary of design considerations enables new Iphion employees to quickly become familiarised with typical P2P pitfalls and concepts.

Iphion application-specific analysis A further analysis of the specific implementation of Iphion's existing application revealed other major issues: 'stateless' packet scheduling prevents P2P communication when deployed on typical cable/xDSL networks, the original streaming of packets from the buffer to the media decoder needs several seconds of (redundant) additional buffering, upload bandwidth of all leaf nodes is unusable because cross links between nodes of the same rank are currently impossible, fixed rank assignment may result in sub-optimal division of resources due to node churn (and depends on a user configurable upload bandwidth), the existing cryptographic signatures do not protect the network against pollution attacks, the original architecture is prone to bugs due to e.g. crypto code being scattered over different functional blocks and has practical disadvantages such that e.g. information about other peers is forgotten too soon (which should not happen on small and/or localised networks).

Most of these issues must be solved before large-scale deployment of the application on typical residential networks is considered: P2P communication on such networks is currently impossible or very inefficient at best. The concept of using relay servers in the network –to reduce zapping delay and reduce inter-ISP traffic– is very good, although this concept is not yet present in the actual application.

Realistic network simulation To verify original and evaluate improved application behaviour, a custom event-based simulator is created with a strong focus on accurately simulating important properties of cable/xDSL networks (see below). The functionality of the original Iphion application has been recreated in this simulator, while already using the newly designed architecture for ease of programming. The simulator allows to instantiate networks of tens to hundreds of nodes with different types of links, latencies, bandwidths, queue sizes, etc. Simulation can be paused at any time, properties of all nodes, links, queues, buffers, packets, etc. can be inspected and modified from the built-in commandline, and graphs can be exported to visualise partnerships, bandwidth usage, etc. Most improvements in this project have been implemented and tested in the simulator.

It is widely known that most residential Internet connections have asymmetric up- and download bandwidths, and this fact is indeed often taken into account when designing modern streaming P2P algorithms. However, most network simulations do *not* take queuing delays in typical cable modems into account, whereas experiments in this report (and other literature) have shown that these delays can be up to 9 seconds! Although typical idle-link RTTs between nearby peers are around 10–50 ms, a queuing delay of 500 ms can easily occur even on fast cable links, yielding a communication delay of 1 second between two peers. This stands in sharp contrast to the rule-of-thumb queue sizes generally used in simulations (roughly taking the bandwidth-delay product of the flow), and has severe consequences for achieving the desired sub-second zapping delays.

Application and protocol improvements Many important issues have been addressed by this project: the Streamer allows to reduce the media decoder buffer from more than 2 seconds down to 400 ms or less (and improves on some smaller issues such as better detection and handling of playback discontinuities).

System-architecture redesign from layers to functional blocks enables cross links, solves peer-record duplication between layers, allows smarter decisions on such records (such as keeping information about some peers if no better peers are available), and delivers cleaner and more flexible code, paving the way for solving other recommended issues.

A simple but very efficient algorithm is designed and implemented in the simulator to allow Iphion's network to be split in several (overlapping, if desired) clouds, with one or more relay servers serving that cloud. The algorithm is based on (easy to obtain) information about ISP networks in which the relay server will be placed, and allows immediate estimation of network distance between peers, without (slow) probing and irrespective of RTTs (which may not reliably indicate distance in practice).

Experiments in the simulator have shown that Iphion's existing application causes excessive packet loss (congestion collapse) and packet duplication on such links (e.g. 300% duplication if the RTT is 'just' over 450 ms) due to the lack of proper congestion control and inefficient packet scheduling. A smarter packet scheduler was therefore designed, that prevents unnecessary resending of in-flight packets (from e.g. 300% to 0%), while still allowing to quickly detect lost packets (e.g. after receiving 2 additional packets, i.e. 20 ms instead of 450 ms) and re-request them. This already solves one part of the problem, but does not yet prevent high queuing delays.

Congestion control for Iphion application To solve the latter, proper congestion control is needed. First, an exploration of Iphion's requirements on congestion control was made, leading to the definition of two types of flows (Peer-to-Peer and Relay-to-Peer), and two directions of flows (to and from peers), each with different requirements: Relay-to-Peer flows are only encountered in the to-peer direction, and are always beneficial for that peer. Peer-to-Peer flows occur both on to-peer and from-peer directions, but because the bottleneck link is likely at the from-peer side, these flows should behave friendly to competing traffic. To still guarantee reliable stream delivery even when Peer-to-Peer flows are supressed, the Relay-to-Peer flows should be divided in a part (a bit more than the stream bitrate) *without*, and the remainder *with* congestion control, additionally yielding more efficient resource usage at relay servers especially during peak hours.

TFRC evaluation and improvements As the Internet Engineering Taskforce (IETF) and literature recommend and/or use TCP-Friendly Rate Control (TFRC) for UDP congestion control (especially in

multimedia applications), this algorithm was implemented in the simulated Iphion application. However, simulations using realistic (cable/xDSL) queue sizes show that TFRC exhibits extremely large queuing delays (well over one second, up to 9 seconds) just as the original Iphion application.

Thus, in contrast to popular belief, TFRC should *not* be used for low-latency multimedia applications on typical cable/xDSL networks!

An improved version of TFRC was then designed, which allows to put a limit on RTT. The modifications to TFRC have deliberately been very small, and only make it behave more 'friendly' towards competing traffic (which is a feature, given Iphion's congestion-control requirements). Although the algorithm greatly improves upon TFRC's original behaviour on slow links (completely preventing congestion losses, reducing the slowstart phase from 30 to 2 seconds, limiting the RTT from 1.5–6.5 s to 200–500 ms), it requires three parameters to be chosen (`MaxRTT`, `RTTMargin` and `ExpectedRate`), and still causes queues of faster links to be filled more than necessary (up to the configured maximum RTT, queuing up e.g. 150 packets instead of 5).

TCP's congestion control for low-latency multimedia It is then shown that control of the amount of in-flight packets based on the bandwidth-delay product (BDP) should be used, instead of control of send rate based on packet loss, to simultaneously satisfy all requirements: 100% link saturation, (almost) no packet loss, *and* short queuing delays (e.g. 4 ms instead of 150 ms for improved TFRC, or even 9 seconds for 'standard' TFRC). Such an approach does not fit well to TFRC's equation-based send-rate control, but fits remarkably well to window-based algorithms such as TCP. TCP Vegas and FreeBSD's implementation of TCP actively include the BDP into their reasoning, which leads to the remarkable recommendation to use the congestion-control part of FreeBSD's TCP implementation for further investigation. The latter was chosen because of its beautifully simple algorithm, and many years of proven performance. It is expected to perfectly suit the needs of Iphion, and likely other low-latency multimedia applications as well.

5.2 Recommendations

The most important recommendations resulting from this project are:

- Investigate the performance of FreeBSD TCP's congestion control, most notably its queue-fill behaviour. Given the results of Section 4.6, it is expected that that algorithm satisfies all requirements defined in Section 4.1 for Peer-to-Peer flows.
- For Relay-to-Peer flows, a mechanism similar to TCP-parameter caching in operating systems may further improve performance during slowstart, by not starting from zero, but a 'known-possible' rate.
- Assuming that the congestion-control algorithm then indeed satisfies the requirements, implement this in both players, and relay servers.
- Although TCP-like congestion control is less prone to being fully suppressed than the RTT-limited TFRC and is thus more likely to maintain at least the stream bitrate, it is recommended to also consider the 'congestion control-less' transmission from relay servers, as this reduces the computational complexity at the servers and allows 'guaranteed' stream delivery. (Note: faster slowstart might also be possible by temporarily increasing the congestion control-less flow.)
- Packet scheduling must be changed to prevent re-sending in-flight packets, but also to quickly recover from lost packets as explained in Section 3.4.
- The current single-server model should be split to include relay servers, with a separate server (possibly on the same physical machine as the streaming server for small networks) as the rendezvous point ('channel-join server'). A protocol to achieve this is described in Section 3.3.
- Although computationally expensive, the media stream itself should be protected with a cryptographic signature.

The previous items can be considered 'deal breaking' for large-scale deployment. Less critical, but useful improvements would be:

- Part of the architectural changes of Section 3.2 have already been implemented in the real application, but cross links and uni-directional partnerships should still be implemented.
- Locality awareness is a challenging (e.g. to maintain a DAG structure to prevent isolated clusters) but important improvement, and should ideally be added in an early stage of development.
- The playback pointer / buffer synchronisation scheme should be modified as explained in Section 3.1. Additionally, a better way of determining the initial playback point selection should be designed, to cope with differing zapping delays due to link speeds and network health.
- When a user switches to a different channel, the application should inform its leechers on the previous channel of its departure, but could still keep sending data to them for a few seconds while the uplink is not yet used for the new channel. This can greatly improve robustness of the network on 'catastrophic' events such as when almost all users simultaneously switch to a different channel when commercials start on a popular channel.
- Incentives (Section 2.1.9) require an administrative system to be created and complicate the division and prioritisation of bandwidth, but should still be considered as they can provide a better overall QoS of the network.
- Buffermaps and requestmaps (especially the 'differential' versions of Section 3.4) can be compressed by truncating them to just include regions with bits actually set. Optionally, some processing power could be saved by aligning all maps to 8- or 32-bit boundaries.
- When implementing congestion control it is wise to keep buffers in the operating system as small as possible, to allow newly arriving requestmaps asking for near-deadline packets to be sent promptly. This can be done by implementing a callback structure or prioritised queue, instead of FIFO enqueuing all packets at once upon receiving a requestmap.

A Existing P2P algorithms

This appendix contains short reviews of existing P2P algorithms, ranging from the first to very recent designs, with a focus on relevance to Iphion. Especially the later proposals include some interesting views and ideas, which may be useful for later consideration.

The following sections are ordered by their main structure (tree, mesh) as this structure already decides on some major algorithm choices (e.g. packet scheduling). Additionally, this yields a nearly chronological ordering.

A.1 Single tree

The single tree protocols were the first designs, and are generally simple to implement, but inefficient and unreliable (when not using dedicated hardware/servers).

Narada One of the first overlay networks not requiring native IP multicast, truly working at the application layer appeared in 2000 with a protocol called Narada [20] by Carnegie Mellon University. It builds its network in two stages: first, a mesh is built between nodes, on which then spanning trees (in case of multiple sources) are created to transport the data. It was designed to cope with the dynamicity of nodes, and constantly tries to optimise the trees it creates. It does so in a distributed way, only using a known rendez-vous server for bootstrapping. Since every node in the systems keeps track of and communicates with every other node, the system does not scale to large networks (every node maintains $\mathcal{O}(N)$ state and the system has $\mathcal{O}(N^2)$ control overhead). The authors themselves state that their network can be used for tens to hundreds of nodes [19].

NICE To improve the scalability of Narada, researchers at the University of Maryland proposed NICE [6]. The main difference with Narada is that it uses a hierarchical structure instead of a mesh to construct the data delivery trees. The hierarchy is built from clusters of nodes, where every cluster is of size between k and $3k-1$. Every cluster has a leader, and the leaders of the clusters in hierarchy level L_0 form clusters in level L_1, etc. There will be at most $\log_k N$ levels. Every node is in level L_0 and the highest level will have just one node. NICE maintains $\mathcal{O}(\log N)$ state, and has control overhead between $\mathcal{O}(k)$ (for non-cluster leaders) and $\mathcal{O}(k \cdot \log N)$ (for the node in the highest layer).

The data tree is implicitly defined by the control structure: a node sends its data through its cluster leader, which will forward it to the nodes in the cluster, etc.

The cluster leader is chosen to be the graph theoretic centre of the cluster. In the paper, this means it has the minimum maximum distance to all other nodes in its cluster, where the distance is defined as its latency. This will lead to quicker join operations, as a joining node contacts the highest level cluster leader first, and is then directed to the leaders of increasingly lower layers. As the leaders also directly determine data paths, this is not necessarily the best path in terms of bandwidth for example.

ZIGZAG An even more efficient technique was proposed by University of Central Florida, called ZIGZAG [71]. Like NICE, it builds a hierarchical structure, but does not use the leaders of the cluster, but a "foreign leader": the leader of a different cluster. Hence the name ZIGZAG. In case the data parent (foreign leader) of a cluster fails, the head (as it is called in ZIGZAG) of a cluster finds a new parent, allowing for a faster repair, using just the local neighbourhood. ZIGZAG also decreases the worst-case degree of the nodes to $\mathcal{O}(k^2)$ (instead of $\mathcal{O}(k \cdot \log N)$). It still does not consider bandwidth heterogeneity, and as it uses a $\mathcal{O}(k)$ star topology to distribute data from a leader to a foreign cluster, this imposes a limit on the total stream rate that can be sustained.

SRMS When a node in a tree breaks, this has consequences for the whole subtree. Scalable Resilient Media Streaming (SRMS) [7] tries to lessen this effect by using a technique they call Probabilistic Resilient Multicast (PRM). The idea is to have nodes in the tree forward blocks to a few (e.g. $\beta = 3$) random other nodes in the tree with a certain low probability (e.g. $r = 0.01$ or $r = 0.03$). Although this will yield a small percentage of duplicate packets, it can greatly enhance the reliability of the tree: if a node now fails, it is possible to get some missing pieces from the children instead, while quickly repairing the tree simultaneously. The paper shows that even for small values of r and β, the overhead to guarantee successful data delivery with a high probability asymptotically decreases to zero for asymptotic increase of group size.

In the experiment in the paper, with an overhead due to PRM of around 3%, delivery ratios of 100% were obtained for 20% of the group, and 90% experienced a loss of less than 5%, compared to the NICE method where 50% of the nodes experienced more than 20% losses. Quite a remarkable improvement for such an apparently simple solution.

A.2 Multiple tree

Multi-tree algorithms already improve on the single tree approach by reducing the amount of repairing needed after one tree fails, and provide somewhat more efficient usage of available bandwidth. Data is still pushed down the trees, so there is no need for requestmaps etc. (which will add to the delay).

SplitStream A different approach was taken by the people from Microsoft Research and Rice University [12]. To overcome some problems of a single tree structure (such as NICE), multiple trees are used. The video stream is encoded using Multiple Description Coding (MDC) [33], such that every description can be decoded separately. Every description is then sent over its own tree, hence the name SplitStream. A failure in one tree will now lead to 'just' a graceful degradation of the video quality.

The paper states an important point about the bandwidth unfairness issue of trees: the fraction of leaf nodes increases with the arity (node degree) k. For example, for a binary tree ($k = 2$) more than half of the nodes is a leaf node. For $k = 16$ (often used for MDC) this is already more than 90%. This means that the full load of the trees has to be carried by less than 10% of the nodes. This is the reason why SplitStream requires "interior node disjointness": every node in SplitStream is an interior node in one tree, and a leaf in the remaining ones. This way the forwarding load is spread across all nodes. Bandwidth heterogeneity is considered in part, by allowing a node to join less than the maximum (in this case 16) number of trees.

SplitStream builds its trees using Scribe [13], which in turn is built on Pastry [67].

Pastry is a so-called Distributed Hash Table (DHT): a completely decentralised database of key-value pairs using hashes as the keys. In Pastry, nodes and objects have random identifiers (nodeIDs and keys, respectively) from a large but sparse ID space. Identifiers are 128 bits wide. Every node routes a message consisting of key and value to the node with nodeID numerically closest to the key. Each node maintains a routing table of size $\mathcal{O}(\log N)$, which is constructed such that a node can forward a message to another node with a nodeID that has a larger prefix in common with the message key. This allows for message routing paths of length $\mathcal{O}(\log N)$. The nodes in the routing table are further chosen to prefer nodes with shorter network delays. Scribe uses Pastry to join a multicast group, which is identified by a groupID. The group consists of all Pastry routes from all members of the group to the source. Note that these routes can include nodes that are not members of the multicast group (in case of multiple sources). The source then transmits its data through the reverse paths back to all members. Joining the multicast group therefore just consists of routing a message towards the multicast groupID until an existing member is reached, and adding this route to the multicast tree.

Interior node disjointness is performed by using nodeIDs which differ in the first part of the prefix, thereby creating a completely different path with high probability of disjointness. To limit the maximum load on a node, an additional limit on the out-degree (on top of the in-degree limit in Pastry) is used. In case a node wants to connect to a parent which is already 'full', the node is "pushed down" the tree until it finds a child that has sufficient capacity available. If this fails, the node uses a feature of Scribe

called "anycast" to locate a member of the special "spare capacity group". This will potentially create long branches, which are not desirable in terms of delay.

CoopNet Again Microsoft Research, this time in cooperation with Carnegie Mellon University, propose CoopNet [59]. It is very similar to SplitStream in a number of respects, but differs by its use of a central server to organise the nodes in an efficient tree. It too tries to create interior node disjoint trees, but with a different philosophy: to make the trees as short as possible. In SplitStream, the maximum out-degree of a parent can hurt the node disjointness, and possibly leads to high trees. In CoopNet, the central server can easily guarantee an even load distribution and interior node disjointness. They argue that using a central server is allowable in this case, as it is also the source of the data. If this server should fail, the source is also gone, so there would be no use to still have a functioning tree anyway.

In CoopNet, a node joins the tree by asking the source, which in turn searches its (internal) tree for a few nodes close to the root with spare bandwidth. A random node is picked from this list and returned to the client. The purpose is to create short, wide trees.

An improvement of the system was made by having each node measure its position relative to certain landmark nodes. The server will now bias the node-picking algorithm with this information, trying to return nodes close to the client. In effect, this achieves the same type of locality that Pastry does for SplitStream.

The paper also mentions the potential node churn in case of a large flash crowd: traces from the MSNBC site at September 11, 2001 show an average node arrival and departure of 180 nodes per second, with a peak rate of 1000 per second. Group sizes being approximately 10000 on average, peaking at 18000.

Chunkyspread Chunkyspread [72] by Cornell University and University of Tokyo also creates multiple trees, and uses MDC (see Section 2.1.2) to split the video. In contrast to SplitStream, it does not use a DHT (e.g. Scribe+Pastry) as the underlying structure, but instead uses an unstructured mesh to find new peers. The mesh is constructed using an algorithm called Swaplinks, described in a separate paper [73]. Another major difference with SplitStream is that Chunkyspread does not strive for node disjoint trees.

Chunkyspread expects a user to configure the nominal and maximum amount of bandwidth to supply, relative to the downstream. It expresses this amount in number of (MDC) slices. The goal of the algorithm is first try to find enough peers to fulfil the bandwidth need, and when this is satisfied, to try to find peers which could improve the delay. The node degree is adapted to the nominal upstream, so that peers that can/will supply more, will have correspondingly more neighbours.

Although potential peers are discovered at random by the Swaplinks algorithm, not every peer qualifies as a parent: the algorithm tries to build trees, which may by definition have no cycles. To prevent cycles, Bloom filters are used: every node has an ID, which will leave a 'fingerprint' in the Bloom filter that is sent on every outgoing packet. Upon reception of a packet, the node inspects the Bloom filter to detect its fingerprint, and if it matches, a loop is declared and that parent is disconnected. An excellent overview of other applications of Bloom filters in network problems (e.g. set reconciliation) is given in [9].

The paper contains some detailed apples-to-apples comparisons between Chunkyspread and SplitStream, also showing some typical numbers of startup times, load balancing, latencies, control overhead, recovery times, etc.

The authors state that the algorithm combines the efficiency of a tree-based approach with much of the simplicity of a swarming style approach, and allowing different optimisation strategies depending on the need of an application. They also feel that there currently is no definitive answer to the question of whether to use tree (like Chunkyspread) or swarming (like CoolStreaming) technology.

Trickle Researchers at National Chiao Tung University invented Trickle [34], a system that also uses multiple trees, but this time not using MDC to split data. Instead, they use the Information Dispersal Algorithm (IDA) [63], which is an erasure code that splits data in n fragments, of which at least m

fragments ($m \leq n$) have to be received correctly to decode the original data. The n fragments are distributed via disjoint trees, and a node subscribes to at least m trees. In case of packet loss, a node can subscribe to more trees to compensate for the losses, as it does not matter which packets are received, just the number of packets. The number of trees to join is determined dynamically: if the number of successful frame recoveries drops below a certain threshold, a node will join more trees, and vice versa. Trees are not dropped and joined at random, but using a circular queue. Additionally, the minimal number of trees a client must join is restricted by a variable r, $m \leq r \leq n$. In the experiments $n = 20$, $r = 17$ and $m = 16$.

The trees are created using Pastry and the source is, just like in SplitStream, assigned n different identities to create interior node disjoint trees. However, in case a potential parent does not have enough bandwidth available to support a new child, the parent will try to recruit a helper node, that will become the actual parent of the child.

One problem with Trickle is that packet overhead is quite large: 25%. The authors suggest this is caused by the packetisation scheme, as H.264 can produce very small packets, which are further divided by IDA. Another problem is that of incentives: it is not clear why a (non-member) node would want to become a helper node.

AnySee AnySee [50], by Huazhong University of Science and Technology and Hong Kong University of Science and Technology, is based on CoolStreaming (described later), but uses optimisation techniques on the (random) mesh to improve locality. It looks at timestamps in received packets, and discards parents with higher delays. It builds trees on top of the locality aware mesh by "reverse tracing": messages are forwarded between peers, that record the full list of nodes the message has passed so far, which allows to 'replay' those paths. To improve robustness, every node maintains two trees: the 'live' tree, and a backup tree.

AnySee claims to improve the startup time from CoolStreaming (around 60 s) to approximately 20 s, using a 40 s buffer (instead of 120 s), and at the same time increasing the playback continuity and resource utilisation.

A.3 Hybrid

The hybrid systems mainly use trees to distribute data, but start to use data pulling for sub-structures.

Bullet Researchers at Duke University invented Bullet [44], a system that uses a single-tree to transport control messages and an initital set of disjoint data, and then uses 'orthogonal' links between random nodes in the tree to distribute the remaining bulk of the data (i.e., to join the data sets again). Advantages of this are that it is no longer needed to maintain a bandwidth optimised tree (as the bulk is transferred through the random mesh), and that the tree is a very efficient way to distribute non-time-critical data. It allows peers to fully utilise their uplink, irrespective of their position in the tree. Bullet uses TFRC (see Section 4.3) as the congestion-control mechanism, and uses Bloom filters to solve the set reconciliation (instead of requestmaps).

A disadvantage is that once a node is assigned a position in the tree, it has no way to change this position. This has direct consequences for the playback position, amount of 'safety margin' (pre-buffer) a node can acquire, and still largely determines the amount of bandwidth the node should supply. Thus, uncooperative nodes near the source can therefore degrade system performance as being a member of the tree still means having to transmit the basic disjoint data set.

BulkTree BulkTree [32] by Tsinghua University creates a single tree to distribute control messages and data, but to overcome the problem of unstable interior nodes, they propose to use clusters of nodes instead, which form a virtual "super node". Every cluster has a leader and vice leader, and the leader ensures the cluster size will remain between k and $3k$ nodes. If it becomes too big, the virtual node is split, and it is merged if the cluster should become too small.

GridMedia GridMedia [83], also by Tsinghua University, uses a combination of two approaches: data pulling and data pushing. The system creates a mesh like that of CoolStreaming, but divides its buffer in two parts: the node first tries to fill the 'front' part of the buffer using a data push strategy, and pulls the remaining missing packets from its neighbours. This is done by using a sliding window over the buffer: as soon as missing packets slide from the push window into the pull window, the pulling phase is started using requestmaps. To prevent duplicates in the push window, so-called "pushmaps" are sent to neighbours to inform them about packets the node is interested in receiving when they become available at that neighbour.

The push-pull system has a slightly lower overhead than the pull-only scenario, and improves mainly on the average end-to-end delay of nodes and the playback continuity. No locality or incentives (just like CoolStreaming) are taken into account.

The algorithm might prove useful for the Iphion application as well: Iphion's relay servers will always have the newest packets, so they can send these packets to any other node (as long as their buffer is not too far behind) without the risk of packet duplication. Nodes can then exchange the remaining missing packets with each other. Additionally, this buffer technique could be used to fill a large part of the buffer using normal IP multicast if it would be available (see also HyMoNet, below).

HON The Hybrid Overlay Network (HON) [86] proposed by Simon Fraser University turns the usual assumptions upside-down: they take a gossip style overlay to distribute the bulk of the data, and create a tree structure on top of the (random) mesh to be able to quickly retrieve missing packets from parents in the tree as the tree is optimised for low delay to the source. (Usually, the mesh is used to retrieve missing packets, e.g. GridMedia [83], Bullet [44] and PULSE [61].)

The system is designed to be used for Video on Demand.

HyMoNet The Hybrid Multicast Overlay Network (HyMoNet) [14] by Tsinghua University tries to use IP multicast if it is available, and uses an overlay network for the remaining interconnectivity. It uses a central service for parent discovery. Node heterogeneity is not considered: a parent must simply forward the full stream to every child.

The idea of using network layer multicast (if available) is nice, and could also be used for other protocols, e.g. GridMedia [83], mTreebone [77] and the Iphion protocol as well.

CDN-P2P This system in [80] by Purdue University combines a Content Distribution Network (CDN) like Akamai with a P2P network for Video on Demand services. The general idea is to use the CDN to bootstrap the network, and if enough peers have buffered (part of) the video, the data distribution is gradually shifted to the P2P network. The CDN servers will still be used to maintain the structure of the network (much like trackers in the BitTorrent scenario) and to maintain credits/debts of peers in the system.

There is a nice analysis in the paper about calculating the "hand-off" moment: when to stop serving the content through the CDN. This tactic might be useful for Iphion to determine the amount of relay servers that will be needed in the network.

mTreebone People from Simon Fraser University and Microsoft Research Asia invented mTreebone [77]. The authors discovered (using log traces of real broadcasts) that usually a P2P overlay system converges to a set of fairly stable nodes with high bandwidths forming the internal nodes of a tree-like structure near the source, and a dynamic set of outskirt nodes and leafs. They also note (like the authors of SplitStream) that it is possible to support such a tree using a small set of strong interior nodes.

This insight led to a system where nodes can promote themselves to be stable enough, depending on the amount of time they are in the session, and the total duration of the session. It is allowed to use this metric, because the durations of nodes in the overlay is not a uniform probability, but follows a heavy-tailed distribution (in particular: a Pareto distribution). These stable nodes will form a tree, so data can be pushed down, decreasing the end-to-end delay. Unstable (i.e. newer) nodes then connect to the leafs of this tree and use pulling techniques to retrieve the data. The stable nodes in the tree use

(like CoolStreaming) the SCAMP protocol [31] to inform each other of potential neighbours in case a stable parent should leave the network. If this happens, the node will use 'standard' requestmaps to reconcile the missing pieces, and quickly tries to re-attach to another parent in the tree.

mTreebone nodes use the same 'two-stage' buffering that GridMedia nodes use, with a sliding window that determines packets to fetch through the data pull approach, in case the tree push part did not deliver all packets.

The paper thoroughly compares mTreebone with Chunkyspread and CoolStreaming, and shows some significant improvements are made. However, the system does not consider asymmetric bandwidths, and purely uses the duration criterion to promote a node to the stable tree structure.

A.4 Unstructured

This category contains the random, gossipping type of networks. Note that the initial version of Iphion's application was based on CoolStreaming/DONet (and still contains its requestmap principle).

CoolStreaming/DONet One of the first and one of the most famous P2P overlays is CoolStreaming/DONet [85], invented by The Chinese University of Hong Kong, Simon Fraser University and Hong Kong University of Science and Technology. DONet is the scientific name for the project, CoolStreaming (which stands for COoperative OverLay Streaming) is the public Internet implementation.

In CoolStreaming, a random mesh is constructed, from which each node selects a limited number (e.g. 4) of partners. The random mesh is constructed using SCAMP [31], a scalable and lightweight gossipping algorithm. It maintains a partial view of the network, and exchanges messages with other nodes informing them about the membership of itself and other nodes. The messages exchanged this way are periodically sent, and have very low overhead. Additionally, the SCAMP algorithm creates a mesh with path lengths of $\mathcal{O}(\log N)$, which helps to constrain the maximum end-to-end delay.

In contrast to the tree-based approaches, data is now supplied by more than one 'parent'. This means that a receiver must coordinate what to fetch from each parent. This is done by requestmaps (RMs): a list of bits, where every bit corresponds to a specific packet. If a bit is set, it means the receiver requests the packet from the parent. A receiver knows what to ask by means of buffermaps (BMs): again a list of bits, but this time denoting the availability of a packet in the buffer of a parent. Parents periodically exchange buffermaps with their partners. A receiver schedules its packets on a rarest-first basis: it first schedules packets which are available at just one partner, then those that two partners have, etc. In case there is a choice, the algorithm chooses the parent that can deliver the packet as fast as possible. The source always has all packets available, so would usually supply a buffermap with all bits set. However, to spread the load, the source can supply buffermaps with bits set in a round-robin fashion.

CoolStreaming employs TFRC (see Section 4.3) for its congestion control, and uses piggybacking of BMs and RMs on the feedback and data packets to decrease the amount of overhead.

The control overhead is mainly caused by the buffermap and requestmap exchanges between partners; the gossipping protocol is very lightweight. Experiments show it can achieve very low overhead: less than 2% for 6 partners and a playback continuity of more than 98% for a network size of 500 nodes and a playback rate of 500 kbps. The buffer used in this case was a hefty 60 seconds though.

DONet is very robust to node churn, and is easy to implement (the authors mention 2000 lines of Python code for the CoolStreaming v0.9 implementation) and scalable (4000 simultaneous users for 450-755 Kbps streams are reported). The biggest disadvantages of the system are that it uses very large buffers (usually 120 seconds) and has very long zapping times (around 30 seconds), and that it does not take locality into account.

Chainsaw Chainsaw [60], by Stony Brook University, is an improvement of Bullet: its goal is to eliminate the tree (hence the name) from Bullet (Section A.3). The authors implemented the system in Macedon [65] (by the authors of Bullet), which allows for a one-to-one comparison with Bullet and SplitStream.

Although the system uses no incentives, the authors state that this can be included if needed, for example using SWIFT [70]. A downside of the algorithm is that the mesh is fully random, which again does not take locality into account.

An important point to consider is the packet scheduling strategy: the algorithm picks packets at random, and it was found that indeed some packets will never appear in the network, and by the time nodes start to request it, it is too late to distribute it to all nodes. This was solved using a heuristic in the source (called "Request Overriding") where the source will force not-yet-requested packets into the network.

A.5 Structured

As noted in the design considerations (Section 2.1), care must be taken when trying to incorporate some kind of structure in a random mesh (such as CoolStreaming). In the general case of structuring a random mesh, weakly connected clusters can appear. To prevent this problem, well-known techniques can be used, e.g. Distributed Hash Tables (DHTs, like Chord, CAN or Pastry[67]), Directed Acyclic Graphs (DAGs [48, 58]) and SkipLists (e.g. [76]).

This section contains proposals that actively promote such a (non-tree) structure.

Dagster Dagster [58], invented by the National University of Singapore, creates a Directed Acyclic Graph (DAG, see Figure 2.1) structure to guarantee a minimal number of distinct paths to the source. This prevents the problem of a single point of failure, as could occur when blindly clustering random meshes. A major goal of the system is to provide incentives to nodes to encourage sharing more bandwidth, allowing 'better' nodes to preempt other nodes, thereby getting closer to the source (in terms of path length).

In Dagster, nodes are capable of re-encoding video streams to a lower bitrate, to allow more challenged nodes down the DAG to still receive a stream, but in a lower quality. MDC is used to solve the problem of set reconciliation: a client assigns every parent a number, and expects that parent to encode and send that frame, modulo the number of parents.

The DAG structure is maintained by a central server, for two reasons: easy and guaranteed organisation of the DAG, and to prevent cheating. As the system allows nodes to preempt each other, some authority must validate and control the preemption commands. This also allows to prioritise certain nodes.

Disadvantages of the system are the arbitrary end-to-end delay and the lack of locality awareness. Additionally, the use of stream recoding makes it unsuitable for use in the Iphion setup.

DagStream Although also using a DAG, researchers from the University of Illinois propose quite a different strategy. DagStream [48], in contrast to Dagster, uses a distributed algorithm to maintain a DAG structure. The structure is built (like in many other approaches) in two steps: first a random mesh is created, in this case using an algorithm called RandPeer [49]. Then, a subset of the random peers is chosen to form the DAG. Every peer sets is level to one more than the level of its parents. It then (periodically) broadcasts its level to its children. To prevent a loop, a clever trick is used: if a loop has formed, this would mean the level update will re-appear as a level update of one of the 'parents'. If this happens too often, a loop is declared, and the connection with the parent is broken. Note that the temporary existence of a loop does not directly hurt the performance of the system, as the DAG is mainly used to prevent weakly connected clusters.

It should be noted that DagStream takes a locality first, bandwidth second approach. The philosophy is to select parents with a low latency/level first, and if this does not satisfy the bandwidth need, select more parents.

DagStream allows peers to 'walk' up and down the DAG, and every node tries to improve its QoS by achieving a lower level, and lower latency. The paper experiments with three different parent selection criteria: delay only, level only and first delay, then level. It appears that the delay+level approach is better than the delay-only approach, which is in turn much better than the level-only approach. By incorporating delay, a node can improve the locality of the DAG.

To speed up finding good parents, DagStream tries to use its "two-hop neighbours" first. This information is obtained by periodically sending (piggybacked on other packets) the list of parents to each parent, which then allows a parent to inform its children about possible good candidates for them. Only if the two-hop neighbours cannot be used (because they are full, or don not satisfy level constraints for example), the RandPeer service is invoked.

Although the name RandPeer suggests fully random nodes will be returned, this is not the case: every node determines a 3-bit position vector which encodes the delay to 3 landmark hosts. This information is then used to return a biased random set of nodes in the neighbourhood.

A disadvantage of DagStream is that, just as with trees, leaf nodes do not contribute back to the network. The general idea of using a simple algorithm to build a DAG structure to prevent isolated clusters is very useful though, see Section 2.1.5.

Cyclon/Vicinity The dissertation of S. Voulgaris of the Vrije Universiteit Amsterdam contains good descriptions of prior work, but also describes a few fairly simple but highly effective algorithms for constructing meshes. The first contribution of the work is the Cyclon algorithm, that is used to build fully random overlays, using gossips. Some strategies for exchanging neighbour information are discussed, focusing on good randomness of the resulting overlay, low bandwidth overhead, short path lengths (in terms of hops) and limited degree.

As Cyclon does not consider locality/clustering, a different protocol is built on top of Cyclon called Vicinity. Although the algorithms have many similarities, it is stated that it is very important to have *both* algorithms working simultaneously to prevent the optimisation of the clusters to get stuck in local maxima. Additionally, having random links helps newly joining nodes to quickly 'fly' to the best cluster. Note that the definition of "vicinity" is left open: it could be used for network latency optimisation, clustering according to IP addresses, but also clustering by nodeID etc.

The Vicinity algorithm could be a very nice option to implement in case a fully distributed protocol is needed to optimise the topology of the streaming process, namely to provide a parent-selection algorithm with a list of potentially good neighbours.

PULSE The inventors of PULSE (France Telecom Division R&D) [61] liked some ideas of Bullet, but see the dependence on a tree as a problem, because of the direct relation between a node's static position in the tree and e.g. the amount of bandwidth it should supply. Therefore, in the PULSE system, nodes are (somewhat similar to the DagStream case) allowed to walk up and down the structure. It is important to note that in this case the node's position in the structure is determined by its buffer position, not the other way around. Note that PULSE does not prevent cycles from forming, which could lead to weakly connected graphs.

PULSE introduces the notion of a Relative Media Clock: variables corresponding to buffer positions are expressed in terms of a delay relative to the clock of the source, instead of a constantly increasing value. This means it is now easier to communicate this information with other nodes as it will remain practically constant, and it is easier to derive the speed of e.g. buffer head progression in relation to the stream rate.

Also, in contrast to the constant buffer size that is used in most systems, PULSE uses a dynamically growing or shrinking buffer around its playback position. It starts with a small buffer, which is gradually expanded if the network allows to proceed with a speed larger than the stream rate, thereby improving the startup time, and still maintaining robustness in case of node failure.

PULSE has two different modes to fill its buffer: Missing and Forward. The Missing recovery is much like the buffermap approaches (data pulling) used in other systems, and is used to reconcile differences between the buffers of nearby (in terms of graph position) nodes. The Forward recovery is used to speed up the distribution of data in way that resembles the data push methodology. Forward recovery can be used on links from nodes near the source, to nodes farther away (or in the case of Iphion: from relay servers to players).

To provide incentives, PULSE incorporates a tit-for-tat policy much like that of BitTorrent [22] in what is called the "Friend" mode, and uses a micro-payment scheme for other modes. This means

a PULSE node will maintain state about other nodes it has communicated with, to remember the debts/credits of these nodes.

The paper makes some interesting points about how to determine the play position of a node, by looking at the state of the nodes around it, which might be useful for the Iphion application.

References

[1] Adar, E. and B. Huberman. FREE RIDING ON GNUTELLA. First Monday, Vol. 5 (2000), No. 10, p. 134–139.

[2] Ali, A. and A. Mathur, H. Zhang. MEASUREMENT OF COMMERCIAL PEER-TO-PEER LIVE VIDEO STREAMING. In: Proceedings of the Workshop in Recent Advances in Peer-to-Peer Streaming, Waterloo, Canada, 10 August 2006. School of Computer Science, Carnegie Mellon University, Pittsburgh, 2006.

[3] Allman, M. and V. Paxson, W. Stevens. TCP CONGESTION CONTROL. RFC 2581 (Proposed Standard). Sterling: Internet Engineering Taskforce (IETF), 1999.

[4] Annapureddy, S. and S. Guha, C. Gkantsidis, D. Gunawardena, P. Rodriguez. IS HIGH-QUALITY VOD FEASIBLE USING P2P SWARMING? In: Proceedings of the 16th international conference on World Wide Web, Banff, Alberta, Canada, May 08–12 2007. New York: ACM Press, 2007. p. 903–912.

[5] Banerjee, S. and B. Bhattacharjee. A COMPARATIVE STUDY OF APPLICATION LAYER MULTICAST PROTOCOLS. IEEE Network, Vol. 4 (2002), p. 3.

[6] Banerjee, S. and B. Bhattacharjee, C. Kommareddy. SCALABLE APPLICATION LAYER MULTICAST. In: Proceedings of SIGCOMM '02: the 2002 conference on Applications, technologies, architectures, and protocols for computer communications, Pittsburgh, Pennsylvania, USA, 19–23 August 2002. New York: ACM Press, 2002. p. 205–217.

[7] Banerjee, S. and S. Lee, R. Braud, B. Bhattacharjee, A. Srinivasan. SCALABLE RESILIENT MEDIA STREAMING. In: Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video, Cork, Ireland, 16–18 June 2004. New York: ACM Press, 2004. p. 4–9.

[8] Bishop, M. and S. Rao, K. Sripanidkulchai. CONSIDERING PRIORITY IN OVERLAY MULTICAST PROTOCOLS UNDER HETEROGENEOUS ENVIRONMENTS. In: Proceedings of the INFOCOMM 2006: 25th IEEE International Conference on Computer Communications, Barcelona, Catalunya, Spain, 23–29 April 2006. Piscataway: IEEE Press, 2006. p. 1–13.

[9] Broder, A. and M. Mitzenmacher. NETWORK APPLICATIONS OF BLOOM FILTERS: A SURVEY. Internet Mathematics, Vol. 1 (2002), No. 4, p. 485–509.

[10] Byers, J. and M. Luby, M. Mitzenmacher, A. Rege. A DIGITAL FOUNTAIN APPROACH TO RELIABLE DISTRIBUTION OF BULK DATA. ACM SIGCOMM Computer Communication Review (1998), p. 56–67.

[11] Calvert, K. and M. Doar, E. Zegura. MODELING INTERNET TOPOLOGY. IEEE Communications Magazine, Vol. 35 (1997), No. 6, p. 160–163.

[12] Castro, M. and P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, A. Singh. SPLITSTREAM: HIGH-BANDWIDTH MULTICAST IN COOPERATIVE ENVIRONMENTS. In: Proceedings of the nineteenth ACM symposium on Operating systems principles, Bolton Landing, NY, USA, 19–22 October 2003. New York: ACM Press, 2003. p. 298–313.

[13] Castro, M. and P. Druschel, A. Kermarrec, A. Rowstron. SCRIBE: A LARGE-SCALE AND DECENTRALIZED APPLICATION-LEVEL MULTICAST INFRASTRUCTURE. IEEE Journal on Selected Areas in Communications, Vol. 20 (2002), No. 8, p. 1489–1499.

[14] Chang, B. and Y. Shi, N. Zhang. HYMONET: A PEER-TO-PEER HYBRID MULTICAST OVERLAY NETWORK FOR EFFICIENT LIVE MEDIA STREAMING. In: Proceedings of AINA'06: the 20th International Conference on Advanced Information Networking and Applications, Vienna, Austria, 18–20 April 2006. Washington: IEEE Computer Society, 2006. p. 15–20.

[15] Chawathe, Y. and others. SCATTERCAST: AN ARCHITECTURE FOR INTERNET BROADCAST DISTRIBUTION AS AN INFRASTRUCTURE SERVICE. Doctoral dissertation. University of California, Berkeley, 2000.

[16] Chen, L. and T. Sun, G. Yang, M. Sanadidi, M. Gerla. MONITORING ACCESS LINK CAPACITY USING TFRC PROBE. Computer Communications, Vol. 29 (2006), No. 10, p. 1605–1613.

[17] Chiang, M. and S. Low, A. Calderbank, J. Doyle. LAYERING AS OPTIMIZATION DECOMPOSITION: A MATHEMATICAL THEORY OF NETWORK ARCHITECTURES. Proceedings of the IEEE, Vol. 95 (2007), No. 1, p. 255–312.

[18] Chou, P. and Y. Wu, K. Jain. PRACTICAL NETWORK CODING. In: Proceedings of 41st Annual Allerton Conference on Communication, Control, and Computing, Monticello, IL, USA, 1–3 October 2003.

[19] Chu, Y. and S. Rao, S. Seshan, H. Zhang. A CASE FOR END SYSTEM MULTICAST. IEEE Journal on Selected Areas in Communications, Vol. 20 (2002), No. 8, p. 1456–1471.

[20] Chu, Y. and S. Rao, H. Zhang. A CASE FOR END SYSTEM MULTICAST (KEYNOTE LOCATION). In: Proceedings of ACM SIGMETRICS, Santa Clara, California, United States, 18–21 June 2000. New York: ACM Press, 2000. Vol. 28, p. 1–12.

[21] Chun, B. and D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, M. Bowman. PLANETLAB: AN OVERLAY TESTBED FOR BROAD-COVERAGE SERVICES. ACM SIGCOMM Computer Communication Review, Vol. 33 (2003), No. 3, p. 3–12.

[22] Cohen, B. INCENTIVES BUILD ROBUSTNESS IN BITTORRENT. In: Proceedings of the First Workshop on Economics of Peer-to-Peer Systems, Berkeley, California, USA, 5–6 June 2003. University of California, Berkeley, 2003.

[23] Dabek, F. and R. Cox, F. Kaashoek, R. Morris. VIVALDI: A DECENTRALIZED NETWORK COORDINATE SYSTEM. In: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications, Portland, Oregon, USA, 30 August – 3 September 2004. New York: ACM Press, 2004. p. 15–26.

[24] Deering, S. MULTICAST ROUTING IN A DATAGRAM INTERNETWORK. Doctoral dissertation. Stanford University, Stanford, 1992.

[25] Deering, S. HOST EXTENSIONS FOR IP MULTICASTING. RFC 1112 (Standard). Sterling: Internet Engineering Taskforce (IETF), 1989.

[26] Diot, C. and B. Levine, B. Lyles, H. Kassem, D. Balensiefen. DEPLOYMENT ISSUES FOR THE IP MULTICAST SERVICE AND ARCHITECTURE. IEEE Network, Vol. 14 (2000), No. 1, p. 78–88.

[27] Dischinger, M. and A. Haeberlen, K. P. Gummadi, S. Saroiu. CHARACTERIZING RESIDENTIAL BROADBAND NETWORKS. In: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement, San Diego, California, USA, 27–31 August 2007. New York: ACM Press, 2007. p. 43–56.

[28] Erikson, H. MBONE: THE MULTICAST BACKBONE. Communications of the ACM, Vol. 37 (1994), No. 8, p. 54–60.

[29] Floyd, S. CONGESTION CONTROL PRINCIPLES. RFC 2914 (Best Current Practice). Sterling: Internet Engineering Taskforce (IETF), 2000.

[30] Floyd, S. and M. Handley, J. Padhye, J. Widmer. EQUATION-BASED CONGESTION CONTROL FOR UNICAST APPLICATIONS. In: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, Stockholm, Sweden, 2 September – 1 August 2000. New York: ACM Press, 2000. p. 43–56.

[31] Ganesh, A. and A. Kermarrec, L. Massoulié. SCAMP: PEER-TO-PEER LIGHTWEIGHT MEMBERSHIP SERVICE FOR LARGE-SCALE GROUP COMMUNICATION. In: Proceedings of the Third International COST264 Workshop on Networked Group Communication, London, UK, 7–9 November 2001. Ed. by J. Crowcroft and M. Hofmann. Berlin: Springer, 2001. Vol. 2233, p. 44–55.

[32] Gong, A. and G. Ding, Q. Dai, C. Lin. BULKTREE: AN OVERLAY NETWORK ARCHITECTURE FOR LIVE MEDIA STREAMING. Journal of Zhejiang University-Science A, Vol. 7 (2006), p. 125–130.

[33] Goyal, V. MULTIPLE DESCRIPTION CODING: COMPRESSION MEETS THE NETWORK. IEEE Signal Processing Magazine, Vol. 18 (2001), No. 5, p. 74–93.

[34] Guo, Y. and J. Zao, W. Peng, L. Huang, F. Kuo, C. Lin. TRICKLE: RESILIENT REAL-TIME VIDEO MULTICASTING FOR DYNAMIC PEERS WITH LIMITED OR ASYMMETRIC NETWORK CONNECTIVITY. In: Proceedings of the 8th IEEE International Symposium on Multimedia, San Diego, California, USA, 11–13 December 2006. Washington: IEEE Computer Society, 2006. p. 391–398.

[35] Habib, A. and J. Chuang. INCENTIVE MECHANISM FOR PEER-TO-PEER MEDIA STREAMING. Piscataway: IEEE Press, 2004. p. 171–180.

[36] Handley, M. and S. Floyd, J. Padhye, J. Widmer. TCP FRIENDLY RATE CONTROL (TFRC): PROTOCOL SPECIFICATION. RFC 3448 (Proposed Standard). Sterling: Internet Engineering Taskforce (IETF), 2003.

[37] He, Y. and I. Lee, L. Guan. DISTRIBUTED RATE ALLOCATION IN P2P STREAMING. In: Proceedings of the IEEE International Conference on Multimedia and Expo, Beijing, China, 2–5 July 2007. Piscataway: IEEE Press, 2007. p. 388–391.

[38] Hei, X. and C. Liang, J. Liang, Y. Liu, K. Ross. A MEASUREMENT STUDY OF A LARGE-SCALE P2P IPTV SYSTEM. Technical Report, Department of Computer and Information Science, Polytechnic University, Brooklyn, 2006.

[39] Ho, T. and M. Medard, J. Shi, M. Effros, D. Karger. ON RANDOMIZED NETWORK CODING. In: Proceedings of 41st Annual Allerton Conference on Communication, Control, and Computing, Monticello, IL, USA, 1–3 October 2003.

[40] Jelasity, M. and A. Montresor, G. Jesi. PEERSIM PEER-TO-PEER SIMULATOR. Available online: http://peersim.sourceforge.net.

[41] Jun, S. and M. Ahamad. INCENTIVES IN BITTORRENT INDUCE FREE RIDING. In: Proceeding of P2PECON '05: the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems, Philadelphia, Pennsylvania, USA, 22 August 2005. New York: ACM Press, 2005. p. 116–121.

[42] Kamvar, S. and M. Schlosser, H. Garcia-Molina. THE EIGENTRUST ALGORITHM FOR REPUTATION MANAGEMENT IN P2P NETWORKS. In: Proceedings of the 12th international conference on World Wide Web, Budapest, Hungary, 20–24 May 2003. New York: ACM Press, 2003. p. 640–651.

[43] Kohler, E. and M. Handley, S. Floyd. DATAGRAM CONGESTION CONTROL PROTOCOL (DCCP). RFC 4340 (Proposed Standard). Sterling: Internet Engineering Taskforce (IETF), 2006.

[44] Kostić, D. and A. Rodriguez, J. Albrecht, A. Vahdat. BULLET: HIGH BANDWIDTH DATA DISSEMINATION USING AN OVERLAY MESH. ACM SIGOPS Operating Systems Review, Vol. 37 (2003), No. 5, p. 282–297.

[45] Larzon, L.-A. and M. Degermark, S. Pink, L.-E. Jonsson, G. Fairhurst. THE LIGHTWEIGHT USER DATAGRAM PROTOCOL (UDP-LITE). RFC 3828 (Proposed Standard). Sterling: Internet Engineering Taskforce (IETF), 2004.

[46] Li, Z. MIN-COST MULTICAST OF SELFISH INFORMATION FLOWS. In: Proceedings of INFOCOM 2007: 26th IEEE International Conference on Computer Communications, Anchorage, Alaska, USA, 6–12 May 2007. Piscataway: IEEE Press, 2007. p. 231–239.

[47] Li, Z. and A. Mahanti. A PROGRESSIVE FLOW AUCTION APPROACH FOR LOW-COST ON-DEMAND P2P MEDIA STREAMING. In: Proceedings of the 3rd international conference on Quality of service in heterogeneous wired/wireless networks, Waterloo, Ontario, Canada, 7–9 August 2006. New York: ACM Press, 2006.

[48] Liang, J. and K. Nahrstedt. DAGSTREAM: LOCALITY AWARE AND FAILURE RESILIENT PEER-TO-PEER STREAMING. In: Proceedings of the 13th SPIE/ACM Multimedia Computing and Networking Conference, San Jose, California, 18–19 January 2006. Ed. by S. Chandra and C. Griwodz. Bellingham: SPIE, 2006. Vol. 6071, p. 224–238.

[49] Liang, J. and K. Nahrstedt. RANDPEER: MEMBERSHIP MANAGEMENT FOR QOS SENSITIVE PEER-TO-PEER APPLICATIONS. In: Proceedings of INFOCOM 2006: the 25th IEEE International Conference on Computer Communications, Barcelona, Catalunya, Spain, 23–29 April 2006. Piscataway: IEEE Press, 2006. p. 1–10.

[50] Liao, X. and H. Jin, Y. Liu, L. Ni, D. Deng. ANYSEE: PEER-TO-PEER LIVE STREAMING. In: Proceedings of INFOCOM 2006: 25th IEEE International Conference on Computer Communications, Barcelona, Catalunya, Spain, 23–29 April 2006. Piscataway: IEEE Press, 2006. p. 1–10.

[51] McCanne, S. and S. Floyd, et al. NETWORK SIMULATOR NS-2. The Vint project, available online: http://www.isi.edu/nsnam/ns.

[52] Meddour, D. and M. Mushtaq, T. Ahmed. OPEN ISSUES IN P2P MULTIMEDIA STREAMING. In: Proceedings of MULTICOMM'06, Istanbul, Turkey, 11–15 June 2006. Piscataway: IEEE Press, 2006. p. 43–48.

[53] Medina, A. and A. Lakhina, I. Matta, J. Byers. BRITE: AN APPROACH TO UNIVERSAL TOPOLOGY GENERATION. In: Proceedings of 9th International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Cincinnati, Ohio, USA, 15–18 August 2001. Washington: IEEE Computer Society, 2001. Vol. 1.

[54] Naicken, S. and B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, D. Chalmers. THE STATE OF PEER-TO-PEER SIMULATORS AND SIMULATIONS. ACM SIGCOMM Computer Communication Review, Vol. 37 (2007), No. 2, p. 95–98.

[55] Ng, T. and H. Zhang. A NETWORK POSITIONING SYSTEM FOR THE INTERNET. In: Proceedings of USENIX Conference, Boston, MA, USA, 27 June – 2 July 2004. Berkeley: USENIX Association, 2004.

[56] Nguyen, T. and A. Zakhor. MULTIPLE SENDER DISTRIBUTED VIDEO STREAMING. IEEE Transactions on Multimedia, Vol. 6 (2004), No. 2, p. 315–326.

[57] Nichols, K. and S. Blake, F. Baker, D. Black. DEFINITION OF THE DIFFERENTIATED SERVICES FIELD (DS FIELD) IN THE IPV4 AND IPV6 HEADERS. RFC 2474 (Proposed Standard). Sterling: Internet Engineering Taskforce (IETF), 1998.

[58] Ooi, W. DAGSTER: CONTRIBUTOR-AWARE END-HOST MULTICAST FOR MEDIA STREAMING IN HETEROGENEOUS ENVIRONMENT. In: Proceedings of SPIE, San Jose, California, USA, 19-20 January 2005. Bellingham: SPIE, 2005. Vol. 5680, p. 77.

[59] Padmanabhan, V. and H. Wang, P. Chou. RESILIENT PEER-TO-PEER STREAMING. In: Proceedings of the 11th IEEE International Conference on Network Protocols, Atlanta, Georgia, USA, 4–7 November 2003. Piscataway: IEEE Press, 2003. p. 16–27.

[60] Pai, V. and K. Kumar, K. Tamilmani, V. Sambamurthy, A. Mohr. CHAINSAW: ELIMINATING TREES FROM OVERLAY MULTICAST. In: Proceedings of International Workshop on Peer-to-Peer Systems (IPTPS), Ithaca, NY, USA, 24–25 February 2005. Ed. by M. Castro and R. van Renesse. Berlin: Springer, 2005. Vol. 3640, p. 127–140.

[61] Pianese, F. P2P LIVE MEDIA STREAMING: DELIVERING DATA STREAMS TO MASSIVE AUDIENCES WITHIN STRICT TIMING CONSTRAINTS. Master's thesis. Institut Eurecom, Sophia-Antipolis, 2004.

[62] Postel, J. USER DATAGRAM PROTOCOL. RFC 768 (Standard). Sterling: Internet Engineering Taskforce (IETF), 1980.

[63] Rabin, M. EFFICIENT DISPERSAL OF INFORMATION FOR SECURITY, LOAD BALANCING, AND FAULT TOLERANCE. Journal of the ACM, Vol. 36 (1989), No. 2, p. 335–348.

[64] Ramaswamy, L. and B. Gedik, L. Liu. A DISTRIBUTED APPROACH TO NODE CLUSTERING IN DECENTRALIZED PEER-TO-PEER NETWORKS. IEEE Transactions on Parallel and Distributed Systems, Vol. 16 (2005), No. 9, p. 814–829.

[65] Rodriguez, A. and C. Killian, S. Bhat, D. Kostić, A. Vahdat. MACEDON: METHODOLOGY FOR AUTOMATICALLY CREATING, EVALUATING, AND DESIGNING OVERLAY NETWORKS. In: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation, San Francisco, CA, USA, 29–31 March 2004. Berkeley: USENIX Association, 2004. Vol. 1, p. 20.

[66] Rosenberg, J. and J. Weinberger, C. Huitema, R. Mahy. STUN - SIMPLE TRAVERSAL OF USER DATAGRAM PROTOCOL (UDP) THROUGH NETWORK LOCATION TRANSLATORS (NATS). RFC 3489 (Proposed Standard). Sterling: Internet Engineering Taskforce (IETF), 2003.

[67] Rowstron, A. and P. Druschel. PASTRY: SCALABLE, DECENTRALIZED OBJECT LOCATION, AND ROUTING FOR LARGE-SCALE PEER-TO-PEER SYSTEMS. In: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, Heidelberg, Germany, 12–16 November 2001. Ed. by R. Guerraoui. Berlin: Springer, 2001. Vol. 2218, p. 329–350.

[68] Saroiu, S. and P. Gummadi, S. Gribble. MEASUREMENT STUDY OF PEER-TO-PEER FILE SHARING SYSTEMS. In: Proceedings of SPIE, San Jose, California, 20–25 January 2002. Bellingham: SPIE, 2001. Vol. 4673, p. 156–170.

[69] Shokrollahi, A. RAPTOR CODES. IEEE/ACM Transactions on Networking, Vol. 14 (2006), p. 2551–2567.

[70] Tamilmani, K. and V. Pai, A. Mohr. SWIFT: A SYSTEM WITH INCENTIVES FOR TRADING. In: Proceedings of the Second Workshop on Economics of Peer-to-Peer Systems, Harvard University, Cambridge, MA, USA, 4–5 June 2004. Harvard University, Cambridge, MA, USA, 2004.

[71] Tran, D. and K. Hua, T. Do. ZIGZAG: AN EFFICIENT PEER-TO-PEER SCHEME FOR MEDIA STREAMING. In: Proceedings of INFOCOM 2003: Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies, San Franciso, CA, USA, 30 March – 3 April 2003. Piscataway: IEEE Press. Vol. 2.

[72] Venkataraman, V. and P. Francis, J. Calandrino. CHUNKYSPREAD: MULTI-TREE UNSTRUCTURED PEER-TO-PEER MULTICAST. In: Proceedings of the 5th International Workshop on Peer-to-Peer Systems, Santa Barbara, CA, USA, 27–28 February 2006. Online at http://iptps06.cs.ucsb.edu/papers/Venkat-chunky06.pdf.

[73] Vishnumurthy, V. and P. Francis. ON HETEROGENEOUS OVERLAY CONSTRUCTION AND RANDOM NODE SELECTION IN UNSTRUCTURED P2P NETWORKS. In: Proceedings of INFOCOM 2006: 25th IEEE International Conference on Computer Communications, Barcelona, Catalunya, Spain, 23–29 April 2006. Piscataway: IEEE Press, 2006. p. 1–12.

[74] Vlavianos, A. and M. Iliofotou, M. Faloutsos. BITOS: ENHANCING BITTORRENT FOR SUPPORTING STREAMING APPLICATIONS. In: Proceedings of INFOCOM 2006: 25th IEEE International Conference on Computer Communications, Barcelona, Catalunya, Spain, 23–29 April 2006. Piscataway: IEEE Press, 2006. p. 1–6.

[75] Voulgaris, S. EPIDEMIC-BASED SELF-ORGANIZATION IN PEER-TO-PEER SYSTEMS. Doctoral dissertation. Vrije Universiteit, Amsterdam, 2006.

[76] Wang, D. and J. Liu. A DYNAMIC SKIP LIST-BASED OVERLAY FOR ON-DEMAND MEDIA STREAMING WITH VCR INTERACTIONS. IEEE Transactions on Parallel and Distributed Systems (preprint) (2007).

[77] Wang, F. and Y. Xiong, J. Liu. MTREEBONE: A HYBRID TREE/MESH OVERLAY FOR APPLICATION-LAYER LIVE VIDEO MULTICAST. In: Proceedings of ICDCS '07: 27th International Conference on Distributed Computing Systems, Toronto, Ontario, Canada, 25–29 June 2007. Piscataway: IEEE Press, 2007. p. 49.

[78] Wang, S. and H. Hsiao. FAST END-TO-END AVAILABLE BANDWIDTH ESTIMATION FOR REAL-TIME MULTIMEDIA NETWORKING. In: Proceedings of 8th Workshop on Multimedia Signal Processing, Victoria, Canada, 3–6 October 2006. Piscataway: IEEE Press, 2006. p. 415–418.

[79] Xie, H. and A. Krishnanmurthy, Y. R. Yang, A. Silberschatz. P4P: PROACTIVE PROVIDER PARTICIPATION FOR P2P. Department of Computer Science, Yale University, CT, USA, 2007. YALEU/DCS/TR-1377.

[80] Xu, D. and S. Kulkarni, C. Rosenberg, H. Chai. ANALYSIS OF A CDN–P2P HYBRID ARCHITECTURE FOR COST-EFFECTIVE STREAMING MEDIA DISTRIBUTION. Multimedia Systems, Vol. 11 (2006), No. 4, p. 383–399.

[81] Ying, L. and A. Basu. TRACEROUTE-BASED FAST PEER SELECTION WITHOUT OFFLINE DATABASE. In: Proceedings of the 8th IEEE International Symposium on Multimedia, San Diego, California, USA, 11–13 December 2006. Piscataway: IEEE Press, 2006. p. 609–614.

[82] Yiu, W. and X. Jin, S. Chan. CHALLENGES AND APPROACHES IN LARGE-SCALE P2P MEDIA STREAMING. IEEE MultiMedia, Vol. 14 (2007), No. 2, p. 50–59.

[83] Zhang, M. and J. Luo, L. Zhao, S. Yang. A PEER-TO-PEER NETWORK FOR LIVE MEDIA STREAMING USING A PUSH-PULL APPROACH. In: Proceedings of the 13th annual ACM international conference on Multimedia, Hilton, Singapore, 6–11 November 2005. New York: ACM Press, 2005. p. 287–290.

[84] Zhang, M. and Y. Xiong, Q. Zhang, S. Yang. ON THE OPTIMAL SCHEDULING FOR MEDIA STREAMING IN DATA-DRIVEN OVERLAY NETWORKS. In: Proceedings of GLOBECOM '06: Global Telecommunications Conference, San Francisco, California, USA, 27 November – 1 December 2006. Piscataway: IEEE Press, 2006. p. 1–5.

[85] Zhang, X. and J. Liu, B. Li, T. Yum. COOLSTREAMING/DONET: A DATA-DRIVEN OVERLAY NETWORK FOR EFFICIENT LIVE MEDIA STREAMING. In: Proceedings of IEEE INFOCOM 2005, Miami, FL, USA, 13–17 March 2005. Piscataway: IEEE Press, 2005.

[86] Zhou, M. and J. Liu. A HYBRID OVERLAY NETWORK FOR VIDEO-ON-DEMAND. In: Proceedings of ICC 2005: the IEEE International Conference on Communications, Seoul, Korea, 16–20 May 2005. Piscataway: IEEE Press, 2005. Vol. 2.