Delft University of Technology
Parallel and Distributed Systems Report Series

Libswift: the PPSPP Reference Implementation

Riccardo Petrocco, Cor-Paul Bezemer,
Johan Pouwelse, Dick H. J. Epema
r.petrocco@gmail.com

December 2014

Report number PDS-2014-004

ISSN 1387-2109

Published and produced by:
Parallel and Distributed Systems Group
Department of Software and Computer Technology
Faculty Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

Information about Parallel and Distributed Systems Report Series:
`reports@pds.ewi.tudelft.nl`

Information about Parallel and Distributed Systems Section:
`http://www.pds.ewi.tudelft.nl/`

© 2014 Parallel and Distributed Systems Group, Department of Software and Computer Technology, Faculty Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the publisher.

Contents

1 Introduction **3**
1.1 Libswift . . . 3
1.2 Outline . . . 3

2 Bins and Binmaps for Data Availability **4**
2.1 Binmap . . . 4
2.2 Availability . . . 5
2.2.1 Availability of Content Between Two Peers . . . 6
2.2.2 Availability of Content in the Swarm . . . 6

3 Libswift's State Machine **7**
3.1 Ping-Pong State . . . 8
3.2 Congestion Avoidance State . . . 8
3.3 Keep-Alive State . . . 9

4 Pull Mechanism **10**
4.1 RTT Estimation . . . 10
4.2 The Data Inter-Arrival Period . . . 11
4.3 Piece Selection . . . 11

5 Experiments **12**
5.1 Experimental Set-up . . . 12
5.2 Network efficiency . . . 13
5.3 Delay . . . 14
5.4 Packet Loss . . . 16
5.5 Performance Limitations . . . 16

6 Conclusions and Future Work **17**

List of Figures

1 Binmap – format: *bin number* (*layer*, *offset*) 5
2 Visual representation of storing the availability information in a binmap. 6
3 The availability array. 7
4 The availability array modifications after a new announce for a bin is received. The new announce is for bin 5, hence only the right hand side of the trees change. 7
5 LIBSWIFT's state machine. 8
6 An example of selecting the rarest content to request from a sending peer. 12
7 The behaviour of LIBSWIFT over a 5 Mbit/s link (the curves almost overlap). . . 13
8 The behaviour of LIBSWIFT with a a network link capacity ranging from 2 to 20 Mbit/s. 14
9 The behaviour of LIBSWIFT with different end-to-end delays over a 10 Mbit/s link. 15
10 The behaviour of LIBSWIFT depending on the packet loss rate over a 10 Mbit/s link. 15
11 The behaviour of LIBSWIFT with a 1.5% packet loss rate over a 10 Mbit/s link. . 16
12 The download progress over time of LIBSWIFT on high capacity networks for varying piece sizes and link capacities. 17

1 Introduction

The Peer-to-Peer Streaming Peer Protocol (PPSPP) is a transmission protocol for disseminating the same content to a group of interested parties in a streaming fashion. PPSPP is currently proposed as a standard and waiting for approval at the IETF [1]. PPSPP supports streaming of both pre-recorded (on-demand) and live audio/video content. It is based on the peer-to-peer (P2P) paradigm, where clients consuming the content are put on equal footing with the servers initially providing the content, to create a system where everyone can potentially provide upload bandwidth. As any standard protocol description, PPSPP [1] defines in great detail how the protocol must behave and how information is exchanged between communicating nodes. While PPSPP, including its behaviour and the message structure, has been clearly described in the draft, the details of the implementation are left to the developer.

To work at its best, a transmission protocol needs to provide several features such as a well defined and extensible communication mechanism, flow and congestion control and a good interface to external programs, and it has to make efficient use of the available resources, without being too intrusive. While the official PPSPP draft clearly describes many of those features such as the communication pattern, the message structure, and the congestion control, some details of the implementation are not presented as they are out of scope for the protocol description. In addition, the PPSPP draft presents several novel ideas and data structures for increasing its efficiency, of which the implementation details have not been discussed before.

1.1 Libswift

Over the last several years, we have designed and implemented the reference implementation of PPSPP and its novel ideas and data structures, called Libswift[1]. It is a useful reference to analyse and validate the protocol's properties and behaviour. Libswift can be used in two different ways: as a stand-alone client, or as a library, and in both of these ways it can run as a background process. This allows for an easy integration into existing applications, offering several interfaces, from a simple command line interface to more complex remote control interfaces, e.g. via HTTP or a socket. Libswift has been widely used for the experiments conducted by us and others during the years of research and development [2, 3, 4, 5, 6, 7, 8]. In earlier work, we have deeply analysed the performance of the initial implementation of Libswift [3]. Driven by the results of these experiments and experiences with real world deployment, we have made several design choices for Libswift, which are not directly relevant to the theoretical description of PPSPP, and therefore, are not discussed in the PPSPP draft. The aim of this document is to provide insight on some of these design choices and approaches to solving implementation problems. We describe these choices with the goal of assisting others who are interested in implementing the protocol.

1.2 Outline

In the remainder of this document we describe four aspects of Libswift's implementation. Section 2 describes the *binmap*, a novel data structure described in the PPSPP draft. While it has already been introduced and discussed in previous work [9, 3], Section 2 presents some implementation details and clearly defines the role of binmaps in Libswift. Section 3 presents Libswift's *state machine* which dictates when the client should send messages to any of its neighbours. This is particularly important in a highly dynamic environment such as P2P networks. Section 4 describes Libswift's pull mechanism. While PPSPP is designed to work

[1] http://libswift.org/

following both a push and pull mechanism [1], our reference implementation mostly relies on a pull mechanism. This choice is mostly motivated by the unstructured network in which LIBSWIFT operates, as peers are usually organised in a mesh type structure. In a structured network, e.g., in a live distribution scenario where peers are organised in a tree structure, a push mechanism might be prefered [10, 11]. Finally, Section 5 presents some experimental results of LIBSWIFT in challenging environments. Our testing environments are characterised by lossy, high latency and low bandwidth networks. The goal of Section 5 is to analyse and discuss the behaviour of LEDBAT [12], which is used as LIBSWIFT's congestion control algorithm.

2 Bins and Binmaps for Data Availability

In PPSPP [1], *bins*, or binary numbers, are introduced as a way of addressing ranges of content. Bins are the nodes and leaves of the *binmap*, a binary tree built on top of the content. This novel approach has been first designed and introduced in the early development stages of LIBSWIFT [9]. It is an optional feature and can be replaced with piece or byte ranges, which are actually the default means of requesting content given their simplicity. Our reference implementation, LIBSWIFT, supports both approaches, but internally it handles data only through bins and binmaps. Bins are used in conjunction with time values in order to keep track of incoming requests, outstanding requests, and data that has been sent and is waiting to be acknowledged. Binmaps are used to store information about the availability of content, to identify what has been retrieved, verified, announced by neighbour peers, and so on.

In this section, we do not describe those data structures in detail, as they have been extensively presented in PPSPP and in previous work [1, 9, 3]. Instead we give an overview of how binmaps fit in the context of the reference implementation, LIBSWIFT, and how they are used.

2.1 Binmap

In P2P systems, the content is divided into a number of pieces in order to optimise its distribution. Those pieces are represented by the leaves of the binmap, a binary tree built on top of the content. Each node of the tree, called *bin*, addresses either a single piece or a range of pieces of the content:

- Single pieces are addressed by leaf nodes, represented by bins with even numbers.
- Piece ranges are addressed by higher layer nodes, represented by bins with odd numbers.

To demonstrate the behaviour of a binmap, we will use the binmap depicted by Figure 1. This binmap is used to address content which is divided into four pieces (which are depicted by bin 0, 2, 4 and 6). Pieces are numbered from left to right, starting with 0 (i.e., the *relative offset*).

Bin numbers are expressed either in a compact form as a single integer, or in an extended form as the layer number and its relative offset, i.e., $(layer, offset)$. When addressing content through a bin number, we refer to the subtree of that bin, either a single piece or a range of pieces. More precisely, the bin, identified by (layer, offset), addresses the range of pieces with numbers in:

$$[offset \cdot 2^{layer}, (offset + 1) \cdot 2^{layer}),$$

Hence, the entire content of Figure 1 can be addressed either as bin *3*, or as bin *(2,0)*, which addresses piece range $[0, 4)$. The bins uniquely address the nodes of the tree, that can be either full, if the content is available, or empty. The bin number can be calculated based on its position in the tree:

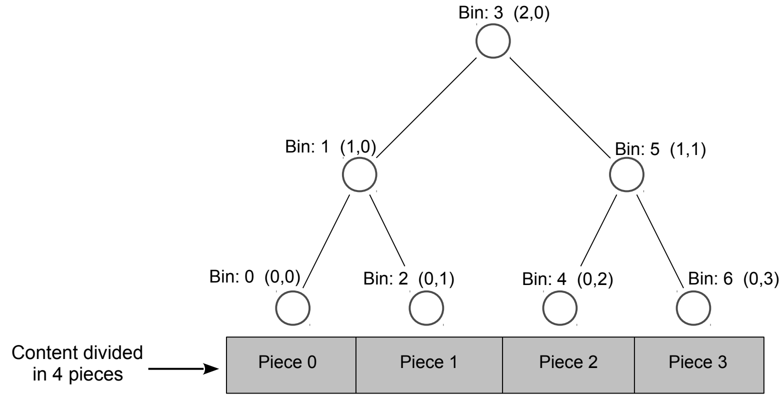


Figure 1: Binmap – format: *bin number* (*layer*, *offset*)

$$bin = 2^{(layer+1)} \cdot offset + 2^{layer} - 1$$

Since LIBSWIFT is written in C++, the bin representation allows for efficient binary operations which provide fast tree traversal and binmap comparison. Examples of such operations in C++, where b is the bin number, are the following:

- Check if the bin is a leaf of the tree: $!(b \; \& \; 1)$ (not an odd number)
- Return the beginning of the range of a bin: $(b \; \& \; (b+1)) \gg 1$, e.g. *bin(0,0)* from *bin(2,0)*.
- Return the length of the range of a bin: $(b+1) \; \& \; -(b+1)$, e.g. 4 for *bin(2,0)*.
- Return the sibling bin: $b \;\hat{}\; (b \;\hat{}\; (b+1)+1)$, e.g. *bin(1,1)* for *bin(1,0)*.

Each node of the tree, referenced by a bin, holds a status, which is either *full*, meaning that the range of content referenced by the bin has been retrieved, *empty*, the opposite of full, or *partially filled*, meaning that only part of the bin has been retrieved. Figure 2 gives a graphical representation on the statuses of the binmap's nodes. Figure 2 also shows another important characteristic of the binmap, the aggregation of information to higher layers. In this example, bins 0 and 2 have been retrieved, hence the information is propagated to bin 1 which is full. This allows to free the memory required to store the status of bin 0 and 2, making the binmap a very lightweight data structure when distributing large content. Furthermore, it allows to quickly determine if a range of content, or bin, has been partially or fully retrieved.

2.2 Availability

The standard choice for storing availability information in P2P systems, e.g., in BitTorrent [13], is the bitmap. In contrast, we decided to implement and develop the binmap data structure, as it provides several advantages over the standard bitmap, such as low hardware requirements, tree compression in case of redundant information (e.g., one side of the tree being completely full or empty), fast tree traversal, and easy comparison [9]. In LIBSWIFT, binmaps are used for two purposes: representing the availability of content 1) between two peers, and 2) in the swarm.

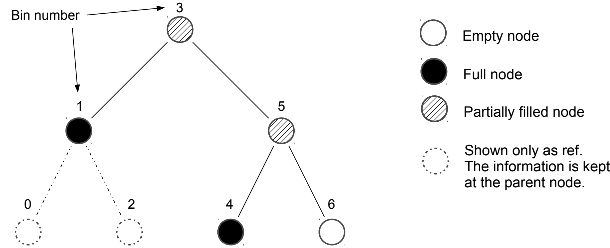


Figure 2: Visual representation of storing the availability information in a binmap.

2.2.1 Availability of Content Between Two Peers

Each peer stores one binmap representing the content it already retrieved and has locally available, and one binmap for each peer it connects to, representing the content that the other peer announces to have locally available. Requests for content and content availability announcements are performed by comparing the local binmap and the binmap the other peer announces.

2.2.2 Availability of Content in the Swarm

Binmaps are also used to represent the overall availability of content in a swarm. In a swarm, each piece of content is available 1 to N times, where N is the total number of peers in the swarm. The local view of the availability of pieces is given by what neighbour peers announce to have, hence it is limited by the number of connections a peer establishes, defined as n. In order to efficiently select the rarest pieces to retrieve, we introduce a novel data structure, called the *availability array*. A peer creates an availability array for each swarm it joins that has more than two active peers. It is an array of binmaps, where the index represents the data replication in the swarm. For example, if only one peer announces to have bin x, than x is filled at the binmap with index 1, and the bins that are filled in the binmap at the n-th index of the availability array represent the content that has been announced by n peers.

The availability array has an important property, which is the complementarity of its binmaps: the combination (defined as intersection) of all the binmaps in the availability array always results in one full binmap. An example of an availability array is provided in Figure 3

- n has already been defined,
- we call the index of the elements of the availability array *rarity index*, as it indicates the content replication in the swarm.

Imagine that the example of the availability array presented in Figure 3 refers to a swarm in which the content is divided into four pieces, following the example of Figure 1. It shows that no peer has announced to have bin 6 (*bin(0,3)*), as it is filled at index 0, one peer has bin 4 (*bin(0,2)*) and two peers have bin 1 (*bin(1,0)*), the first 2 pieces of content. The binmaps for indices 3 to n are not shown as the first 3 indexes are complementary and create a full binmap, meaning that the intersection of these binmaps results in a full tree. In other words, the replication of all 4 pieces of the content is known, hence the remaining indexes of the availability array necessarily point to empty binmaps.

The availability array is updated if either a new announcement is received, e.g, a connection with a new peer is established and it announces what it already retrieved, an existing peer updates

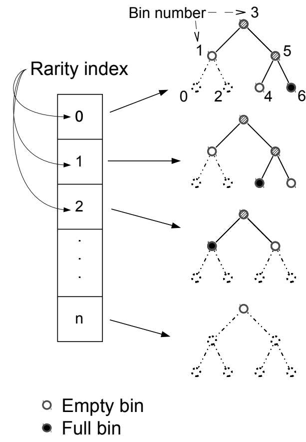


Figure 3: The availability array.

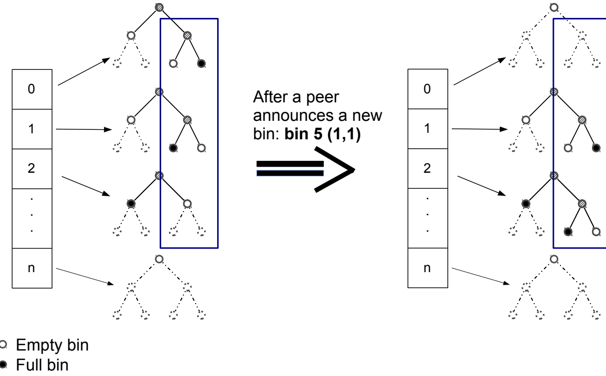


Figure 4: The availability array modifications after a new announce for a bin is received. The new announce is for bin 5, hence only the right hand side of the trees change.

its announcement, or if the connection with a peer is closed. Figure 4 depicts the situation after an announcement for bin 5 (*bin(1,1)*) is received. The availability array from Figure 3 is now updated as one peer has piece 3 available, and two peers have piece 2 available.

The size of the availability array, that is, the length of the array, is a customisable parameter and directly affects the accuracy of the availability view. By default, it is set to n, the maximum number of connections that will be initialised by a peer. In some circumstances, such as on hardware-limited devices which simultaneously participate in several popular swarms, it can be reduced to a lower size in order to lower memory and computational requirements. In case more connections are established than can be represented by the availability array, some loss of accuracy occurs as the omitted information is combined on the last index of the array.

3 Libswift's State Machine

The official PPSPP draft [1] defines how the protocol should react to general dynamics of a P2P network environment, such as peers unexpectedly leaving the system, but it does not define how the protocol should behave in other circumstances, such as intermitted content requests. This section aims to clarify how in LIBSWIFT we determine when to send data, when to wait, and when to close a connection. Contrary to the client-server scenario, P2P protocols need more than a single sending state in order to provide efficient data transmission. This is mainly caused by the absence of a single uninterrupted byte stream: generally, peers want to keep active connections with each other, as even peers that do not have any useful data to offer might retrieve new pieces of content at any time. In LIBSWIFT, peers have three different sending states:

- *Ping-pong*
- *Congestion avoidance*
- *Keep-alive*

For each connection a peer establishes, it switches between the three sending states based on the requests and acknowledgements it receives. Figure 5 presents the state diagram of the state machine. In the remainder of this section we analyse each sending state in detail.

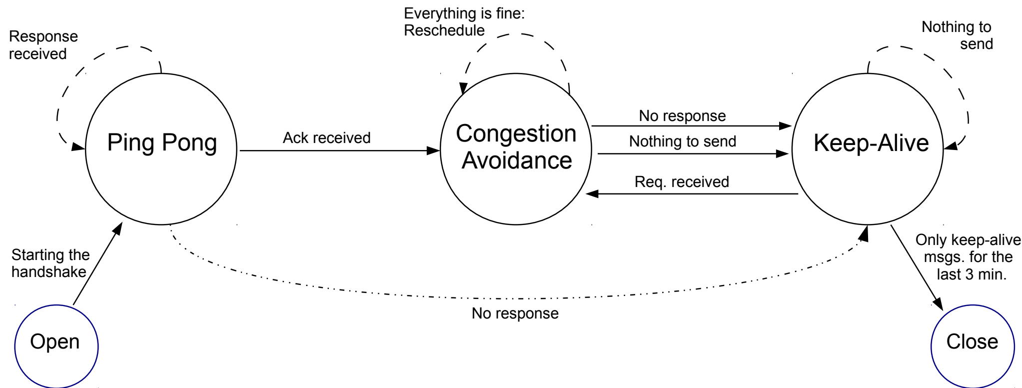


Figure 5: LIBSWIFT's state machine.

3.1 Ping-Pong State

When initialising a connection, during the initial handshake, a peer starts in the ping-pong state. During this state, the peer sends a packet and waits for a reply before sending the following packet. The sending state changes when either no reply is received within a certain period, moving to the keep-alive state, or when an acknowledgement is received, in which case the state changes to congestion avoidance.

3.2 Congestion Avoidance State

The congestion avoidance state is a sending state in which a peer is actively sending data to another peer in the network. It is based on a congestion window (cw), defined as the number of allowed outstanding packets at any time. It is divided into two stages:

1. Initially a *slow start* approach is applied, as defined in [14],
2. then the state moves to the *congestion control* stage.

During the slow-start stage, cw is initialised to one packet, and increased by one for each acknowledgement received. The congestion window is linearly increased until one of the following happens:

- An acknowledgement is not received on time, following the retransmission time-out presented in [14]. Result: cw is halved.
- The following applies: $si < 100ms$, where si is the sending interval, defined as rtt/cw, and rtt is the estimated round-trip time (RTT) average. Result: the stage moves from the slow-start stage to the congestion control stage.

The congestion control stage follows the Low Extra Delay Background Transport (LEDBAT) standard [12], which adjusts the sending rate based on the one-way delay samples (owd). The owd is measured over the data path, hence it is the time required for data packets to arrive

from the sender to the receiver. LEDBAT is a delay-based congestion control which aims to utilise the available bandwidth without interfering with concurrent traffic, and has been selected given its widespread adoption [15, 13, 16, 17]. Previous work provides an in-depth analysis of LEDBAT [18, 19, 20].

The connection will stay in the congestion avoidance state and switches to the keep-alive state if one of the following conditions occurs:

- No packet has been received during the last $8 \cdot rtt$ seconds.
- si is greater than the maximum of 4 seconds and $4 \cdot rtt$.
- There are no outstanding requests to be served.

For the first two conditions, the state switches to keep-alive as expected, resetting the cw and other parameters, while the third condition presents an exception: It differs from the other two as the connection might still be active, and should not be restored as it would take longer to redetermine the optimal sending rate. There might not be any outstanding requests for three different reasons:

1. The peer's requests might be lost, if on a lossy network such as wireless or mobile links, or they might be dropped by a router if the link is heavily congested.
2. It frequently occurs during a live streaming transmission when the peer's download speed is higher than the video stream bitrate. In this case the peer can retrieve the content faster than it is generated at the source, therefore it will not have enough requests for content to fully utilise its link.
3. The requesting peer might already have the pieces announced by the sending peer, but considering the P2P environment, new pieces might become available at any time, hence the connection must be kept alive.

Hence, in case new requests are received within a short period of time, set to $4 \cdot rtt$ as default but possibly changed based on the specific environment, the sending state switches back to the congestion avoidance state resuming the transmission at the same rate as before.

3.3 Keep-Alive State

The keep-alive state and its transition to other states are presented in the official draft [1]. We discuss it in this document for clarity. If there is nothing to be sent, as no request has been received, a peer stays in the keep-alive state. In this state, a peer sends packets with no payload, containing only the keep-alive signal, to notify the other peer of his presence. A typical scenario is a leecher communicating with a seeder. In this scenario the leecher stays in keep-alive state while the seeder switches between sending states based on the leecher's incoming requests and acknowledgements. If two connected peers are both in keep-alive state, e.g., two seeders, the interval between keep-alive packets increases over time, up to a maximum of 3 minutes, after which the connection is closed. A peer switches to the congestion avoidance state once it receives a new request for content.

4 Pull Mechanism

Our reference implementation applies a pull mechanism in order to retrieve data from other peers. By directly requesting the data, the receiver peer (RP) has control over what will be received from any of his neighbours, referred to in this section as sending peers (SPs). In the official draft [1] there is no description of the specific pull mechanism strategy, as it is left to the implementer to decide, hence this section presents the approach implemented in Libswift.

The general goal in a P2P network is to retrieve the content as soon as possible, which is achieved by maximising the rate of incoming data. Hence, the aim of the RP is to guarantee that the SP always has an outstanding queue of requests that need to be served, assuring that it never stays idle. If the SP runs out of requests to serve, it will switch to the *keep-alive* state and the flow of data is halted until new requests are received (see Section 3.2). On the other hand, the RP should not overwhelm the SP with requests, as the number of requests affects the reaction time to sudden network changes, such as a congested links or peers leaving the network. Furthermore, some other aspects need to be considered, which might affect the efficiency of data retrieval for the individual peer or for the entire network. For example, in a streaming context, it was shown to be more beneficial to retrieve the content at a download rate close to the stream bit-rate [21], rather than retrieving it as fast as possible like in a file-sharing context.

In the remainder of this section we present the strategies implemented in Libswift in order to optimise the rate of incoming traffic in every scenario, whether it is static or dynamic content. It is left to the implementer to choose the strategy that best fits his needs.

4.1 RTT Estimation

In many protocols which apply a pull mechanism, the standard way of determining the amount of content that should be requested relies on the time required for a request to be served, hence by estimating the round-trip time (RTT). In Libswift, we do not rely exclusively on the RTT estimation, as it might be unreliable for the RP. In LEDBAT, only the peer which is actively sending data has a correct estimation of the RTT, as it is calculated based on the received acknowledgements (acks). Since acks are sent by the RP as soon as possible [12], the SP calculates the RTT as the time interval between sending the data packet and receiving its acknowledgement. The SP needs to keep a correct RTT estimation as it is used in Libswift to determine the retransmission time-out, using the same approach as TCP [22].

On the other hand, the RP cannot estimate the RTT accurately. The only way for the RP to calculate the RTT would be to analyse the time elapsed between requesting a chunk of content and receiving it. This time is influenced by several factors, as the time needed for the SP to serve the request depends on:

- The number of outstanding requests.
- The processing time at the SP, as it might vary depending on the number of hashes that need to be sent.
- The number of concurrent streams, as the SP divides the available bandwidth on its upload link over all the RPs from the swarms in which it is participating.

All this information is unavailable to the RP, hence the RTT cannot be used as the main parameter to determine how much content should be requested.

4.2 The Data Inter-Arrival Period

In Libswift, we rely on the data inter-arrival period, at the RP, to determine how much content should be requested by the RP to the SP. The data inter-arrival period (dip) is defined as the amount of time elapsed between successive arrivals of packets containing data messages. Hence, the dip is the most accurate way of estimating the actual sending rate of the SP. The RP determines whether to request content to the SP based on Algorithm 1.

Algorithm 1 Determine the amount of data to be requested.

```
1: rt ← max(1s, 4 ∗ RTT)              ▷ rt: request time, usually 1 second
2: rp ← max(1, rt/dip)                ▷ rp: request packets, number of pkts in rt sec.
3: ra ← max(0, rp − ro)               ▷ ra: allowed requests; ro: outstanding req.
4: rs ← min(ra, rate allowed)         ▷ rs: req. size; rate allowed: take rate limits into account
5: if rs > rg then                    ▷ rg: req. granularity, avoids req. fragmentation.
6:     req ← Pick(rs)
7:     if req = 0 then
8:         return                     ▷ end-game: nothing can be requested
9: return req
```

The first line of Algorithm 1 might be misleading as the RTT estimation is used. We consider the RTT estimation only for extremely slow connections where the RTT is greater than 250ms. The RTT value is always initialised during the handshake, as peers reply to handshakes immediately, and is updated only through acks. Therefore, even peers communicating with seeders have a rough estimation of the RTT, but it is never updated and unreliable. The RTT is only used to guarantee that there are enough outstanding requests, while lines 1 to 5 of Algorithm 1 calculate how much content to request considering the number of outstanding requests, the potential rate limit, and the desired requests granularity (a configurable parameter). The aim of Algorithm 1 is to limit the size of the request, or bin layer, while the *Pick* function of line 6, described in the following section, determines which bin to request.

4.3 Piece Selection

In Libswift we provide different piece-picking algorithms depending on the required retrieval pattern, e.g., static content (file-sharing), time-critical content (VoD) or dynamically generated content (live stream). The piece-picking function *Pick* (line 6 of Algorithm 1) is called each time new content needs to be requested from another peer. Libswift features four piece-picking algorithms, which differ in the way they prioritise ranges of content:

1. *Sequential*, the simplest piece-picker as it just retrieves the content sequentially. A small variation can be introduced by setting a random variance used for traversing the binmap.

2. *Rarest-first*, selects the rarest bin, within the limits of rs from Algorithm 1, that the SP has to offer across the entire content.

3. *Video-on-Demand*, divides the content in different sets of priorities, from high to low, selecting the rarest content the SP has to offer starting from the highest priority set. The algorithm has been presented in [3].

4. *Live stream*, returns the largest bin available from the current playback position in the stream.

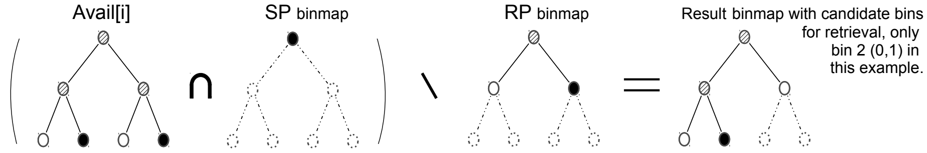


Figure 6: An example of selecting the rarest content to request from a sending peer.

The rarest-first and Video-on-Demand algorithms apply the same strategy in order to select the rarest available bin that can be retrieved from the SP. They only differ in the range of pieces which are considered at each execution. They iterate through the availability array (*Avail*), described in Section 2.2.2, from the lowest to the highest index. While iterating through *Avail*, from the rarest to the most popular content, the algorithm performs a three-way comparison with the binmap representing the content that the SP has to offer, and the binmap representing what the RP has already retrieved, both defined in Section 2.2.1. The three-way comparison can be expressed using the set intersection and complement as $(\text{Avail[i]} \cap \text{SP}_{binmap}) \setminus \text{RP}_{binmap}$, where i is the index of the availability array addressing what i neighbour peers announce to have. An example of selecting the rarest content from a seeder is presented in Figure 6. In this example RP selects the second piece of content, bin 2 or (0,1), as it is the only full bin in the resulting binmap. In case the resulting binmap has more than one full bin, the RP selects the bin to request depending on which piece picking algorithm is used, e.g., the bin closer to the current playback position or randomly.

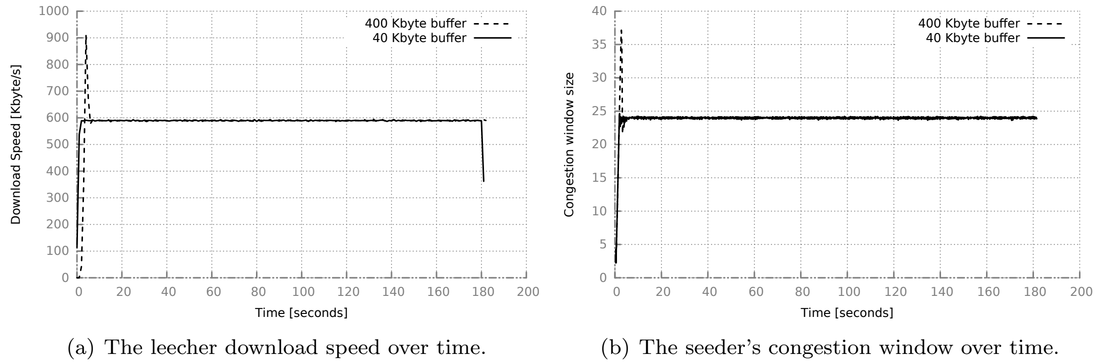
As a final note regarding the pull mechanism, it should be noted that the number of requests sent to each peer also influences the rate at which data will be retrieved, as lowering the number of requests automatically leads to a lower link utilisation. This characteristic can be exploited in order to prioritise specific connections over others, e.g., connections which feature a low delay can be dedicated to the most time-critical requests (as for missing content close to the current playback position).

5 Experiments

In this section we provide an evaluation of Libswift. We analyse its behaviour by emulating different network conditions, and present the results.

5.1 Experimental Set-up

To simplify our analysis we emulate only two peers exchanging content. Analysing only two peer allows us to correctly identify the behaviour of the protocol, highlighting its downsides when operating in challenging environments. In previous work we have analysed the behaviour of Libswift in big swarms [3]. In our emulations one peer is the content provider, the *seeder*, while the other peer is the content consumer, the *leecher*. The leecher is initialised once the seeder loads the content, has generated the required hashes, and is ready to distribute it. The leecher network link speed is set to 1 Gbit/s (hence it is practically not limited) and we vary the connection speed limit on the seeder's network link. We configure the end-to-end delay between the leecher and seeder to be 60ms. The content is 100 MByte in size and is divided in 1 Kbyte chunks, the default value.



(a) The leecher download speed over time. (b) The seeder's congestion window over time.

Figure 7: The behaviour of LIBSWIFT over a 5 Mbit/s link (the curves almost overlap).

We consider the *completion time* as the most important metric to evaluate the performance of our protocol. We present the *download progress* for each experiment, showing both the completion time and the download progress over time. We also present the *download speed* over time, discussing the efficiency of the protocol in exploiting the available bandwidth. The download speed is directly related to LEDBAT's *congestion window*, as the congestion control dictates the sending rate, hence we also present it in some experiments to clarify LIBSWIFT's behaviour.

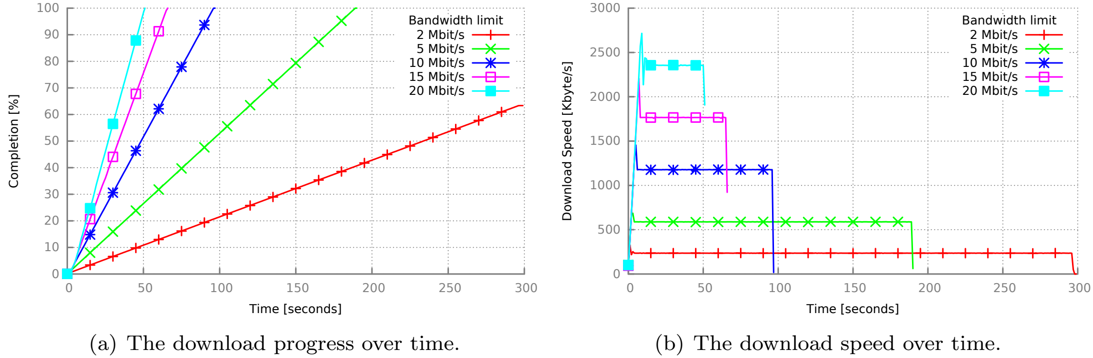
Our experimental environment uses LXC containers[2] and netem[3] to emulate various network conditions. An LXC container is a lightweight environment which provides full resource isolation and resource control for an application or a system. LXC containers can be considered a lightweight virtualization method for running multiple isolated Linux systems (containers) on the same host. All experiments were conducted on a single machine with a single dual Intel Xeon CPU 2.40GHz and 8GB of memory, running Debian with kernel 3.12. We used a customised experiment framework, called Gumby[4], to conduct the experiments in an automated fashion. For the specific container setup details, we refer the reader to the Gumby documentation[5].

We start each peer in the experiment in its own container. To emulate various network conditions, we limit outgoing and incoming traffic to the containers using netem. Netem allows the definition of network traffic control rules on a network device. By setting these rules for the network devices inside the container, we can specify traffic rules for one peer without affecting the network speed of the other peer in the experiment. For every container, we can specify the download and upload rate in bits/s, the delay in ms and the percentage of lost packets on the outgoing traffic. In addition, we can specify the burst sizes for the incoming and outgoing connection.

5.2 Network efficiency

In this section we show how LIBSWIFT efficiently exploits the available bandwidth. Figure 7 shows its performance with a network link limited to 5 Mbit/s. Figure 7(a) presents the seeder's upload speed while Figure 7(b) presents its congestion window. Both metrics are important to analyse the protocol's performance. The stability of the sender's congestion window shows that

[2] `https://linuxcontainers.org/`
[3] `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`
[4] `https://github.com/tribler/gumby`
[5] `https://github.com/Tribler/gumby/tree/devel/experiments/libswift`



(a) The download progress over time.

(b) The download speed over time.

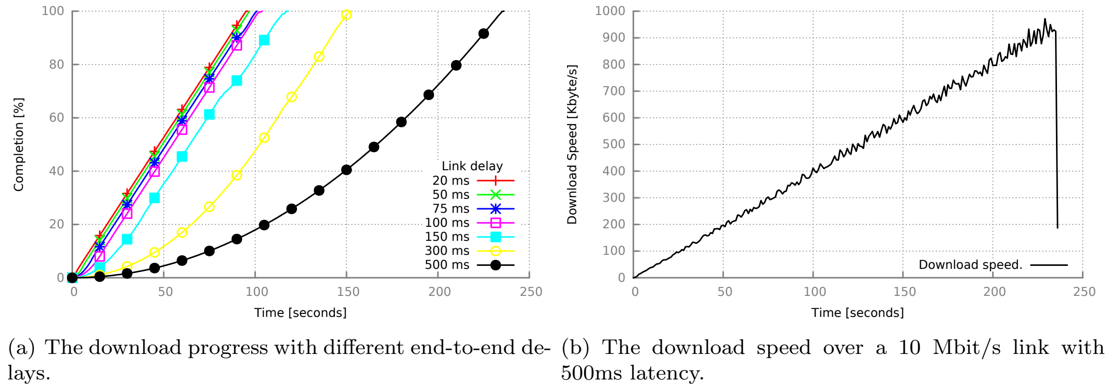
Figure 8: The behaviour of LIBSWIFT with a a network link capacity ranging from 2 to 20 Mbit/s.

LEDBAT correctly identifies the optimal sending rate, which results in a stable download speed for the leecher. In both figures we present two experiments which show how buffers on the data path, usually introduced by routers, influence the time required to reach the optimal network speed. A stable download speed is especially important in streaming applications, where it is used to determine if it is possible to provide a constant playback for the end user. LEDBAT, hence LIBSWIFT, increases the sending rate until either a packet is lost, or the one-way-delay starts to increase. This behaviour can be seen at the beginning of the transmission before the sending rate reaches a stable value, causing the small initial spike shown in Figures 7(a) and 7(b). The intensity of the spike, and the time required to reach a stable speed depend on two factors, the end-to-end latency and the intermediate buffers. The higher the latency, the longer it will take to receive acknowledgements for sent data. Hence it will take longer to adapt the sending rate. This behaviour is investigated in the following section. The buffers present on the data path, e.g., router's buffers, interfere with the sending rate calculation. Small buffers lead to a quicker adaptation, while large buffers delay the adaptation process. In Figure 7 we present the effect of changing the router's buffer from 40Kbyte, the continuous line, to 400Kbyte, the dashed line. While a 40KByte buffer quickly leads to the optimal link utilization, the larger 400Kbyte buffer introduces an initial spike, as LEDBAT requires more time to determine the proper upload speed. Figure 8 presents LIBSWIFT's behaviour with different bandwidth capacities. This experiment shows that the sending rate properly adapts to the available bandwidth.

5.3 Delay

We investigate the effect of changing the delay between the seeder and the leecher. The leecher's delay is fixed at 20ms, both on the outgoing and incoming packets, while the seeder's delay on the data path varies from 25ms to 500ms depending on the experiment.

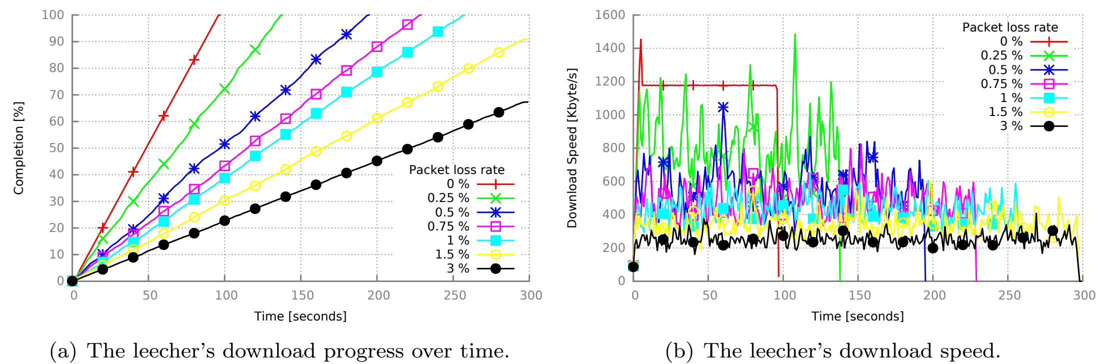
Figure 9(a) shows the download progress over time, and clearly illustrates how a longer delay influences the time required to reach the optimal speed and fully utilise the network link. The delays showen in Figure 9(a) are the one-way-delay (owd) values on the data path, as those influence LEDBAT, hence the protocol's behaviour. The difference in download progress for owd delays lower than 100ms is rather marginal. For end-to-end delays greater that 150ms the congestion control, LEDBAT [12], requires a long time to adapt to the available bandwidth. Figure 9(b) shows how with a delay of 500ms it never actually reaches a sending rate of 10



(a) The download progress with different end-to-end delays.

(b) The download speed over a 10 Mbit/s link with 500ms latency.

Figure 9: The behaviour of LIBSWIFT with different end-to-end delays over a 10 Mbit/s link.



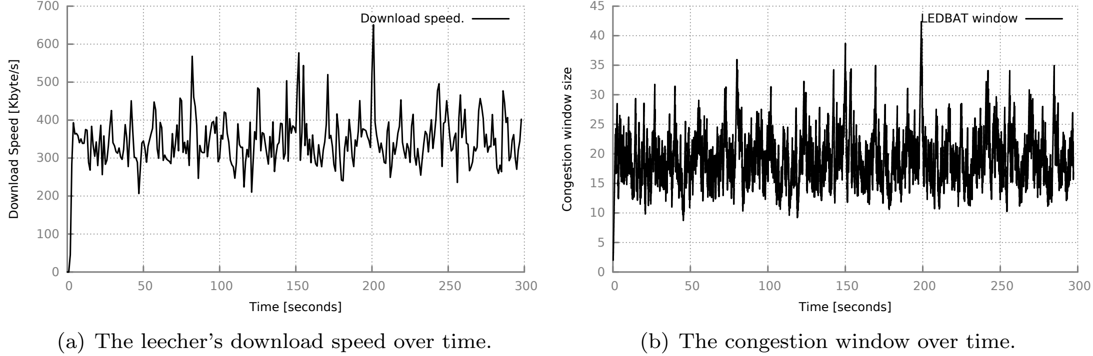
(a) The leecher's download progress over time.

(b) The leecher's download speed.

Figure 10: The behaviour of LIBSWIFT depending on the packet loss rate over a 10 Mbit/s link.

Mbit/s when sending the 100 MByte test file.

This behaviour is the result of LEDBAT's design decisions, as it was designed as a low delay congestion control. LEDBAT tries to reach a *target* delay on the data path. The target delay is defined as the maximum queueing delay that LEDBAT may introduce in the network. With large delays, LEDBAT doesn't increase the sending rate as fast as with low delays, in the eventuality that the higher delay values might be caused by other traffic. LEDBAT's RFC specifies that the target delay needs to be less than 100ms, which is the same value used in LIBSWIFT and in the UTP protocol [15], used by other P2P protocols such as BitTorrent [13]. This explains the performance degradation when the delay is greater than 100ms, see Figure 9(a). In the real world, a target delay value of 100ms has proven to be a suitable candidate for peer-to-peer communications. The main exception is for high latency network links, such as satellite links, where a longer time is required to reach optimal download rates.



(a) The leecher's download speed over time.

(b) The congestion window over time.

Figure 11: The behaviour of LIBSWIFT with a 1.5% packet loss rate over a 10 Mbit/s link.

5.4 Packet Loss

An important performance characteristic that needs to be considered when using LEDBAT, as we do in LIBSWIFT, is its behaviour on a lossy network. LEDBAT does not distinguish between packet losses and congestion. Hence, packet losses are interpreted as a sign of a congested network link, causing the congestion window to be reduced, therefore decreasing the sending rate. Figure 10 shows the performance degradation related to the packet loss rate. We present the result of 7 experiments, each characterized by a different packet loss rate. The packet loss rates we investigate are between 0% and 3%. Figure 10(a) presents the effects on the download progress, while Figure 10(b) clearly illustrates how the protocol is able to reach a stable sending rate only when the link presents no packet loss.

Figure 11 shows in more detail the relation between the leecher's download rate (Figure 11(a)) and the seeder's congestion window (Figure 11(b)). Here, the performance degradation on lossy networks becomes clear, as opposed to the behaviour on a perfect network (see Figure 7).

5.5 Performance Limitations

The biggest limitation of LIBSWIFT is the computational requirement of the content integrity protection scheme, which is particularly evident on high capacity network links. LIBSWIFT applies a data atomic principle, which states that every packet transferring data should hold all the information needed to verify the data it contains. This means that for every packet containing a data chunk, the sender needs to include all the hashes required to verify it. A peer starts retrieving the content with only the root hash as content identifier, which is therefore the only hash it can initially use to verify incoming data. The receiver, after retrieving a data packet, needs to verify the data against a valid hash, might it be the root hash or a hash of an intermediate node which has been previously validated. Only after validation, it will send an acknowledgement to the sender and it will notify other peers of the newly downloaded chunk. This content validation is the most resource consuming process in LIBSWIFT, taking up to 50% of its execution time [6].

The default piece/chunk size in PPSPP is 1 Kbyte, as it fits well within the boundaries of the average Ethernet MTU, which is 1500 bytes. Furthermore, a small piece size has also proven to be a good candidate for live streaming systems [23]. LIBSWIFT is a single-threaded application, and the performance it can achieve with the default piece size of 1 KByte is limited by the

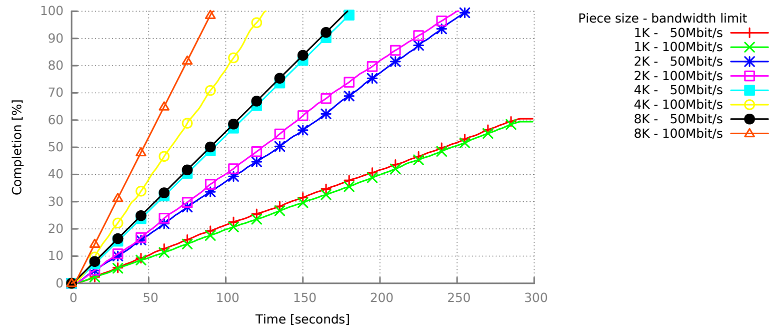


Figure 12: The download progress over time of LIBSWIFT on high capacity networks for varying piece sizes and link capacities.

capabilities of the CPU. While it should not be a limiting factor for live streams, as the protocol can easily retrieve a high bitrate stream on a low-end CPU, 20 Mbit/s in our experiments, the piece size might be increased to reach higher download speeds.

Figure 12 presents the download progress over two high speed links, 50 Mbit/s and 100 Mbit/s, when varying the piece size from 1 Kbyte to 8 Kbyte. For the experiments presented in Figure 12, the test file is 1 GByte in size and both clients are executed on the same CPU. Using the default piece size of 1 Kbyte does not allow LIBSWIFT to fully exploit the available bandwidth when running on a low-end CPU. This behaviour can be observed in Figure 12, where the lines for 1 Kbyte piece size show how the protocol only retrieves ~60% of the 1 GByte test file during the first 300 seconds. On this specific CPU, to fully utilise a 50 Mbit/s link, the piece size should be increased to 4 Kbyte, while a piece size of 8 Kbyte is a better candidate for links with (more than) 100 Mbit/s capacity.

The downside of increasing the chunk size is that in case a packet containing data is dropped in the network, more content needs to be retransmitted. Considering the case of a link with 1500 Byte MTU, if the content is divided in 1 Kbyte chunks and one packet is lost, only that packet will need to be retransmitted. On the other hand, if the content is divided in 4 Kbyte chunks, the transmission of a single data chunk involves up to 4 packets, including the required hashes and other messages, and the loss of a single packet might require the retransmission of all 4 packets.

6 Conclusions and Future Work

The Peer-to-Peer Streaming Peer Protocol (PPSPP) is a P2P transmission protocol for disseminating the same content to a group of interested parties in a streaming fashion. PPSPP is currently proposed as a standard and is waiting approval by the IETF. While the official Peer-to-Peer Streaming Peer Protocol draft defines in great detail how the protocol must behave, the details of the implementation are left to the developer.

In this document, we have discussed several design choices for LIBSWIFT, our reference implementation of PPSPP. First, we have discussed the implementation of the availability map, our data structure for representing the overall availability of content in a swarm. Second, we have discussed how we have made LIBSWIFT suitable for reacting to the dynamics in a P2P network environment. Finally, we discussed the pull mechanism used by LIBSWIFT to retrieve data from other peers. In addition, we have presented a number of experiments in which we demonstrate

the behaviour of LIBSWIFT under various network conditions.

Future work involves fine tuning LIBSWIFT and developing a simple GUI for running and configuring LIBSWIFT.

References

[1] A. Bakker, R. Petrocco, and V. Grishchenko, "Peer-to-peer streaming peer protocol (ppspp)," June 2014. IETF draft. 3, 4, 7, 9, 10

[2] A. Bakker, R. Petrocco, M. Dale, J. Gerber, V. Grishchenko, D. Rabaioli, and J. Pouwelse, "Online video using bittorrent and html5 applied to wikipedia," in *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*, pp. 1–2, IEEE, 2010. 3

[3] R. Petrocco, J. Pouwelse, and D. H. Epema, "Performance analysis of the libswift p2p streaming protocol," in *Peer-to-Peer Computing (P2P), 2012 IEEE 12th International Conference on*, pp. 103–114, IEEE, 2012. 3, 4, 11, 12

[4] R. Petrocco, M. Capotă, J. Pouwelse, and D. H. Epema, "Hiding user content interest while preserving p2p performance," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pp. 501–508, ACM, 2014. 3

[5] P. M. Eittenberger, K. M. Schneider, and U. R. Krieger, "Performance evaluation of next generation content delivery proposals," in *Next Generation Mobile Apps, Services and Technologies (NGMAST), 2013 Seventh International Conference on*, pp. 136–141, IEEE, 2013. 3

[6] G. V. Sanchez, "Libswift-ppspp information centric router: Sha1 accelerator," Master's thesis, Delft University of Technology, Delft, Netherlands, August 2013. 3, 16

[7] V. Grishchenko, F. Osmani, R. Jimenez, J. Pouwelse, and H. Sips, "On the design of a practical information-centric transport," tech. rep., PDS Technical Report PDS-2011-006, 2011. 3

[8] F. Osmani, V. Grishchenko, R. Jimenez, and B. Knutsson, "Swift: The missing link between peer-to-peer and information-centric networks," in *Proceedings of the First Workshop on P2P and Dependability*, P2P-Dep '12, (New York, NY, USA), pp. 4:1–4:6, ACM, 2012. 3

[9] J. P. Victor Grishchenko, "Binmaps: Hybridizing bitmaps and binary trees," tech. rep., TUDelft, The Netherlands, PDS Technical Report PDS-2011-005, 2011. 3, 4, 5

[10] D. Wu, Y. Liu, and K. W. Ross, "Queuing network models for multi-channel p2p live streaming systems," in *INFOCOM 2009, IEEE*, pp. 73–81, IEEE, 2009. 4

[11] Q. Huang, H. Jin, and X. Liao, "P2p live streaming with tree-mesh based hybrid overlay," in *Parallel Processing Workshops, 2007. ICPPW 2007. International Conference on*, pp. 55–55, IEEE, 2007. 4

[12] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind, "Low extra delay background transport (ledbat), rfc6817," Dec. 2012. 4, 8, 10, 14

[13] "Bittorrent protocol 1.0." `http://www.bittorrent.org`. 5, 9, 15

[14] M. Allman, V. Paxson, and E. Blanton, "Tcp congestion control, rfc5681," 2009. 8

[15] "utorrent transport protocol." http://www.bittorrent.org/beps/bep_0029.html. 9, 15

[16] "Libtorrent." http://www.libtorrent.org/. 9

[17] "the utorrent bt client." http://www.utorrent.com/. 9

[18] D. Rossi, C. Testa, and S. Valenti, "Yes, we ledbat: Playing with the new bittorrent congestion control algorithm," in *Passive and Active Measurement*, pp. 31–40, Springer, 2010. 9

[19] C. Testa and D. Rossi, "On the impact of utp on bittorrent completion time," in *Peer-to-Peer Computing (P2P), 2011 IEEE International Conference on*, pp. 314–317, IEEE, 2011. 9

[20] G. Carofiglio, L. Muscariello, D. Rossi, and S. Valenti, "The quest for ledbat fairness," *CoRR*, vol. abs/1006.3018, 2010. 9

[21] L. D. et al., "Peer selection strategies for improved qos in heterogeneous bittorrent-like vod systems," ISM '10, (Washington, DC, USA), pp. 89–96, 2010. 10

[22] V. Paxson and M. Allman, "Computing tcp's retransmission timer, rfc2988," 2000. 10

[23] J. J. D. Mol, J. A. Pouwelse, M. Meulpolder, D. H. J. Epema, and H. J. Sips, "Give-to-Get: Free-riding-resilient video-on-demand in P2P systems," in *Multimedia Computing and Networking*, vol. 6818, (San Jose, USA), 2008. 16