216217 P2P-Next

Deliverable number 4.0.6

Next-Share Platform M48—Specification Part

Contractual Date of Delivery to the CEC:	**31 December 2011**
Actual Date of Delivery to the CEC:	**31 December 2011**
Author(s):	**A. Bakker, V. Grishchenko, D. Gabrijelčič, R. Deaconescu, V. Jovanovikj, T. Klobučar, D. Gkorou, T. Vinkó, N. Chiluka, J. Pouwelse, D. Epema**
Participant(s):	**TUD, JSI, UPB**
Workpackage:	**WP4**
Est. person months:	**68**
Security:	**PU**
Nature:	**P**
Version:	**1.0**
Total number of pages:	**84**

Abstract:

This document contains the architecture and protocol specifications for the Next-Share content delivery platform, release M48. As such it complements the M48 source code which is distributed separately. Together the source code (referred to as D4.0.6--Code Part) and this document form the D4.0.6 deliverable. This document builds on the D4.0.1-5 --Specification Part deliverables.

Keyword list: Next-Share, API, architecture, protocol specification

Contents

Revision History

This specification document describes only the new features added to the Next-Share content delivery platform (and resulting architecture and API changes) since the previous release. Hence, the document is not an update of the previous version, and to obtain a complete specification of the platform all D4.0.* documents should be read together. The D4.0.1, 4.0.2, 4.0.3, 4.0.4, 4.0.5 deliverables are sometimes referred to as D4.0.1a, 4.0.1b, D4.0.1c, D4.0.1d, and D4.0.1e respectively.

Date	Version	Status	Comments
30/06/09	4.0.1 (revised)	Approved	At the request of the reviewers the D4.0.1 deliverable was split into a Code part and a Specification part. This document is the Specification part.
30/04/09	4.0.2	Approved	Specification of the new features that were added to M8 and the resulting architecture and API changes.
31/12/09	4.0.3	Approved	Specification of the new features that were added to M16 and the resulting architecture and API changes.
31/08/10	4.0.4	Approved subject to conditions	Specification of the new features that were added to M24 and the resulting architecture and API changes.
15/07/11	4.0.4 (revised)	Submitted	Revised following reviewers' comments at year 3 technical review: • Scientific articles about the reputation service are now added as appendices to this document. According Annex I V2.2b, WP4 should deliver a software platform. Following the reviewers' comments at the year 1 technical review we started providing a specifications document with the software delivery. Now we also supply the relevant scientific output as part of the deliverable. • Purpose of SIP integration described more clearly. • Roadmap for year 4 included. • Added a revision history. • Network fabric aspects will be described in D4.0.6.

31/04/11	4.0.5	Submitted	Specification of the new features that were added to M40 and the resulting architecture and API changes.
31/12/11	4.0.6 (this)	Submitted	Specification of the new features that were added to M48 and the resulting architecture and API changes.

Chapter 1 Introduction

This document contains the architecture, Application Programmer Interfaces (APIs) and protocol specifications for the Next-Share content delivery platform, version M48. The reader is assumed to be familiar with the BitTorrent protocol [1][2] on which Next-Share M48 is based and deliverable D4.0.1 (revised) and D4.0.2-5 which describe the M8, M16, M24, M32 and M40 versions of the platform, respectively. For convenience, we include a brief summary of BitTorrent below.

This document is structured as follows. After an introduction of the BitTorrent background, we start with a brief description of the new extensions to the platform. Next, we present the revised architecture of the platform. Subsequently, the protocols underlying the extensions are described in detail, each in a separate chapter. Finally, we present the revisions to the platform's APIs.

1.1 BitTorrent Background

BitTorrent has an interesting design that enables each individual downloader to maximize his own download rate and locks out users who do not contribute to the system. A peer wishing to download a particular file through BitTorrent first needs to obtain a *torrent* metafile for the file from, for example, a Web site or RSS news feed. The metafile gives the peer the address of a *tracker* for the file and checksums to verify downloaded parts of the file. The peer then contacts the tracker to obtain a list of peers currently involved in downloading the file, implying they have pieces of the file to share.

Next, the peer contacts a random peer to obtain a first piece of the file itself. With this piece in hand, the peer starts to contact other peers in the list to see if they will trade its piece for another part of the file. If so, the contacted peer sends a few blocks of the negotiated piece, and continues to do so as long as the other does the same. This tit-for-tat mechanism automatically locks out peers who are unwilling to upload themselves. By monitoring the download rate obtained from its current set of peers and randomly trying other peers to see if faster peers are available, a user can maximize its download rate. By always selecting a rare part of the file from the pieces on offer, a peer ensures it always has a piece of the file that other peers are interested in. These policies for piece selection and bandwidth trading lead to a balanced economy with suppliers meeting demand and achieving their own goal (fast download) at the same time. Once the peer has obtained the complete file it will become a *seeder* and altruistically provide pieces to other peers without any return. The set of all peers currently actively exchanging pieces of the file is called the file's *swarm*.

1.2 New Features

For the M48 release we have the following new extensions or updates to report:

1.2.1.1. Swift Became Basis for IETF P2P Streaming Protocol

In December 2011, the Peer-to-Peer Streaming Protocol (PPSP) working group of the Internet Engineering Task Force (IETF) chose swift as the basis for the P2P streaming protocol standard they are defining. The current IETF draft, extensively rewritten by TUD since M40 is included in Sec. 3.2.

1.2.1.2. Swift in the Linux Kernel

Swift has been designed to act as a BitTorrent-like replacement for TCP and be able to be implemented as a Transport layer protocol in the operating-system networking stack. Towards this latter goal, an initial porting of the protocol in the Linux kernel networking stack has been undertaken by UPB. We have designed the kernel space/user space component and created a Linux kernel protocol development framework. In order to allow rapid development, a user space raw socket-based implementation has been created. It provides a BSD socket-like API making it compatible with the kernel syscall interface. Doing a compatible implementation in user space is important to reduce the high development and debugging time that is common to kernel development. These porting efforts are described in Section 3.3.

1.2.1.3. Seamless Integration of swift into NextShare Core

For M48 we extended the NextShare Core API to support swift downloads. Developers can now start swift downloads as easy as the original BitTorrent-based downloads. The actual download takes place in a separate process running the C++ swift implementation that is controlled by the Python implementation of NextShare Core via a TCP socket. This fact, is, however, totally transparent to the developer who just deals with the Python interface. This integration is different from the hybrid peer-to-peer engine we investigated in D4.0.5 where the Python Core and swift C++ download engine were actually running in the same process. The new integration is described in Sections 2.1 and 3.4.

1.2.1.4. Extended swift Protocol Implementation

Since M40 we have improved the swift C++ implementations with a number of features. First, we added rate limiting such that video-on-demand content can be downloaded at the bitrate, rather than as-fast-as-possible which causes problems on the limited set-top box hardware. Second, the size of the chunks in which the content data is transferred by swift has now been made configurable. This allows for experiments to find the optimal chunk size for various scenarios (fast networks, memory-limited hardware). Third, the verification of content against the cryptographic hashes that ensure its integrity has been optimized. Content download in a previous swift session no longer has to be rechecked when the download or

sharing is resumed in a subsequent session. In addition, the number of hashes checked in the Merkle hash tree when a chunk of content is received has also been reduced by ~33% (measured over a complete download). The improved swift source code can be found in the Next-Share/source-swift/ directory of the Code Part of this deliverable.

1.2.1.5. Enhanced Closed Swarms Protocol

The Closed swarms (CS) protocol is an access control mechanism for controlling the P2P content delivery process, described in D4.0.3 and published in [10]. It acts on peer level – enables peers to recognize the authorized peers and to avoid communication with the non-authorized ones. The distinction between authorized and non-authorized peers in the swarm is made based on possession of an authorization credential called proof-of-access (PoA). The peers exchange their credentials right after they establish connection, in a challenge-response messages exchange process. The CS protocol can provide access control in an innovative business content delivery system, in which users would receive graded service – the authorized users would receive additional or better service than the non-authorized ones, for example access to high speed seeders for better performance.

However, the CS protocol lacks of flexibility in the access control – it is applicable only under the same conditions for all authorized users. Moreover, it is vulnerable to man-in-the-middle attacks. An attacker can interfere in the communication between two authorized peers by simply relaying the messages between them. After the authorized peers successfully finish the protocol and start the content delivery, the attacker will be able to read all the exchanged content pieces, since they are not encrypted. Therefore, we add several enhancements to the CS protocol in order to overcome these shortcomings. The enhancements provide additional flexibility in the access control mechanism and fulfil a number of content providers' requirements. In addition, they promise efficient and secure content delivery. The Enhanced Closed Swarms Protocol designed by JSI is described in Chapter 4 and a paper published on it at the Fifth International Conference on Emerging Security Information, Systems and Technologies, (SECURWARE) is included as Appendix C.

1.2.1.6. Monitoring of P2P Service Provisioning

For professional content providers and users as well it is crucial that content ingest service could be properly monitored. Systems users would like to follow distribution effectiveness and spot any issues in service provisioning in real time. The plans for monitoring of the ingest service were described in the deliverable D3.2.1.b. Chapter 5 describes the client-side implementation of this monitoring system by JSI and how it provides a way to sample the monitoring parameters and to export these parameters via a web based interface for other services to use.

1.2.1.7. Reputation Service Progress: Reducing the History in Reputation Systems

Following the reviewers' comments at the year 3 technical review we now also supply the relevant scientific output on the reputation service as part of the deliverable. In particular, we include the following papers by TUD as Appendices A and B, respectively:

R. Delaviz, N. Andrade, and J. Pouwelse. *"Improving Accuracy and Coverage in an Internet-Deployed Reputation Mechanism"*. In Proceedings 10th IEEE International Conference on Peer-to-Peer Computing (IEEE P2P'10), Delft, the Netherlands, August 25-27, 2010.

D. Gkorou, T. Vinkó, N. Chiluka, J. Pouwelse and D. Epema. *"Reducing the History in Reputation Systems"*. In Proceedings 17th Annual Conference of the Advanced School for Computing and Imaging (ASCI '11), Veldhoven, the Netherlands, November 2011.

1.3 Categorization of Features

WP4.1: IPvNext

- Enhanced Closed Swarms

WP4.2: Network Awareness

- Monitoring of P2P Service Provisioning

WP4.3: P2P data exchange

- Swift standardization within IETF
- Swift in the Linux-kernel
- Integration of Swift into NextShareCore
- Swift implementation improvements

WP4.3: Micropayments

-

Chapter 2
Revised Architecture

2.1 Changes Since M40

Figure 2.1 shows the architecture of the Next-Share M48 platform. In this high-level overview the changes for M48 are not visible. Figure 2.2 zooms into the "TorrentShare" component that we extended with support for swift downloads. For every swift download that is started via the Session.start_download() API call, a SwiftDownload object is created. This object represents the download in the Core.

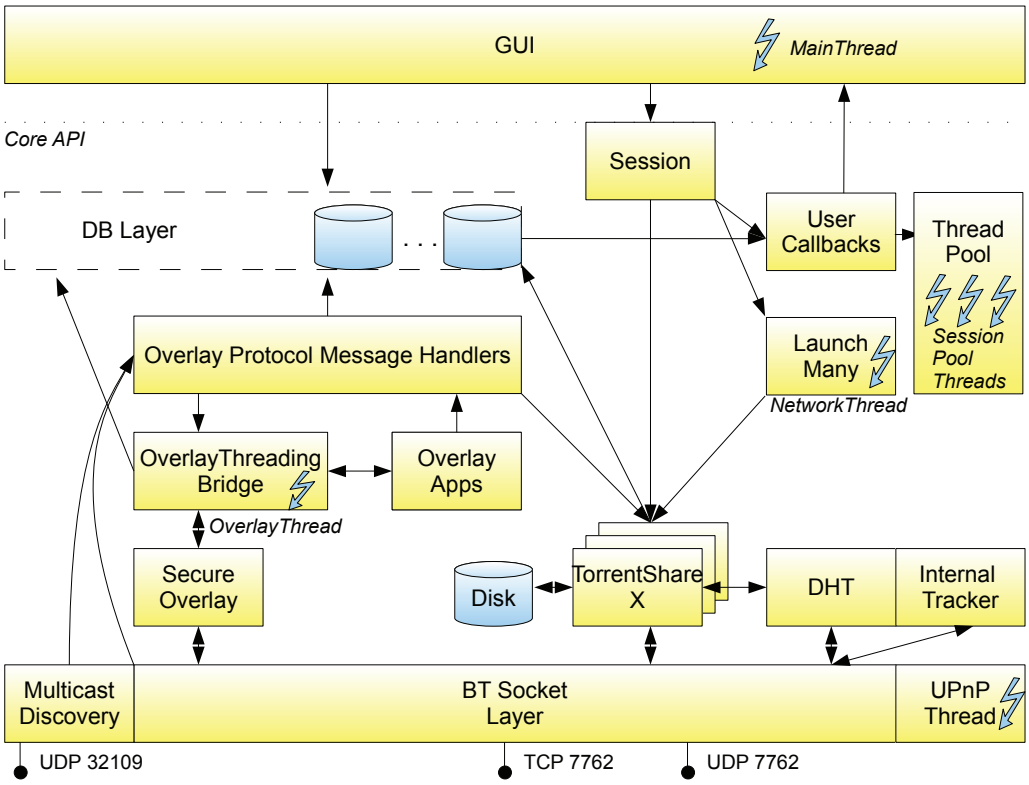


Figure 2.1: Current architecture of the NextShare Core. Boxes represent classes or components, cylinders represent databases or disk storage and bolts of lightning represent threads.

The actual downloading and sharing of the content is done in a separate process, shown on the right in Figure 2.2. This process is controlled via a TCP socket that is used to send commands and receive back e.g. statistics about the download's progress. The SwiftProcess object takes care of this interaction and translates the high-level API commands (stop/restart download) into commands sent over TCP and turns received info into upcalls, Every swift

download may be carried out by its own swift operating-system process, or one swift process can perform multiple downloads in parallel, depending on some policy. The SwiftProcMgr is the component that implements the policy that governs the mapping from download to process.

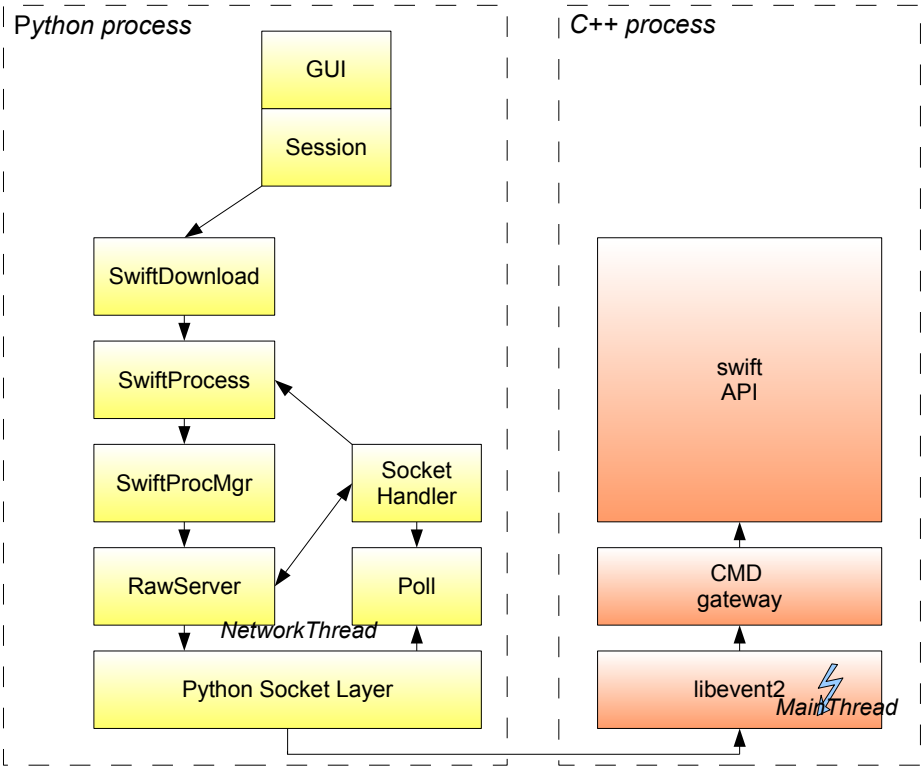


Figure 2.2: Architecture of the TorrentShare component for swift downloads. Components in yellow are implemented in Python, components in red in C++.

The swift process shown on the right in Figure 2.2 uses the libevent library (http://libevent.org/), version 2 to facilitate socket communication. The command gateway (appreviated CMD gateway in the figure) receives the commands sent over the TCP sockets and calls the appropriate swift API methods. It also periodically retrieves download and sharing statistics from the swift API and communicates them back to the controlling Python Core.

Figure 2.3 shows the structure of the implementation of the swift API. For every swift download in progress there is a FileTransfer object that represents it. The FileTransfer object links to a HashTree object that stores the Merkle tree of SHA1 hashes that is used for content integrity in swift (see Chapter 3). It also links to a PiecePicker object that decides which pieces of the content are downloaded from whom, following e.g. an economic model that optimizes upload utilization. Swift supports different PiecePicker objects implementing different policies. Finally, it maintains a list of Channel objects which represent a (virtual) connection to a peer. Per peer we record which pieces it currently has in a Peer binmap datastructure [9]. The HashTree object records which pieces we already downloaded in its

"Own" binmap data structure. The Channel objects handle sending and receiving of the swift protocol messages and do congestion control when needed. A swift process can use one or more network ports and they are centrally controlled by the "Socket Mgmt" component. Finally, at the top of Figure 2.3 we find the HTTP gateway that takes care of delivering the content downloaded via swift to a video player over HTTP, and the statistics gateway (abbreviated "STATS gateway" in the figure). The latter provides statistics to the SwarmPlayer browser plugin developed by WP6 when swift is used as its backend (see D6.5.5/D6.5.6).

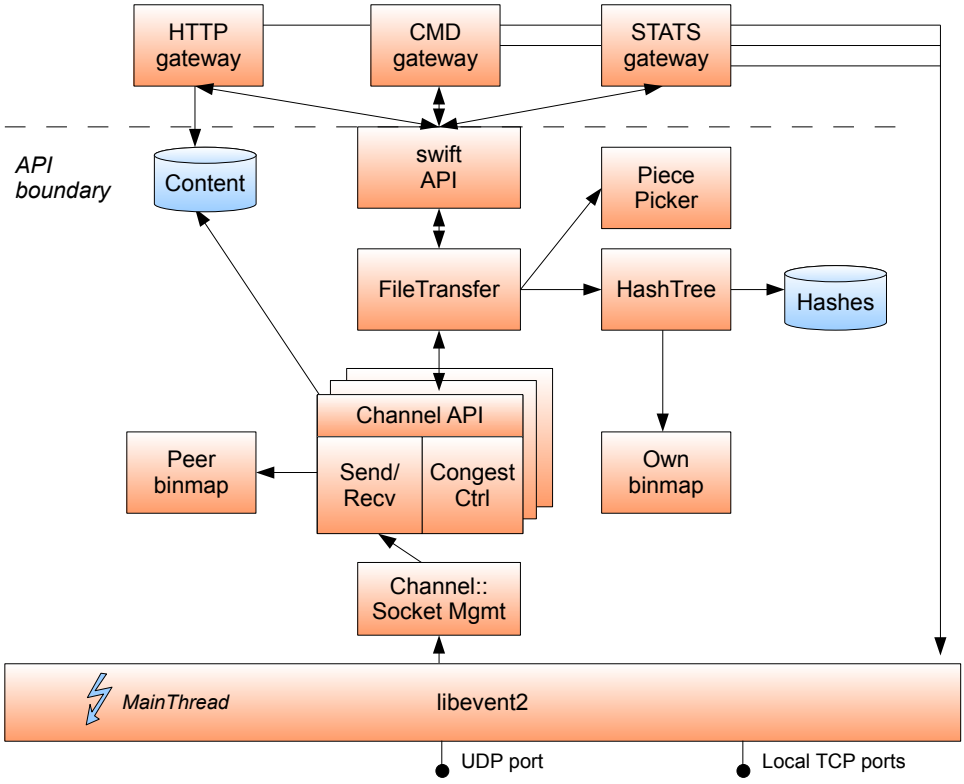


Figure 2.3: Architecture of the swift implementation.

For a full description of the architecture we refer to D4.0.1. The changes we made to the Core API in M48 are discussed in Chapter 6.

Chapter 3
The Generic Multiparty Transport Protocol (swift)

3.1 Introduction

This chapter describes our work on the swift multiparty transport protocol. In December 2011, the Peer-to-Peer Streaming Protocol (PPSP) working group of the Internet Engineering Task Force (IETF) chose swift as the basis for the P2P streaming protocol standard they are defining. The current IETF draft, extensively rewritten since M40 is included in Sec. 3.2 and is also available from *https://http://datatracker.ietf.org/doc/draft-ietf-ppsp-peer-protocol/*.

In addition, this chapter presents the work of UPB on integrating swift into the Linux kernel (Section 3.3). Finally, Section 3.4 describes how TUD extended the NextShare Core API to support swift downloads, such that developers can now start swift downloads as easy as the original BitTorrent-based downloads.

3.2 IETF Internet Draft

```
PPSP                                                        V. Grishchenko
Internet-Draft                                                   A. Bakker
Intended status: Informational                                   TU Delft
Expires: April 28, 2012                                  October 26, 2011

        The Generic Multiparty Transport Protocol (swift)
            <draft-grishchenko-ppsp-swift-03.txt>

Abstract

   The Generic Multiparty Protocol (swift) is a peer-to-peer based
   transport protocol for content dissemination. It can be used for
   streaming on-demand and live video content, as well as conventional
   downloading. In swift, the clients consuming the content participate
   in the dissemination by forwarding the content to other clients via a
   mesh-like structure.  It is a generic protocol which can run directly
   on top of UDP, TCP, HTTP or as a RTP profile. Features of swift are
   short time-till-playback and extensibility. Hence, it can use
   different mechanisms to prevent freeriding, and work with different
   peer discovery schemes (centralized trackers or Distributed Hash
   Tables). Depending on the underlying transport protocol, swift can
   also use different congestion control algorithms, such as LEDBAT, and
   offer transparent NAT traversal. Finally, swift maintains only a
   small amount of state per peer and detects malicious modification of
   content. This documents describes swift and how it satisfies the
   requirements for the IETF Peer-to-Peer Streaming Protocol (PPSP)
   Working Group's peer protocol.
```

Status of this memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at

http://www.ietf.org/shadow.html.

Table of Contents

1. Introduction

1.1. Purpose

This document describes the Generic Multiparty Protocol (swift), designed from the ground up for the task of disseminating the same content to a group of interested parties. Swift supports streaming on-demand and live video content, as well as conventional downloading, thus covering today's three major use cases for content distribution. To fulfil this task, clients consuming the content are put on equal footing with the servers initially providing the content

to create a peer-to-peer system where everyone can provide data. Each peer connects to a random set of other peers resulting in a mesh-like structure.

Swift uses a simple method of naming content based on self-certification. In particular, content in swift is identified by a single cryptographic hash that is the root hash in a Merkle hash tree calculated recursively from the content [ABMRKL]. This self-certifying hash tree allows every peer to directly detect when a malicious peer tries to distribute fake content. It also ensures only a small amount of information is needed to start a download (just the root hash and some peer addresses).

Swift uses a novel method of addressing chunks of content called "bin numbers". Bin numbers allow the addressing of a binary interval of data using a single integer. This reduces the amount of state that needs to be recorded per peer and the space needed to denote intervals on the wire, making the protocol light-weight. In general, this numbering system allows swift to work with simpler data structures, e.g. to use arrays instead of binary trees, thus reducing complexity.

Swift is a generic protocol which can run directly on top of UDP, TCP, HTTP, or as a layer below RTP, similar to SRTP [RFC3711]. As such, swift defines a common set of messages that make up the protocol, which can have different representations on the wire depending on the lower-level protocol used. When the lower-level transport is UDP, swift can also use different congestion control algorithms and facilitate NAT traversal.

In addition, swift is extensible in the mechanisms it uses to promote client contribution and prevent freeriding, that is, how to deal with peers that only download content but never upload to others. Furthermore, it can work with different peer discovery schemes, such as centralized trackers or fast Distributed Hash Tables [JIM11].

This documents describes not only the swift protocol but also how it satisfies the requirements for the IETF Peer-to-Peer Streaming Protocol (PPSP) Working Group's peer protocol [PPSPCHART,I-D.ietf-ppsp-reqs]. A reference implementation of swift over UDP is available [SWIFTIMPL].

1.2. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

1.3. Terminology

message
: The basic unit of swift communication. A message will have different representations on the wire depending on the transport protocol used. Messages are typically multiplexed into a datagram for transmission.

datagram
: A sequence of messages that is offered as a unit to the underlying transport protocol (UDP, etc.). The datagram is swift's Protocol Data Unit (PDU).

content
: Either a live transmission, a pre-recorded multimedia asset, or a file.

bin
: A number denoting a specific binary interval of the content (i.e., one or more consecutive chunks).

chunk
: The basic unit in which the content is divided. E.g. a block of N kilobyte.

hash
: The result of applying a cryptographic hash function, more specifically a modification detection code (MDC) [HAC01], such as SHA1 [FIPS180-2], to a piece of data.

root hash
: The root in a Merkle hash tree calculated recursively from the content.

swarm
: A group of peers participating in the distribution of the same content.

swarm ID
: Unique identifier for a swarm of peers, in swift the root hash of the content (video-on-demand,download) or a public key (live streaming).

tracker
: An entity that records the addresses of peers participating in a swarm, usually for a set of swarms, and makes this membership information available to other peers on request.

choking
When a peer A is choking peer B it means that A is currently not willing to accept requests for content from B.

2. Overall Operation

The basic unit of communication in swift is the message. Multiple messages are multiplexed into a single datagram for transmission. A datagram (and hence the messages it contains) will have different representations on the wire depending on the transport protocol used (see Sec. 6).

2.1. Joining a Swarm

Consider a peer A that wants to download a certain content asset. To commence a swift download, peer A must have the swarm ID of the content and a list of one or more tracker contact points (e.g. host+port). The list of trackers is optional in the presence of a decentralized tracking mechanism. The swarm ID consists of the swift root hash of the content (video-on-demand, downloading) or a public key (live streaming).

Peer A now registers with the tracker following e.g. the PPSP tracker protocol [I-D.ietf.ppsp-reqs] and receives the IP address and port of peers already in the swarm, say B, C, and D. Peer A now sends a datagram containing a HANDSHAKE message to B, C, and D. This message serves as an end-to-end check that the peers are actually in the correct swarm, and contains the root hash of the swarm. Peer B and C respond with datagrams containing a HANDSHAKE message and one or more HAVE messages. A HAVE message conveys (part of) the chunk availability of a peer and thus contains a bin number that denotes what chunks of the content peer B, resp. C have. Peer D sends a datagram with just a HANDSHAKE and omits HAVE messages as a way of choking A.

2.2. Exchanging Chunks

In response to B and C, A sends new datagrams to B and C containing HINT messages. A HINT or request message indicates the chunks that a peer wants to download, and contains a bin number. The HINT messages to B and C refer to disjunct sets of chunks. B and C respond with datagrams containing HASH, HAVE and DATA messages. The HASH messages contains all cryptographic hashes that peer A needs to verify the integrity of the content chunk sent in the DATA message, using the content's root hash as trusted anchor, see Sec. 3.5. Using these hashes peer A verifies that the chunks received from B and C are

correct. It also updates the chunk availability of B and C using the information in the received HAVE messages.

After processing, A sends a datagram containing HAVE messages for the chunks it just received to all its peers. In the datagram to B and C it includes an ACK message acknowledging the receipt of the chunks, and adds HINT messages for new chunks. ACK messages are not used when a reliable transport protocol is used. When e.g. C finds that A obtained a chunk (from B) that C did not yet have, C's next datagram includes a HINT for that chunk.

Peer D does not send HAVE messages to A when it downloads chunks from other peers, until D decides to unchoke peer A. In the case, it sends a datagram with HAVE messages to inform A of its current availability. If B or C decide to choke A they stop sending HAVE and DATA messages and A should then rerequest from other peers. They may continue to send HINT messages, or periodic KEEPALIVE messages such that A keeps sending them HAVE messages.

Once peer A has received all content (video-on-demand use case) it stops sending messages to all other peers that have all content (a.k.a. seeders). Peer A MAY also contact the tracker or another source again to obtain more peer addresses.

2.3. Leaving a Swarm

Depending on the transport protocol used, peers should either use explicit leave messages or implicitly leave a swarm by stopping to respond to messages. Peers that learn about the departure should remove these peers from the current peer list. The implicit-leave mechanism works for both graceful and ungraceful leaves (i.e., peer crashes or disconnects). When leaving gracefully, a peer should deregister from the tracker following the (PPSP) tracker protocol.

3. Messages

In general, no error codes or responses are used in the protocol; absence of any response indicates an error. Invalid messages are discarded.

For the sake of simplicity, one swarm of peers always deals with one content asset (e.g. file) only. Retrieval of large collections of files is done by retrieving a directory list file and then recursively retrieving files, which might also turn to be directory lists, as described in Sec. 3.11.

3.1. HANDSHAKE

As an end-to-end check that the peers are actually in the correct swarm, the initiating peer and the addressed peer SHOULD send a HANDSHAKE message in the first datagrams they exchange. The only payload of the HANDSHAKE message is the root hash of the content.

After the handshakes are exchanged, the initiator knows that the peer really responds. Hence, the second datagram the initiator sends MAY already contain some heavy payload. To minimize the number of initialization roundtrips, implementations MAY dispense with the HANDSHAKE message. To the same end, the first two datagrams exchanged MAY also contain some minor payload, e.g. HAVE messages to indicate the current progress of a peer or a HINT (see Sec. 3.6).

3.3. HAVE

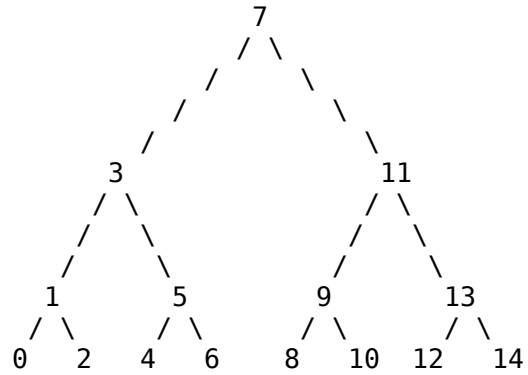
The HAVE message is used to convey which chunks a peers has available, expressed in a new content addressing scheme called "bin numbers".

3.3.1. Bin Numbers

Swift employs a generic content addressing scheme based on binary intervals ("bins" in short). The smallest interval is a chunk (e.g. a N kilobyte block), the top interval is the complete 2**63 range. A novel addition to the classical scheme are "bin numbers", a scheme of numbering binary intervals which lays them out into a vector nicely. Consider an chunk interval of width W. To derive the bin numbers of the complete interval and the subintervals, a minimal balanced binary tree is built that is at least W chunks wide at the base. The leaves from left-to-right correspond to the chunks 0..W in the interval, and have bin number I*2 where I is the index of the chunk (counting beyond W-1 to balance the tree). The higher level nodes P in the tree have bin number

```
binP = (binL + binR) / 2
```

where binL is the bin of node P's left-hand child and binR is the bin of node P's right-hand child. Given that each node in the tree represents a subinterval of the original interval, each such subinterval now is addressable by a bin number, a single integer. The bin number tree of a interval of width W=8 looks like this:



So bin 7 represents the complete interval, 3 represents the interval of chunk 0..3 and 1 represents the interval of chunks 0 and 1. The special numbers 0xFFFFFFFF (32-bit) or 0xFFFFFFFFFFFFFFFF (64-bit) stands for an empty interval, and 0x7FFF...FFF stands for "everything".

3.3.2. HAVE Message

When a receiving peer has successfully checked the integrity of a chunk or interval of chunks it MUST send a HAVE message to all peers it wants to interact with. The latter allows the HAVE message to be used as a method of choking. The HAVE message MUST contain the bin number of the biggest complete interval of all chunks the receiver has received and checked so far that fully includes the interval of chunks just received. So the bin number MUST denote at least the interval received, but the receiver is supposed to aggregate and acknowledge bigger bins, when possible.

As a result, every single chunk is acknowledged a logarithmic number of times. That provides some necessary redundancy of acknowledgments and sufficiently compensates for unreliable transport protocols.

To record which chunks a peer has in the state that an implementation keeps for each peer, an implementation MAY use the "binmap" data structure, which is a hybrid of a bitmap and a binary tree, discussed in detail in [BINMAP].

3.4. ACK

When swift is run over an unreliable transport protocol, an implementation MAY choose to use ACK messages to acknowledge received data. When a receiving peer has successfully checked the integrity of a chunk or interval of chunks C it MUST send a ACK message containing

the bin number of its biggest, complete, interval covering C to the sending peer (see HAVE). To facilitate delay-based congestion control, an ACK message contains a timestamp.

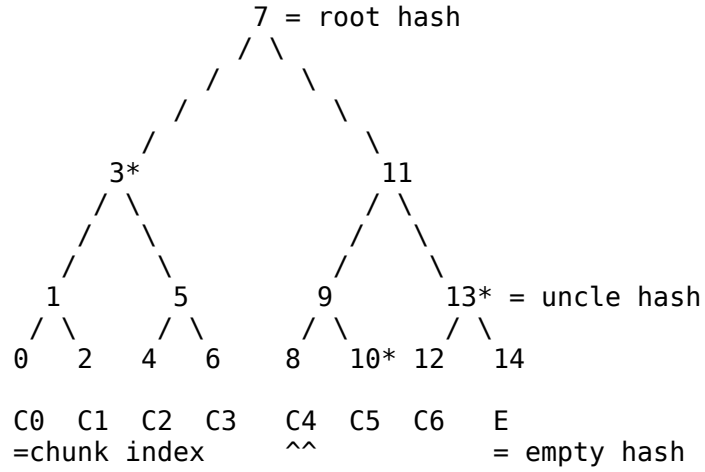
3.5. DATA and HASH

The DATA message is used to transfer chunks of content. The associated HASH message carries cryptographic hashes that are necessary for a receiver to check the the integrity of the chunk. Swift's content integrity protection is based on a Merkle hash tree and works as follows.

3.5.1. Merkle Hash Tree

Swift uses a method of naming content based on self-certification. In particular, content in swift is identified by a single cryptographic hash that is the root hash in a Merkle hash tree calculated recursively from the content [ABMRKL]. This self-certifying hash tree allows every peer to directly detect when a malicious peer tries to distribute fake content. It also ensures only a small the amount of information is needed to start a download (the root hash and some peer addresses). For live streaming public keys and dynamic trees are used, see below.

The Merkle hash tree of a content asset that is divided into N chunks is constructed as follows. Note the construction does not assume chunks of content to be fixed size. Given a cryptographic hash function, more specifically a modification detection code (MDC) [HAC01], such as SHA1, the hashes of all the chunks of the content are calculated. Next, a binary tree of sufficient height is created. Sufficient height means that the lowest level in the tree has enough nodes to hold all chunk hashes in the set, as before, see HAVE message. The figure below shows the tree for a content asset consisting of 7 chunks. As before with the content addressing scheme, the leaves of the tree correspond to a chunk and in this case are assigned the hash of that chunk, starting at the left-most leaf. As the base of the tree may be wider than the number of chunks, any remaining leaves in the tree are assigned a empty hash value of all zeros. Finally, the hash values of the higher levels in the tree are calculated, by concatenating the hash values of the two children (again left to right) and computing the hash of that aggregate. This process ends in a hash value for the root node, which is called the "root hash". Note the root hash only depends on the content and any modification of the content will result in a different root hash.



3.5.2. Content Integrity Verification

Assuming a peer receives the root hash of the content it wants to download from a trusted source, it can can check the integrity of any chunk of that content it receives as follows. It first calculates the hash of the chunk it received, for example chunk C4 in the previous figure. Along with this chunk it MUST receive the hashes required to check the integrity of that chunk. In principle, these are the hash of the chunk's sibling (C5) and that of its "uncles". A chunk's uncles are the sibling Y of its parent X, and the uncle of that Y, recursively until the root is reached. For chunk C4 its uncles are bins 13 and 3, marked with * in the figure. Using this information the peer recalculates the root hash of the tree, and compares it to the root hash it received from the trusted source. If they match the chunk of content has been positively verified to be the requested part of the content. Otherwise, the sending peer either sent the wrong content or the wrong sibling or uncle hashes. For simplicity, the set of sibling and uncles hashes is collectively referred to as the "uncle hashes".

In the case of live streaming the tree of chunks grows dynamically and content is identified with a public key instead of a root hash, as the root hash is undefined or, more precisely, transient, as long as new data is generated by the live source. Live streaming is described in more detail below, but content verification works the same for both live and predefined content.

3.5.3. The Atomic Datagram Principle

As explained above, a datagram consists of a sequence of messages. Ideally, every datagram sent must be independent of other datagrams,

so each datagram SHOULD be processed separately and a loss of one datagram MUST NOT disrupt the flow. Thus, as a datagram carries zero or more messages, neither messages nor message interdependencies should span over multiple datagrams.

This principle implies that as any chunk is verified using its uncle hashes the necessary hashes MUST be put into the same datagram as the chunk's data (Sec. 3.5.4). As a general rule, if some additional data is still missing to process a message within a datagram, the message SHOULD be dropped.

The hashes necessary to verify a chunk are in principle its sibling's hash and all its uncle hashes, but the set of hashes to sent can be optimized. Before sending a packet of data to the receiver, the sender inspects the receiver's previous acknowledgments (HAVE or ACK) to derive which hashes the receiver already has for sure. Suppose, the receiver had acknowledged bin 1 (first two chunks of the file), then it must already have uncle hashes 5, 11 and so on. That is because those hashes are necessary to check packets of bin 1 against the root hash. Then, hashes 3, 7 and so on must be also known as they are calculated in the process of checking the uncle hash chain. Hence, to send bin 12 (i.e. the 7th chunk of content), the sender needs to include just the hashes for bins 14 and 9, which let the data be checked against hash 11 which is already known to the receiver.

The sender MAY optimistically skip hashes which were sent out in previous, still unacknowledged datagrams. It is an optimization tradeoff between redundant hash transmission and possibility of collateral data loss in the case some necessary hashes were lost in the network so some delivered data cannot be verified and thus has to be dropped. In either case, the receiver builds the Merkle tree on-demand, incrementally, starting from the root hash, and uses it for data validation.

In short, the sender MUST put into the datagram the missing hashes necessary for the receiver to verify the chunk.

3.5.4. DATA and HASH Messages

Concretely, a peer that wants to send a chunk of content creates a datagram that MUST consist of one or more HASH messages and a DATA message. The datagram MUST contain a HASH message for each hash the receiver misses for integrity checking. A HASH message MUST contain the bin number and hash data for each of those hashes. The DATA message MUST contain the bin number of the chunk and chunk itself. A peer MAY send the required messages for multiple chunks in the same datagram.

3.6. HINT

While bulk download protocols normally do explicit requests for certain ranges of data (i.e., use a pull model, for example, BitTorrent [BITTORRENT]), live streaming protocols quite often use a request-less push model to save round trips. Swift supports both models of operation.

A peer MUST send a HINT message containing the bin of the chunk interval it wants to download. A peer receiving a HINT message MAY send out requested pieces. When it receives multiple HINTs (either in one datagram or in multiple), the peer SHOULD process the HINTs sequentially. When live streaming, it also may send some other chunks in case it runs out of requests or for some other reason. In that case the only purpose of HINT messages is to coordinate peers and to avoid unnecessary data retransmission, hence the name.

3.7. Peer Address Exchange and NAT Hole Punching

Peer address exchange messages (or PEX messages for short) are common for many peer-to-peer protocols. By exchanging peer addresses in gossip fashion, peers relieve central coordinating entities (the trackers) from unnecessary work. swift optionally features two types of PEX messages: PEX_REQ and PEX_ADD. A peer that wants to retrieve some peer addresses MUST send a PEX_REQ message. The receiving peer MAY respond with a PEX_ADD message containing the addresses of several peers. The addresses MUST be of peers it has recently exchanged messages with to guarantee liveliness.

To unify peer exchange and NAT hole punching functionality, the sending pattern of PEX messages is restricted. As the swift handshake is able to do simple NAT hole punching [SNP] transparently, PEX messages must be emitted in the way to facilitate that. Namely, once peer A introduces peer B to peer C by sending a PEX_ADD message to C, it SHOULD also send a message to B introducing C. The messages SHOULD be within 2 seconds from each other, but MAY not be, simultaneous, instead leaving a gap of twice the "typical" RTT, i.e. 300-600ms. The peers are supposed to initiate handshakes to each other thus forming a simple NAT hole punching pattern where the introducing peer effectively acts as a STUN server [RFC5389]. Still, peers MAY ignore PEX messages if uninterested in obtaining new peers or because of security considerations (rate limiting) or any other reason.

The PEX messages can be used to construct a dedicated tracker peer.

3.8. KEEPALIVE

A peer MUST send a datagram containing a KEEPALIVE message periodically to each peer it wants to interact with in the future but has no other messages to send them at present.

3.9. VERSION

Peers MUST convey which version of the swift protocol they support using a VERSION message. This message MUST be included in the initial (handshake) datagrams and MUST indicate which version of the swift protocol the sending peer supports.

3.10. Conveying Peer Capabilities

Peers may support just a subset of the swift messages. For example, peers running over TCP may not accept ACK messages, or peers used with a centralized tracking infrastructure may not accept PEX messages. For these reasons, peers SHOULD signal which subset of the swift messages they support by means of the MSGTYPE_RCVD message. This message SHOULD be included in the initial (handshake) datagrams and MUST indicate which swift protocol messages the sending peer supports.

3.11. Directory Lists

Directory list files MUST start with magic bytes ".\n..\n". The rest of the file is a newline-separated list of hashes and file names for the content of the directory. An example:

```
.
..
1234567890ABCDEF1234567890ABCDEF12345678  readme.txt
01234567890ABCDEF1234567890ABCDEF1234567  big_file.dat
```

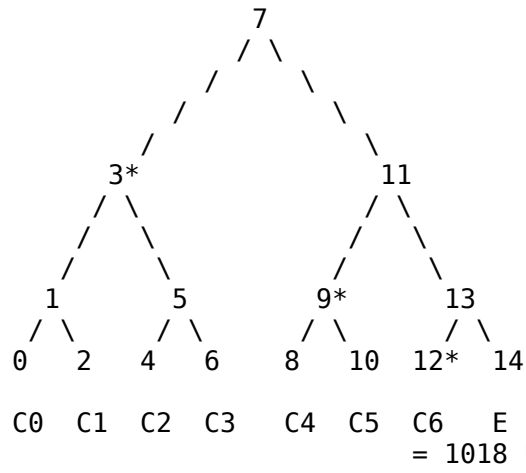
4. Automatic Detection of Content Size

In swift, the root hash of a static content asset, such as a video file, along with some peer addresses is sufficient to start a download. In addition, swift can reliably and automatically derive the size of such content from information received from the network when fixed sized chunks are used. As a result, it is not necessary to include the size of the content asset as the metadata of the content, in addition to the root hash. Implementations of swift MAY use this automatic detection feature.

4.1. Peak Hashes

The ability for a newcomer peer to detect the size of the content depends heavily on the concept of peak hashes. Peak hashes, in general, enable two cornerstone features of swift: reliable file size detection and download/live streaming unification (see Sec. 5). The concept of peak hashes depends on the concepts of filled and incomplete bins. Recall that when constructing the binary trees for content verification and addressing the base of the tree may have more leaves than the number of chunks in the content. In the Merkle hash tree these leaves were assigned empty all-zero hashes to be able to calculate the higher level hashes. A filled bin is now defined as a bin number that addresses an interval of leaves that consists only of hashes of content chunks, not empty hashes. Reversely, an incomplete (not filled) bin addresses an interval that contains also empty hashes, typically an interval that extends past the end of the file. In the following figure bins 7, 11, 13 and 14 are incomplete the rest is filled.

Formally, a peak hash is a hash in the Merkle tree defined over a filled bin, whose sibling is defined over an incomplete bin. Practically, suppose a file is 7162 bytes long and a chunk is 1 kilobyte. That file fits into 7 chunks, the tail chunk being 1018 bytes long. The Merkle tree for that file looks as follows. Following the definition the peak hashes of this file are in bins 3, 9 and 12, denoted with a *. E denotes an empty hash.



Peak hashes can be explained by the binary representation of the number of chunks the file occupies. The binary representation for 7 is 111. Every "1" in binary representation of the file's packet length corresponds to a peak hash. For this particular file there are indeed three peaks, bin numbers 3, 9, 12. The number of peak hashes

for a file is therefore also at most logarithmic with its size.

A peer knowing which bins contain the peak hashes for the file can therefore calculate the number of chunks it consists of, and thus get an estimate of the file size (given all chunks but the last are fixed size). Which bins are the peaks can be securely communicated from one (untrusted) peer A to another B by letting A send the peak hashes and their bin numbers to B. It can be shown that the root hash that B obtained from a trusted source is sufficient to verify that these are indeed the right peak hashes, as follows.

Lemma: Peak hashes can be checked against the root hash.

Proof: (a) Any peak hash is always the left sibling. Otherwise, be it the right sibling, its left neighbor/sibling must also be defined over a filled bin, because of the way chunks are laid out in the leaves, contradiction. (b) For the rightmost peak hash, its right sibling is zero. (c) For any peak hash, its right sibling might be calculated using peak hashes to the left and zeros for empty bins. (d) Once the right sibling of the leftmost peak hash is calculated, its parent might be calculated. (e) Once that parent is calculated, we might trivially get to the root hash by concatenating the hash with zeros and hashing it repeatedly.

Informally, the Lemma might be expressed as follows: peak hashes cover all data, so the remaining hashes are either trivial (zeros) or might be calculated from peak hashes and zero hashes.

Finally, once peer B has obtained the number of chunks in the content it can determine the exact file size as follows. Given that all chunks except the last are fixed size B just needs to know the size of the last chunk. Knowing the number of chunks B can calculate the bin number of the last chunk and download it. As always B verifies the integrity of this chunk against the trusted root hash. As there is only one chunk of data that leads to a successful verification the size of this chunk must be correct. B can then determine the exact file size as

```
    (number of chunks -1) * fixed chunk size + size of last chunk
```

4.2. Procedure

A swift implementation that wants to use automatic size detection MUST operate as follows. When a peer B sends a DATA message for the first time to a peer A, B MUST include all the peak hashes for the content in the same datagram, unless A has already signalled earlier in the exchange that it knows the peak hashes by having acknowledged

any bin, even the empty one. The receiver A MUST check the peak hashes against the root hash to determine the approximate content size. To obtain the definite content size peer A MUST download the last chunk of the content from any peer that offers it.

5. Live streaming

In the case of live streaming a transfer is bootstrapped with a public key instead of a root hash, as the root hash is undefined or, more precisely, transient, as long as new data is being generated by the live source. Live/download unification is achieved by sending signed peak hashes on-demand, ahead of the actual data. As before, the sender might use acknowledgements to derive which content range the receiver has peak hashes for and to prepend the data hashes with the necessary (signed) peak hashes. Except for the fact that the set of peak hashes changes with time, other parts of the algorithm work as described in Sec. 3.

As with static content assets in the previous section, in live streaming content length is not known on advance, but derived on-the-go from the peak hashes. Suppose, our 7 KB stream extended to another kilobyte. Thus, now hash 7 becomes the only peak hash, eating hashes 3, 9 and 12. So, the source sends out a SIGNED_HASH message to announce the fact.

The number of cryptographic operations will be limited. For example, consider a 25 frame/second video transmitted over UDP. When each frame is transmitted in its own chunk, only 25 signature verification operations per second are required at the receiver for bitrates up to ~12.8 megabit/second. For higher bitrates multiple UDP packets per frame are needed and the number of verifications doubles.

6. Transport Protocols and Encapsulation

6.1. UDP

6.1.1. Chunk Size

Currently, swift-over-UDP is the preferred deployment option. Effectively, UDP allows the use of IP with minimal overhead and it

also allows userspace implementations. The default is to use chunks of 1 kilobyte such that a datagram fits in an Ethernet-sized IP packet. The bin numbering allows to use swift over Jumbo frames/datagrams. Both DATA and HAVE/ACK messages may use e.g. 8 kilobyte packets instead of the standard 1 KiB. The hashing scheme stays the same. Using swift with 512 or 256-byte packets is theoretically possible with 64-bit byte-precise bin numbers, but IP fragmentation might be a better method to achieve the same result.

6.1.2. Datagrams and Messages

When using UDP, the abstract datagram described above corresponds directly to a UDP datagram. Each message within a datagram has a fixed length, which depends on the type of the message. The first byte of a message denotes its type. The currently defined types are:

```
HANDSHAKE = 0x00
DATA = 0x01
ACK = 0x02
HAVE = 0x03
HASH = 0x04
PEX_ADD = 0x05
PEX_REQ = 0x06
SIGNED_HASH = 0x07
HINT = 0x08
MSGTYPE_RCVD = 0x09
VERSION = 0x10
```

Furthermore, integers are serialized in the network (big-endian) byte order. So consider the example of an ACK message (Sec 3.4). It has message type of 0x02 and a payload of a bin number, a four-byte integer (say, 1); hence, its on the wire representation for UDP can be written in hex as: "02 00000001". This hex-like two character-per-byte notation is used to represent message formats in the rest of this section.

6.1.3. Channels

As it is increasingly complex for peers to enable UDP communication between each other due to NATs and firewalls, swift-over-UDP uses a multiplexing scheme, called "channels", to allow multiple swarms to use the same UDP port. Channels loosely correspond to TCP connections and each channel belongs to a single swarm. When channels are used, each datagram starts with four bytes corresponding to the receiving channel number.

6.1.4. HANDSHAKE and VERSION

A channel is established with a handshake. To start a handshake, the initiating peer needs to know:

(1) the IP address of a peer
(2) peer's UDP port and
(3) the root hash of the content (see Sec. 3.5.1).

To do the handshake the initiating peer sends a datagram that MUST start with an all 0-zeros channel number followed by a VERSION message, then a HASH message whose payload is the root hash, and a HANDSHAKE message, whose only payload is a locally unused channel number.

On the wire the datagram will look something like this:

```
00000000 10 01
04 7FFFFFFF 1234123412341234123412341234123412341234
00 00000011
```

(to unknown channel, handshake from channel 0x11 speaking protocol version 0x01, initiating a transfer of a file with a root hash 123...1234)

The receiving peer MUST respond with a datagram that starts with the channel number from the sender's HANDSHAKE message, followed by a VERSION message, then a HANDSHAKE message, whose only payload is a locally unused channel number, followed by any other messages it wants to send.

Peer's response datagram on the wire:

```
00000011 10 01
00 00000022  03 00000003
```

(peer to the initiator: use channel number 0x22 for this transfer and proto version 0x01; I also have first 4 chunks of the file, see Sec. 4.3)

At this point, the initiator knows that the peer really responds; for that purpose channel ids MUST be random enough to prevent easy guessing. So, the third datagram of a handshake MAY already contain some heavy payload. To minimize the number of initialization roundtrips, the first two datagrams MAY also contain some minor payload, e.g. a couple of HAVE messages roughly indicating the current progress of a peer or a HINT (see Sec. 3.6). When receiving the third datagram, both peers have the proof they really talk to each other; three-way handshake is complete.

A peer MAY explicit close a channel by sending a HANDSHAKE message that MUST contain an all 0-zeros channel number.

On the wire:
00 00000000

6.1.5. HAVE

A HAVE message (type 0x03) states that the sending peer has the complete specified bin and successfully checked its integrity:
03 00000003
(got/checked first four kilobytes of a file/stream)

6.1.6. ACK

An ACK message (type 0x02) acknowledges data that was received from its addressee; to facilitate delay-based congestion control, an ACK message contains a timestamp, in particular, a 64-bit microsecond time.
02 00000002 12345678
(got the second kilobyte of the file from you; my microsecond timer was showing 0x12345678 at that moment)

6.1.7. HASH

A HASH message (type 0x04) consists of a four-byte bin number and the cryptographic hash (e.g. a 20-byte SHA1 hash)
04 7FFFFFFF 1234123412341234123412341234123412341234

6.1.8. DATA

A DATA message (type 0x01) consists of a four-byte bin number and the actual chunk. In case a datagram contains a DATA message, a sender MUST always put the data message in the tail of the datagram. For example:
01 00000000 48656c6c6f20776f726c6421
(This message accommodates an entire file: "Hello world!")

6.1.9. KEEPALIVE

Keepalives do not have a message type on UDP. They are just simple datagrams consisting of a 4-byte channel id only.

On the wire:
00000022

6.1.10. Flow and Congestion Control

Explicit flow control is not necessary in swift-over-UDP. In the case of video-on-demand the receiver will request data explicitly from peers and is therefore in control of how much data is coming towards it. In the case of live streaming, where a push-model may be used, the amount of data incoming is limited to the bitrate, which the receiver must be able to process otherwise it cannot play the stream. Should, for any reason, the receiver get saturated with data that situation is perfectly detected by the congestion control. Swift-over-UDP can support different congestion control algorithms, in particular, it supports the new IETF Low Extra Delay Background Transport (LEDBAT) congestion control algorithm that ensures that peer-to-peer traffic yields to regular best-effort traffic [LEDBAT].

6.2. TCP

When run over TCP, swift becomes functionally equivalent to BitTorrent. Namely, most swift messages have corresponding BitTorrent messages and vice versa, except for BitTorrent's explicit interest declarations and choking/unchoking, which serve the classic implementation of the tit-for-tat algorithm [TIT4TAT]. However, TCP is not well suited for multiparty communication, as argued in Sec. 9.

6.3. RTP Profile for PPSP

In this section we sketch how swift can be integrated into RTP [RFC3550] to form the Peer-to-Peer Streaming Protocol (PPSP) [I-D.ietf-ppsp-reqs] running over UDP. The PPSP charter requires existing media transfer protocols be used [PPSPCHART]. Hence, the general idea is to define swift as a profile of RTP, in the same way as the Secure Real-time Transport Protocol (SRTP) [RFC3711]. SRTP, and therefore swift is considered ``a "bump in the stack" implementation which resides between the RTP application and the transport layer. [swift] intercepts RTP packets and then forwards an equivalent [swift] packet on the sending side, and intercepts [swift] packets and passes an equivalent RTP packet up the stack on the receiving side.'' [RFC3711].

In particular, to encode a swift datagram in an RTP packet all the non-DATA messages of swift such as HINT and HAVE are postfixed to the RTP packet using the UDP encoding and the content of DATA messages is sent in the payload field. Implementations MAY omit the RTP header for packets without payload. This construction allows the streaming application to use of all RTP's current features, and with a modification to the Merkle tree hashing scheme (see below) meets

swift's atomic datagram principle. The latter means that a receiving peer can autonomously verify the RTP packet as being correct content, thus preventing the spread of corrupt data (see requirement PPSP.SEC-REQ-4).

The use of ACK messages for reliability is left as a choice of the application using PPSP.

6.3.1. Design

6.3.1.1. Joining a Swarm

To commence a PPSP download a peer A must have the swarm ID of the stream and a list of one or more tracker contact points (e.g. host+port). The list of trackers is optional in the presence of a decentralized tracking mechanism. The swarm ID consists of the swift root hash of the content, which is divided into chunks (see Discussion).

Peer A now registers with the PPSP tracker following the tracker protocol [I-D.ietf.ppsp-reqs] and receives the IP address and RTP port of peers already in the swarm, say B, C, and D. Peer A now sends an RTP packet containing a HANDSHAKE without channel information to B, C, and D. This serves as an end-to-end check that the peers are actually in the correct swarm. Optionally A could include a HINT message in some RTP packets if it wants to start receiving content immediately. B and C respond with a HANDSHAKE and HAVE messages. D sends just a HANDSHAKE and omits HAVE messages as a way of choking A.

6.3.1.2. Exchanging Chunks

In response to B and C, A sends new RTP packets to B and C with HINTs for disjunct sets of chunks. B and C respond with the requested chunks in the payload and HAVE messages, updating their chunk availability. Upon receipt, A sends HAVE for the chunks received and new HINT messages to B and C. When e.g. C finds that A obtained a chunk (from B) that C did not yet have, C's response includes a HINT for that chunk.

D does not send HAVE messages, instead if D decides to unchoke peer A, it sends an RTP packet with HAVE messages to inform A of its current availability. If B or C decide to choke A they stop sending HAVE and DATA messages and A should then rerequest from other peers. They may continue to send HINT messages, or exponentially slowing KEEPALIVE messages such that A keeps sending them HAVE messages.

Once A has received all content (video-on-demand use case) it stops sending messages to all other peers that have all content (a.k.a. seeders).

6.3.1.3. Leaving a Swarm

Peers can implicitly leave a swarm by stopping to respond to messages. Sending peers should remove these peers from the current peer list. This mechanism works for both graceful and ungraceful leaves (i.e., peer crashes or disconnects). When leaving gracefully, a peer should deregister from the tracker following the PPSP tracker protocol.

More explicit graceful leaves could be implemented using RTCP. In particular, a peer could send a RTCP BYE on the RTCP port that is derivable from a peer's RTP port for all peers in its current peer list. However, to prevent malicious peers from sending BYEs a form of peer authentication is required (e.g. using public keys as peer IDs [PERMIDS].)

6.3.1.4. Discussion

Using swift as an RTP profile requires a change to the content integrity protection scheme (see Sec. 3.5). The fields in the RTP header, such as the timestamp and PT fields, must be protected by the Merkle tree hashing scheme to prevent malicious alterations. Therefore, the Merkle tree is no longer constructed from pure content chunks, but from the complete RTP packet for a chunk as it would be transmitted (minus the non-DATA swift messages). In other words, the hash of the leaves in the tree is the hash over the Authenticated Portion of the RTP packet as defined by SRTP, illustrated in the following figure (extended from [RFC3711]). There is no need for the RTP packets to be fixed size, as the hashing scheme can deal with variable-sized leaves.

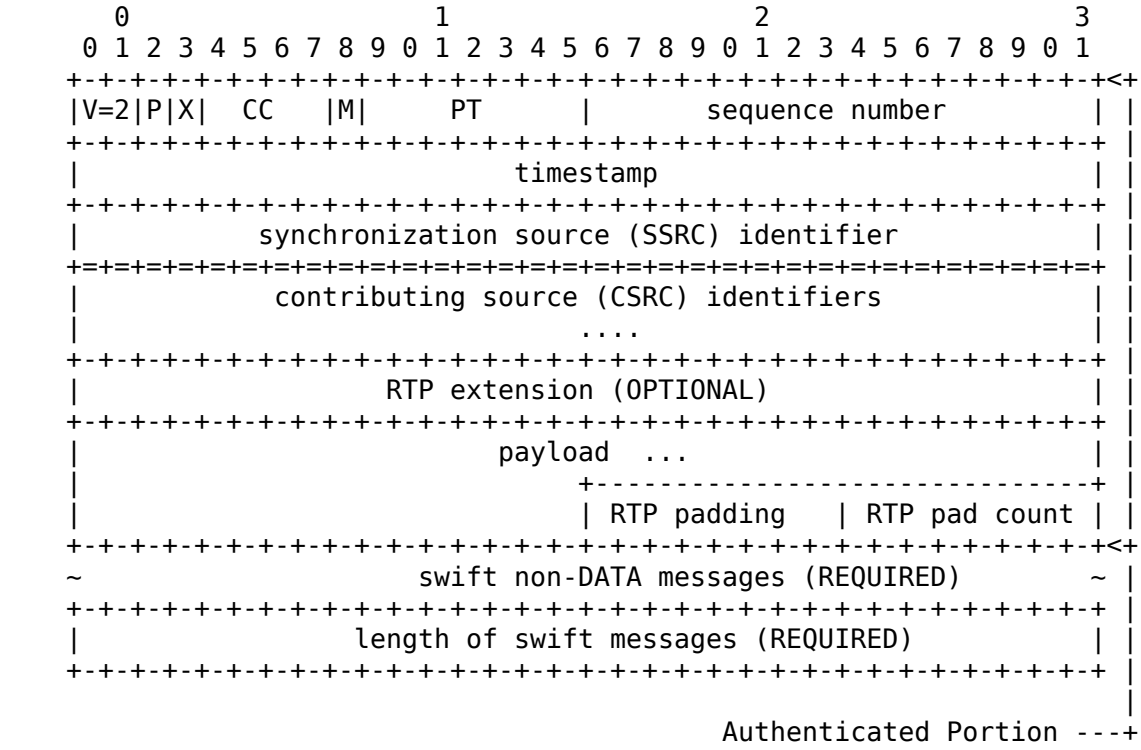


Figure: The format of an RTP-Swift packet.

As a downside, with variable-sized payloads the automatic content size detection of Section 4 no longer works, so content length MUST be explicit in the metadata. In addition, storage on disk is more complex with out-of-order, variable-sized packets. On the upside, carrying RTP over swift allow decryption-less caching.

As with UDP, another matter is how much data is carried inside each packet. An important swift-specific factor here is the resulting number of hash calculations per second needed to verify chunks. Experiments should be conducted to ensure they are not excessive for, e.g., mobile hardware.

At present, Peer IDs are not required in this design.

6.3.2. PPSP Requirements

6.3.2.1. Basic Requirements

- PPSP.REQ-1: The swift PEX message can also be used as the basis for

a tracker protocol, to be discussed elsewhere.

- PPSP.REQ-2: This draft preserves the properties of RTP.

- PPSP.REQ-3: This draft does not place requirements on peer IDs, IP+port is sufficient.

- PPSP.REQ-4: The content is identified by its root hash (video-on-demand) or a public key (live streaming).

- PPSP.REQ-5: The content is partitioned by the streaming application.

- PPSP.REQ-6: Each chunk is identified by a bin number (and its cryptographic hash.)

- PPSP.REQ-7: The protocol is carried over UDP because RTP is.

- PPSP.REQ-8: The protocol has been designed to allow meaningful data transfer between peers as soon as possible and to avoid unnecessary round-trips. It supports small and variable chunk sizes, and its content integrity protection enables wide scale caching.

6.3.2.2. Peer Protocol Requirements

- PPSP.PP.REQ-1: A GET_HAVE would have to be added to request which chunks are available from a peer, if the proposed push-based HAVE mechanism is not sufficient.

- PPSP.PP.REQ-2: A set of HAVE messages satisfies this.

- PPSP.PP.REQ-3: The PEX_REQ message satisfies this. Care should be taken with peer address exchange in general, as the use of such hearsay is a risk for the protocol as it may be exploited by malicious peers (as a DDoS attack mechanism). A secure tracking / peer sampling protocol like [PUPPETCAST] may be needed to make peer-address exchange safe.

- PPSP.PP.REQ-4: HAVE messages convey current availability via a push model.

- PPSP.PP.REQ-5: Bin numbering enables a compact representation of chunk availability.

- PPSP.PP.REQ-6: A new PPSP specific Peer Report message would have to be added to RTCP.

- PPSP.PP.REQ-7: Transmission and chunk requests are integrated in this protocol.

6.3.2.3. Security Requirements

- PPSP.SEC.REQ-1: An access control mechanism like Closed Swarms [CLOSED] would have to be added.

- PPSP.SEC.REQ-2: As RTP is carried verbatim over swift, RTP encryption can be used. Note that just encrypting the RTP part will allow for caching servers that are part of the swarm but do not need access to the decryption keys. They just need access to the swift HASHES in the postfix to verify the packet's integrity.

- PPSP.SEC.REQ-3: RTP encryption or IPsec [RFC4303] can be used, if the swift messages must also be encrypted.

- PPSP.SEC.REQ-4: The Merkle tree hashing scheme prevents the indirect spread of corrupt content, as peers will only forward chunks to others if their integrity check out. Another protection mechanism is to not depend on hearsay (i.e., do not forward other peers' availability information), or to only use it when the information spread is self-certified by its subjects.

Other attacks, such as a malicious peer claiming it has content but not replying, are still possible. Or wasting CPU and bandwidth at a receiving peer by sending packets where the DATA doesn't match the HASHes.

- PPSP.SEC.REQ-5: The Merkle tree hashing scheme allows a receiving peer to detect a malicious or faulty sender, which it can subsequently ignore. Spreading this knowledge to other peers such that they know about this bad behavior is hearsay.

- PPSP.SEC.REQ-6: A risk in peer-to-peer streaming systems is that malicious peers launch an Eclipse [ECLIPSE] attack on the initial injectors of the content (in particular in live streaming). The attack tries to let the injector upload to just malicious peers which then do not forward the content to others, thus stopping the distribution. An Eclipse attack could also be launched on an individual peer. Letting these injectors only use trusted trackers that provide true random samples of the population or using a secure peer sampling service [PUPPETCAST] can help negate such an attack.

- PPSP.SEC.REQ-7: swift supports decentralized tracking via PEX or additional mechanisms such as DHTs [SECDHTS], but self-certification of addresses is needed. Self-certification means For example, that each peer has a public/private key pair [PERMIDS] and creates self-certified address changes that include the swarm ID and a timestamp, which are then exchanged among peers or stored in DHTs. See also discussion of PPSP.PP.REQ-3 above. Content distribution can continue as long as there are peers that have it available.

- PPSP.SEC.REQ-8: The verification of data via hashes obtained from a trusted source is well-established in the BitTorrent protocol [BITTORRENT]. The proposed Merkle tree scheme is a secure extension of this idea. Self-certification and not using hearsay are other lessons learned from existing distributed systems.

- PPSP.SEC.REQ-9: Swift has built-in content integrity protection via self-certified naming of content, see SEC.REQ-5 and Sec. 3.5.1.

6.4. HTTP (as PPSP)

In this section we sketch how swift can be carried over HTTP [RFC2616] to form the PPSP running over TCP. The general idea is to encode a swift datagram in HTTP GET and PUT requests and their replies by transmitting all the non-DATA messages such as HINTs and HAVEs as headers and send DATA messages in the body. This idea follows the atomic datagram principle for each request and reply. So a receiving peer can autonomously verify the message as carrying correct data, thus preventing the spread of corrupt data (see requirement PPSP.SEC-REQ-4).

A problem with HTTP is that it is a client/server protocol. To overcome this problem, a peer A uses a PUT request instead of a GET request if the peer B has indicated in a reply that it wants to retrieve a chunk from A. In cases where peer A is no longer interested in receiving requests from B (described below) B may need to establish a new HTTP connection to A to quickly download a chunk, instead of waiting for a convenient time when A sends another request. As an alternative design, two HTTP connections could be used always., but this is inefficient.

6.4.1. Design

6.4.1.1. Joining a Swarm

To commence a PPSP download a peer A must have the swarm ID of the stream and a list of one or more tracker contact points, as above. The swarm ID as earlier also consists of the swift root hash of the

content, divided in chunks by the streaming application (e.g. fixed-size chunks of 1 kilobyte for video-on-demand).

Peer A now registers with the PPSP tracker following the tracker protocol [I-D.ietf-ppsp-reqs] and receives the IP address and HTTP port of peers already in the swarm, say B, C, and D. Peer A now establishes persistent HTTP connections with B, C, D and sends GET requests with the Request-URI set to /<encoded roothash>. Optionally A could include a HINT message in some requests if it wants to start receiving content immediately. A HINT is encoded as a Range header with a new "bins" unit [RFC2616,$14.35].

B and C respond with a 200 OK reply with header-encoded HAVE messages. A HAVE message is encoded as an extended Accept-Ranges: header [RFC2616,$14.5] with the new bins unit and the possibility of listing the set of accepted bins. If no HINT/Range header was present in the request, the body of the reply is empty. D sends just a 200 OK reply and omits the HAVE/Accept-Ranges header as a way of choking A.

6.4.1.2. Exchanging Chunks

In response to B and C, A sends GET requests with Range headers, requesting disjunct sets of chunks. B and C respond with 206 Partial Content replies with the requested chunks in the body and Accept-Ranges headers, updating their chunk availability. The HASHES for the chunks are encoded in a new Content-Merkle header and the Content-Range is set to identify the chunk [RFC2616,$14.16]. A new "multipart-bin ranges" equivalent to the "multipart-bytes ranges" media type may be used to transmit multiple chunks in one reply.

Upon receipt, A sends a new GET request with a HAVE/Accept-Ranges header for the chunks received and new HINT/Range headers to B and C. Now when e.g. C finds that A obtained a chunk (from B) that C did not yet have, C's response includes a HINT/Range for that chunk. In this case, A's next request to C is not a GET request, but a PUT request with the requested chunk sent in the body.

Again, working around the fact that HTTP is a client/server protocol, peer A periodically sends HEAD requests to peer D (which was virtually choking A) that serve as keepalives and may contain HAVE/Accept-Ranges headers. If D decides to unchoke peer A, it includes an Accept-Ranges header in the "200 OK" reply to inform A of its current chunk availability.

If B or C decide to choke A they start responding with 204 No Content replies without HAVE/Accept-Ranges headers and A should then re-request from other peers. However, if their replies contain HINT/Range headers A should keep on sending PUT requests with the

desired data (another client/server workaround). If not, A should slowly send HEAD requests as keepalive and content availability update.

Once A has received all content (video-on-demand use case) it closes the persistent connections to all other peers that have all content (a.k.a. seeders).

6.4.1.3. Leaving a Swarm

Peers can explicitly leave a swarm by closing the connection. This mechanism works for both graceful and ungraceful leaves (i.e., peer crashes or disconnects). When leaving gracefully, a peer should deregister from the tracker following the PPSP tracker protocol.

6.4.1.4. Discussion

As mentioned earlier, this design suffers from the fact that HTTP is a client/server protocol. A solution where a peer establishes two HTTP connections with every other peer may be more elegant, but inefficient. The mapping of swift messages to headers remains the same:

```
HINT = Range
HAVE = Accept-Ranges
HASH = Content-Merkle
PEX = e.g. extended Content-Location
```

The Content-Merkle header should include some parameters to indicate the hash function and chunk size (e.g. SHA1 and 1K) used to build the Merkle tree.

6.4.2. PPSP Requirements

6.4.2.1. Basic Requirements

- PPSP.REQ-1: The HTTP-based BitTorrent tracker protocol [BITTORRENT] can be used as the basis for a tracker protocol, to be discussed elsewhere.

- PPSP.REQ-2: This draft preserves the properties of HTTP, but extra mechanisms may be necessary to protect against faulty or malicious peers.

- PPSP.REQ-3: This draft does not place requirements on peer IDs,

IP+port is sufficient.

- PPSP.REQ-4: The content is identified by its root hash (video-on-demand) or a public key (live streaming).

- PPSP.REQ-5: The content is partitioned into chunks by the streaming application (see 6.4.1.1.)

- PPSP.REQ-6: Each chunk is identified by a bin number (and its cryptographic hash.)

- PPSP.REQ-7: The protocol is carried over TCP because HTTP is.

6.4.2.2. Peer Protocol Requirements

- PPSP.PP.REQ-1: A HEAD request can be used to find out which chunks are available from a peer, which returns the new Accept-Ranges header.

- PPSP.PP.REQ-2: The new Accept-Ranges header satisfies this.

- PPSP.PP.REQ-3: A GET with a request-URI requesting the peers of a resource (e.g. /<encoded roothash>/peers) would have to be added to request known peers from a peer, if the proposed push-based PEX/~Content-Location mechanism is not sufficient. Care should be taken with peer address exchange in general, as the use of such hearsay is a risk for the protocol as it may be exploited by malicious peers (as a DDoS attack mechanism). A secure tracking / peer sampling protocol like [PUPPETCAST] may be needed to make peer-address exchange safe.

- PPSP.PP.REQ-4: HAVE/Accept-Ranges headers convey current availability.

- PPSP.PP.REQ-5: Bin numbering enables a compact representation of chunk availability.

- PPSP.PP.REQ-6: A new PPSP specific Peer-Report header would have to be added.

- PPSP.PP.REQ-7: Transmission and chunk requests are integrated in this protocol.

6.4.2.3. Security Requirements

- PPSP.SEC.REQ-1: An access control mechanism like Closed Swarms [CLOSED] would have to be added.

- PPSP.SEC.REQ-2: As swift is carried over HTTP, HTTPS encryption can be used instead. Alternatively, just the body could be encrypted. The latter allows for caching servers that are part of the swarm but do not need access to the decryption keys (they need access to the swift HASHES in the headers to verify the packet's integrity).

- PPSP.SEC.REQ-3: HTTPS encryption or the content encryption facilities of HTTP can be used.

- PPSP.SEC.REQ-4: The Merkle tree hashing scheme prevents the indirect spread of corrupt content, as peers will only forward content to others if its integrity checks out. Another protection mechanism is to not depend on hearsay (i.e., do not forward other peers' availability information), or to only use it when the information spread is self-certified by its subjects.

Other attacks such as a malicious peer claiming it has content, but not replying are still possible. Or wasting CPU and bandwidth at a receiving peer by sending packets where the body doesn't match the HASH/Content-Merkle headers.

- PPSP.SEC.REQ-5: The Merkle tree hashing scheme allows a receiving peer to detect a malicious or faulty sender, which it can subsequently close its connection to and ignore. Spreading this knowledge to other peers such that they know about this bad behavior is hearsay.

- PPSP.SEC.REQ-6: A risk in peer-to-peer streaming systems is that malicious peers launch an Eclipse [ECLIPSE] attack on the initial injectors of the content (in particular in live streaming). The attack tries to let the injector upload to just malicious peers which then do not forward the content to others, thus stopping the distribution. An Eclipse attack could also be launched on an individual peer. Letting these injectors only use trusted trackers that provide true random samples of the population or using a secure peer sampling service [PUPPETCAST] can help negate such an attack.

- PPSP.SEC.REQ-7: swift supports decentralized tracking via PEX or additional mechanisms such as DHTs [SECDHTS], but self-certification of addresses is needed. Self-certification means For example, that

each peer has a public/private key pair [PERMIDS] and creates self-certified address changes that include the swarm ID and a timestamp, which are then exchanged among peers or stored in DHTs. See also discussion of PPSP.PP.REQ-3 above. Content distribution can continue as long as there are peers that have it available.

- PPSP.SEC.REQ-8: The verification of data via hashes obtained from a trusted source is well-established in the BitTorrent protocol [BITTORRENT]. The proposed Merkle tree scheme is a secure extension of this idea. Self-certification and not using hearsay are other lessons learned from existing distributed systems.

- PPSP.SEC.REQ-9: Swift has built-in content integrity protection via self-certified naming of content, see SEC.REQ-5 and Sec. 3.5.1.

7. Security Considerations

As any other network protocol, the swift faces a common set of security challenges. An implementation must consider the possibility of buffer overruns, DoS attacks and manipulation (i.e. reflection attacks). Any guarantee of privacy seems unlikely, as the user is exposing its IP address to the peers. A probable exception is the case of the user being hidden behind a public NAT or proxy.

8. Extensibility

8.1. 32 bit vs 64 bit

While in principle the protocol supports bigger (>1TB) files, all the mentioned counters are 32-bit. It is an optimization, as using 64-bit numbers on-wire may cost ~2% practical overhead. The 64-bit version of every message has typeid of 64+t, e.g. typeid 68 for 64-bit hash message:

44 000000000000000E 01234567890ABCDEF1234567890ABCDEF1234567

8.2. IPv6

IPv6 versions of PEX messages use the same 64+t shift as just mentioned.

8.3. Congestion Control Algorithms

Congestion control algorithm is left to the implementation and may even vary from peer to peer. Congestion control is entirely implemented by the sending peer, the receiver only provides clues,

such as hints, acknowledgments and timestamps. In general, it is expected that servers would use TCP-like congestion control schemes such as classic AIMD or CUBIC [CUBIC]. End-user peers are expected to use weaker-than-TCP (least than best effort) congestion control, such as [LEDBAT] to minimize seeding counter-incentives.

8.4. Piece Picking Algorithms

Piece picking entirely depends on the receiving peer. The sender peer is made aware of preferred pieces by the means of HINT messages. In some scenarios it may be beneficial to allow the sender to ignore those hints and send unrequested data.

8.5. Reciprocity Algorithms

Reciprocity algorithms are the sole responsibility of the sender peer. Reciprocal intentions of the sender are not manifested by separate messages (as BitTorrent's CHOKE/UNCHOKE), as it does not guarantee anything anyway (the "snubbing" syndrome).

8.6. Different crypto/hashing schemes

Once a flavor of swift will need to use a different crypto scheme (e.g., SHA-256), a message should be allocated for that. As the root hash is supplied in the handshake message, the crypto scheme in use will be known from the very beginning. As the root hash is the content's identifier, different schemes of crypto cannot be mixed in the same swarm; different swarms may distribute the same content using different crypto.

9. Rationale

Historically, the Internet was based on end-to-end unicast and, considering the failure of multicast, was addressed by different technologies, which ultimately boiled down to maintaining and coordinating distributed replicas. On one hand, downloading from a nearby well-provisioned replica is somewhat faster and/or cheaper; on the other hand, it requires to coordinate multiple parties (the data source, mirrors/CDN sites/peers, consumers). As the Internet progresses to richer and richer content, the overhead of peer/replica coordination becomes dwarfed by the mass of the download itself. Thus, the niche for multiparty transfers expands. Still, current, relevant technologies are tightly coupled to a single use case or even infrastructure of a particular corporation. The mission of our

project is to create a generic content-centric multiparty transport protocol to allow seamless, effortless data dissemination on the Net.

TABLE 1. Use cases.

	mirror-based	peer-assisted	peer-to-peer
data	SunSITE	CacheLogic VelociX	BitTorrent
VoD	YouTube	Azureus(+seedboxes)	SwarmPlayer
live	Akamai Str.	Octoshape, Joost	PPlive

The protocol must be designed for maximum genericity, thus focusing on the very core of the mission, contain no magic constants and no hardwired policies. Effectively, it is a set of messages allowing to securely retrieve data from whatever source available, in parallel. Ideally, the protocol must be able to run over IP as an independent transport protocol. Practically, it must run over UDP and TCP.

9.1. Design Goals

The technical focus of the swift protocol is to find the simplest solution involving the minimum set of primitives, still being sufficient to implement all the targeted usecases (see Table 1), suitable for use in general-purpose software and hardware (i.e. a web browser or a set-top box). The five design goals for the protocol are:

1. Embeddable kernel-ready protocol.
2. Embrace real-time streaming, in- and out-of-order download.
3. Have short warm-up times.
4. Traverse NATs transparently.
5. Be extensible, allow for multitude of implementation over diverse mediums, allow for drop-in pluggability.

The objectives are referenced as (1)-(5).

The goal of embedding (1) means that the protocol must be ready to function as a regular transport protocol inside a set-top box, mobile device, a browser and/or in the kernel space. Thus, the protocol must have light footprint, preferably less than TCP, in spite of the necessity to support numerous ongoing connections as well as to constantly probe the network for new possibilities. The practical overhead for TCP is estimated at 10KB per connection [HTTP1MLN]. We aim at <1KB per peer connected. Also, the amount of code necessary to make a basic implementation must be limited to 10KLoC of C. Otherwise, besides the resource considerations, maintaining and auditing the code might become prohibitively expensive.

The support for all three basic usecases of real-time streaming, in-order download and out-of-order download (2) is necessary for the manifested goal of THE multiparty transport protocol as no single usecase dominates over the others.

The objective of short warm-up times (3) is the matter of end-user experience; the playback must start as soon as possible. Thus any unnecessary initialization roundtrips and warm-up cycles must be eliminated from the transport layer.

Transparent NAT traversal (4) is absolutely necessary as at least 60% of today's users are hidden behind NATs. NATs severely affect connection patterns in P2P networks thus impacting performance and fairness [MOLNAT,LUCNAT].

The protocol must define a common message set (5) to be used by implementations; it must not hardwire any magic constants, algorithms or schemes beyond that. For example, an implementation is free to use its own congestion control, connection rotation or reciprocity algorithms. Still, the protocol must enable such algorithms by supplying sufficient information. For example, trackerless peer discovery needs peer exchange messages, scavenger congestion control may need timestamped acknowledgments, etc.

9.2. Not TCP

To large extent, swift's design is defined by the cornerstone decision to get rid of TCP and not to reinvent any TCP-like transports on top of UDP or otherwise. The requirements (1), (4), (5) make TCP a bad choice due to its high per-connection footprint, complex and less reliable NAT traversal and fixed predefined congestion control algorithms. Besides that, an important consideration is that no block of TCP functionality turns out to be useful for the general case of swarming downloads. Namely,

1. in-order delivery is less useful as peer-to-peer protocols often employ out-of-order delivery themselves and in either case out-of-order data can still be stored;
2. reliable delivery/retransmissions are not useful because the same data might be requested from different sources; as in-order delivery is not required, packet losses might be patched up lazily, without stopping the flow of data;
3. flow control is not necessary as the receiver is much less likely to be saturated with the data and even if so, that situation is perfectly detected by the congestion control;
4. TCP congestion control is less useful as custom congestion control is often needed [LEDBAT].

In general, TCP is built and optimized for a different usecase than

we have with swarming downloads. The abstraction of a "data pipe" orderly delivering some stream of bytes from one peer to another turned out to be irrelevant. In even more general terms, TCP supports the abstraction of pairwise _conversations_, while we need a content-centric protocol built around the abstraction of a cloud of participants disseminating the same _data_ in any way and order that is convenient to them.

Thus, the choice is to design a protocol that runs on top of unreliable datagrams. Instead of reimplementing TCP, we create a datagram-based protocol, completely dropping the sequential data stream abstraction. Removing unnecessary features of TCP makes it easier both to implement the protocol and to verify it; numerous TCP vulnerabilities were caused by complexity of the protocol's state machine. Still, we reserve the possibility to run swift on top of TCP or HTTP.

Pursuing the maxim of making things as simple as possible but not simpler, we fit the protocol into the constraints of the transport layer by dropping all the transmission's technical metadata except for the content's root hash (compare that to metadata files used in BitTorrent). Elimination of technical metadata is achieved through the use of Merkle [MERKLE,ABMRKL] hash trees, exclusively single-file transfers and other techniques. As a result, a transfer is identified and bootstrapped by its root hash only.

To avoid the usual layering of positive/negative acknowledgment mechanisms we introduce a scale-invariant acknowledgment system (see Sec 4.4). The system allows for aggregation and variable level of detail in requesting, announcing and acknowledging data, serves in-order and out-of-order retrieval with equal ease. Besides the protocol's footprint, we also aim at lowering the size of a minimal useful interaction. Once a single datagram is received, it must be checked for data integrity, and then either dropped or accepted, consumed and relayed.

9.3. Generic Acknowledgments

Generic acknowledgments came out of the need to simplify the data addressing/requesting/acknowledging mechanics, which tends to become overly complex and multilayered with the conventional approach. Take the BitTorrent+TCP tandem for example:

1. The basic data unit is a byte of content in a file.
2. BitTorrent's highest-level unit is a "torrent", physically a byte range resulting from concatenation of content files.

3. A torrent is divided into "pieces", typically about a thousand of them. Pieces are used to communicate progress to other peers. Pieces are also basic data integrity units, as the torrent's metadata includes a SHA1 hash for every piece.
4. The actual data transfers are requested and made in 16KByte units, named "blocks" or chunks.
5. Still, one layer lower, TCP also operates with bytes and byte offsets which are totally different from the torrent's bytes and offsets, as TCP considers cumulative byte offsets for all content sent by a connection, be it data, metadata or commands.
6. Finally, another layer lower, IP transfers independent datagrams (typically around 1.5 kilobyte), which TCP then reassembles into continuous streams.

Obviously, such addressing schemes need lots of mappings; from piece number and block to file(s) and offset(s) to TCP sequence numbers to the actual packets and the other way around. Lots of complexity is introduced by mismatch of bounds: packet bounds are different from file, block or hash/piece bounds. The picture is typical for a codebase which was historically layered.

To simplify this aspect, we employ a generic content addressing scheme based on binary intervals, or "bins" for short.

Acknowledgements

Victor Grishchenko and Arno Bakker are partially supported by the P2P-Next project (http://www.p2p-next.org/), a research project supported by the European Community under its 7th Framework Programme (grant agreement no. 216217). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the P2P-Next project or the European Commission.

The authors would like to thank the following people for their contributions to this draft: Mihai Capota, Raul Jiminez, Flutra Osmani, Riccardo Petrocco, Johan Pouwelse, and Raynor Vliegendhart.

References

[RFC2119] Key words for use in RFCs to Indicate Requirement Levels
[HTTP1MLN] Richard Jones. "A Million-user Comet Application with Mochiweb", Part 3. http://www.metabrew.com/article/a-million-user-comet-application-with-mochiweb-part-3

[MOLNAT] J.J.D. Mol, J.A. Pouwelse, D.H.J. Epema and H.J. Sips:
"Free-riding, Fairness, and Firewalls in P2P File-Sharing"
[LUCNAT] submitted
[BINMAP] V. Grishchenko, J. Pouwelse: "Binmaps: hybridizing bitmaps
and binary trees"
http://www.tribler.org/download/binmaps-alenex.pdf
[SNP] B. Ford, P. Srisuresh, D. Kegel: "Peer-to-Peer Communication
Across Network Address Translators",
http://www.brynosaurus.com/pub/net/p2pnat/
[FIPS180-2]
Federal Information Processing Standards Publication 180-2:
"Secure Hash Standard" 2002 August 1.
[MERKLE] Merkle, R. "A Digital Signature Based on a Conventional
Encryption Function". Proceedings CRYPTO'87, Santa Barbara, CA,
USA, Aug 1987. pp 369-378.
[ABMRKL] Arno Bakker: "Merkle hash torrent extension", BEP 30,
http://bittorrent.org/beps/bep_0030.html
[CUBIC] Injong Rhee, and Lisong Xu: "CUBIC: A New TCP-Friendly
High-Speed TCP Variant",
http://www4.ncsu.edu/~rhee/export/bitcp/cubic-paper.pdf
[LEDBAT] S. Shalunov: "Low Extra Delay Background Transport (LEDBAT)"
http://www.ietf.org/id/draft-ietf-ledbat-congestion-00.txt
[TIT4TAT] Bram Cohen: "Incentives Build Robustness in BitTorrent", 2003,
http://www.bittorrent.org/bittorrentecon.pdf
[BITTORRENT] B. Cohen, "The BitTorrent Protocol Specification",
February 2008, http://www.bittorrent.org/beps/bep_0003.html
[RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and V.
Jacobson, "RTP: A Transport Protocol for Real-Time
Applications", STD 64, RFC 3550, July 2003.
[RFC3711] M. Baugher, D. McGrew, M. Naslund, E. Carrara, K. Norrman,
"The Secure Real-time Transport Protocol (SRTP), RFC 3711, March
2004.
[RFC5389] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing,
"Session Traversal Utilities for NAT (STUN)", RFC 5389, October 2008.
[I-D.ietf-ppsp-reqs] Zong, N., Zhang, Y., Pascual, V., Williams, C.,
and L. Xiao, "P2P Streaming Protocol (PPSP) Requirements",
draft-ietf-ppsp-reqs-05 (work in progress), October 2011.
[PPSPCHART] Stiemerling et al. "Peer to Peer Streaming Protocol (ppsp)
Description of Working Group"
http://datatracker.ietf.org/wg/ppsp/charter/
[PERMIDS] A. Bakker et al. "Next-Share Platform M8--Specification
Part", App. C. P2P-Next project deliverable D4.0.1 (revised),
June 2009.
http://www.p2p-next.org/download.php?
id=E7750C654035D8C2E04D836243E6526E
[PUPPETCAST] A. Bakker and M. van Steen. "PuppetCast: A Secure Peer
Sampling Protocol". Proceedings 4th Annual European Conference on
Computer Network Defense (EC2ND'08), pp. 3-10, Dublin, Ireland,
11-12 December 2008.

[CLOSED] N. Borch, K. Michell, I. Arntzen, and D. Gabrijelcic: "Access control to BitTorrent swarms using closed swarms". In Proceedings of the 2010 ACM workshop on Advanced video streaming techniques for peer-to-peer networks and social networking (AVSTP2P '10). ACM, New York, NY, USA, 25-30. http://doi.acm.org/10.1145/1877891.1877898

[ECLIPSE] E. Sit and R. Morris, "Security Considerations for Peer-to-Peer Distributed Hash Tables", IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems, pp. 261-269, Springer-Verlag, London, UK, 2002.

[SECDHTS] G. Urdaneta, G. Pierre, M. van Steen, "A Survey of DHT Security Techniques", ACM Computing Surveys, vol. 43(2), June 2011.

[HTTP] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC2616, June 1999.

[SWIFTIMPL] V. Grishchenko, et al. "Swift M40 reference implementation", http://swarmplayer.p2p-next.org/download/Next-Share-M40.tar.bz2 (subdirectory Next-Share/TUD/swift-trial-r2242/), July 2011.

[CCNWIKI] http://en.wikipedia.org/wiki/Content-centric_networking

[HAC01] A.J. Menezes, P.C. van Oorschot and S.A. Vanstone. "Handbook of Applied Cryptography", CRC Press, October 1996 (Fifth Printing, August 2001).

[JIM11] R. Jimenez, F. Osmani, and B. Knutsson. "Sub-Second Lookups on a Large-Scale Kademlia-Based Overlay". 11th IEEE International Conference on Peer-to-Peer Computing 2011, Kyoto, Japan, Aug. 2011

Authors' addresses

A. Bakker
Technische Universiteit Delft
Department EWI/ST/PDS
Room HB 9.160
Mekelweg 4
2628CD Delft
The Netherlands

Email: arno@cs.vu.nl

3.3 Multiparty Protocol in the Linux Kernel

swift has been designed to act as a BitTorrent-like replacement for TCP and be able to be implemented as a Transport layer protocol in the operating system networking stack. In this respect, a initial porting of the protocol in the Linux kernel networking stack has been undertaken. We have designed the kernel space/user space component and created a Linux kernel protocol development framework. In order to allow rapid development, a user space raw socket-based implementation has been created. It provides a BSD socket-like API making it compatible with the kernel syscall interface. Doing a compatible implementation in user space is important to reduce the high development and debugging time that is common to kernel development. The source code can be found in the Next-Share/UPB/kernel-swift-r3192 directory of the Code Part of this deliverable.

The major challenge of designing and implementing a multiparty protocol is splitting features in swift in user space components and kernel space components. The current specification of swift makes it unfeasible to place everything in kernel space, due to a number of issues:

- resource consumption: memory consumption for each piece of content (be it file or stream) is pretty large (binmaps and other data structures);
- it is difficult and not recommended to work with files from kernel space; that is storing in files data received from the network and reading data from files to send it on the network; the recommended practice is receiving/sending data from/to user space and deliver it to the network;
- provided we implement a Transport-layer protocol, it should be only that: a Transport layer protocol responsible for transferring data between processes running on top of different systems on the network; other features should be done in user space as part of a user space Application-layer protocol;
- it would be difficult to provide a socket-like API that is common to all protocol (socket, bind, listen, accept, connect, close, shutdown); adding completely new system calls would make it a very intrusive and very unlikely to be accepted in the mainstream kernel; it would also break compatibility with existing Transport-layer protocols such as TCP, UDP, SCTP, DCCP.

Deciding what components go where (user space or kernel space) is an ongoing activitity. We have decided on a split, but it is still preliminary; updates will adjust the design and implementation according to needs and issues that may be encountered. When creating the split we have focused on the following design choices:

- ensure a socket-like API, make it compatible to other Transport layer protocols; ideally, it should be possible for an existing Application layer protocol to be running on top of the multiparty protocol;

- the protocol implementation should have low memory footprint as described in the original specification; complex features such as binmap management, file/content management should be done in user space;
- the aim is to build multiparty protocol acting as a BitTorrent-like Transport-layer protocol; Peer-to-Peer features have to be present and ensure increased speed/performance when compared to a classical two endpoints communication; redundancy, unordered delivery and internal data transport features of swift should be present in kernel space as part of the multiparty protocol implementation;
- focus on performance; that is consider the overhead of system calls and user space/kernel space code; as much work should be done in kernel space to ensure reduced number of system calls and context switches, but only if it makes sense to do that (see the choices above); zero copying and scatter-gather should be used where possible;
- no decision was made on placing the hash checking and peer discovery modules; due to the binmap management module being present in user space, the integrity checking would probably also happen there; it would be good if this could be transferred to kernel space for reduced overhead (packets could be dropped in the kernel space, without transmitting them to user space); the peer discovery module is planned to be implemented as a user space component running on top of UDP; placing it as a separate service would ensure maximum flexibility while also making it difficult to distribute that information to the kernel space multiparty protocol.

3.3.1 Current Design Proposal of Multiparty Protocol

The multiparty protocol will be implemented at user space level through a raw socket layer to validate our architecture and as a kernel based module in the Linux kernel These two approaches are described in the figure below..

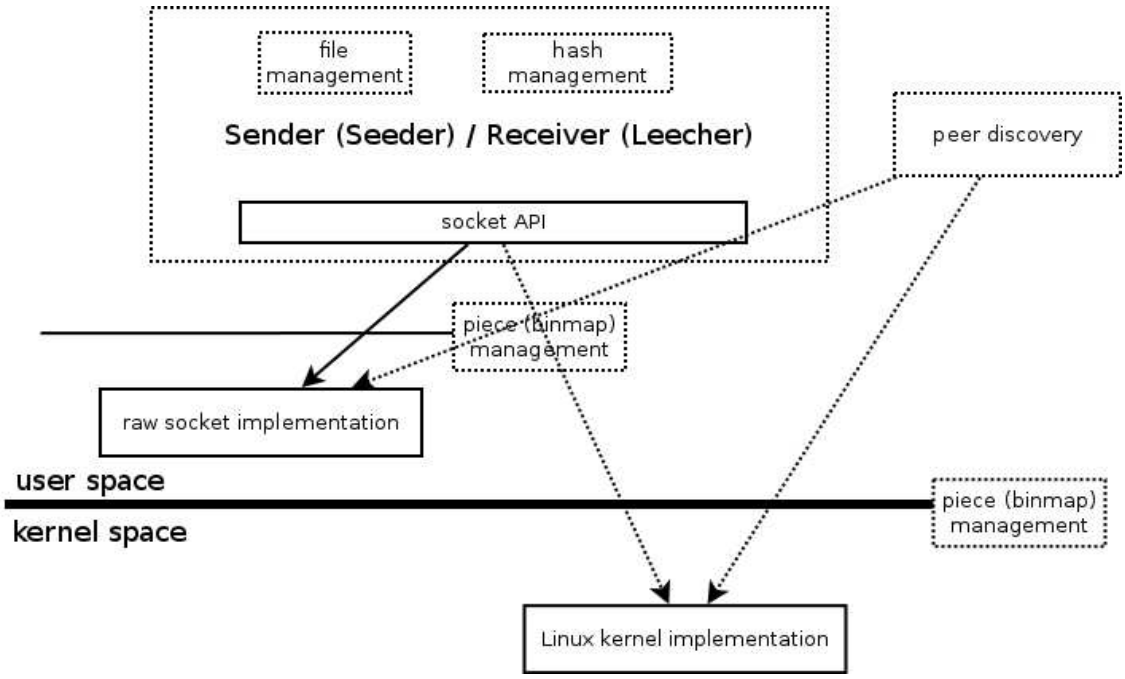


Illustration 1: Architectural Overview

As mentioned above, file/content management will take place in user space. Currently, the peer discovery component is designed as a user space component. The piece management is partially designed to take place in kernel space, with large memory footprint components thought to be implemented in user space. It is still ongoing work on how to split the piece/binmap management such that the protocol would provide less system calls but require little memory consumption in kernel space.

We differentiate between a receiver (also called leecher in BitTorrent environments) and a sender (also called seeder). For the sake of simplicity, we will consider the sender having access to the complete content to serve.

The receiver is the one that requests data data. In order to do this it must connect to the multiparty protocol by creating and binding to a multiparty socket. When it binds to a socket, it uses the hash as a parameter to find a connection with a peer that accesses the respective file. In the current implementation, the hash is equivalent to a port for TCP. This was chosen to ensure a single data channel for each piece of content, though current thoughts are to update it to a port. The discovery is done through the separate peer discovery overlay. The receiver then waits for packets from senders. Requests will be sent by the multiparty protocol where available and their results will be provided to user space. Ideally, a request hash would map to several data pieces such that multiple requests may be completed in a single system call.

A sender is the entity that serves data to receivers. In order to do this it must connect to the multiparty protocol by creating, binding and listening to a multiparty socket. When binding, the sender uses the hash as a parameter. This means that for every hashed there will be a socket on which the seeder that may receive and serve requests. The sender then waits for requests and sends data packets as requested. It isn't yet defined how a request would get to the sender process; it would be best if it could be directly served by the kernel implementation – otherwise, serving it from user space would make the sender nothing more than an UDP-

based sender, giving out packets upon request. This would, however, imply a caching facility in the kernel protocol implementation. At the same time, there needs to be a kernel space to user space communication of requests such that the application may serve them – this remains to be defined.

3.3.2 Raw Socket-based Implementation

Raw mode is basically there to allow to bypass the kernel implementation. It ensures a compatible socket-based API on top of which the user space part may be implemented. Rather than going through the normal layers of encapsulation/decapsulation that the TCP/IP stack on the kernel does, the packet is passed the application that requires it. No TCP/IP processing is done in the kernel networking stack – it's not a processed packet, but a raw packet. The application using the packet is now responsible for stripping off the headers, analyzing the packet, and, more generally, everything the TCP/IP stack normally performs in kernel space.

The raw socket implementation is designed to support all system calls; it acts as a copy of the kernel implementation.

3.3.3 Linux Kernel Framework for Transport-layer Protocol Implementation

In order to implement o transport protocol in the Linux kernel, several implementation tasks have to be completed:

- Defining IPPROTO_$$. This macro will identify the transport protocol. This will be subsequently used for creating a transport protocol socket.
- Defining a transport header. The framework we used defined two 8 bit ports, a source port and a destination port and a 16 bit length field. The latter field is the data length, including the Transport-layer protocol header.

After the above have been completed, several data structures have to be defined, as mentioned below. A data structure that defines the new socket type. This is where information regarding the socket state, such as the destination or the source port, must be saved.

```
struct swift_sock {
   struct inet_sock sock;
   /* swift socket specific data */
   uint8_t src;
   uint8_t dst;
};
```

The protocol definition, used by the socket, including its name and size is defined in a struct proto structure.

```
static struct proto swift_prot = {
   .obj_size = sizeof(struct swift_sock),
```

```
    .owner = THIS_MODULE,
    .name = "SWIFT",
};
```

The most important structure to be defined describes the operations supported by a socket of a given type. For a datagram sending socket, the implementation of release, connect, sendmsg and recvmsg functions is sufficient.

```
static const struct proto_ops swift_ops = {
    .family = PF_INET,
    .owner = THIS_MODULE,
    .release = swift_release,
    .bind = swift_bind,
    .connect = swift_connect,
    …
    .sendmsg = swift_sendmsg,
    .recvmsg = swift_recvmsg,
    .mmap = sock_no_mmap,
    .sendpage = sock_no_sendpage,
};
```

The header for new network packets is set up in the net_protocol header. New packets received directly from the network will fill the protocol field in the IP header with the value of the implemented Transport-layer protocol.

```
static const struct net_protocol swift_protocol = {
    .handler = swift_rcv,
    .no_policy = 1,
    .netns_ok = 1,
};
```

Another data structure has to connect the new protocol to the operations on the its socket. This structure uses two pointers to the struct proto and struct proto_ops data structures:

```
static struct inet_protosw swift_protosw = {
    .type = SOCK_DGRAM,
    .protocol = IPPROTO_SWIFT,
```

```
    .prot = &swift_prot,
    .ops = &swift_ops,
    .no_check = 0,
};
```

As soon as the above structures are defined, the protocol defined will be added to the kernel. The sequence of operations used to establish this step is described below.

```
proto_register(&swift_prot, 1);
inet_add_protocol(&swift_protocol, IPPROTO_SWIFT);
inet_register_protosw(&swift_protosw);
```

Before compiling and inserting the module in the kernel, several socket operations functions have to be filled; for the proposal of a multiparty protocol (a datagram based protocol), this would mean release, bind, connect, sendmsg and recvmsg. A handler for packets received from the network must also be implemented. At protocol level, one must keep a mapping between a port and a socket, meaning that the socket is bound to that port.

Each implemented socket operation function fills different roles:

- release is used for freeing resources (most likely memory) used by the socket.
- bind checks the availability of the port and ties the socket to that port and a source IP address.
- connect maps the current socket to a destination port and IP address.
- sendmsg extracts the destination IP address and port, provided they were passed as arguments from user space. If they were not provided, the module checks whether the socket is connected (through a previous connect call), and, in case no socket is connected, an error is returned. After establishing the receiving end (IP address and port), an skb structure is allocated, specifying the protocol header and the rest of the data. The multiparty header is filled, user space data is copied and the packet is routed. After the routing process, required data is copied and is queued for transmit using the ip_queue_xmit function call.
- recvmsg is responsible for the reverse operation of sendmsg. A datagram is read from the receiving queue through the use of the skb_recv_datagram call. Serveral integrity checks are employed, after which data is copied from the skb structure to user space. At the time, in case the sender address (IP address and port) has been requested (as used by the recvmsg and recvfrom socket API call), required

information is filled in the user space buffer. In the end the skb structure is freed through the use of the skb_free_datagram call.

- The function passed as a handler in the net_protocol structure is invoked when a packed is received. The protocol field in the packet IP header will be used to de-multiplex the packet and call the required function. When the packet is received, several validations will take place. Subsequently, the destination port information will be extracted. A lookup operation returns the socked mapped to the destination port. The skb->cb field is initialized to information regarding the sender. The packed is then added to the receive queue through the use of ip_queue_rcv_skb.

3.3.4 Testing Protocol Implementation

In order to test the validity of the protocol implementation, a set of unit tests have been designed and implemented. We have implemented a test suite for every socket system call, which tests calling scenarios. Unit tests are created to ensure the code quality and to validate if the system call behaves in a similar manner to other system calls.

Secondly, we have implemented a functional test suite to validate a simple work flow. For this purpose we tested a peer that acts as the seeder and a receiver which behaves as a leecher. A small file is transferred between the two entities to ensure proper delivery.

The test suite is mainly implementing using arrays of function pointers or function pointer structures. A top level structure defines test suites for each function exported by the implementation. Each test suite is a series of methods that test a variant of the call of the function.

The top-level test suite is defined by the test_fun_array as seen below:

```
static void (*test_fun_array[])(void) = {
    NULL,
    test_dummy,
    socket_test_suite,
    bind_test_suite,
    getsockname_test_suite,
    getsockopt_test_suite,
    sendto_test_suite,
    recvfrom_test_suite,
    sendmsg_test_suite,
    recvmsg_test_suite,
```

```
    close_test_suite,
};
```

Each member of the array is a test suite for the functions exported by the implementation. Exported functions are basic socket API functions. Each such test suite is a sequential caller of individual test methods, as shown below:

```
void recvfrom_test_suite(void)
{
    start_suite();
    recvfrom_dummy();
    recvfrom_invalid_descriptor();
    recvfrom_descriptor_is_not_a_socket();
    recvfrom_socket_is_not_bound();
    recvfrom_after_sendto_ok();
    recvfrom_after_sendmsg_ok();
}
```

Each individual method is used to test a single aspect of the test suite. For example, the recvfrom_socket_is_not_bound shown below tests the implementation for sending out an error when receiving from a socket that is not bound (that is no bind call was provided).

```
static void recvfrom_socket_is_not_bound(void)
{
    int sockfd;
    struct sockaddr_sw local_addr;
    struct sockaddr_sw remote_addr;
    ssize_t bytes_recv;
    char buffer[BUFSIZ];

    sockfd = sw_socket(PF_INET, SOCK_DGRAM, IPPROTO_SWIFT);
    DIE(sockfd < 0, "sw_socket");

    fill_sockaddr_sw(&local_addr, &remote_addr, "127.0.0.1", "myHash", "127.0.0.1");
        bytes_recv = sw_recvfrom(sockfd, buffer, BUFSIZ, 0, (struct sockaddr *) &remote_addr, sizeof(remote_addr));
```

```
    test( bytes_recv < 0 && errno == EAFNOSUPPORT );
}
```

Tests will be implemented to highlight the requested improvements of the kernel-based implementation over the swift user space implementation. Metrics to be taken into consideration are number of system calls, number of context switches, function call overhead.

3.4 The Control Protocol for Swift Processes

The protocol used by the NextShareCore to control the external swift process doing the actual content sharing as shown in Chapter 2 is defined as follows. Note that this protocol is very similar to the NSSA API protocol defined in D6.5.x by which the browser plugin controls the NextShareCore BitTorrent-based downloads. All commands end with a \r\n sequence. The notation + means the values are concatenated into a single string where the values are separated by spaces:

- **START** + SWIFTURL: Sent by the Core to signal to swift that it should start downloading the swift swarm identified by the URL. A SWIFTURL consists of the scheme tswift followed by a server part identifying the swift tracker (IP/hostname + port) followed by a path consisting of the root hash of the content asset in 2-digit hexadecimal notation. The path may optionally be postfixed with an $ sign followed by the chunk size of the swarm in bytes. Also, the path may optionally be postfixed with an @ sign followed by the duration of the content in seconds. This information will be returned in a X-Content-Duration header by swift's local HTTP server when accessed after a PLAY command, and is necessary for proper playback of videos by the Firefox browser. An example tswift URL looks as follows:
  `tswift://127.0.0.1:20003/12a1a5efc007710f165a9537edfc2de6f3670f2e@888`

- **REMOVE** + roothash_hex + removestate + removecontent: Sent by the Core to instruct swift to stop and remove the download identified by the root hash in 2-digit hexadecimal notation (see above). The first integer “removestate” (1 or 0) says whether all the download state (i.e., checkpoint and persistent hash trees) should be removed from disk. The second integer “removecontent” says whether the (partially) downloaded content asset should also be removed from disk.
- **CHECKPOINT** + roothash_hex: Sent by the Core to let swift create a checkpoint for the download identified by the root hash in 2-digit hexadecimal notation (see above). A checkpont currently writes the “Own” binmap to disk in a .mbinmap file (see Chapter 2).
- **MAXSPEED** + roothash_hex + direction + speed: Sent by the Core to let swift limit the speed (in KiB/s as a float) in the given “direction” (DOWNLOAD or UPLOAD)

for the download identified by the root hash in 2-digit hexadecimal notation (see above).

- **SETMOREINFO** + roothash_hex + onoff: Sent by the Core to let swift send detailed upload and download statistics in JSON format (http://json.org/) via a MOREINFO command. “onoff” should be “1” or “0” to enable or disable, respectively.
- **MOREINFO** + roothash_hex + jsondata: Sent by swift to report detailed up and download statistics following the JSON specification given below.
- **SHUTDOWN:** Sent by the Core to instruct the swift procress to exit.
- **INFO** + roothash_hex + dlstatus + complete/total + dlspeed + ulspeed + numleech + numseeds: Sent by swift to report the download status (see below), the progress in terms of bytes completely downloaded out of the total size (note the total size may vary due to the dynamic content size determination mechanism of swift). The dlspeed and ulspeed are floats in KiB/s. Numleech and numseeds indicate the number of leechers, respectively, seeders the peer is currently connected to. Download status can take the following values:
  - 1 = DLSTATUS_WAITING4HASHCHECK
  - 2 = DLSTATUS_HASHCHECKING
  - 3 = DLSTATUS_DOWNLOADING
  - 4 = DLSTATUS_SEEDING
  -
- **PLAY** + roothash_hex + HTTPURL: Sent by swift when the content asset previously STARTed is playable from the specified HTTPURL (a HTTP URL to swift's HTTP gateway component, see Chapter 2).
- **ERROR** + message + … : Sent by swift when an unknown or malformed command is received (used for debugging purposes).

The jsondata field in the MOREINFO message has the following format:

```
{
    "timestamp":"<float>",
    "channels":[
        {"ip":"<string>",
         "port":<int>,
         "raw_bytes_up":<int>,
         "raw_bytes_down":<int>,
         "bytes_up":<int>,
         "bytes_down":<int>}
    ],
    "raw_bytes_up":<int>,
    "raw_bytes_down":<int>,
    "bytes_up":<int>,
    "bytes_down":<int>
}
```

The timestamp is a number of seconds as a float. The bytes* fields are in bytes, with the raw_ variants counting the bytes sent over the wire, as opposed to the just content bytes sent

and received. The receiver of the MOREINFO message is supposed to calculate the average up and download speeds itself using its preferred metric from the information in consecutive MOREINFO messages.

Chapter 4
Enhanced Closed Swarms Protocol

4.1 Introduction

The Closed swarms (CS) protocol [10][11] is an access control mechanism for controlling the P2P content delivery process. It acts on peer level – enables peers to recognize the authorized peers and to avoid communication with the non-authorized ones. The distinction between authorized and non-authorized peers in the swarm is made based on possession of an authorization credential called proof-of-access (PoA). The peers exchange their credentials right after they establish connection, in a challenge-response messages exchange process. The CS protocol can provide access control in an innovative business content delivery system, in which users would receive graded service – the authorized users would receive additional or better service than the non-authorized ones, for example access to high speed seeders for better performance.

However, the CS protocol lacks of flexibility in the access control – it is applicable only under the same conditions for all authorized users. Moreover, it is vulnerable to man-in-the-middle attacks. An attacker can interfere in the communication between two authorized peers by simply relaying the messages between them. After the authorized peers successfully finish the protocol and start the content delivery, the attacker will be able to read all the exchanged content pieces, since they are not encrypted. Therefore, we add several enhancements to the CS protocol in order to overcome these shortcomings. The enhancements provide additional flexibility in the access control mechanism and fulfil a number of content providers' requirements. In addition, they promise efficient and secure content delivery.

The Enhanced Closed Swarms (ECS) protocol differs from the CS protocol in that it controls the content delivery only in one direction – from the swarm member to the protocol initiator. Moreover, it supports: restriction of the content delivery based on location, provision on different content quality in the same swarm, temporal constraints and load balancing and optimization of the delivery process. The format of the authorization credential and the message exchange process of the ECS protocol, as well as its possible application are explained in detail in Appendix C: Access Control in BitTorrent P2P Networks Using the Enhanced Closed Swarms Protocol .

In this chapter we describe the implementation details of the ECS protocol (Section 4.2) and its integration in the Next-Share platform (Section 4.3). Moreover, we present how ECS protocol can be used for creation and maintenance of a hierarchically structured swarm (Section 4.4). The source code can be found in the Next-Share/BaseLib/Core/ClosedSwarm/ECS* files of the Code Part of this deliverable.

4.2 Implementation

The ECS protocol is implemented in Python programming language. The main functional elements of the ECS protocol are contained in two modules. For detailed specification please refer to the documentation (BaseLib/Core/ClosedSwarm/README).

- The ECS_ClosedSwarms module is based on the CS protocol implementation. It inherits most of the CS protocol implementation interfaces, and furthermore adapts them to provide the added enhancements. The inherited classes and interfaces are abstractions of the credential and the protocol message exchange process. In addition, they fix a few glitches discovered in the CS protocol implementation. Moreover, this module contains classes responsible for management of the ECS protocol, on per swarm and cross swarm basis.
- The ECS_AuthorizationEngine module was developed from scratch. It contains classes for interpretation and evaluation of the expressive and flexible access control policies supported by the ECS protocol, as well as for coordination of this process. The lexer and the parser developed for the specific ECS protocol grammar[1], are according to the Yacc module specifications.

The ECS protocol suggests content encryption as a countermeasure for man-in-the-middle attacks. The encryption should be light for good performance, but secure enough for preventing the attacker to decrypt the content in reasonable time. We decided to use the AES algorithm for this purpose, in CBC mode and with key length of 128 bits. It performed better than Blowfish in our test, which can be seen from the table below. We measured the encryption and decryption times (in ms) and operations, with and without rekeying, for different piece sizes. We used the M2Crypto module in the both, implementation and tests.

Table 1: Performance measurements of encryption/decryption of different piece sizes with the AES and Blowfish algorithms

Process/Performance	Without rekeying		With rekeying	
	Time (ms)	Operations	Time (ms)	Operations
AES Encryption of 256kB	1.79533+-0.68%	557	1.86494+-5.14%	536.2
AES Decryption of 256kB	1.96816+-3.45%	508.1	1.85258+-4.45%	539.8
AES Encryption of 512kB	4.63914+-2.72%	215.6	4.48835+-2.4%	222.8
AES Decryption of 512kB	4.72314+-2.55%	211.7	4.41845+-2.43%	226.3
AES Encryption of 1MB	4.66194+-2.5%	214.5	4.44107+-2.82%	225.2
AES Decryption of 1MB	4.74228+-2.32%	210.9	4.48398+-4.11%	223
BF Encryption of 256kB	2.49674+-0.8%	400.5	2.49198+-0.48%	401.3
BF Decryption of 256kB	2.52692+-2.99%	395.7	2.47133+-0.42%	404.6
BF Encryption of 512kB	5.94719+-1.83%	168.1	6.00546+-2.06%	166.5
BF Decryption of 512kB	6.01544+-1.9%	166.2	6.0129+-2.01%	166.3

[1] Actually, it is slightly different than the grammar in the ECS design. There is only one group of conditions (General), because the other group was mainly for the purpose of provisioning of different content quality in the same swarm, something that is not currently supported.

BF Encryption of 1MB	6.01137+-2.19%	166.4	6.03989+-1.98%	165.6
BF Decryption of 1MB	6.06232+-2.24%	165	5.77029+-4.03%	173.3

In the ECS implementation peer B, instead of sending encrypted key, sends an unencrypted random number that will be used as an initialization vector. This is due to the fact that Elliptic Curves (EC) are used for public-private cryptography, which enables peers to compute a shared secret with the Elliptic Curve Diffie-Hellman protocol. Each of the peers only needs to know the other peer's public key – which is simple because it is contained in its credential.

4.3 Integration

The ECS protocol is designed as BitTorrent extension. It is integrated in BitTorrent with the help of the Extension protocol, using the same message IDs as the CS protocol. In addition, we defined new message ID for the additional message it has.

The ECS protocol implementation aims at having minimal impact on the other Next-Share core implementation. It mainly follows the CS protocol integration, which is done in the (BaseLib/Core/BitTornado/BT1) Connecter module. Furthermore, the management part is integrated in the (BaseLib/Core/APIImplementation) LauncManyCore and the Connecter modules.

We explain the ECS integration in Next-Share through a simple scenario. Peer A starts the BitTorrent protocol with a closed swarm member - peer B. After they successfully complete the ECS protocol initiated by peer A, peer B starts to upload content pieces. In a while, peer B becomes interested in downloading content from peer A and initiates the ECS protocol. At some point of time, peer A's credential expires, and peer B terminates the ECS protocol initiated by peer A, as well as the established connection to it. Figures 4.1-4.4 illustrate sequence diagrams related to this scenario, where all objects on the lifelines are Python objects.

At the beginning, both of the peers start the Next-Share system. When the torrent file is provided, Session object calls add() method from the TriblerLaunchMany object (Fig. 4.1). At this point, ECS_Manager object registers the torrent and initializes an ECS_SwarmManager for it.

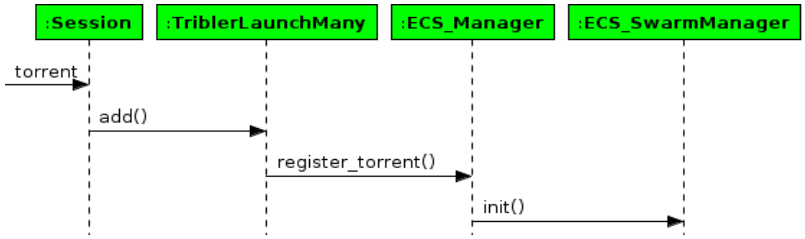


Figure 4.1: Diagram of the ECS management objects start up

Furthermore, peer A starts the BitTorrent protocol to peer B (Fig. 4.2). Upon receiving the extend protocol handshake (containing support for ECS protocol), the responsible peer B's ECS_SwarmManager registers the Connection (Connecter.py/Connection class) object, and couples it with an ECS_Connection, which will coordinate the protocol message exchange process. The Connection object uses the send_cs_message() method for sending the outgoing messages, and the got_cs_message() method for transferring the incoming messages to the

ECS_Connection object. ECS_Connection object processes the received messages and creates the appropriate responses utilizing the EnhancedClosedSwarm object methods. In this case, this is the 'upload' EnhancedClosedSwarm object – it is responsible for controlling whether peer B should upload content to peer A, according to its authorizations. Moreover, the ECS_Connection object sets the environment in the ECS_AuthorizationEngine object for proper rules evaluation, and schedules next rules evaluations and credential validations. If peer A's request is according to its authorizations, peer B starts to upload content.

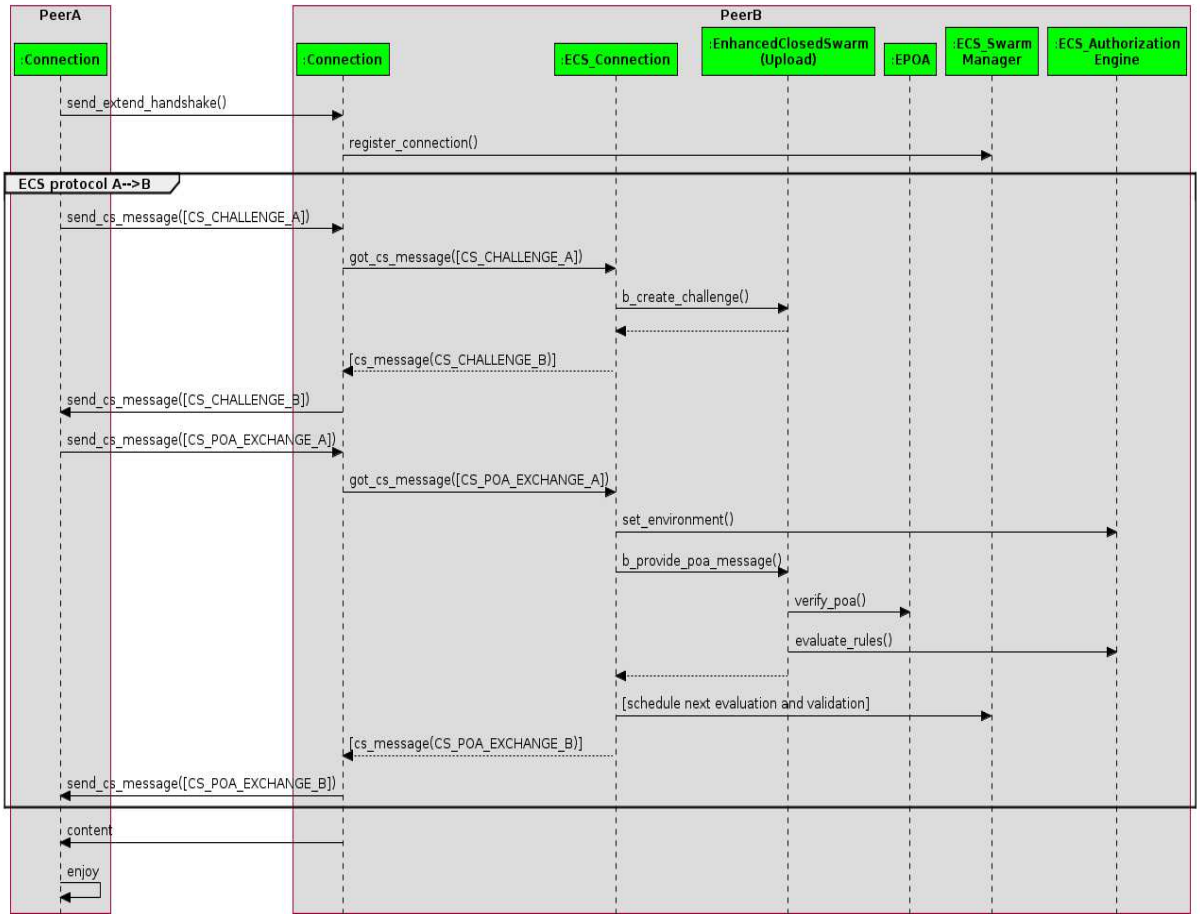


Figure 4.2: Diagram of the ECS message exchange process initiated by the protocol initiator

Next, peers A a only exception is

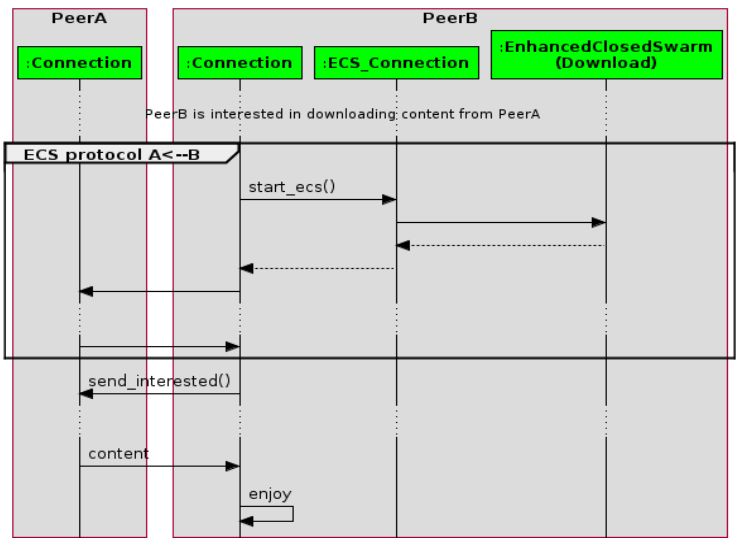


Figure 4.3: Diagram of the ECS message exchange process initiated by the closed swarm member

Finally, at some point of time, peer A's credential expires, which is detected during scheduling next rules evaluations and credential validations. Peer B creates notification (Info) message and sends it to peer A (Fig. 4.4). Then, it immediately closes the connection.

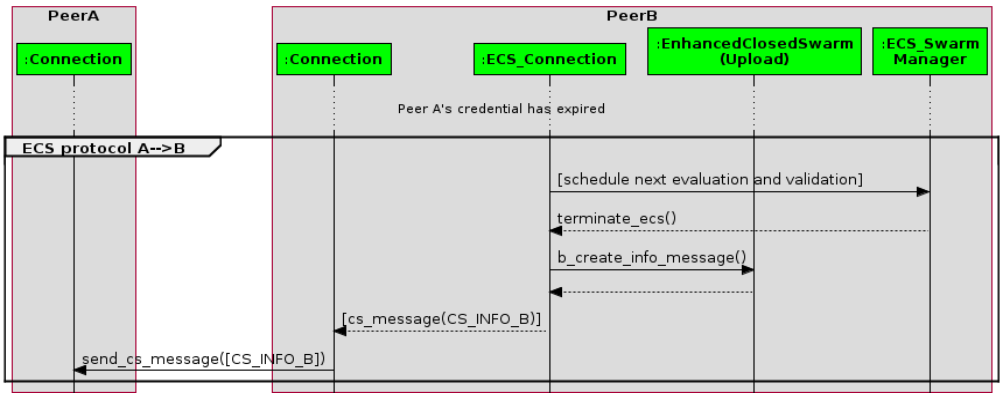


Figure 4.4: Diagram of the ECS protocol termination

4.4 Using ECS for Creation and Maintenance of a Hierarchically Structured Swarm

A hierarchically structured swarm promises means for load balancing and optimization of the delivery process, especially in case of live streaming content. The hierarchical structure is formed by separation of the seeders into layers (levels) according to the priority assigned to them by the content provider (Fig. 4.5). In BitTorrent live streaming swarm, seeders are special peers with outstanding properties (e.g., high bandwidth), which are always unchoked by the content injector, and are often purposely set by the content provider to improve the other peer's (leechers) download performance. The greater the priority of the seeders a layer contains is - the higher it appears in the structure. The leechers are placed in the most outer layer and do not have any priority. The value of the priority defines the level of precedence a seeder has among the other peers in the live streaming swarm (seeders and leechers). Normally, the content injector and the seeders establish a connection to any peer in the swarm regardless of its priority, as long as they have a free connection. However, when a lack of free connection occurs, the connections with seeders having lower priorities or with leechers will be terminated in favour of seeders having greater priorities. This promises quick transport of the content from the content injector towards the lowest level of the structure of seeders, and consequently to the leechers.

Two mechanisms are needed for the process of creation and maintenance of the hierarchical structure:

- Automatic introduction of seeders: Seeders explicitly know each other by maintaining lists of their identifiers (e.g., IP address and port number). However, these lists are maintained manually - something that becomes impractical for a large or hierarchically structured swarm.
- Suitable peer discovery: This mechanism is needed to enable seeders to fit in the appropriate layer according to their priority

ECS protocol provides these mechanisms by:

- supporting expressive and flexible access control policies (in the Rules field)
- enabling peers to exchange swarm members between themselves (in the Peers field)

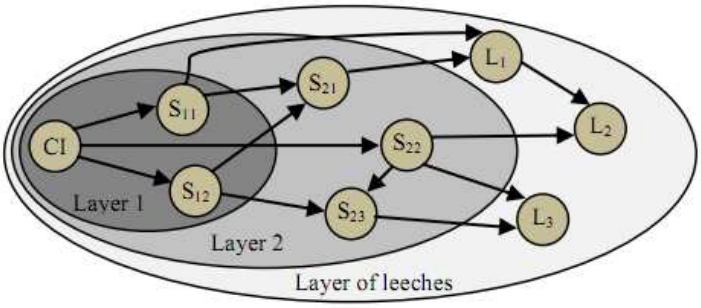


Figure 4.5: Hierarchical structure of a live streaming swarm: the content injector (CI) is not part of any layer; the seeds (S) from layer 1 have a priority (e.g., 20) greater than the seeds from layer 2 (e.g., 10); the leechers (L) are all placed in one layer and do not have any priority

The process of creation of the hierarchically structured swarm goes as follows. During the design the content provider decides about the number of the needed seeders and the number of the layers of the structure. Then it assigns the seeders to specific layer, according to their capabilities (e.g., bandwidth) and issues them appropriate credentials, specifying their priority in the policy. All peers first contact the tracker to discover other members of the closed swarm. Then they initiate the peer wire protocol to the returned peers. The ECS protocol starts after the Extension protocol handshake. With the ECS protocol peers exchange list of other swarm members, sorted by their priority. One half of the peers in the list is selected from their download connections, while the other half from their upload connections. In this way, peers are able to move up and down in the hierarchical structure, until they fit in the appropriate level according to their priority. The content provider can maintain the structure by introducing new seeders in cases when a seeder goes down or when a flash crowd occurs, only by issuing appropriately credentials to them. In addition, logs from a simulation of the described process are given in Appendix D: Logs from a Simulation of Creation of a Hierarchically Structured Swarm.

Chapter 5
Monitoring of P2P Service Provisioning

5.1 Introduction

For professional content providers and users as well it is crucial that content ingest service could be properly monitored. Systems users would like to follow distribution effectiveness and spot any issues in service provisioning in real time. P2P-Next project road map wise the plans for the ingest service Monitoring were described in the deliverable D.3.2.1.b.

Monitoring service implementation primary goal is to provide transparent and easy access to the monitoring parameters of the Next-Share PC implementation. For this reason Monitor provides a way to sample the monitoring parameters and to export the parameters via web based interface for other services usage.

Monitoring implementation can be used for following tasks:

1. Programmable, periodical sampling of various numerical and non numerical information pertinent to content provisioning and consumption process obtained via instrumenting Next-Share implementation. Implementation keeps a backlog of a number of sampled information and provides basic statistics for numerical information.
2. Export information about content provisioning process from Living Lab live streams ingest points. Collection of information is done via scripts and cron, MRTG[2] (Multi Router Traffic Grapher) is used for live displaying of the data. The same process could be used for example to feed other monitors, like Nagios.[3]
3. Collect monitoring information from client Next-Share PC instances, locally. The approach works well and can be intermixed with SNMP based collection of the other client os/network/application data.
4. Export information on content consumption process on Next-Share PC client. Monitored information is presented in a graphical way through web interface on the client.
5. Extended access to Next-Share implementation and export of its internals through client internal web service interface for utility and debugging purposes, like access to debug logging, python garbage collection, up-time, etc. Information exported through internal web interfaces can be graphically presented on end user client locally or accessed from a command line at ingest point.

[2] See http://oss.oetiker.ch/mrtg/ for details.

[3] See http://www.nagios.org/ for details.

5.2 Design and Implementation Overview

Monitoring is designed on decoupling of (1) sampling of monitor information, (2) exporting the information for potential consumers and (3) its presentation.

Sampling support a number of different information to sample, from numerical, non numerical and compound information, like lists and dictionaries. Sampling is possible on compound samples as well. Sampling is designed to be minimally interweaved with the instrumented environment; it requires that the sampling variable is defined before its usage and after that the variable sampling is done via single line of code. Triggering of the sampling requires external mechanisms, provided by the instrumented environment. For sampled variables an adjustable backlog of sampled samples is kept that provides flexibility in range of possible supported monitored information consumers.

Sampled monitoring information is exported through simple web based interface. The reason for this is that the web interface can be used by a number of information consumers, for example from command line, management scripts like cron jobs and from within a web browser. The interface supports a number of output formats suitable for the consumers, for example ascii, for command line usage, or json for web browser environment consumption. In this way the presentation of the monitored information could be left to the consumer. But minimal presentation capabilities needs to be build in, for example, easier command line presentation and web based APIs need to be flexible enough to support specific consumer requirements. For example, to build responsive web based presentation of monitored data, the exporting web interface supports ranges based on information on last accessed time to the API so the minimal range needed for the consumer to refresh the information presentation is passed through the interface. In this manner refresh time of a sampled variable graph in web browser can be reduced significantly.

Monitoring implementation sampling is coded in Python, in object oriented manner. Programming wise the main entry point is Monitor manager. The manager uses a singleton patter therefore a consistent view on sampled monitored information is available in single python process instance. For web interface export urlrelay package[4] is used, a simplest but flexible relay for binding implementation methods to web API interfaces. No additional python packages besides core python distribution are needed. Implementation provides two different ways to bind the Monitor with Next-Share implementation: for live streaming monitoring the createlivestream.py script has been modified and for Next-Share PC client its BackgroundProcess.py implementation has been instrumented with Monitor sampling. Both ways of binding reuse Next-Share callback mechanism capabilities to trigger the sampling at user specified intervals.

A number of consumers has been used and implemented. For simple on line monitoring of the content ingest curl tool has been used from the command line. For continuous on-line monitoring scripts for MRTG were developed. Scripts are run via usual cron mechanism in UNIX environment and MRTG takes care of the monitored information presentation via web browser. A simple collector has been developed that allows continuous collection of

[4] See http://pypi.python.org/pypi/urlrelay/ for details.

monitored information via command line from Next-Share PC clients as well from SNMP (Simple Network Management Protocol) enabled Windows environment. Finally, for web based presentation on the Next-Share PC client a set of javascript scripts, combined with javascript plotting library for jQuery flot,[5] has been developed, forming an easy to embed browser widgets. The widgets are served through the same web interface as is used for exporting monitored information.Web interface has been extended with simple utilities for serving static and dynamic pages and basic templates, enabling the widgets to be self contained, e.g. without external dependencies.

5.3 Using the Monitor

Monitor implementation provides modified createlivestream.py script that initialize the monitoring variables, sample them at specified interval (default 10s) and exports the variables and the Monitor functionality via web interface on specified port (default 9212, for live stream 1000 about listen port so multiple monitors could run on the same server simultaneously).

Additional createlivestream.py script parameters for turning on the monitor and specifying monitor listening port are:

```
--monitor <arg>
          Enable monitoring of live stream (defaults to False)

--monitorport <arg>
          Monitoring port, if undefined will be set to 1000 above the listen
          port (defaults to '')
```

For Next-Share PC client a simple patch is provided in Next-Share/JSI/Monitor/tools directory that enables monitoring of the content consumption process on the client. The patch needs to be applied to latest version of Next-Share and the PC client rebuild.

5.4 Usage Examples

Monitor capabilities can be quickly grasped by running Monitor example:

```
topaz:~/src/M40/Next-Share:{734}> export PYTHONPATH=$(pwd):.
topaz:~/src/M40/Next-Share:{735}> python JSI/Monitor/Monitor.py
Reporter listening on host 127.0.0.1, port 9212
Check http://127.0.0.1:9212/help for help.
Std error write test, if this can be seen the feature is not enabled
localhost - - [24/Jul/2011 14:17:47] "GET /index HTTP/1.1" 200 923
```

The default host is 127.0.0.1 (localhost) and port 9212. The last line already presents a logged request for index page, which returns the layout of the Monitor service implementation:

```
topaz:~/:{658}> curl -s http://127.0.0.1:9212/index

MonitorService index:
/
  about
  debug
   |-- hpy (?maxlines=lines[&index=i[&q=[bysize|byclodo|byid|byrcs|byvia|setref|
maxlines|theone]]|&r=[list|clear|ref])
   |-- uncollectable (?set=flags|?type=short|?level=level)
  examples
  help
  images
  index
  logging
   |-- log (?max=lines)
   |-- loglevel (?q=logger[&set=loglevel])
```

[5] See http://code.google.com/p/flot/ for details.

```
 |-- stderr (?q=[maxlines|count]|?setmaxl=n|?avoid=[str|*])
static
test-monitor
 |-- download (?q=[list|*]|?jsoncallback=?[&attr=x])
 |-- graphs (?q=[ptype|reload][&w=w&h=h&n=num&x=num&y=num&p=pos]|[&t=name])
 |-- id
 |-- peer_list
 |-- range (?num=n[&(?)var=variable][&q=flot|flotavg])
 |-- report
 |-- speed (?q=[list|*]|?jsoncallback=?[&attr=x])
 |-- swarm (?id=x&attr=y)
 |-- upload (?q=[list|*]|?jsoncallback=?[&attr=x])
 |-- variables
uptime (?q=condensed)
utils
 |-- attributes (?attr=attribute[&q=type])
 |-- settable (?attr=attribute[&set=value])
```

Besides some expected links there are some more interesting:

- help: provides terminal oriented help for the links
- test-monitor:

  - download/speed/upload: numerical monitor variables

  - id/peer_list: info about the peers

  - swarm: access to aggregated peer list information

  - report: terminal report in unix uptime style

  - range: gives a specified range of values for numerical variables

  - graphs: graphing the numerical variables, for a browser consumption, by default returns a standalone page of numerical variables graph

  - /test-monitor/graphs?q=embed gives a jquery-ui based widget for embedding into a page

  - /test-monitor/graphs?q=reload, handy for development, reloads templates dynamically even in the plugin, useful for presentation development
- uptime: uptime of the monitor tool
- static, images: static web pages served from disk, use for example *static* to list the content, access to jQuery javascripts and images
- examples: dynamic web pages based on templates, served from dict, embedding example is in embed-mon-graphs.html
- debug/hpy: provides web interface to heapy[6] debugging tool
- debug/uncollectable: uses Python garbage collector for discovering uncollectable objects
- log/logging, log/stderr: gives access to logging information, could be used to turn logging to stderr in Python logging style, could redirect stderr to a buffer
- utils: provide read/write access to certain monitoring implementation attributes

As can be noticed form the output the index page provides quick help for the query part of the link, which can be handy for the terminal usage.

[6] See http://pypi.python.org/pypi/guppy/ for details.

5.4.1 Usage via command line

Command line curl tool can be used to access exported monitored information, for example, the following command will return report related information of ingest of RTV Slovenia TV channel SLO1:

```
stream:~:{939}> curl -s http://127.0.0.1:10951/slo1_mpegts/report
slo1_mpegts report,  uptime: 13h:59m:59s
Variable    Samples        Value      Average     Average2     Averege3
   peers       4998         2.00         1.10         1.05         1.03
  upload       4998       770.07       438.00       431.84       456.71
download       4998         0.00         0.00         0.00         0.00
   speed       4998   1016295.95    481440.23    456451.10    475756.54
```

The report outputs numerical variables monitored by Monitor and their averages (5, 10 and 15 minutes). Since the ingest point is a live stream seeder its download speed is zero. Speed parameter is estimation of cross swarm speed based on collected peer lists.

The following example shows how to access ids of the peers in the swarm or any other attributes available in the peer list, for example IP numbers of the peers.

```
stream:~:{939}> curl -s http://127.0.0.1:10951/slo1_mpegts/swarm
N407-----mxb3NOa9pvG
N242-----G0EHv0fwEub
stream:~:{940}> curl -s http://127.0.0.1:10951/slo1_mpegts/swarm\?attr=ip
193.138.1.106
193.138.1.244
```

5.4.2 Usage in MRTG and cron scripts

The same tool as is described in the previous section can be used with MRTG for collecting information from multiple ingest processes. The following example is MRTG script, that collects overall information about peers accessing JSI/RTV Living Lab live streams (6 channels):

```
#!/bin/zsh

MAX_CH=6
BASE=1095
count=0
average=0

for i in {1..$MAX_CH}
do
  monitor=$(curl -s http://127.0.0.1:$BASE$i/utils/attributes\?attr=monitor)
  count=$(($count + $(curl -s http://127.0.0.1:$BASE$i/$monitor/peers)))
  average=$(($average + $(curl -s http://127.0.0.1:$BASE$i/$monitor/peers\?average)))
done
echo $count
echo $average
echo 0
echo $1 connection count
```

The script provides actual overall number of peers connected to all channels and five minute average value in MRTG compliant format. MRTG scripts are run via host cron service every five minutes (hard MRTG default). The average value helps to catch an indication about clients that are connected less then five minutes to either of the streams.

5.4.3 Next-Share client PC usage

NextShare PC can use Monitor implementation to graphically represent the monitored information in the web browser of the client. On the following figure parts of two Firefox windows are presented, the first showing the monitor graphs for each sampled variable and the second, smaller one, the widget presenting the variables in condensed form. In this

example only download, upload and the number of peers in swarm are presented. In the widget flot feature which enables popups with values is used to avoid cluttering the interface with exact values. The following is a link on the client host to the widget:

```
http://127.0.0.1:9212/examples/embed-mon-graphs.html
```

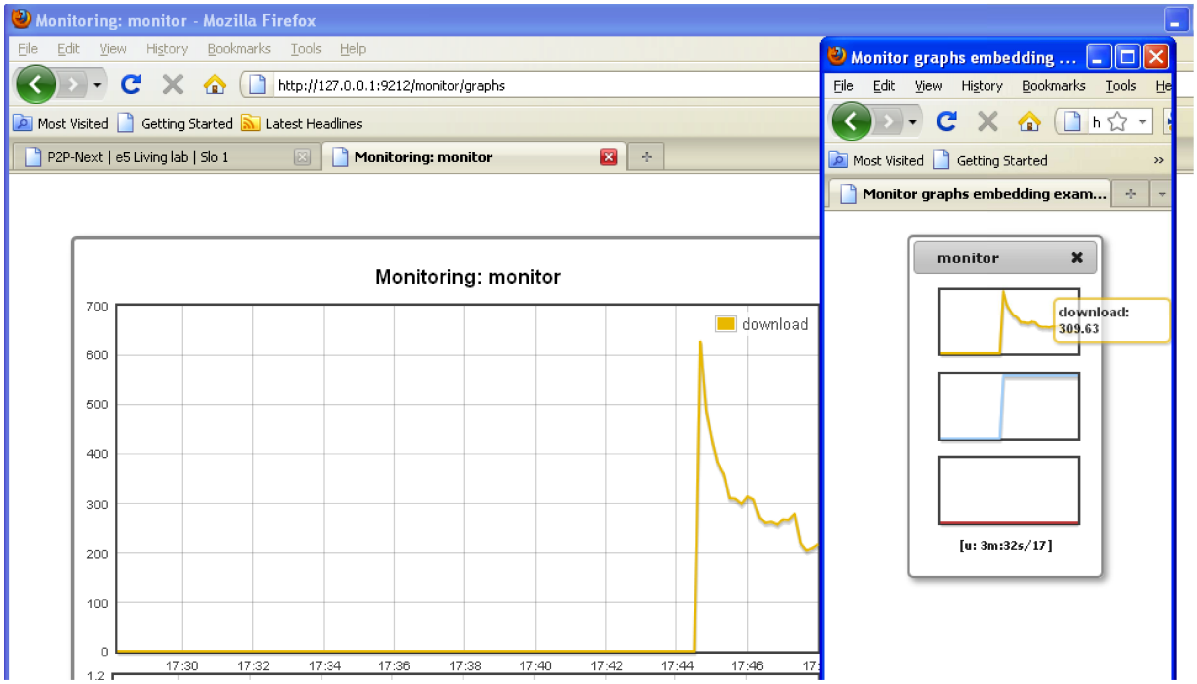


Figure 5.1: Monitoring graphs and widget

Before the widget can be embedded into other pages one needs to address the issues of 'same origin policy'[7] enforced by most browsers. There exists a number of methods to solve this problem but the Monitor implementation currently doesn't use or suggest any particular solution.

5.4.4 Other usage

The Monitoring implementation has been used for other purposes as well. Through same web interface it was easy to export Python garbage collector information about non-collectable objects or redirecting to it debugging information ordinary delivered to standard error. Such features were used for automated collection of information across multiple services for longer period of time. In this way it was easier to analyse and diagnose issues that has emerged during the NextShare implementation development.

5.5 Security Considerations

Though some care has been taken not to expose too much information via Monitor implementation the intended scope of running the software is localhost. If really needed, external exposure should be done via protected proxying of the internal web server (via apache, nginx, etc.)

[7] See http://en.wikipedia.org/wiki/Same_origin_policy for details.

Chapter 6
Revised APIs

The Next-Share platform currently has one defined API, referred to as the NextShareCore or BaseLibCore API, shown in Figure 2.1. The full API specification for Next-Share M8 was published in D4.0.1. As the changes to the API made for M48 are small we only specify the changes here since M40. The BaseLibCore API for M48 carries version number 1.2.0.

- Added support for swift downloads: The TorrentDef class was refactored into a generic ContentDefinition class and two subclasses TorrentDef and SwiftDef. The SessionConfig class was extended with configuration parameters for the swift processes:
    - s/get_swift_proc: to enable/disable support for swift Downloads via an external swift C++ process.
    - s/get_swift_path: to set/get file-systen path to swift binary
    - s/get_swift_downloads_per_process: Set/get number of downloads per swift process. When exceeded, a new swift process is created.
- To optimize sharing on the NextShareTV set-top box we made a number of Core parameters configurable which previously were not:
  SessionConfig.s/get_socket_write_always()
  SessionConfig.s/get_socket_rcvbuf_size()
  SessionConfig.s/get_socket_sndbuf_size()
  DownloadConfig.s/get_live_drop_pieces()
  DownloadConfig.s/get_upload_while_prebuf()
  DownloadConfig.s/get_ratelim_min_resched_time()
- Added a peer-assisted mode of operation where the NS tracker sends the address of a well-known seeder in every reply. It automatically distributes the load over the available seeders. To this extent, we added the following methods to the API:
    - TorrentDef.[s/g]et_predef_seeders()
    - SessionConfig.[s/g]et_tracker_send_predefseeds()

Bibliography

[1] Cohen, B. Incentives to Build Robustness in BitTorrent. In Proceedings 1st Workshop on Economics of Peer-to-Peer Systems (Berkeley, CA, June 2003). http://BitTorrent.com/bittorrentecon.pdf

[2] Cohen, B. The BitTorrent Protocol Specification. BEP 3, BitTorrent Community Forum, www.BitTorrent.org, Jan. 2008.

[3] Gertjan Halkes and Johan Pouwelse, UDP NAT and Firewall Puncturing in the Wild, IFIP Networking 2011, Valencia, Spain, 9-13 May, 2011.

[4] Burford, M. WebSeed – HTTP/FTP Seeding (GetRight style). BEP 19, BitTorrent Community Forum, www.bittorrent.org, Feb. 2008.

[5] Mol, J.J.D., Bakker, A. Pouwelse, J., Epema, D. and Sips, H.. The Design and Deployment of a BitTorrent Live Video Streaming Solution. In Proceedings International Symposium on Multimedia (ISM 2009), San Diego, CA, USA, Dec 14-16, 2009.

[6] P. H. J. Perälä, J. P. Paananen, M. Mukhopadhyay, and J.-P. Laulajainen, “A Novel Testbed for P2P Networks,” in Testbeds and Research Infrastructures. Development of Networks and Communities: 6th International ICST Conference, TridentCom 2010. Berlin, Germany, May 2010, pp. 69–83.

[7] M. Devera, “Htb home”, http://luxik.cdi.cz/devik/qos/htb/

[8] S. Hemminger, “Network Emulation with Netem”, in Proceedings of the 6th Australia’s National Linux Conference (LCA2005), Canberra, Australia., April 2005.

[9] Victor Grishchenko, Johan Pouwelse, “Binmaps: Hybridizing Bitmaps and Binary Trees”, Technical Report PDS-2011-005, Dept. of Software and Computer Technology, Faculty Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, Delft, The Netherlands.

[10] Borch N.T., Arntzen, I., Mitchell K., Gabrijelčič D.: Access control to BitTorrent swarms using Closed Swarms. In: Proceedings of the 2010 ACM Workshop on Advanced Video Streaming Techniques for Peer-to-Peer Networks and Social Networking, AVSTP2P '10, pp. 25-30. ACM, New York (2010)

[11] P2P-Next Deliverable number 4.0.3 : Next-Share Platform M24—Specification Part

Chapter 7
Appendix A: Improving Accuracy and Coverage in an Internet-Deployed Reputation Mechanism

R. Delaviz, N. Andrade, and J. Pouwelse. In *Proceedings 10th IEEE International Conference on Peer-to-Peer Computing (IEEE P2P'10)*, Delft, the Netherlands, August 25-27, 2010.

Improving Accuracy and Coverage in an Internet-Deployed Reputation Mechanism

Rahim Delaviz, Nazareno Andrade, and Johan A. Pouwelse

Parallel and Distributed Systems Group
Department of Computer Science, Delft University of Technology, The Netherlands
Email: r.delavizaghbolagh@tudelft.nl

***Abstract*—P2P systems can benefit from reputation mechanisms to promote cooperation and help peers to identify good service providers. However, in spite of a large number of proposed reputation mechanisms, few have been investigated in real situations. BarterCast is a distributed reputation mechanism used by our Internet-deployed Bittorent-based file-sharing client Tribler. In BarterCast, each peer uses messages received from other peers to build a weighted, directed subjective graph that represents the upload and download activity in the system. A peer calculates the reputations of other peers by applying the maxflow algorithm to its subjective graph. For efficiency reasons, only paths of at most two hops are considered in this calculation.**

In this paper, we identify and assess three potential modifications to BarterCast for improving its accuracy and coverage (fraction of peers for which a reputation value can be computed). First, a peer executes maxflow from the perspective of the node with the highest betweenness centrality in its subjective graph instead of itself. Second, we assume a gossiping protocol that gives each peer complete information about upload and download activities in the system, and third, we lift the path length restriction in the maxflow algorithm. To assess these modifications, we crawl the Tribler network and collect the upload and download actions of the peers for three months. We apply BarterCast with and without the modifications on the collected data and measure accuracy and coverage.

I. Introduction

P2P systems can benefit from *reputation mechanisms* through which peers evaluate the reputations of the participants of the system and are therefore able to identify good service providers. Two central properties of a reputation mechanism are its *accuracy*, that is, how well a peer can approximate "objective" reputation values when calculating the reputation of other peers, and its *coverage*, that is, the fraction of peers for which an interested peer is able to compute reputation values. Inaccurate or partial reputation evaluation may lead to misjudgment, poor behavior, and finally, system degradation. The BarterCast mechanism [1] is an Internet-deployed reputation mechanism that is used by the Tribler Bittorent-based file-sharing client [2] to select good bartering partners and to prevent free-riding. In this paper we evaluate the accuracy and the coverage of BarterCast, propose three modifications to this mechanism, and evaluate these modifications with respect to accuracy and coverage. The evaluation is performed using empirical data collected by crawling the Tribler P2P network over a 3-month period.

P2P file-sharing systems are characterized by large populations and high turnover. In such setting, two participants interacting will often have no previous experience with each other, and will be thus unable to estimate each others' behavior in the system. If choosing among potential interaction partners is important, such configuration is an issue. The fundamental idea behind a reputation mechanism is that individual behavior does not usually change radically over time, and past activity is a good predictor of future actions [3]. Using this idea, a reputation mechanism collects information on the past behavior of the participants in a system and quantifies these information into reputation values. In a distributed reputation mechanism, depending on how the information about peers behavior are disseminated or how the reputation values are computed, each participant may have different reputation values for the same participants.

We have previously designed and implemented the BarterCast reputation mechanism in our Bittorent-based P2P client Tribler. In BarterCast, peers exchange messages about their upload and download actions, and use the collected information to calculate reputations. From the BarterCast messages it receives, each peer builds a local weighted, directed graph with nodes representing peers and with edges representing amounts of transferred data. This subjective graph is then used by each peer to calculate the reputation values of other peers by applying the maxflow algorithm to the graph, interpreting the edge weights as "flows."

In this paper we propose three modifications to the BarterCast reputation mechanism, and we evaluate the accuracy and the coverage of the original BarterCast reputation mechanism and of all combination of these three modifications. First, rather than have each peer execute the maxflow algorithm to compute reputations from its own perspective, we make each peer do so from the perspective of the node with the highest *betweenness centrality* [4] in its subjective graph. The second modification consists in using a gossiping protocol that fully disseminates the BarterCast records in the whole system rather than limiting the exchange of these records to one hop. In the third modification we increase the maximal path length in the maxflow algorithm to 4 or 6 instead of 2 as in the original BarterCast. In order to evaluate the original BarterCast reputation mechanism and our three modifications, we have crawled the Tribler P2P system for 83 days to obtain as many BarterCast records of the Tribler

peers as possible. From the records obtained from each peer, we emulate its reputation computations by reconstructing its *subjective view*, represented by the subjective graph of the peer (in this paper the terms subjective graph and subjective view are synonyms). We then use this graph to execute the maxflow algorithm with and without modifications.

In the rest of the paper, we first explain the BarterCast mechanism in detail and we define the metrics accuracy and coverage. In Section III, we explain the crawler and the data collecting process, and we describe the collected data. In Section IV, we state the problem of the current version of BarterCast and explain the modifications we propose. In Section V, first we explain the experimental setup then the experimental results are presented.

II. The BarterCast Reputation Mechanism

In this section, we first explain the BarterCast mechanism in detail and then we formulate the metrics accuracy and coverage in this mechanism.

A. The BarterCast Mechanism

The BarterCast mechanism is used by the Tribler Bittorent client to rank peers according to their upload and download behavior. In this mechanism, a peer whose upload is much higher than its download gets a high reputation, and other peers give a high priority to it when selecting a bartering partner. In BarterCast, when two peers exchange content, they both log the amount of transferred data and the identity of the exchange partner in a BarterCast record; these records store the total cumulative amounts of data transferred in both directions since the first data exchange between the peers. In BarterCast, each peer regularly contacts other peers in order to exchange BarterCast records. Peer sampling for selecting to whom to send BarterCast records is done through a gossip protocol called BuddyCast, which is at the basis of Tribler. In BuddyCast, peers regularly contact each other in order to exchange lists of known peers and content.

Using the BarterCast message exchange mechanism, each peer creates its own current local view of the upload and download activity in the system. Formally, the receiver of BarterCast records creates and gradually expands its *subjective graph*. The subjective graph of peer i is $G_i = (V_i, E, \omega)$, where V_i is the set of nodes representing the peers about whose activity i has heard through BarterCast records, and E is the set of weighted directed edges (u, v, w), with u and $v \in V_i$ and w the total amount of data transferred from u to v. Upon reception of a BarterCast record (u, v, w), peer i either adds (a) new node(s) and a new edge to its subjective graph if it did not know u and/or v, or only (a) new directed edge(s) if it did know u and v but did not know about the data transfer activity between them, or adapts the weight(s) of the existing edge(s) between u and v. If peer i receives two BarterCast records with the same sender u and the same receiver v from different peers, it keeps the record that indicates lower amounts of data transferred in order to avoid invalid reports from malicious peers that try to inflate their uploads. Furthermore, the direct experience of the peer has higher priority than received reports from others.

In order to calculate the reputation of an arbitrary peer $j \in V_i$ at some time, peer i applies the maxflow algorithm [5] to its current subjective graph to find the maximal flow from itself to j and vice versa. Maxflow is a classic algorithm in graph theory for finding the maximal flow from a source node to a destination node in a weighted graph. When applying maxflow to the subjective graph, we interpret the weights of the edges, which represent amounts of data transferred, as flows. The original maxflow algorithm by Ford-Fulkerson [5] tries all possible paths from the source to the destination, but in BarterCast, in order to limit the computation overhead, only paths of length at most 2 are considered. The rationale for expecting that this limit is sufficient is that the majority of peers may have indirect relationships through popular intermediaries [6], in which case using two hops in maxflow provides sufficient data for the evaluation of reputations. Using the values $F_2(.,.)$ as computed with the 2-hops maxflow algorithm, the subjective reputation of peer j from peer i's point of view is calculated as:

$$S_{ij} = \frac{\arctan(F_2(j,i) - F_2(i,j))}{\pi/2}, \qquad (1)$$

and so $S_{ij} \in [-1, +1]$. If the destination node j is more than two hops away from i, then its reputation is set to 0.

In Figure 1 a simple subjective graph is shown in which peer i as the owner of the graph evaluates the reputation of peer j. In this graph, $F_2(i,j) = 11$ and $F_2(j,i) = 5$, and so $R_i(j) = -0.89$.

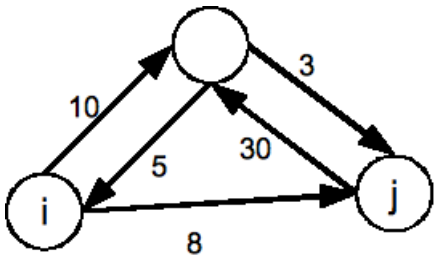


Fig. 1. A sample subjective graph.

Using a flow algorithm (e.g., maxflow in BarterCast) is like doing a collaborative inference where the knowledge of all involved nodes is included in the computation of the final value. Beside this, flow algorithms like maxflow are more resilient against sybil attacks that trivial operations like averaging or summation are not [7]. The BarterCast mechanism can be generalized in the form of *flow-based* mechanisms. Such mechanisms have two common features. First, the relation between participants is shown as a graph. Second, there is a function ϕ which calculates the flow of a specified metric from a node set I to a destination node set J, and the obtained flow is used to calculate the final reputation value.

B. Accuracy and Coverage

As the term *accuracy* indicates, it is a measure of how close an estimated reputation value is to an "objective"

or real value. In a distributed mechanism like BarterCast, depending on how the feedback records are disseminated, peers may have different opinions about the reputation of a peer at the same time. Each peer also at each point in time has an *objective reputation* value, O_j, that is calculable only if the evaluator peer has a global view of the activity of all peers. In our case, only the crawler has such a view and using the collected data we can calculate the objective reputations. If U_j and D_j are the total upload and download by peer j, then its objective reputation is

$$O_j = \frac{\arctan(U_j - D_j)}{\pi/2} \quad (2)$$

Using the objective and subjective reputations, the estimation error is defined as the absolute value of the difference between the subjective and objective values:

$$e(i,j) = abs(S_{ij} - O_j) \quad (3)$$

Higher estimation errors mean lower accuracy and vice versa.

Coverage is another important metric that expresses how well a node is located and can reach other nodes in the graph. Denoting by $F_h(.,.)$ the maximum flow computed with the maxflow algorithm using all paths of length less than or equal to h, in the subjective graph G the h-hop coverage of node i is defined as

$$c_G(i,h) = |\{u | F_h(i,u) > 0 \ or \ F_h(u,i) > 0\}| \quad (4)$$

So the coverage of node i in a graph is the number of nodes at a distance at most h from node i with non-zero maximum flow to or from i. Dividing the coverage by the number of nodes normalizes it into the interval of $[0,1]$ and makes it possible to compare this metric in graphs of different size.

C. Related Work

The BarterCast mechanism was designed by Meulpolder et al. to distinguish free-riders and cooperative peers in file-sharing environments. After the first release, Seuken et al. [8] proposed an improvement to make it more resilient against misreporting attacks. Their solution is based on ignoring some of the feedback reports. Also, this solution could cut down the severity of the attack, but on the other hand it increases the feedback sparsity. Xiong et al. [9] show that the feedback sparsity is an issue in large distributed systems, and that a lack of enough feedback can lead to lower accuracy and coverage.

Besides BarterCast, several other distributed reputation mechanisms have been proposed for P2P systems, but they use different methods to calculate reputation values. EigenTrust [10] is based on summation of direct observations and indirect data and uses centralized *matrix operations* to compute the left eigen vector. The CORE system [11] uses *arithmetic weighted averaging* on historical data to calculate reputation values. The BarterCast mechanism best fits in a class of mechanisms which use flow-based reputation function as defined by Cheng et al. [7].

III. The Crawler and the Collected Data

To collect the required dataset consisting of the BarterCast records of all (or at least, many) Tribler peers for analysis, we have crawled the Tribler network for 83 days, from June 20 until September 9, 2009. Except for some slight differences, the crawler works as an ordinary Tribler client. Discovery of the new peers is done through the BuddyCast protocol, which is the gossiping engine of the Tribler client. When a new peer is discovered with this protocol, it is added to a list. The crawler hourly contacts all peers in this list and asks them for their latest BarterCast records by including the timestamp of the latest record it does have of each peer. Using the BarterCast records received by the crawler from each peer, we can reconstruct the subjective graph of that peer in the same way the peer builds it.

The discovered peers have different ages, some of them having been installed and running for months and others just for a few days or even hours. So, when the crawler asks a peer for BarterCast records for the first time, it might receive very old records that are useless because they correspond to peers that were online in the past but no longer participate in the system. To mitigate this problem, when the crawler contacts a peer for the first time, it uses the start time of the crawl, that is, 00:00 hours on June 20, 2009, so that the discovered peers will only include BarterCast records fresher than the crawl start time in their replies.

Another problem in doing the crawling is the size of the reply messages. If a peer is asked for all its records at once, the reply message might be large and sending it may be problematic. To prevent this intrusive effect in the crawling, in each contact peers are only asked for 50 records that they have not sent already. Because of a potentially high churn rate, this limitation causes a side effect and for some of the peers that go offline the crawler is unable to fetch all their records. To have a reliable analysis, such incomplete views should be removed. Because in each contact a peer is limited to send at most 50 records, so with a high probability, having a multiple of 50 records from a peer means that it has not sent all its records. As a consequence, to filter out incomplete views, all views of the size of a multiple of 50 are removed.

To be able to sort the collected records and to account for the time difference with remote peers, the crawler asks peers to send for their local time as well. When the crawler receives such information, it logs the remote peer's time and its own local time. Using these two local times and the timestamp of the record (available in the record payload) the collected records can be sorted. If t_p and t_c denote the local time of the remote peer and the crawler, respectively, and t_r is the record timestamp, then the relative record occurrence time is:

$$t_c - t_p + t_r \quad (5)$$

This relative time is used in the experiments to sort the BarterCast records.

During the crawling period, the crawler collected 547,761 BarterCast records from 2,675 different peers. After filtering

out the incomplete views, 416,061 records were left, collected from 1,442 peers, which means that although 46% of the views are incomplete, they contain only 24% of the collected records. All the subsequent processing and analysis in this paper is based only on complete views.

IV. Problem Statement and Proposed Modifications

An analysis of the collected data set shows that the accuracy and the coverage with the current BarterCast mechanism are low and need to be improved. The mean of the estimation error is 0.664, which is the same as the average difference between two random values in the interval of possible reputation values, $[-1, +1]$. This means that a random guess for the subjective reputation value has the same precision as using the BarterCast mechanism. Similarly, the coverage of the BarterCast mechanism is very low at 0.032. In order to remedy this situation, we propose the following three modifications to the BarterCast mechanism.

A. Modification 1: Using Betweenness Centrality

Betweenness centrality has been introduced by Freeman [4] as a measure of the number of shortest paths passing through a node. In a graph $G = (V, E)$, if δ_{st} is the number of shortest paths between two arbitrary nodes s, t of G, and $\delta_{st}(v)$ is the number of these paths that pass through node v, then the betweenness centrality of node v is $\beta(v) = \sum_{s \neq v \neq t} \frac{\delta_{st}(v)}{\delta_{st}}$. A higher betweenness centrality means a higher participation of the node in connecting other nodes, and also a higher flow that passes through it. Another feature of this measure is that in contrast to connectivity (the sum of in and out degrees of a node), which is a local quantity, betweenness centrality is a quantity across the whole graph; nodes with many connections may have a low betweenness centrality and vice versa [12]. Betweenness centrality has been used in the analysis of various topics, like transportation, social networks, and biological networks, but to the best of our knowledge it has not been used in reputation systems.

In the original BarterCast mechanism, a peer i as the owner of the subjective graph G_i, in evaluating the reputation of peer j, runs the maxflow algorithm to compute the maximum flow from itself to j and from j to itself. In the proposed modification, first node i finds the node with the highest betweenness centrality in G_i, and then replaces itself with that node in the maxflow execution. By this change, the evaluator peer benefits from the centrality feature of the central node and uses the collected data in a better way.

B. Modification 2: Using Full Gossip

The second modification is obtained by changing the way BarterCast records are disseminated. In the original version, peers only use *1-hop* message passing and they are not allowed to forward the received records. Peers only report their own download and upload activities to the peers that are discovered by the BuddyCast protocol. This method limits the effect of misreporting but it is not efficient in spreading the BarterCast records. Specially if a peer goes offline, its upload and download activity are not disseminated, and when it comes online again, very few peers know about its activities. In this modification, instead of using 1-hop message passing, we assume that there is a *full gossiping* protocol that spreads records without the hop limitation, so that in principle all online peers eventually receive all propagated records.

C. Modification 3: Lifting the Maxflow Hop-Count Restriction

In the third modification we lift the restriction of 2 on the hop count in the maxflow algorithm and increase it to 4 or 6 hops. With this change, more nodes are involved in the maxflow algorithm and the chance of reaching a node, and so increasing the coverage, is increased.

V. Experimental Setup and Results

In this section we first explain our experimental set-up for assessing the accuracy and coverage of the original BarterCast mechanism and of the proposed modifications. In short, we emulate the creation of subjective graphs using the BarterCast records received by the crawler, and we emulate their computation of the reputation values of those peers to which they appear to have uploaded data. Then we present the experimental results and compare the effect of the proposed modifications on accuracy and coverage. At the end we do some statistical tests and determine whether the improvement level in accuracy is statistically significant or not.

A. Emulation of Full Gossiping

The subjective views collected by the crawler are only based on the standard 1-hop dissemination of BarterCast records. In order to evaluate the modification obtained with full-gossiping mode, we create artificial subjective views from the 1-hop subjective views. The full-gossip view at a certain point in time is same for all peers, and is built from *all* BarterCast records received from all peers with a timestamp lower than that time. So here we assume perfect full gossip in that all BarterCast records with a certain timestamp have been received by all peers at the time indicated by the timestamp. It should be noted that when using full gossiping, the reputation computations may still yield different results when maxflow is executed from the perspective of the local peer, but will give the same results when the local peer is replaced by the node with the highest betweenness centrality.

B. Experiment Design

In a large scale system like the one that the BarterCast mechanism is designed for, it is not required that every peer is able to evaluate the reputation of every other peer; peers just need to evaluate the reputations of the peers that they *encounter*. In the file-sharing system that we are studying, encountering means that a peer d contacts a peer s and asks s for some content, and peer s before responding to the request of d evaluates its reputation. When such an event happens, we say that s encounters d. In our experiment we try to

emulate the encountering events and only do a reputation evaluation when processing a BarterCast record in order to build up a subjective view that indicates such an event.

Another point we consider in the experiment is that in a decentralized reputation mechanism like BarterCast, we cannot expect that immediately after joining the system, a peer is able to give a good evaluation of the reputations of the peers it encounters. The newly joining peers should be allowed to collect information during a *training* phase from already existing peers and grow their subjective views before starting the evaluation of reputations of others during the *testing* phase.

The starting point of our experiment consists of the time-ordered sequences of BarterCast records the crawler has received from all peers, which we can use to build their subjective views. We define the *availability interval* of a peer as the interval between the timestamps of the first and last record in the sequence of BarterCast records the crawler has received from it. In our experiment, every peer goes through two phases, a *training* phase and a *testing* phase. In the training phase of a peer, we reconstruct its subjective view starting from the empty view by adding in sequence the BarterCast records of the first 80% of its availability interval. Only in the testing phase, peers evaluate the reputations of the peers they encounter. The testing phase is like the training phase, except that before adding an edge to its subjective view, a peer checks to see whether the conditions for encountering are satisfied. By checking these conditions we can detect the occurrence of an encountering event between two peers, and if required run the reputation evaluation process.

In the discussion below, we assume that the format of a BarterCast record is $[s, d, D, U, t]$, with t a relative timestamp and with D (U) the amount of data downloaded (uploaded) by peer s from (to) peer d until time t. When in the testing phase record $[s, d, D, U, t]$ of the subjective view G_i of peer i is processed, it is determined whether the reputation of peer d should be evaluated by peer i. This is only done if the following two conditions are satisfied:

I) $i = s$: The peer that uploads is also the owner of the subjective graph, and it is the peer that should do the reputation evaluation.

II) $U > 0$: The record indicates an actual data upload.

In other words, if a record passes the above conditions, the reputation of the peer that does the downloading is evaluated by the peer that does the uploading, and the latter coincides with the peer for which the BarterCast record is processed (s evaluates d, and i and s coincide). The meaning of the two conditions on the BarterCast records is that apparently, peer i has done an upload to d, and when the BarterCast reputation mechanism would have been in use, this would have been the time that peer i should have invoked it.

When processing BarterCast records in the testing phase, the peers whose reputations should be evaluated by other peers, are categorized as *newcomers* or *existing peers*. The newcomers are those peers that have not done any download or upload activity in the past (before the relative time of the record that is processed), but the existing peers have done so and the crawler knows about their activity. To detect newcomers, let $[s, d, D, U, t]$ be the record that is being processed, and assume it has passed the above encountering checks, so peer s should evaluate d. To determine whether peer d is a newcomer or not, we consider all current subjective views, and if in any of these there exists a record $[s', d', D', U', t']$ with $s' = d$ or $d' = d$, $t' < t$, and $U' > 0$ or $D' > 0$, then d has been active in the past and is not a newcomer; otherwise it is.

Reputation evaluation for newcomers is meaningless, as without any previous information about a peer, there is no reputation to be calculated. So, in the results of the accuracy and the coverage below, only the existing nodes are considered and the newcomers are excluded. In our experiment, in which the training and testing phases take 80% and 20% of the availability intervals of the peers, respectively, the numbers of newcomers and existing peers are 140 and 123, respectively.

The explained experiment is run for each view one-by-one and in all combinations of the proposed modifications. For each combination, we assess the values of the accuracy and the coverage, and when all views are processed, the results are aggregated to compare the performance of the different combinations.

C. Coverage

The barchart in Figure 2 shows the number of covered peers for all combinations of the proposed modifications. It is expected that only existing peers can be covered by the evaluator peers, and so in all of our experiments the maximum possible value for the coverage is 123 (the number of existing peers). The left half of the graph shows the cases in which the central node is used in the maxflow algorithm and the right half the view owner itself. As the graph shows, full gossiping boosts the coverage dramatically. Using the central node increases the coverage too, specially in 2-hops maxflow, but for a larger number of hops, it is less effective. Increasing the number of hops has more or less the same influence as using the central node, and in both dissemination methods the biggest improvement is seen when we go from 2 to 4 hops.

D. Accuracy

In Figure 3 we show the fractions of nodes for which either the central node in the subjective graph or the local peer provides a better estimation of the reputation value for different numbers of hops in maxflow and in both 1-hop and full-gossip dissemination. In practice, equal reputation estimation means that both reputation values are equal to 0. As the left hand of the figure (1-hop dissemination) shows, in more than 80% of the cases the central node and the view owner give the same estimation. When we move to full gossiping, the situation changes considerably, and using the central node gives better estimations. Especially with 4 and 6 hops, the number of cases that the central node is better is twice the number of cases that the view owner is better.

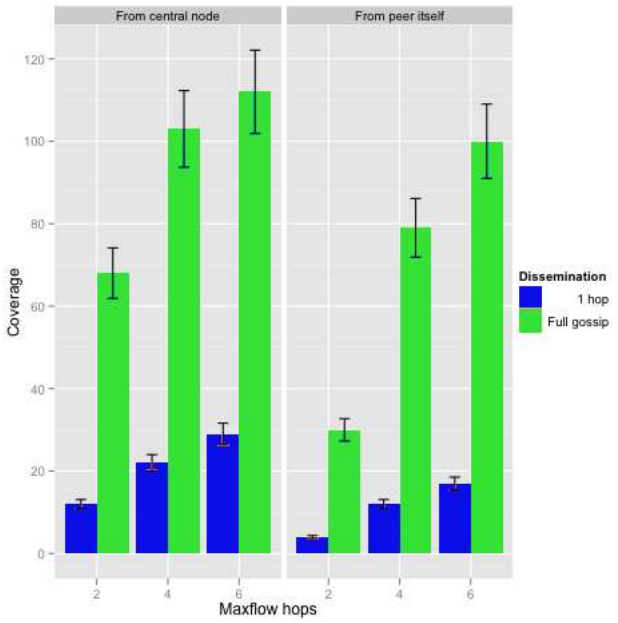


Fig. 2. The coverage of the BarterCast mechanism in different scenarios. (Error bars show the standard error of mean.)

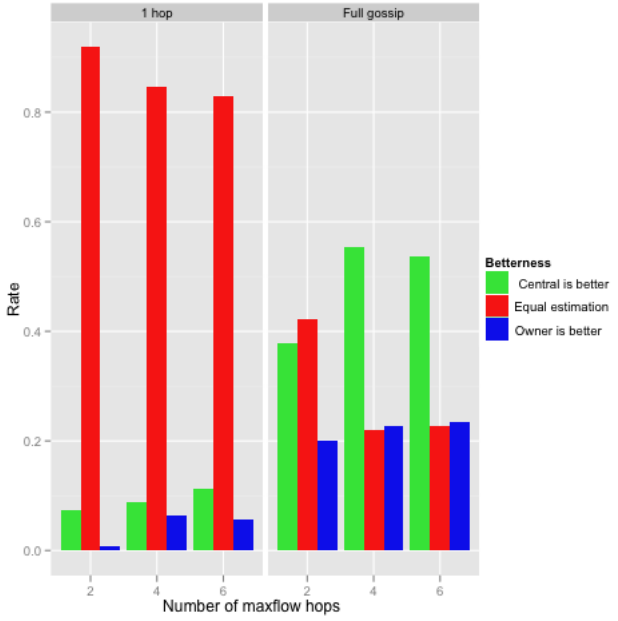


Fig. 3. Comparing the accuracy of the central node against the view owner in the BarterCast mechanism.

Figure 3 only shows which combination of the methods is better, but it does not tell how much they are better. To have a grasp of the improvement rate we compare the mean and the median of estimation errors. Figure 4 shows the mean and its standard error for all combinations of the modifications. As the graph shows, only changing the number of hops or using the central node does not improve much, and using the full gossiping is needed. Then, using both the central node and a higher number of hops decrease the estimation error, and when all modifications are applied, the mean of the errors becomes 0.404.

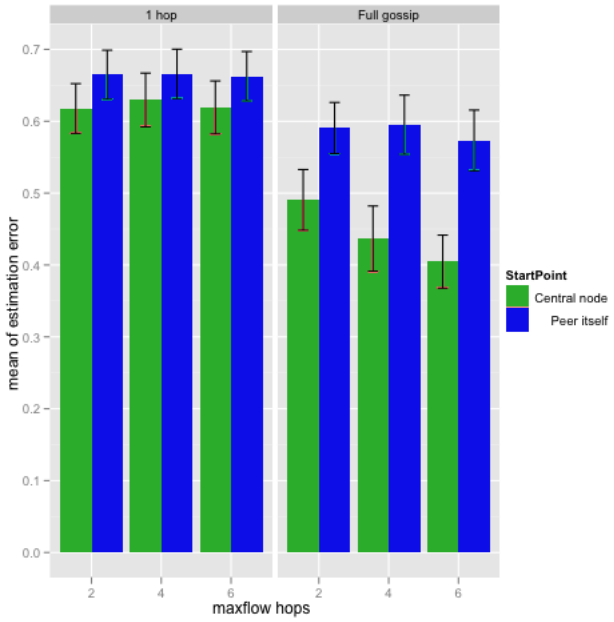


Fig. 4. The mean of the estimation error in the BarterCast mechanism. (Error bars show the standard error of mean.)

As the mean value may be biased by a small number of very high or very low values, we compare also the median of estimation errors in different scenarios. Figure 5 shows the median of the error in various situations. As it is seen, in 1-hop dissemination using the central node or higher maxflow hops only have a little influence on the decrease of the median and using the full gossiping is required. In full gossiping mode using the central node is very effective and when it is combined with higher maxflow hops the median is decreased by a factor of 10 and in the ultimate case it is pushed to below 0.09.

E. Statistical Analysis

The graphs with mean and median give an indication of the difference of these statistics in various scenarios, but they do not assess the significance of the differences. Figure 6 shows the density plots of the estimation errors in full gossiping and 6 hops maxflow for both central node and view owner. As it is seen from these plots the estimation error values do not follow a Gaussian distribution. Because of the non-Gaussian property of the distributions, using a non-parametric test is preferred and in our analysis we use

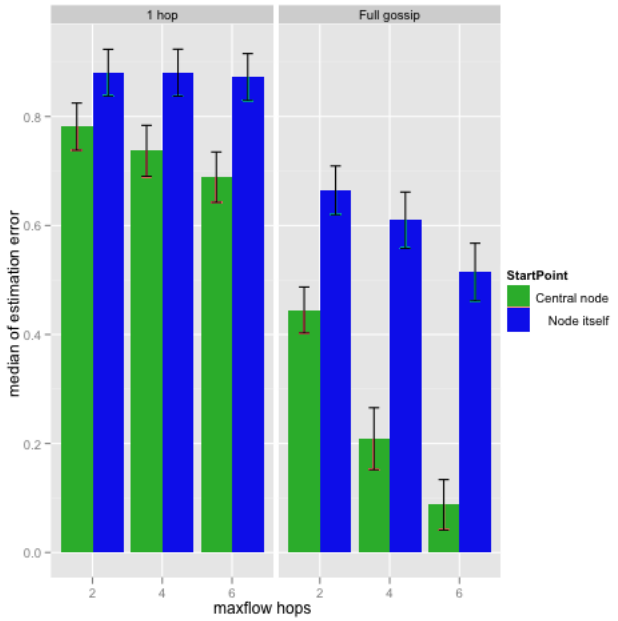


Fig. 5. The median of the estimation error in the BarterCast mechanism. (Error bars show the standard error of median.)

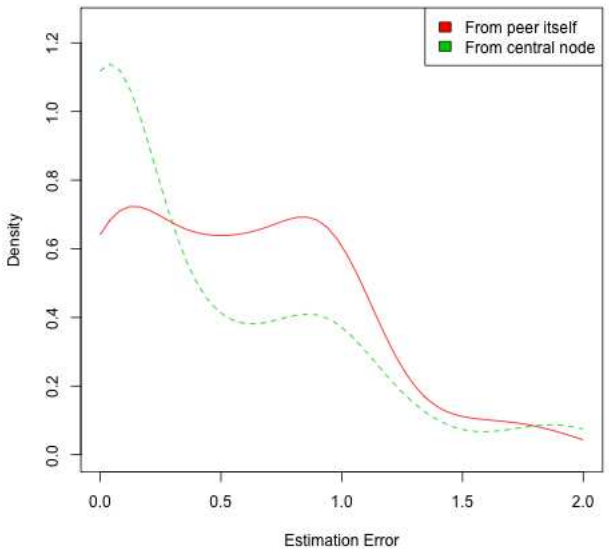


Fig. 6. Density plot of the error values with full gossiping and 6-hop maxflow

the Wilcoxon signed-rank test to compare the change of estimation errors. In this test, the null hypothesis is the equality of the medians and we test whether it is rejected or not. In each test we change one system parameter related to the three modifications of BarterCast discussed in this paper, and test whether the change in error values is significant. Tables I to IV contain the p-Value and the test results. In the first table we change the dissemination method and compare error values in various combinations of the other parameters (first and second column). The third column shows the p-Value, and the fourth column says whether with significant level of 95% the equality of the medians is rejected or not.

Table II is the same as Table I except that in this table the central node is compared with the view owner and as the last column shows, using the central node is only effective in full gossiping. Tables III and IV contain the test results of changing the number of hops in 1-hop and full gossiping dissemination, respectively. In Table III we do not see any rejection, and increasing the number of hops in maxflow does not help in 1-hop dissemination. In Table IV, which tests the effect of increasing the maxflow hop number in full gossiping mode, we just see a single rejection in the last row, meaning that using more hops is useful only if the central node and full gossiping are applied.

TABLE I
WILCOXON TEST FOR THE DIFFERENCE IN ESTIMATION ERRORS FOR 1-HOP DISSEMINATION AND FULL GOSSIPING.

number of hops in maxflow	maxflow start point	p-Value	rejected? (95% sig. level)
2	view owner	0.0898	No
4	view owner	0.0585	No
6	view owner	0.0126	Yes
2	central node	0.0015	Yes
4	central node	1.617e-05	Yes
6	central node	2.031e-07	Yes

TABLE II
WILCOXON TEST FOR THE DIFFERENCE IN ESTIMATION ERRORS FOR USING CENTRAL NODE AND VIEW OWNER.

number of hops in maxflow	dissemination	p-Value	rejected? (95% sig. level)
2	1-hop	0.3032	No
4	1-hop	0.3309	No
6	1-hop	0.2863	No
2	full gossip	0.0160	Yes
4	full gossip	0.0008	Yes
6	full gossip	0.0003	Yes

TABLE III
WILCOXON TEST FOR THE DIFFERENCE IN ESTIMATION ERRORS USING 1-HOP DISSEMINATION AND CHANGING NUMBER OF HOPS IN MAXFLOW ALGORITHM.

change in number of hops in maxflow	maxflow start point	p-Value	rejected? (95% sig. level)
2 to 4	view owner	0.9686	No
4 to 6	view owner	0.8719	No
2 to 6	view owner	0.8318	No
2 to 4	central node	0.9721	No
4 to 6	central node	0.8297	No
2 to 6	central node	0.7936	No

F. Maxflow Runtime

In the BarterCast mechanism one of the reasons for limiting the maxflow hops to 2 is to decrease the computation overhead. To analyze the overhead of increasing hop numbers, we run the maxflow algorithm in 2 and 6 hops and compare the runtimes. Like the experiment for accuracy and coverage, also in this experiment the destination node in the maxflow algorithm is an existing node which is evaluated by the source node (a view owner). In the complete graph (a

TABLE IV
WILCOXON TEST FOR THE DIFFERENCE IN ESTIMATION ERRORS USING FULL GOSSIP DISSEMINATION AND CHANGING NUMBER OF HOPS IN MAXFLOW ALGORITHM.

change in number of hops in maxflow	maxflow start point	p-Value	rejected? (95% sig. level)
2 to 4	view owner	0.7633	No
4 to 6	view owner	0.4640	No
2 to 6	view owner	0.2862	No
2 to 4	central node	0.1700	No
4 to 6	central node	0.2440	No
2 to 6	central node	0.0117	Yes

graph combined of subjective graphs of all peers) for each pair of source and destination peers, we run the maxflow algorithm 100 times and average the runtime. Figure 7 shows the experiment result, sorted by the runtime in 2 hops and compares it with the runtime in 6 hops. As it is seen, the runtimes are bounded between 200 and 450 ms and from the performance point of view their difference is negligible.

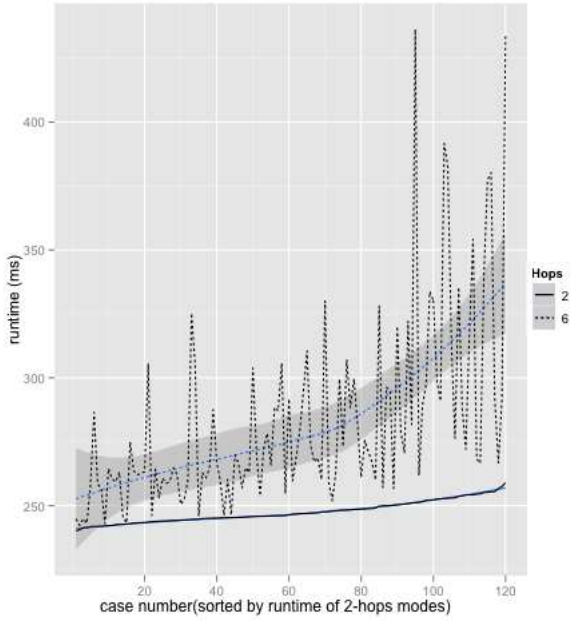


Fig. 7. Runtime of the maxflow algorithm in 2 and 6 hops

VI. CONCLUSION AND FUTURE WORK

In this paper we have performed an empirical analysis of the accuracy and the coverage of the BarterCast reputation mechanism and proposed three applicable modifications to improve these values: using betweenness centrality, using full gossip instead of 1-hop dissemination of BarterCast records, and increasing the path length in the maxflow algorithm. Our results show that using full gossip leads to the large improvement according to our metrics. Moreover, the other two modifications provide significant improvements, but only if combined with full gossip.

After understanding the improvements leveraged by changes in the design of BarterCast, some open questions related to the proposed improvements need now to be addressed. Also full gossiping increases the dissemination performance, but it is more vulnerable to misreporting attacks, and the indirect reports should be treated carefully. A possible solution for this problem could be to put the indirect reports in a secondary view and to add them to the primary view, used for reputation evaluation, if they are received from more than a certain number of peers or by highly reputed peers. Another method to address the misreporting attack is the use of double signatures. In this solution, before disseminating a record, the content sender and receiver sign the associated BarterCast record using their private keys. Using this technique no other peer can eavesdrop and change the record.

The second problem related to the proposed solutions is the performance of finding the node with the highest betweenness centrality in the subjective graphs. The complexity for this calculation is considerable, $O(nm)$ for unweighted and $O(mn+n^2 \log n)$ for weighted graphs [13], where n and m are the number of nodes and edges respectively. Our results are based on unweighted version and even with $O(nm)$ complexity the usage patterns observed so far hint at its practical feasibility. Our preliminary analysis suggests that the subjective graphs in BarterCast have a scale-free property, and the highest central node in them does not change very often. In this setting, the computation of the central node can be done periodically and in relatively long periods, reducing the amount of computation overhead in the system.

Finally, we believe that our proposed solution and especially the betweenness centrality can be applied not only in BarterCast, but in any flow-based reputation mechanism that has similar features as described in Section II-A. Both addressing the open questions in implementing the proposed improvements and evaluating the generality of the use of betweenness centrality are challenging topics for future research.

ACKNOWLEDGMENT

This work was partially supported by the European Community's 7th Framework Program through the P2P-Next and QLectives projects (grant no. 216217, 231200).

REFERENCES

[1] M. Meulpolder, J. Pouwelse, D. Epema, and H. Sips, "BarterCast: A practical approach to prevent lazy freeriding in P2P networks," 2009.
[2] J. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. Epema, M. Reinders, M. Van Steen, and H. Sips, "Tribler: A social-based peer-to-peer system," *Concurrency and Computation–Practice and Experience*, vol. 20, no. 2, pp. 127–138, 2008.
[3] P. Resnick and R. Zeckhauser, "Trust among strangers in Internet transactions: Empirical analysis of eBay's reputation system," *Advances in Applied Microeconomics: A Research Annual*, vol. 11, pp. 127–157, 2002.
[4] L. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, no. 1, pp. 35–41, 1977.
[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press and McGraw-Hill, 2001, pp. 651–664.
[6] M. Piatek, T. Isdal, A. Krishnamurthy, and T. Anderson, "One hop reputations for peer to peer file sharing workloads," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2008, pp. 1–14.
[7] A. Cheng and E. Friedman, "Sybilproof reputation mechanisms," in *Proceedings of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems*. ACM, 2005, p. 132.

[8] S. Seuken, J. Tang, and D. C. Parkes, "Accounting Mechanisms for Distributed Work Systems," in *Proceedings 24th AAAI Conference on Artificial Intelligence (AAAI '10)*, 2010.

[9] L. Xiong, L. Liu, and M. Ahamad, "Countering feedback sparsity and manipulation in reputation systems," in *Proceedings of the 2007 International Conference on Collaborative Computing: Networking, Applications and Worksharing*. Citeseer, 2007, pp. 203–212.

[10] S. Kamvar, M. Schlosser, and H. Garcia-Molina, "The eigentrust algorithm for reputation management in p2p networks," in *Proceedings of the 12th international conference on World Wide Web*. ACM New York, NY, USA, 2003, pp. 640–651.

[11] S. Buchegger and J. Le Boudec, "A robust reputation system for mobile ad-hoc networks," *Proceedings of P2PEcon, June*, 2004.

[12] M. Barthélemy, "Betweenness centrality in large complex networks," *The European Physical Journal B-Condensed Matter and Complex Systems*, vol. 38, no. 2, pp. 163–168, 2004.

[13] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

Chapter 8
Appendix B: Reducing the History in Reputation Systems

D. Gkorou, Tamás Vinkó, Nitin Ciluka, J.A. Pouwelse, and D.H.J. Epema,
In *Proceedings 17th Annual Conf. of the Advanced School for Computing and Imaging (ASCI '11)*, Veldhoven, the Netherlands, Nov. 2011.

Reducing the history in reputation systems

Dimitra Gkorou, Tamás Vinkó, Nitin Chiluka, Johan Pouwelse, Dick Epema
Delft University of Technology
{D.Gkorou, T.Vinko, N.J.Chiluka, J.A.Pouwelse, D.H.J.Epema}@tudelft.nl

Abstract

Many reputation systems are characterized by high computational cost and high memory requirements, and such systems have to be able to deal with high population turnover. Consequently, the use of the complete historical information about the past behavior of nodes in the system when computing reputation values, is prohibitive. In this paper we propose the use of a subset of the complete history in the computation of reputations defining local and global criteria for the choice of this subset. Next, we evaluate our approach focusing on max-flow based reputation systems using both synthetic graphs and graphs derived from our Internet-deployed Bartercast reputation system. Our experimental analysis shows the effectiveness and limitations of our approach.

1 Introduction

Reputation systems have been widely used in large Internet-deployed systems (such as eBay, YouTube and Amazon) to establish trust among users and provide incentives by rewarding good behavior. In reputation systems, nodes store information about the past behavior of other nodes, and based on this information, they compute reputations in order to take decisions about their future interactions. A family of reputation systems useful in many applications is flow-based reputation systems which consist in two categories, systems using max-flow (such as Bazaar [1], Bartercast [2], and the system proposed by Feldman et al. [3]), and systems aggregating the flows of reputations (such as EigenTrust [4], PeerTrust [5], Pagerank [6]). The amount of historical information on the interactions in the system maintained for each node affects the performance and the characteristics of the reputation system. Most proposed systems use the complete history of interactions in the system, but this approach is inefficient in terms of the cost of computing the reputations, the memory storage of the system, and the ability to capture the dynamic behavior nodes. Therefore, in this paper we propose and assess a scheme for reducing the amount of history maintained for each node and its impact on the computed reputations.

There are two important reasons for reducing the history of the interactions. First, networks such as popular online markets, social networks, and file-sharing peer-to-peer systems, which benefit from a reputation system by improving the trust of users' interactions, consist of hundreds of thousands or even millions of active users. Because many reputation algorithms require a computational cost proportional to the square of the number of nodes, using the complete history for computing the reputation of nodes in a large and growing network is prohibitive. Secondly, the complete history of interactions in the computation of reputations accumulates old information, impeding the nodes from capturing the dynamic behavior of the system. In highly dynamic systems where many nodes enter and leave the system within a short period, using long-term history results in including information from inactive nodes. Furthermore, a long-term history creates the possibility that a previously well-behaved node will exploit its good reputation by acting maliciously. For these reasons, some widely used reputation mechanisms, such as those of eBay and eLance, use historical information of a 6-month window.

In this study, we use only a subset of the complete history of interactions to approximate reputations. We model the network of the *complete history* as a growing graph G and the corresponding subset of the *reduced history* as a subgraph G' of G. The reduced history G' is derived from G by deleting the least important edges and nodes. We define the importance of a node according to its age, its activity level, its reputation, and its position in the graph, while the importance of an edge is defined according to its age, its weight, and its position in the graph. Then we evaluate our approach in terms of approximating the reputations using synthetic random and scale-free graphs, and a real-world graph derived from the Bartercast [2] the reputation system of our BitTorrent-based P2P client Tribler [7]. In this study, our definition of reputation is based on max-flow because our main motivation is Bartercast. However, our approach can be generalized to other definitions of reputations as well.

2 Motivation and Description of the Problem

In this section, we discuss the main motivation for our approach and we formalize the problem we study.

2.1 Motivation

The main motivation for using a subset of the complete history of interactions in a network is the computational cost and the storager requirements of reputation algorithms. Reputation systems, such as that of eBay, cover hundreds of

thousands of active users while reputation algorithms (e.g., the Ford-Fulkerson algorithm for max-flow, Eigentrust, and PageRank) usually have a high computational complexity. Personalized reputation and recommendation systems are even more computationally intensive.

The dynamic behavior of many reputation systems makes the use of the complete history ineffective. In systems with a high population turnover such as peer-to-peer networks, only a few nodes remain for a long period in the system while the majority of nodes enters the system performing some interactions and then leaves it. As a result, the graph representing the complete history of interactions has a dense core with a few long living and very active nodes, and a periphery with many loosely connected nodes that are inactive for a long time. Moreover, most new links tend to be attached to this dense core. This behavior has also been observed in Bartercast. In these systems, using a long-term history results in considering information from inactive nodes in the computation of reputations, decreasing their accuracy.

2.2 Description of the problem

Our problem consists in finding a representative subset of the complete history of interactions in a reputation system and using this representative subset to approximate the real values of reputations. We model the interactions among the nodes of a network as a directed weighted graph $G = (V, E)$, where the vertices represent the nodes and the edges the interactions. The weight of an edge represents its importance; for instance, in Bartercast, the weight of an edge between nodes represents the amount of data transferred in the corresponding direction. The graph is growing and allows existing nodes to create new edges and new nodes to join the graph, since in real networks the existing nodes create new links and new nodes enter the system. The graph G represents the *complete history* of interactions in the network.

Given the large, growing graph G, our target is to create a subgraph of G, denoted by G', with similar properties regarding the reputations of nodes and to dynamically maintain G' as the complete history grows. The graph G' will be used for the computation of reputations, and represents the *reduced history* of interactions in the network.

3 The Bartercast Reputation Mechanism

In this section, we provide the necessary background concerning the Bartercast reputation system [2,8]. Using Bartercast, each peer in Tribler computes reputations of other peers based on the history of their upload and download activities. A peer achieves a high reputation by uploading much more than downloading, and other peers give a higher priority to it when selecting a bartering partner. In Bartercast, when a peer exchanges content with another peer, they both store a *Bartercast record* with the amount of data transferred and the identity of the corresponding peer. Regularly, peers contact another peer to exchange Bartercast records using a gossip-like protocol. Therefore, each peer keeps a history not only of the interactions in which it was directly involved, but of interactions among other peers as well.

From the Bartercast records it receives, every peer i dynamically creates a weighted, directed *subjective graph* G_i, the nodes of which represent the peers about whose activity i has heard through Bartercast records, and the edges of which represent the total amounts of data transferred between two nodes in the corresponding directions. Each peer i computes its subjective reputation of every other peer j by applying the Ford-Fulkerson algorithm [9] to its current subjective graph and computing the value of $\arctan(f_{jk} - f_{ik})/(\pi/2)$, where node k represents the node with the maximum betweenness centrality[1], f_{jk} represents the maximum flow from node j to node k in the network and f_{kj} is the maximum flow in the reverse direction. The flow f_{ji} is limited to the sum of the weights of the in-links of peer i, no matter what uploads peer j reports.

In our experimental analysis, we will assume *full-gossip* in which peers are allowed to forward the records they receive from other peers, and so all peers eventually receive all the propagated records. Thus, the graph derived from Bartercast can be considered as the subjective graph of all nodes. In our experiments, we will use the definition of reputation used in Bartercast.

4 Creating the Reduced History

The basic idea of creating G' consists in removing the least important elements, either nodes or edges, from G. On the one hand, a graph is stored as a list of edges, so the removal of edges is a natural way of decreasing its size. On the other hand, in a dynamic network, nodes leave the network and we do not want to consider their information in the computation of reputations. Therefore, a node removal process is needed in conjunction with the edge removal. The ratio of removed nodes versus removed edges depends on the dynamics of the network. In highly dynamic networks, a high number of new nodes join the system and many old nodes leave the system, and as a result, the dominant process is the node removal. Nevertheless, edge removal implies node removal and vice versa. More precisely, edge removal can lead to disconnecting a node from the graph and node removal results in deleting the adjacent edges of the removed node.

4.1 Parameters of node removal

The selection of a node to be removed is based on its age, its activity level, its reputation, or its position in the graph. The first factor determining the node removal is the age, since we want to keep fresh information for two reasons. First,

[1] The betweenness centrality of a node measures the sum of the fractions of the numbers of shortest paths among all pairs of vertices that pass through that node [10].

computing the reputations using fresh information allows our computation to capture the dynamic behavior of nodes. Secondly, maintaining too old information may result in considering information of inactive nodes or nodes without any recent activity. The age of node i is expressed as $\tau_i = t - t_i$ where t is the current time and t_i is the time instance node i joined the system. In most networks [11], [12], [13], the age of a node i affects its behavior in a non-linear way, and so, instead of its age, we consider its aging factor $f(\tau_i)$, with f a decreasing function with $f(0) = 1$, which can be exponential (e.g., $f(\tau) = e^{-b\tau}$, where τ represents the age of a node and b is a constant).

Another important factor is the activity level d_i of a node i and it is represented by its degree. Nodes with a high activity level have to be maintained in the reduced history graph, since these nodes participated in many interactions, and so, they provide much information.

Furthermore, in the reduced history, we want to preserve the information of nodes with high reputations, since these nodes are the most reliable in the network. Moreover, allowing nodes with high reputations to contribute to the computation of reputations longer is a kind of rewarding the most trusted nodes. We assume that the values of reputations are normalized to $[-1, 1]$, and we denote by r_i the reputation of node i.

The last factor we consider for the removal of a node is its position in the graph. Removing nodes from the graph can result in destroying its structure by creating many disconnected components. Therefore, keeping nodes that keep the graph connected is necessary. The importance of the position of node i in the graph is defined by its betweenness centrality, denoted by $C_B(i)$.

The first three factors represent the behavior of node i while the fourth factor is added just for preserving the structure of the graph during the deletion process. Based on the factors described above, the probability $P_n(i)$ of deleting a node i can be expressed as:

$$P_n(i) = \alpha P_A(d_i, r_i, \tau_i) + \beta P_B(C_B(i)), \qquad (1)$$

where $P_A(d_i, r_i, \tau_i)$ expresses the probability of deleting node i based on its activity level, aging factor and reputation, and $P_B(C_B(i))$ expresses the probability of deleting node i according to its position in the graph. The parameters α and $\beta = 1 - \alpha$ take values in $[0, 1]$ and can be chosen according to the graph properties. If the structure of a graph (such as random graph) is sensitive to node removal, it is critical to maintain nodes with high betweenness to keep the graph connected, whereas in graphs such as power-law graphs, node removal does not affect the graph so much but the removal of the highly connected nodes does.

We define the probability P_A as:

$$P_A(d_i, r_i, \tau_i) = \frac{n - d_i r_i f(\tau_i)}{n^2 - \sum_j d_j r_j f(\tau_j)},$$

where n is the number of nodes in the graph, and where the denominator acts as a normalization so that the sum of the probabilities sum to 1. Clearly, a node with a higher age, a lower activity level, or a lower reputation is more likely to be removed. Although the maximum value of $d_i r_i f(\tau_i)$ is equal to $n-1$ (corresponding to $d_i = n-1, r_i = 1$ and $f(\tau_i) = 1$), for simplicity, we approximate it by n. Similarly, P_B can be expressed as:

$$P_B(C_B(i)) = \frac{n^2 - C_B(i)}{n^3 - \sum_j C_B(j)}.$$

Again, even though the maximum value of $C_B(i)$ is equal to $(n-1)(n-2)$, we approximate it by n^2 like in the definition of P_A.

4.2 Parameters of edge removal

The removal of an edge is determined by its age, its weight, and position in the graph. Similarly to the age of a node, we define the age of an edge e_{ij} connecting nodes i and j, as $\tau_{ij} = t - t_{ij}$, where t is the current time and t_{ij} the time of its creation. The aging factor of edge e_{ij} is a decaying function $f(\tau_{ij})$ and can be, e.g., an exponential function.

The weight of edge e_{ij}, denoted by w_{ij}, indicates its importance; for instance, in Bartercast it represents the amount of data transferred data in the direction of the link. Since interactions with a high cost are more important for the computation of reputations, edges with high weights have to be preserved in the graph. The importance of the edge e_{ij} in terms of its position in the graph is defined by its edge betweenness centrality $C_E(e_{ij})$. The betweenness centrality of an edge is defined as the sum of the ratios of shortest paths between all pairs of nodes containing this edge. The weight and the aging factor of an edge represent the importance of an edge in terms of its contribution to the computation of reputations, while the edge betweenness centrality helps in preserving the structure of the graph.

Similarly to node deletion, the probability of removing edge e_{ij} can be expressed as:

$$P_e(e_{ij}) = \alpha P_S(w_{ij}, \tau_{ij}) + \beta P_F(C_E(e_{ij})), \qquad (2)$$

where α and β are the parameters used in the definition of P_n to control the topology of the derived graph. The probabilities P_S and P_F are defined similarly to P_A and P_B, respectively. More precisely, we define

$$P_S(w_{ij}, \tau_{ij}) = \frac{1 - w_{ij} f(\tau_{ij})}{m - \sum_{s,t} w_{st} f(\tau_{st})},$$

where m is the number of edges in the graph, and

$$P_F(C_E(e_{ij})) = \frac{n^2 - C_E(e_{ij})}{n^3 - \sum_{s,t} C_E(e_{st})}.$$

Therefore, edges with lower age, lower weight, and lower betweenness centrality are more likely to be removed.

5 Dataset

In order to assess our method for creating the reduced history, we consider both synthetic graphs and graphs derived from real networks. Our synthetic graphs include random and scale-free graphs. In most networks, two processes occur simultaneously: first, new nodes enter the system, and secondly, the already existing nodes interact, thus creating new links. To capture this behavior in our models, we define the probability p_c which represents the probability of adding a new node at each time step to the graph, and the probability $1 - p_c$ which represents the probability of adding new links between existing nodes. In highly dynamic systems, the appearance of new nodes is dominant, and so the value of p_c is high. In our models for synthetic graphs, we allow the occurrence of multiple edges formulating weighted graphs.

For our experiments, we create the complete history and the corresponding reduced history of a graph in parallel. In the complete history, we store all the new information, while for the construction of the reduced history we keep its size (almost) constant to a maximum number of stored nodes n_{max}, which represents the computational or memory limitation of the system. We control the size of the reduced history by removing nodes or edges from the graph as new information is stored as described in the previous section. Below, we describe in detail our models for random graphs and scale-free graphs, the properties of the Bartercast graph, and the construction of the corresponding reduced histories.

5.1 Random Graphs

A random graph [14], denoted by $R(n, p_r)$, is composed of n nodes, and each potential edge connecting two nodes occurs independently with probability p_r. Based on this model, we generate a growing directed random graph $R(n_t, p_r)$ representing the complete history of interactions.

To create the graph $R(n_t, p_r)$ with n_t nodes at time t, starting from a single node, we perform continuously the following operations:

- With probability p_c we add a new node with each of its potential directed edges existing with some probability p.
- With probability $1 - p_c$ we add pn_t new directed edges adjacent to randomly chosen existing nodes.

From the properties of random graphs, we can show that $p_r \sim \frac{p}{2p_c}$ (proof omitted due to the space limitation). In accordance with R, we create the reduced history graph R'. The reduced history R' is equal to R up to the maximum number of nodes n_{max}. After having reached n_{max} nodes, R' is created by performing continuously the following operations:

- When a new node is added to R, we also add this node to R' along with its edges, and then we remove one node together with its edges with probability P_n from Eq. (1).
- When new edges are added to R, we add the same edges to R'. Then we remove from R' the same number of edges according to the probabilities P_e from Eq. (2).

Note that some edges in R may connect nodes that have been removed from R'; in this case, these edges are not added to R'.

5.2 Scale-free Graphs

Scale-free graphs are characterized by their degree distribution following a power law, i.e., their degree distribution satisfies $P(k) \approx ck^{-\gamma}$, where $P(k)$ is the fraction of nodes of degree k, γ denotes the power-law exponent, and c is a constant. We create a growing directed scale-free graph using the preferential attachment model [15], according to which a new node joining the network is attached to nodes already existent with probability proportional to their degrees. Similarly to the procedure for random graphs, we generate two directed graphs S and S' corresponding to the complete history and the reduced-history.

We create $S(n_t)$ by starting with a small seeding graph with m_0 nodes connected by $m_0 - 1$ edges and then performing the following steps:

- With probability p_c we add a new node with m directed edges, with $m \leqslant m_0$. From these m links adjacent to the new node. Each edge is adjacent to an already existing node i with probability $\Pi(i) = d_i / \sum_j d_j$, with d_i the degree of node i.
- With probability $1 - p_c$ we add m new directed edges. Each of these edges are adjacent to an existent node i with probability $\Pi(i)$.

One can show that S is scale-free with power-law exponent equal to $\gamma = 1 + 2/(2 - p_c)$ (proof omitted due to space limitation). In line with S, we build the reduced history S' using the same procedure as for random graphs.

5.3 Bartercast Graphs

We have crawled the Tribler system from September 1 to December 23, 2010, collecting information of the Bartercast reputation system from $29,716$ nodes. Based on this information, we build the corresponding graph B where vertices represent the nodes of Bartercast and edges represent the data transferred between two nodes across the corresponding direction. The graph derived from Bartercast is not connected and so, we proceed in the analysis using its largest connected component containing $8,838$ nodes and $25,899$ edges. Bartercast is characterized by high population turnover and thus, the derived graph consists in a dense core with very few long living and active nodes and a periphery with many loosely connected nodes of low activity.

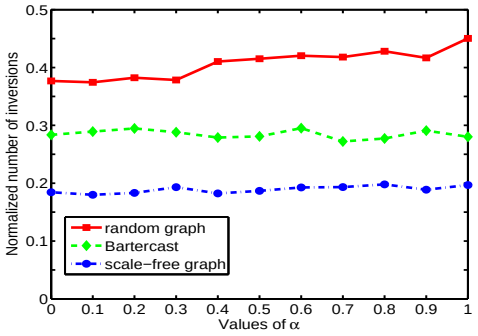


Figure 1: Impact of α on the number of inversions between the sequences of reduced and complete history in random and scale-free graphs, and the graph derived from Bartercast.

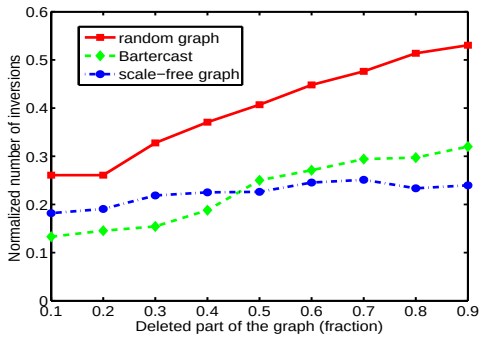


Figure 2: Fraction of inversions between the sequences of reduced and complete history in random and scale-free graphs, and the graph derived from Bartercast when a fraction of the graph has been removed.

Similarly to the procedure for random and scale-free graphs, we build the complete and the reduced history for Bartercast, B and B' respectively. For the complete history B, we use all the information crawled from the Bartercast system and for the reduced history B', we remove information using Equation 1 and 2 as new information is added and in this way, we keep its size constant.

6 Evaluation

In this section, we evaluate experimentally the *accuracy* achieved when we use the reduced history instead of the complete history, showing the effectiveness and the limitations of our approach. As a metric of accuracy, we use the preservation of the ranking of the nodes in the reduced history according their reputation values in comparison with the complete history. More precisely, we consider the sequences of the Unique Identifiers (UIDs) of the nodes in the complete and the corresponding reduced history of our graphs and we compute the minimum number of inversions needed in the sequence of the reduced history to get all the common nodes in their correct order in the complete history. The minimum

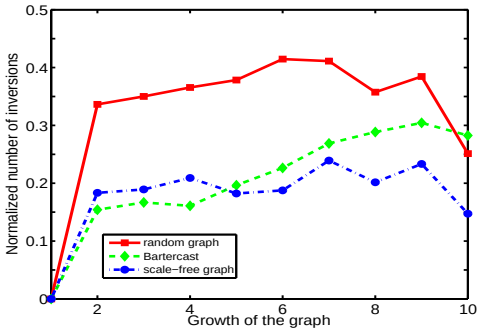


Figure 3: Fraction of inversions between the sequences of reduced and complete history in random and scale-free graphs, and the graph derived from Bartercast when the initial graph grows i times for $i = 1, \ldots, 10$.

number of inversions is normalized over the worst case.

First, we study the impact of the parameter α in Eq. (1), thus assessing whether the use of betweenness centrality for the deletion of nodes and edges is significant for the accuracy between the sequences of the reduced and the complete history. We use a random graph $R(n, p_r)$, a scale-free graph $S(n)$ with $n = 2,000$ nodes, both constructed as described in Section 5 with $p_r = 0.02$, and with $m = 3$ and $\gamma = 2.2$, respectively and the last instance of Bartercast graph B_3. The fraction of nodes and edges removed from each graph is equal to 50%. In Figure 1, we can observe that in random graphs, increasing α results in deterioration of the accuracy by 19% from its initial value. In contrast, in scale-free graphs and in the graph derived from Bartercast varying α does not affect the accuracy since in these graphs, betweenness centrality and degree are correlated. Low values of α mean that the main parameter of node and edge deletion is betweenness centrality. The structure of random graphs is more sensitive to node deletion, and so, lower values of α result in higher accuracy, while scale-free graphs and the graph derived from Bartercast are robust to node and edge removal (as long as the most central nodes are preserved) and the value of α does not affect the accuracy.

Next, we examine the preservation of the ranking of the nodes according to their reputations when we remove a fraction of the graph starting with 0.1 up to 0.9. We use a random, and a scale-free graph with $2,000$ nodes with the same characteristics as in the previous experiment, and we remove a part of the graph each time evaluating its impact on the ranking of reputations. We observe that in random graphs, the ranking of reputations deteriorates a lot when the size of the part being deleted from the graph increases, while in scale-free graphs, even if the deleted part is large, the ranking of reputations is highly preserved, as we can see in Figure 2. In random graphs, all nodes have similar statistical properties and so, when a part of the graph is removed, the ranking on reputation of the remaining nodes changes a lot compared to the complete history. In contrast, in scale-free graphs, only a

few nodes gather the majority of links and as a result, these nodes are the most central and have high reputations, while the vast majority of nodes has a very small connectivity. The highly connected central nodes are preserved with very high probability and the rest of nodes has very small connectivity. Since there is a large gap between the characteristics of different nodes in scale-free graphs, the ranking of reputations is well preserved even if we remove a part of the graph. In Bartercast graph, the ranking of reputations is well preserved as in scale-free graphs because it is characterized by a dense core with very few highly connected nodes and a periphery containing the majority of nodes which have very low connectivity. Similarly to scale-free graphs, the highly connected nodes are preserved with very high probability and as a result, the ranking of reputations is highly preserved.

Finally, we evaluate the accuracy of using the reduced history instead of the complete history in growing graphs. We consider random and scale-free graphs growing from 500 to $5,000$ nodes, while the corresponding reduced history is of constant size of 500 nodes. The Bartercast graph grows from 884 to $8,838$ nodes and its corresponding reduced history has constant size of 884 nodes. In scale-free graphs, the ranking of reputations of the reduced history follows the ranking of the complete history with higher accuracy than in random graphs, as is presented in Figure 3. In scale-free graphs and in Bartercast graph, the highly connected nodes are preserved with high probability in the reduced history and the rest of the nodes has very small centrality and connectivity. Consequently, the reduced history approximates the ranking of reputations accurately because of the gap between the properties of the most central nodes and the rest of the network. However, in random graphs our approach does not perform accurately due to the similar properties of nodes.

We have presented our experiments for ranking nodes based on their reputations. Furthermore, we have also performed the same experiments with the relative error in the reputation values as a metric, but we do not exhibit the corresponding figures here due to space limitations. According to our experiments, the reduced history does not approximate accurately the values of reputations. In random graphs, the relative error in the values of the reputations in the reduced history compared to the complete history is about 0.9 for all experiments, in the scale-free graphs the relative error is about 0.5, and in the graph derived from Bartercast the relative error is about 0.9. Therefore, the relative values of reputations are very sensitive to removal of information in the graph while the ranking is more robust.

7 Conclusion and Future Work

Using the complete history of interactions in a reputation system is not efficient due to its high computational cost and high memory requirements, and to the high population turnover. We have proposed the use of the reduced history instead of the complete history defining the main parameters for choosing the nodes participating in it. Next, we have evaluated our approach experimentally exploring both theoretical graph models (random and scale-free) and the graph derived from the Bartercast reputation system. We conclude that for scale-free graphs the reduced history is accurate and efficient while for random graphs due to their structural properties, the reduced graphs are not accurate enough. In the graph derived from Bartercast, the use of the reduced history performs with good accuracy. In this study, our analysis focuses on max-flow based reputation but as a future work, we plan to explore the effectiveness of our approach for other definitions of reputation such as those based on eigenvector centrality.

References

[1] A. Post, V. Shah, and A. Mislove, "Bazaar: Strengthening user reputations in online marketplaces," in *NSDI*, 2011.

[2] M. Meulpolder, J. Pouwelse, D. Epema, and H. Sips, "Bartercast: A practical approach to prevent lazy freeriding in p2p networks," in *IEEE IPDPS (Hot-P2P)*, 2009.

[3] M. Feldman, K. Lai, I. Stoica, and J. Chuang, "Robust incentive techniques for peer-to-peer networks," in *ACM EC*, 2004.

[4] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina, "The eigentrust algorithm for reputation management in p2p networks," in *WWW*, 2003.

[5] L. Xiong and L. Liu, "Peertrust: Supporting reputation-based trust for peer-to-peer electronic communities," *IEEE TKDE*, 2004.

[6] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Stanford InfoLab, Technical Report 1999-66, 1999.

[7] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. R. van Steen, and H. J. Sips, "Tribler: a social-based peer-to-peer system," *Concurr. Comput.: Pract. Exper.*, 2008.

[8] D. Gkorou, J. Pouwelse, and D. Epema, "Betweenness centrality approximations for an internet deployed p2p reputation system," in *IEEE IPDPSW (HotP2P)*, 2011.

[9] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. The MIT Press, 1990.

[10] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, 1977.

[11] K. Hajra and P. Sen, "Aging in citation networks," *Physica A: Statistical Mechanics and its Applications*, 2005.

[12] H. Zhu, X. Wang, and J. Zhu, "The effect of aging on network structure," *Phys. Rev. E*, 2003.

[13] L. A. N. Amaral, A. Scala, M. Barthélémy, and H. E. Stanley, "Classes of small-world networks," *Proc. Natl. Acad. Sci. U.S.A*, 2000.

[14] P. Erdős and A. Rényi, "On random graphs I," *Publicationes Mathematicae (Debrecen)*, 1959.

[15] A. L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, 1999.

Chapter 9

Appendix C: Access Control in BitTorrent P2P Networks Using the Enhanced Closed Swarms Protocol

V. Jovanovikj, D. Gabrijelčič and T. Klobučar. In Proceedings of the Fifth International Conference on Emerging Security Information, Systems and Technologies, SECURWARE 2011, pp. 97-102, IARIA, 2011.

Access Control in BitTorrent P2P Networks Using the Enhanced Closed Swarms Protocol

Vladimir Jovanovikj, Dušan Gabrijelčič, Tomaž Klobučar
Laboratory for Open Systems and Networks
Jožef Stefan Institute
Ljubljana, Slovenia
E-mail: vladimir@e5.ijs.si

Abstract— **The future content delivery platforms are predicted to be efficient, user-centric, low-cost and participatory systems, with social and collaborative connotation. The peer-to-peer (P2P) architectures, especially ones based on BitTorrent protocol, give a solid basis for provision of such future systems. However, current BitTorrent P2P networks lack flexible access control mechanisms. In this paper enhancements to existing access control mechanism for BitTorrent systems – the Closed Swarms protocol are presented, providing additional flexibility in access control mechanism, enabling fine grained security policies specification and enforcement. The enhancements fulfill a number of content providers' requirements and promise efficient, flexible and secure content delivery in future content delivery scenarios.**

Keywords-access control, P2P, BitTorrent, flexible policy, Closed Swarms

I. INTRODUCTION

It is envisaged that in the future people will consume 3D content enriched with additional media types and technologies that will engage more of our senses and will provide us immersive experience. People will have the ability to create virtual and personalized environments that will correctly simulate the real world and could have a variety of everyday applications. The virtual environments together with enriched 3D content will bring the communication between people to a higher level, and at the same time will enhance the users' entertainment. Moreover, they will foster human creativity even more and the current trend of people to be not only consumers but also producers of media content is expected to grow [1].

Future content delivery platforms will need to be able to provide efficient delivery of such high quality media content (streaming and stored), on-demand or live to the consumers, with an excellent quality of service. They are predicted to be user-centric and capable of considering the social aspects of the users, as well as the data being delivered. In order to become economically successful, the future content delivery platforms will have to be suitable for large and small size content providers and to be low-cost. This can be achieved if they are participatory and collaborative systems, in which all customers will become actively involved in the content delivery process. Because of its characteristics the peer-to-peer (P2P) architectures gives a solid basis for future provision of such systems. Indeed, one of its most prominent representative [2] – the BitTorrent protocol [3] has already proved to be scalable, robust and efficient in delivery of large audio and video data, and suitable for live streaming and social interaction between its users [4][5][6]. Thus, BitTorrent promises to be a suitable P2P protocol for future P2P-based content delivery platforms.

In short, with BitTorrent peers exchange small and fixed size pieces of the content file. A group of peers sharing the same file is called a swarm. A peer needs to acquire a so called torrent file in order to start downloading. The torrent file contains the needed information for the protocol initiation. The sharing process is coordinated either by a central server – the tracker or by the participants themselves – using the DHT [7] protocol. BitTorrent uses tit-for-tat policies to provide fairness in the delivery process [8]. Peers that continue to upload after they have downloaded the whole content file (seeds) improve the downloading process of the other peers (leeches).

The future P2P-based content delivery platforms need to be secure and trusted in order to be widely accepted and used. The importance of security as well as the main security requirements for P2P networks have already been emphasized in [9][10]. Among them access control is considered basic and standard, especially by content providers. The access control in the P2P-based content delivery systems is quite difficult to accomplish because of the basic properties of the system: 1) the content consumers are directly involved in the process of content distribution, i.e. peers exchange the content among themselves; and 2) the system tends towards full decentralization, without even a single central party for administration.

The main goal of this paper is to propose several enhancements of an existing access control mechanism for BitTorrent P2P networks – the Closed Swarms protocol [11], that we believe will provide a flexible access control mechanism for future P2P-based content delivery platforms applicable in various scenarios. First, we give an overview of the existing approaches for access control in BitTorrent P2P networks in Section II. Then, we describe the motivation for enhancing the Closed Swarms protocol in Section III. We present our proposed enhancements in Section IV and furthermore discuss them in Section V. Finally, we conclude the paper and present our future work in Section VI.

 ISBN: 978-1-61208-010-9

II. RELATED WORK

Access control in P2P content delivery systems can be achieved either directly protecting the content being delivered or controlling the content delivery process.

An access control mechanism directly protecting the content is proposed by Zhang et al. [12]. It is basically a digital rights management (DRM) mechanism for BitTorrent, based on using trusted tracker and initial seed, as well as using trusted content viewer on the client side. The main idea behind their schema is existence of a single plaintext copy of the content being delivered, the one at the trusted initial seed, while all the other copies of the content, resting at the peers being part of the content delivery system are uniquely encrypted for every peer, piece by piece. The peers consume the content only with a trusted content viewer, which is responsible for decrypting the content according to the purchased license from the content provider. This scheme is highly dependable on the tracker, which is far from full decentralization and is an obvious security risk – a single point of failure. Moreover, it doesn't provide means for applying flexible content usage policies, even though it is possible to define copyright related usage policies into the license. All this makes this scheme not appropriate for the future P2P-based content delivery platforms. In addition, the encryption and increased communication with the tracker certainly have impact on the performance of the content delivery. It is worth mentioning that providing copyright protection in a fully decentralized environment that favours open source software is a task very difficult to fulfill.

Another mechanism for direct protection of the content is described by Jimenez et al. [13]. In their scheme, the provider first encrypts the content before it is being distributed among the peers. Only peers that commit a payment and satisfy the provider's policies are authorized to receive the decryption key and consequently are able to decrypt the content. Although this mechanism is capable for implementing a certain access control policies (for example based on geolocation), it depends only on one cryptographic key, which makes it to be easily compromised.

Private tracker [14] extension of the BitTorrent protocol is an access control mechanism for controlling the delivery process. It restricts access in the system by simply not giving information about the participants to unauthorized users, i.e. users that do not meet a certain criteria, such as minimum upload-to-download ratio. This mechanism is not appropriate for future P2P-based content delivery platforms as it is highly centralized. Also, it depends on peers using only one private tracker at a time as a peer discovery mechanism, which makes it be easily subverted.

Closed swarms (CS) protocol [11] is an access control mechanism for controlling the delivery process that acts on peer level. It enables peers to recognize the authorized peers and to avoid communication with the non-authorized ones. The distinction between authorized and non-authorized peers in the swarm is made based on possession of an authorization credential called proof-of-access (PoA). The peers exchange their credentials right after they establish connection, in a challenge-response messages exchange. In most severe case, with the CS protocol only the authorized peers receive service (content). Nevertheless, it is possible to design a system in which both users would receive service (content), but graded – the authorized users would receive additional or better service than the non-authorized ones, for example access to high speed seeds for better performance. The CS protocol can provide access control in an innovative business content delivery system, but only under the same conditions for all authorized users. Moreover, this protocol is vulnerable to man-in-the-middle attacks.

Another access control mechanism for controlling the delivery process that acts on peer level is Lockr [15]. It is a privacy preserving access control mechanism for social networks in general. It is also applicable in BitTorrent P2P networks, for people to control the delivery of their personal content via them. The content owner issues digitally signed social attestations to all persons it has a social interaction with. A social attestation certifies the social relationship between two persons. In order to start exchanging pieces of the content, two peers need to verify their attestations during a social handshake, a form of zero-knowledge protocol. This access control mechanism is fine example of improved privacy in content delivery and in social networks in general. However, it still lacks support of flexible access control policies for the future P2P-based content delivery platforms.

III. MOTIVATION

To motivate our work we describe the following scenario. An international TV broadcaster (a content provider) wants to distribute live TV program to its clients (authorized users) using a P2P-based content delivery platform, based on the BitTorrent protocol. The TV broadcaster aims at achieving fine grained load balancing and optimization of its program delivery process, and restricting its program's availability only in one country (e.g., only in Slovenia) because of the digital rights issues, although it is broadcasting other programs in several countries. Furthermore, the TV broadcaster decides to deliver a service to clients under different conditions. Premium clients, for example, would receive higher content quality (e.g., HD video) for a certain amount of money, whereas basic clients would receive lower content quality (e.g., SD video) for free. This is beneficial from business perspective, as it can increase indirect earnings, and from technical perspective, since it can improve content delivery. Moreover, clients should be able to purchase certain service packages in which they will receive high content quality only during certain time periods, e.g., every day from 18 till 20 hours, during the most popular show. Analysis of the scenario elicited the following requirements.

Requirement 1: Fine grained load balancing and optimization of the delivery process: In BitTorrent live streaming swarm, none of the peers, except the content injector, has the whole content in advance, as seeds in regular swarms do. Instead, seeds in live streaming swarm are special peers with outstanding properties (e.g., high bandwidth), which are always unchoked by the content injector and have the same role in content delivery as the regular seeds – they improve the other peers' download

 ISBN: 978-1-61208-010-9

performance. The seeds are often purposely set by the content provider and behave as a small Content Delivery Network (CDN) [6].

In order to achieve fine grained load balancing and optimization of the delivery process, the content provider (TV broadcaster) can create and maintain a hierarchical structure of seeds in the live streaming swarm, analogical to a hierarchical CDN [16]. This structure is formed by separation of the seeds into layers (levels) according to the priority assigned to them by the content provider (Fig. 1) and placing the seeds at strategic locations. The greater the priority of the seeds a layer contains is – the higher it appears in the structure. The value of the priority defines the level of precedence a seed has among the other peers in the live streaming swarm (seeds and leeches). Normally, the content injector and the seeds establish a connection to any peer in the swarm regardless of its priority, as long as they have a free connection. However, when a lack of free connection occurs, the connections with seeds having lower priorities or with leeches will be terminated in favour of seeds having greater priorities.

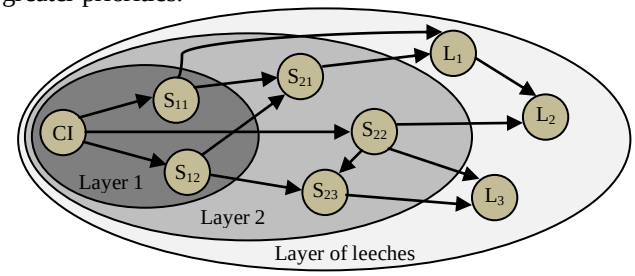


Figure 1. Hierarchical structure of a live streaming swarm: the content injector (CI) is not part of any layer; the seeds (S) from layer 1 have a priority (e.g., 20) greater than the seeds from layer 2 (e.g., 10); the leeches (L) are all placed in one layer and do not have any priority.

Two mechanisms are needed for the process of creation and maintenance of a hierarchical structure of seeds in a live streaming swarm.

Sub-requirement 1.1: Automatic introduction of a seed: Seeds download content only from the content injector or other seeds, which explicitly know them by maintaining lists of their identifiers (e.g., IP address and port number). However, these lists are maintained manually – something that becomes impractical for a large swarm (like in the scenario above) and very difficult for creation and maintenance of a hierarchical layered structure. Therefore, a mechanism for automatic introduction of a seed in the live streaming swarm is needed, that will also place the seed in a specific layer of the hierarchical structure.

Sub-requirement 1.2: Suitable peer discovery: This mechanism is needed to enable quick transport of the content from the content injector towards the lowest level of the structure of seeds, and consequently to the regular peers (clients). Currently, none of the peer discovery mechanisms the BitTorrent protocol supports (e.g., the tracker [3] or the DHT [7] protocol), takes into consideration a hierarchical structure of a live streaming swarm.

Requirement 2: Restriction of the content delivery based on peer location: According to this requirement, only peers inside one country are allowed to receive an authorization credential and join the swarm. The physical location of a peer on country level can be determined by using the Internet geolocation technology. Although tactics for evasion of this technology do exist, it is considered sufficient for compliance with the legal regulations [17]. The CS protocol needs to be properly extended in order to take into consideration the output of the Internet geolocation technology in the access control decision.

Requirement 3: Provision of different content quality in the same swarm: The content provider needs to create only one content stream by using a scalable video coding technique, but encoded in several layers [18]. Then, by specifying in the authorization credentials which layers the holders are allowed to receive, peers can easily determine which content pieces should provide to them. For example, premium clients would be authorized to receive content pieces from all the encoding layers, while other clients – only from fewer layers.

Requirement 4: Temporal constraints: In addition to the previous requirement, the authorization credential can also specify temporal constraints, for example, when the allowed content layers would be provided to the clients. This can even provide a basis for business models in the content delivery process by creating different service packages for the clients.

IV. THE ENHANCED CLOSED SWARMS PROTOCOL

We believe that after enhancement, the Closed Swarms protocol fulfills the requirements from Section III and becomes resistant to man-in-the-middle attacks. Before presenting the proposed enhancements of the CS protocol, in short we describe the format of the authorization credential and the message exchange process in the CS protocol, explained in detail in [11].

The authorization credential (called Proof of Access) of an arbitrary peer A (1) contains information about: the specific swarm – its identifier (SwarmID) and public key (K_S); the credential holder, defined by its public key (K_A); and the expiry time of the credential (ET). The credential issuer, usually the content provider in correlation with a payment system[1], digitally signs this information with the private key of the swarm (K_S^{-1}). The authorization credential is valid only when all the fields and the digital signature are correct.

$$SwarmID, K_S, K_A, ET, \{SwarmID, K_S, K_A, ET\}_{K_S^{-1}} \quad (1)$$

Two peers, an initiator – peer A, and a swarm member – peer B, exchange their credentials in a challenge-response message exchange process:

$$A \rightarrow B: SwarmID, N_A \quad (2)$$

$$A \leftarrow B: SwarmID, N_B \quad (3)$$

$$A \rightarrow B: PoA_A, \{N_A, N_B, PoA_A\}_{K_A^{-1}} \quad (4)$$

$$A \leftarrow B: PoA_B, \{N_A, N_B, PoA_B\}_{K_B^{-1}} \quad (5)$$

[1] The credential issuer signs credential for all swarms it is responsible for, by using their private keys. Although there is no specific protocol of issuing the credentials, the process is explained in detail in [11].

First, with (2) and (3) they exchange the identifier of the swarm they want to join/are part of and randomly generated nonces (N_A/N_B). Then, with (4) and (5) they exchange their credentials (PoA_A/PoA_B) followed by a concatenation of the previously exchanged nonces and the credential, digital signed with their private keys (K_A^{-1}/K_B^{-1}).

The requirements from Section III can be satisfied by using an access control based on flexible authorization framework and proper policy enforcement. A number of distributed frameworks have already been proposed in the past [19]. Here, we aim at integrating such distributed authorization framework in the CS protocol. Furthermore, although the protocol uses authorization credentials containing public key for owner identification, random nonces for message freshness and digital signatures for message authentication, it still remains vulnerable to man-in-the-middle attacks. An attacker can interfere in the communication between two authorized peers by simply relaying the messages between them. After the authorized peers successfully finish the protocol and start the content delivery, the attacker will be able to read all the exchanged content pieces, since they are not encrypted. We propose encryption of the content pieces with a shared secret key as a countermeasure for this attack.

The format of the extended authorization credential is as follows:

$$SwarmID, K_S, K_A, ET, Rules_A, \{SwarmID, K_S, K_A, ET, Rules_A\}_{K_S^{-1}} \quad (6)$$

The newly introduced field – Rules contains conditions under which the credential holder is authorized by the credential issuer to join the swarm and receive the requested service (e.g., content quality, level of prioritized treatment). The format of this field, described with the ABNF notation [20], is given below:

Rules = [General] [Per-piece] (7)
General = conditions (8)
Per-piece = conditions (9)
conditions = condition [log-operator conditions] / "(" conditions ")" (10)
log-operator = "and" / "or" (11)
condition = variable operator value / variable operator variable (12)
operator = "=" / "!=" / "<" / "<=" / ">" / ">=" (13)
variable = 1ALPHA *99(ALPHA / DIGIT) (14)
value = 1*10DIGIT / 1*10DIGIT "." DIGIT / """ 1*10ALPHA """ (15)

The Rules field contains two groups of conditions: a general group and per-piece group. The former contains conditions evaluated every time the credential holder connects to another peer, as well as at specific time (in case of time conditions), whereas the latter contains conditions evaluated on every piece request from the credential holder. In each condition a value of a specific environment variable is compared to other variable or a predefined value. The values of the environment variables are dynamically assigned from the environment of the evaluating peer or from another field, as described later. Each group of conditions is positively evaluated only if the compound logical sentence produces a truth value.

Having on mind the roles of peers A and B, the extended and modified message exchange process goes as follows:

$$A \rightarrow B: Version_A, SwarmID, N_A \quad (16)$$
$$A \leftarrow B: Version_B, SwarmID, N_B \quad (17)$$
$$A \rightarrow B: PoA_A, ReqService_A, \{N_A, N_B, PoA_A, ReqService_A\}_{K_A^{-1}} \quad (18)$$
$$A \leftarrow B: PoA_B, Info_B, Peers_B, \{K_{AB}\}_{K_A}, \{N_A, N_B, PoA_B, Info_B, Peers_B, \{K_{AB}\}_{K_A}\}_{K_B^{-1}} \quad (19)$$
$$A \leftarrow B: Info_B, \{N_A, N_B, Info_B\}_{K_B^{-1}} \quad (20)$$

First, the peers exchange the latest version of the protocol they support (Version), together with the swarm identifier and the randomly generated nonce, with (16) and (17). Then, peer A sends its authorization credential and specifies the service properties (ReqService) it wants to receive with (18). Next, peer B evaluates the service request. If it is according to peer A's authorizations and if peer B can provide the requested service (for example it has an available connection – a free or one to a peer with lower priority that can be terminated), it will enable upload to peer A. Otherwise, upload will be disabled. In both cases, it will first send (19) in order to clarify the process outcome (Info) and to recommend other swarm members for contacting (Peers) to peer A. In positive case (19) will also contain a symmetric key (K_{AB}), generated by peer B and encrypted with peer A's public key, which will be used for encryption of the provided service – the content pieces. On the other hand, in negative case this field will be empty. After a positive (19), peer B starts to upload content to peer A. It also continues to verify the validity of the peer A's credential and to evaluate every piece request according to its authorizations. When a violation occurs, it will send (20) as notification and it will stop uploading. In addition, the protocol will be aborted when one of the peers sends an invalid credential, an incorrect digital signature or a different swarm identifier.

The positive outcome of the message exchange process is one way upload, from peer B to peer A. If peer B is also interested in downloading content from peer A while uploading, it needs to start the same exchange process, but now acting as an initiator.

The formats of the newly introduced fields are as follows. First, the Version field is two bytes and states the protocol version. For example, 02_{HEX} denotes the enhanced CS protocol version. Since this or any future protocol extension or modification results in a new version, peers need to be aware of the versions they support in order to have successful communication. In this way, means for backward compatibility between CS protocol versions can be possible. Next, the description of the ReqService field format, by using the ABNF notation, is:

ReqService = ["(" assignment ")"] ["," ReqService] (21)
assignment = variable "," value (22)
variable = 1ALPHA *99(ALPHA / DIGIT) (23)
value = 1*10DIGIT / 1*10DIGIT "." DIGIT / """ 1*10ALPHA """ (24)

It contains pairs that actually define an assignment of a certain value to a specific environment variable at the

evaluating peer. These values must be assigned to the environment variables before evaluation of the conditions in the Rules field, since they influence the evaluation of the policies from the Rules field. The ReqService field can contain information such as requested content quality or level of prioritized treatment. Furthermore, the Info field is two bytes and specifies the identifier of predefined information that clarifies the protocol outcome. This information can confirm that the upload is enabled or state the reason why it is disabled. For example 01_{HEX} means unauthorized service properties requested. Finally, the Peers field is a set of maximum 5 pairs, each denoting a swarm member. A pair contains either IP address (IPv4 or IPv6) or DNS name of the member, together with its port number.

In conclusion, the enhanced CS protocol is an access control mechanism that acts on a peer level that enables peers to exchange the authorization credential and requested service properties between them in a secure manner.

V. DISCUSSION

Together with our proposed enhancements, the CS protocol fulfills the requirements from Section III, and becomes resistant to man-in-the-middle attacks.

To begin with, the introduction of Rules and ReqService fields fully satisfies the desired requirements 2-4, as well as the sub-requirement 1.1. The Rules field provides creation of expressive and flexible access control policies. These policies are contained in the authorization credential itself which makes their modification easy and dynamic. The policies can be tailored to several groups of peers in the swarm, distinguishable on the basis of various criteria, such as role in a swarm (seed or leech), priority, location and allowed content quality (number of stream layers), during different time periods. Now, seeds can automatically join the hierarchical swarm only by receiving appropriate authorization credentials. However, every peer must first explicitly request the properties of the service it wants to receive by specifying them in the ReqService field, in order to have its policy evaluated correctly. In this way, together with the help of the notifications in the Info field, they can even negotiate (to some extent) the service properties they want to receive.

The access control diagram of a request for service is illustrated in Figure 2 (based on [21]). The initiator – peer A sends its authorization credential and specifies the service properties it wants to receive to the swarm member – peer B with message (18). The Rules field is passed to the peer B's Access Control Decision Function (ADF), where the embodied polices are evaluated. On the other hand, the values from the ReqService field are assigned to specific environment variables and together with other environment variables are taken into account during evaluation of the policies. The peer B's Access Control Enforcement Function (AEF) grants or denies the access to the requested service according to the evaluation of the specified policies.

An example of information provided to the evaluating peer's ADF, i.e. the contents of the Rules and ReqService fields sent by a seed, together with needed environment variables, applicable in the described scenario above is given in Figure 3. The Rules field denotes that the seed is authorized by the credential issuer to join the swarm only when it is located in Slovenia and requests high content quality (e.g., HD video) and prioritized treatment appropriate for level 2 seed. According to the explicitly requested service properties in the ReqService field by the seed and the values of the specific environment variables at the evaluating peer, this policy is positively evaluated at the ADF and the seed is granted access to the swarm.

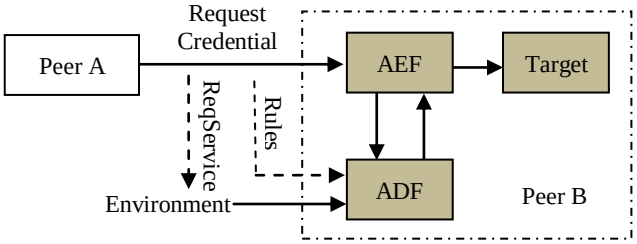


Figure 2. Access control diagram of request for service with the enhanced CS protocol (based on [21]).

Rules:
General:
GEOLOCATION = 'SI' and
PRIORITY <= 10 and
CONTENT_QUALITY <= 3
Per piece:
ReqService:
(PRIORITY, 10),
(CONTENT_QUALITY, 3)
Environment:
GEOLOCATION = 'SI'

Figure 3. Contents of the *Rules* and *ReqService* fields sent to a closed swarm member by a level 2 seed (Fig. 1) and the values of the environment variables at the swarm member.

Furthermore, the newly introduced Peers field provides a peer discovery mechanism applicable in hierarchically structured live streaming swarm, which satisfies sub-requirement 1.2 from Section III. The peer discovery mechanism goes as follows. Every peer first contacts the content injector using the CS protocol. If it is authorized to enter the swarm, it will receive by the content injector a list of swarm members from the layer with the highest priority. Then it continues to contact the returned members and to receive information about other members from the swarm, until it creates the number of connections it needs. Peers return information about members from the same layer or the layer with one level lower priority, as long as this priority is greater than or equal to the initiator's priority. This guarantees that peers will always download content only from peers with the same or higher priority in the structure.

In addition, the Version field provides means for backward compatibility. After two peers exchange the

 ISBN: 978-1-61208-010-9

protocol version they support, the peer supporting the higher version can adapt and send appropriate messages to the version the other peer supports. However, this is applicable in specific cases and only to those peers that are not directly concerned with the higher version changes. For example, if the original CS protocol did contain the Version field, the basic clients from the described scenario could use the original protocol, but only when the requirement for restriction of the content delivery based on peer location is not mandatory.

Finally, by encrypting the exchanged content with a secret key we prevent a malicious peer to read it in an unauthorized manner. However, the purpose of the encryption is not to provide confidentiality of the provided service, but only to fight man-in-the-middle attacks. Also, it does not prevent explicit leakage of content to unauthorized peers, which still depends on the good behavior of the authorized peers.

In conclusion, all the desired requirements from Section III can be satisfied with our proposed enhancements, and means for backward compatibility can be achieved. Also, the vulnerability of the protocol to man-in-the-middle attack is fixed.

VI. CONCLUSION AND FUTURE WORK

In this paper we have proposed several enhancements of an existing access control mechanism for BitTorrent P2P networks – the Closed Swarms protocol. The enhancements provide additional flexibility in access control mechanism, enabling fine grained security policies specification and enforcement. The enhancements fulfill a number of content providers' requirements and promise efficient, flexible and secure content delivery in future content delivery scenarios. Our future work includes integration of the proposed enhancements into the P2P-Next delivery platform (http://p2p-next.org) and their evaluation.

REFERENCES

[1] Future Media Internet Task Force: Research on Future Media Internet. A white paper, (2009). Obtained on April 5, 2011 from: ftp://ftp.cordis.europa.eu/pub/fp7/ict/docs/netmedia/research-on-future-media-internet-2009-4072_en.pdf

[2] Schulze, H., Mochalski, K.: Internet Study 2008/2009 (2009). Obtained on April 5, 2011 from: http://www.ipoque.com/userfiles/file/ipoque-Internet-Study-08-09.pdf

[3] Cohen, B.: BitTorrent protocol specification, (2008). Obtained on April 5, 2011 from: http://www.bittorrent.org/beps/bep_0003.html

[4] Pouwelse, J. A., Garbacki, P., Wang, J., Bakker, A., Yang, J., Iosup, A., Epema, D. H. J., Reinders, M., van Steen M. R., Sips H. J.: Tribler: A social-based based peer to peer system. 5th Int'l Workshop on Peer-to-Peer Systems (IPTPS), Santa Barbara, USA (2006)

[5] Pandey, R.R., Patil KK.: Study of BitTorrent based Video on Demand Systems. International Journal of Computer Applications, Vol.1, No.11, pp. 29-33. Foundation of Computer Science, (2010)

[6] Mol, J.J.D., Bakker, A., Pouwelse, J.A., Epema, D.H.J., Sips, H.J.: The Design and Deployment of a BitTorrent Live Video Streaming Solution. In: 11th IEEE International Symposium on Multimedia, ISM '09, pp. 342-349. IEEE Computer Society, Los Alamitos (2009)

[7] Loewenstern, A.: DHT Protocol, (2008). Obtained on April 5, 2011 from: http://bittorrent.org/beps/bep_0005.html

[8] Cohen, B.: Incentives Build Robustness in BitTorrent, In: Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems, Berkeley, USA, (2003)

[9] International Telecommunication Union – Telecommunication Standardization Sector (ITU-T): Data networks, open system communications and security, Framework for secure peer-to-peer communications. ITU-T Recommendation X.1161, (2008)

[10] Gheorghe, G., Lo Cigno, R., Montresor, A.: Security and privacy issues in P2P streaming systems: A survey. Peer-to-Peer Networking and Applications. Springer, New York (2010)

[11] Borch N.T., Arntzen, I., Mitchell K., Gabrijelčič D.: Access control to BitTorrent swarms using Closed Swarms. In: Proceedings of the 2010 ACM Workshop on Advanced Video Streaming Techniques for Peer-to-Peer Networks and Social Networking, AVSTP2P '10, pp. 25-30. ACM, New York (2010)

[12] Zhang, X., Liu, D., Chen, S., Zhang, Z., Sandhu, R.: Towards digital rights protection in BitTorrent-like P2P systems. In: Proceedings of the 15th ACM/SPIE Multimedia Computing and Networking, San Jose, USA (2008)

[13] Jimenez, R., Eriksson, L, Knutsson, B.: P2P-Next: Technical and Legal Challenges. The Sixth Swedish National Computer Networking Workshop and Ninth Scandinavian Workshop on Wireless Adhoc Networks, Uppsala, Sweden (2009)

[14] Harrison, D.: Private Torrents, (2008). Obtained on April 5, 2011 from: http://bittorrent.org/beps/bep_0027.html

[15] Tootoonchian, A., Saroiu, S., Ganjali, Y., Wolman, A.: Lockr: better privacy for social networks. In: Proceedings of the 5th international conference on Emerging networking experiments and technologies, CoNEXT '09, pp. 169-180. ACM, New York (2009)

[16] Andreev, K., Maggs, B.M., Meyerson, A., Sitaraman, R.K.: Designing overlay multicast networks for streaming, In: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures, SPAA '03, pp. 149-158. ACM, New York (2003)

[17] Muir, J.A., van Oorschot, P.C.: Internet geolocation and evasion. Technical Report TR-06-05, Carleton University, School of Computer Science (2006)

[18] Asioli, S., Ramzan, N., Izquierdo, E.: Efficient Scalable Video Streaming over P2P Network. User Centric Media, LNICST, vol. 40, pp. 153-160. Springer, Heidelberg (2010)

[19] Chapin, P., Skalka, C., Wang, X.S.: Authorization in trust management: Features and foundations. ACM Computing Surveys, Vol. 40, pp.1-48. ACM, New York (2008)

[20] Crocker, D., Overell, P.: Augmented BNF for Syntax Specifications: ABNF. Request for Comments (RFC) 5234, (2008)

[21] International Telecommunication Union – Telecommunication Standardization Sector (ITU-T): Data networks, open system communications and security, Information technology – Open systems interconnection – Security frameworks for open systems: Access control framework. ITU-T Recommendation X.812, (1995)

 ISBN: 978-1-61208-010-9

Chapter 10
Appendix D: Logs from a Simulation of Creation of a Hierarchically Structured Swarm

A process of creation of a hierarchically structured swarm with 7 peers was simulated. One of the peers was the content injector (Seeder), whereas the other peers were considered an auxiliary seeders (Peer 1-6). The structure was designed to have 2 layers of seeders: the upper with 2 peers and the lower with 4 peers. Appropriate credentials were issued to the peers, specifying their allowed priority. The content delivery process was run for 50 seconds. The process output, presenting the communication between peers, is given below. The ratios in the brackets show the ratio of the current and maximum number of upload ECS connections.

```
Peer 1 initiates DL_ECS to Seeder.
Seeder receives UL_ECS initiation from Peer 1. Number of ECS connections 0/2.
Seeder completes UL_ECS to Peer 1. Info: Valid request, suggested peers: [].
Peer 3 initiates DL_ECS to Seeder.
Peer 3 initiates DL_ECS to Peer 1.
Peer 1 receives UL_ECS initiation from Peer 3. Number of ECS connections 0/2.
Seeder receives UL_ECS initiation from Peer 3. Number of ECS connections 1/2.
Peer 1 completes UL_ECS to Peer 3. Info: Valid request, suggested peers: [Seeder].
Seeder completes UL_ECS to Peer 3. Info: Valid request, suggested peers: [Peer 1].
Peer 2 initiates DL_ECS to Seeder.
Peer 2 initiates DL_ECS to Peer 1.
Peer 2 initiates DL_ECS to Peer 3.
Seeder receives UL_ECS initiation from Peer 2. Number of ECS connections 2/2.
Peer 3 receives UL_ECS initiation from Peer 2. Number of ECS connections 0/3.
Peer 1 receives UL_ECS initiation from Peer 2. Number of ECS connections 1/2.
Peer 1 completes UL_ECS to Peer 2. Info: Valid request, suggested peers: [Seeder, Peer 3].
Peer 3 completes UL_ECS to Peer 2. Info: Valid request, suggested peers: [Seeder, Peer 1].
Seeder terminates UL_ECS to Peer 3. Reason: No longer valid request or terminated in favour to
other peer.
Seeder terminates connection to Peer 3.
Seeder completes UL_ECS to Peer 2. Info: Valid request, suggested peers: [Peer 1, Peer 3].
Peer 3 terminates connection to Seeder.
Peer 3 initiates DL_ECS to Peer 2. This is 2nd ECS for these peers.
Peer 3 terminates connection to Peer 2.
Peer 4 initiates DL_ECS to Peer 1.
Peer 4 initiates DL_ECS to Peer 2.
Peer 4 initiates DL_ECS to Peer 3.
Peer 4 initiates DL_ECS to Seeder.
Peer 1 receives UL_ECS initiation from Peer 4. Number of ECS connections 2/2.
Seeder receives UL_ECS initiation from Peer 4. Number of ECS connections 2/2.
Peer 3 receives UL_ECS initiation from Peer 4. Number of ECS connections 0/3.
Peer 2 receives UL_ECS initiation from Peer 4. Number of ECS connections 0/2.
Peer 1 completes UL_ECS to Peer 4. Info: Apologies for not having available connection,
suggested peers: [Seeder, Peer 2, Peer 3].
Peer 1 terminates connection to Peer 4.
Seeder completes UL_ECS to Peer 4. Info: Apologies for not having available connection,
suggested peers: [Peer 2, Peer 1].
Seeder terminates connection to Peer 4.
Peer 4 terminates connection to Seeder.
Peer 4 terminates connection to Peer 1.
Peer 2 completes UL_ECS to Peer 4. Info: Valid request, suggested peers: [Seeder, Peer 1].
Peer 5 initiates DL_ECS to Peer 2.
Peer 5 initiates DL_ECS to Seeder.
Peer 5 initiates DL_ECS to Peer 4.
Peer 5 initiates DL_ECS to Peer 3.
Peer 5 initiates DL_ECS to Peer 1.
Peer 2 receives UL_ECS initiation from Peer 5. Number of ECS connections 1/2.
```

```
Peer 4 receives UL_ECS initiation from Peer 5. Number of ECS connections 0/3.
Seeder receives UL_ECS initiation from Peer 5. Number of ECS connections 2/2.
Peer 3 receives UL_ECS initiation from Peer 5. Number of ECS connections 1/3.
Peer 1 receives UL_ECS initiation from Peer 5. Number of ECS connections 2/2.
Seeder completes UL_ECS to Peer 5. Info: Apologies for not having available connection,
suggested peers: [Peer 2, Peer 1].
Seeder terminates connection to Peer 5.
Peer 2 completes UL_ECS to Peer 5. Info: Valid request, suggested peers: [Seeder, Peer 1, Peer
4].
Peer 4 completes UL_ECS to Peer 5. Info: Valid request, suggested peers: [Peer 2, Peer 3].
Peer 1 completes UL_ECS to Peer 5. Info: Apologies for not having available connection,
suggested peers: [Seeder, Peer 2, Peer 3].
Peer 1 terminates connection to Peer 5.
Peer 5 terminates connection to Peer 1.
Peer 3 completes UL_ECS to Peer 5. Info: Valid request, suggested peers: [Peer 1, Peer 4].
Peer 5 terminates connection to Seeder.
Peer 6 initiates DL_ECS to Peer 4.
Peer 6 initiates DL_ECS to Peer 5.
Peer 6 initiates DL_ECS to Peer 3.
Peer 6 initiates DL_ECS to Peer 2.
Peer 6 initiates DL_ECS to Seeder.
Peer 6 initiates DL_ECS to Peer 1.
Peer 5 receives UL_ECS initiation from Peer 6. Number of ECS connections 0/3.
Peer 4 receives UL_ECS initiation from Peer 6. Number of ECS connections 1/3.
Peer 2 receives UL_ECS initiation from Peer 6. Number of ECS connections 2/2.
Seeder receives UL_ECS initiation from Peer 6. Number of ECS connections 2/2.
Peer 3 receives UL_ECS initiation from Peer 6. Number of ECS connections 2/3.
Peer 1 receives UL_ECS initiation from Peer 6. Number of ECS connections 2/2.
Peer 2 completes UL_ECS to Peer 6. Info: Apologies for not having available connection,
suggested peers: [Seeder, Peer 1, Peer 5, Peer 4].
Peer 2 terminates connection to Peer 6.
Seeder completes UL_ECS to Peer 6. Info: Apologies for not having available connection,
suggested peers: [Peer 2, Peer 1].
Seeder terminates connection to Peer 6.
Peer 5 completes UL_ECS to Peer 6. Info: Valid request, suggested peers: [Peer 2, Peer 3, Peer
4].
Peer 1 completes UL_ECS to Peer 6. Info: Apologies for not having available connection,
suggested peers: [Seeder, Peer 2, Peer 3].
Peer 1 terminates connection to Peer 6.
Peer 6 terminates connection to Peer 1.
Peer 6 terminates connection to Peer 2.
Peer 4 completes UL_ECS to Peer 6. Info: Valid request, suggested peers: [Peer 2, Peer 3, Peer
5].
Peer 3 completes UL_ECS to Peer 6. Info: Valid request, suggested peers: [Peer 1, Peer 5, Peer
4].
Peer 6 terminates connection to Seeder.
Peer 2 terminates connection to Peer 1.
Seeder terminates connection to Peer 1.
Peer 1 terminates connection to Seeder.
Peer 3 terminates connection to Peer 1.
Peer 1 terminates connection to Peer 2.
Peer 1 terminates connection to Peer 3.
Peer 4 terminates connection to Peer 3.
Peer 6 terminates connection to Peer 3.
Peer 5 terminates connection to Peer 3.
Peer 3 terminates connection to Peer 4.
Peer 3 terminates connection to Peer 5.
Peer 3 terminates connection to Peer 6.
Seeder terminates connection to Peer 2.
Peer 5 terminates connection to Peer 2.
Peer 2 terminates connection to Peer 5.
Peer 2 terminates connection to Seeder.
Peer 2 terminates connection to Peer 4.
Peer 5 terminates connection to Peer 4.
Peer 4 terminates connection to Peer 5.
```

Peer 6 terminates connection to Peer 4.