PPSP
Internet-Draft
Intended status: Standards Track
Expires: December 19, 2014

A. Bakker
Vrije Universiteit Amsterdam
R. Petrocco
V. Grishchenko
Technische Universiteit Delft

June 17, 2014

Peer-to-Peer Streaming Peer Protocol (PPSPP)
draft-ietf-ppsp-peer-protocol-10

Abstract

The Peer-to-Peer Streaming Peer Protocol (PPSPP) is a protocol for disseminating the same content to a group of interested parties in a streaming fashion. PPSPP supports streaming of both pre-recorded (on-demand) and live audio/video content. It is based on the peer-to-peer paradigm, where clients consuming the content are put on equal footing with the servers initially providing the content, to create a system where everyone can potentially provide upload bandwidth. It has been designed to provide short time-till-playback for the end user, and to prevent disruption of the streams by malicious peers. PPSPP has also been designed to be flexible and extensible. It can use different mechanisms to optimize peer uploading, prevent freeriding, and work with different peer discovery schemes (centralized trackers or Distributed Hash Tables). It supports multiple methods for content integrity protection and chunk addressing. Designed as a generic protocol that can run on top of various transport protocols, it currently runs on top of UDP using LEDBAT for congestion control.

Status of this Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

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."

This Internet-Draft will expire on December 19, 2014.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 6
1.1. Purpose 6
1.2. Requirements Language 7
1.3. Terminology 7
2. Overall Operation 9
2.1. Example: Joining a Swarm 9
2.2. Example: Exchanging Chunks 10
2.3. Example: Leaving a Swarm 11
3. Messages 11
3.1. HANDSHAKE 11
3.2. HAVE 12
3.3. DATA 12
3.4. ACK 12
3.5. INTEGRITY 13
3.6. SIGNED_INTEGRITY 13
3.7. REQUEST 13
3.8. CANCEL 14
3.9. CHOKE and UNCHOKE 14
3.10. Peer Address Exchange 14
3.10.1. PEX_REQ and PEX_RES Messages 14
3.11. Channels 16
3.12. Keep Alive Signalling 17
4. Chunk Addressing Schemes 17
4.1. Start-End Ranges 18
4.1.1. Chunk Ranges 18
4.1.2. Byte Ranges 18
4.2. Bin Numbers 18
4.3. In Messages 20
4.3.1. In HAVE Messages 20
4.3.2. In ACK Messages 20

5. Content Integrity Protection 21
5.1. Merkle Hash Tree Scheme 21
5.2. Content Integrity Verification 22
5.3. The Atomic Datagram Principle 23
5.4. INTEGRITY Messages 24
5.5. Discussion and Overhead 24
5.6. Automatic Detection of Content Size 25
5.6.1. Peak Hashes 26
5.6.2. Procedure 28
6. Live Streaming 28
6.1. Content Authentication 28
6.1.1. Sign All 29
6.1.2. Unified Merkle Tree 30
6.1.2.1. Signed Munro Hashes 30
6.1.2.2. Munro Signature Calculation 33
6.1.2.3. Procedure 33
6.1.2.4. Secure Tune In 33
6.2. Forgetting Chunks 34
7. Protocol Options 35
7.1. End Option 35
7.2. Version 35
7.3. Minimum Version 36
7.4. Swarm Identifier 36
7.5. Content Integrity Protection Method 37
7.6. Merkle Tree Hash Function 37
7.7. Live Signature Algorithm 38
7.8. Chunk Addressing Method 38
7.9. Live Discard Window 39
7.10. Supported Messages 40
8. UDP Encapsulation 40
8.1. Chunk Size 40
8.2. Datagrams and Messages 41
8.3. Channels 42
8.4. HANDSHAKE 43
8.5. HAVE 44
8.6. DATA 45
8.7. ACK 45
8.8. INTEGRITY 46
8.9. SIGNED_INTEGRITY 47
8.10. REQUEST 48
8.11. CANCEL 48
8.12. CHOKE and UNCHOKE 49
8.13. PEX_REQ, PEX_RESv4, PEX_RESv6 and PEX_REScert 49
8.14. KEEPALIVE 51
8.15. Detecting a Dead Peer 51
8.16. Flow and Congestion Control 52
8.17. Example of Operation 52
9. Extensibility 57

9.1. Chunk Picking Algorithms 57
9.2. Reciprocity Algorithms 57
10. Acknowledgements 57
11. IANA Considerations 58
11.1. PPSP Peer Protocol Message Type Registry 58
11.2. PPSP Peer Protocol Option Registry 58
11.3. PPSP Peer Protocol Version Number Registry 58
11.4. PPSP Peer Protocol Content Integrity Protection Method Registry 58
11.5. PPSP Peer Protocol Merkle Hash Tree Function Registry . . 58
11.6. PPSP Peer Protocol Chunk Addressing Method Registry . . . 58
12. Manageability Considerations 58
12.1. Operations 59
12.1.1. Installation and Initial Setup 59
12.1.2. Requirements on Other Protocols and Functional Components 60
12.1.3. Migration Path 60
12.1.4. Impact on Network Operation 60
12.1.5. Verifying Correct Operation 60
12.1.6. Configuration 61
12.2. Management Considerations 61
12.2.1. Management Interoperability and Information 62
12.2.2. Fault Management 62
12.2.3. Configuration Management 62
12.2.4. Accounting Management 63
12.2.5. Performance Management 63
12.2.6. Security Management 63
13. Security Considerations 63
13.1. Security of the Handshake Procedure 63
13.1.1. Protection Against Attack 1 64
13.1.2. Protection Against Attack 2 65
13.1.3. Protection Against Attack 3 65
13.2. Secure Peer Address Exchange 66
13.2.1. Protection against the Amplification Attack 66
13.2.2. Example: Tracker as Certification Authority 67
13.2.3. Protection Against Eclipse Attacks 68
13.3. Support for Closed Swarms (PPSP.SEC.REQ-1) 68
13.4. Confidentiality of Streamed Content (PPSP.SEC.REQ-2+3) . . 68
13.5. Strength of the Hash Function for Merkle Hash Trees . . . 69
13.6. Limit Potential Damage and Resource Exhaustion by Bad or Broken Peers (PPSP.SEC.REQ-4+6) 69
13.6.1. HANDSHAKE 69
13.6.2. HAVE 69
13.6.3. DATA 70
13.6.4. ACK 70
13.6.5. INTEGRITY and SIGNED_INTEGRITY 70
13.6.6. REQUEST 71
13.6.7. CANCEL 71

13.6.8. CHOKE 71
13.6.9. UNCHOKE 71
13.6.10. PEX_RES 72
13.6.11. Unsolicited Messages in General 72
13.7. Exclude Bad or Broken Peers (PPSP.SEC.REQ-5) 72
14. References 72
14.1. Normative References 72
14.2. Informative References 73
Appendix A. Revision History 77
Authors' Addresses 95

1. Introduction

1.1. Purpose

This document describes the Peer-to-Peer Streaming Peer Protocol (PPSPP), designed for disseminating the same content to a group of interested parties in a streaming fashion. PPSPP supports streaming of both pre-recorded (on-demand) and live audio/video content. It is based on the peer-to-peer paradigm where clients consuming the content are put on equal footing with the servers initially providing the content, to create a system where everyone can potentially provide upload bandwidth.

PPSPP has been designed to provide short time-till-playback for the end user, and to prevent disruption of the streams by malicious peers. Central in this design is a simple method of identifying content based on self-certification. In particular, content in PPSPP is identified by a single cryptographic hash that is the root hash in a Merkle hash tree calculated recursively from the content [MERKLE][ABMRKL]. This self-certifying hash tree allows every peer to directly detect when a malicious peer tries to distribute fake content. The tree can be used for both static and live content. Moreover, it ensures only a small amount of information is needed to start a download and to verify incoming chunks of content, thus ensuring short start-up times.

PPSPP has also been designed to be extensible for different transports and use cases. Hence, PPSPP is a generic protocol which can run directly on top of UDP, TCP, or other protocols. As such, PPSPP 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 allows, PPSPP can also use different congestion control algorithms.

At present, PPSPP is set to run on top of UDP using LEDBAT for congestion control [RFC6817]. Using LEDBAT enables PPSPP to serve the content after playback (seeding) without disrupting the user who may have moved to different tasks that use its network connection.

PPSPP is also flexible and 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. It also allows different schemes for chunk addressing and content integrity protection, if the defaults are not fit for a particular use case. In addition, it can work with different peer discovery schemes, such as centralized trackers or fast Distributed Hash Tables [JIM11]. Finally, in this default setup, PPSPP maintains only a small amount of state per peer. A reference implementation of

PPSPP over UDP is available [SWIFTIMPL].

1.2. Requirements Language

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 RFC 2119 [RFC2119].

1.3. Terminology

message
: The basic unit of PPSPP 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 PPSPP's Protocol Data Unit (PDU).

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

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

chunk ID
: Unique identifier for a chunk of content (e.g. an integer). Its type depends on the chunk addressing scheme used.

chunk specification
: An expression that denotes one or more chunk IDs.

chunk addressing scheme
: Scheme for identifying chunks and expressing the chunk availability map of a peer in a compact fashion.

chunk availability map
: The set of chunks a peer has successfully downloaded and checked the integrity of.

bin
: A number denoting a specific binary interval of the content (i.e., one or more consecutive chunks) in the bin numbers chunk addressing scheme (see Section 4).

content integrity protection scheme
: Scheme for protecting the integrity of the content while it is being distributed via the peer-to-peer network. I.e. methods for receiving peers to detect whether a requested chunk has been maliciously modified by the sending peer.

hash
: The result of applying a cryptographic hash function, more specifically a modification detection code (MDC) [HAC01], such as SHA-1 [FIPS180-4], to a piece of data.

Merkle hash tree
: A tree of hashes whose base is formed by the hashes of the chunks of content, and its higher nodes are calculated by recursively computing the hash of the concatenation of the two child hashes (see Section 5.1).

root hash
: The root in a Merkle hash tree calculated recursively from the content (see Section 5.1).

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

swarm ID
: Unique identifier for a swarm of peers, in PPSPP a sequence of bytes. When Merkle hash trees are used for content integrity protection, the identifier is the so-called root hash of the content (video-on-demand). For live streaming, the swarm ID is a public key.

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.

seeding
: Peer A is said to be seeding when A has downloaded a static content asset completely and is now offering it for others to download.

leeching
: Peer A is said to be leeching when A has not completely downloaded a static content asset yet or is not offering to upload it to others.

channel
: A logical connection between two peers. The channel concept allows peers to use the same transport address for communicating with different peers.

channel ID
: Unique, randomly chosen identifier for a channel, local to each peer. So the two peers logically connected by a channel each have a different channel ID for the channel.

In this document the prefixes kilo, mega, etc. denote base 1024.

2. Overall Operation

The basic unit of communication in PPSPP 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 Section 8).

The overall operation of PPSPP is illustrated in the following examples. The examples assume that UDP is used for transport, the Merkle Hash Tree scheme is used for content integrity protection, and that a specific policy is used for selecting which chunks to download.

2.1. Example: Joining a Swarm

Consider a user who wants to watch a video. To play the video, the user clicks on the play button of a HTML5 <video> element shown in his PPSPP-enabled browser. Imagine this element has a PPSPP URL (to be defined elsewhere) identifying the video as it source. The browser passes this URL to its PPSP protocol handler. Let's call this protocol handler peer A. Peer A parses the URL to retrieve the transport address of a PPSP tracker and swarm ID of the content. The tracker address may be optional in the presence of a decentralized tracking mechanism.

Peer A now registers with the tracker following the PPSP tracker protocol [I-D.ietf-ppsp-base-tracker-protocol] 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 conveys protocol options, in particular, peer A includes the ID of the swarm as the destination peers can listen for multiple swarms on the same transport address.

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 chunk specification that denotes what chunks of the content peer B, resp. C have. Peer D sends a datagram with a HANDSHAKE and HAVE messages, but also with a CHOKE message. The latter indicates that D is not willing to upload chunks to A at present.

2.2. Example: Exchanging Chunks

In response to B and C, A sends new datagrams to B and C containing REQUEST messages. A REQUEST message indicates the chunks that a peer wants to download, and thus contains a chunk specification. The REQUEST messages to B and C refer to disjunct sets of chunks. B and C respond with datagrams containing HAVE, DATA and, in this example, INTEGRITY messages. In the Merkle hash tree content protection scheme (see Section 5.1), the INTEGRITY messages contain all cryptographic hashes that peer A needs to verify the integrity of the content chunk sent in the DATA message. 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. In addition, it passes the chunks of video to the user's browser for rendering.

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 REQUEST 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 REQUEST for that chunk.

Peer D also sends HAVE messages to A when it downloads chunks from other peers. When D is willing to accept REQUESTs from A, D sends a datagram with an UNCHOKE message to inform A. If B or C decide to choke A they sending a CHOKE message and A should then re-request from other peers. B and C may continue to send HAVE, REQUEST, 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 can also contact the tracker or another source again to obtain more peer addresses.

2.3. Example: Leaving a Swarm

To leave a swarm in a graceful way, peer A sends a specific HANDSHAKE message to all its peers (see Section 8.4) and deregisters from the tracker following the (PPSP) tracker protocol. Peers receiving the datagram should remove A from their current peer list. If A crashes ungracefully, peers should remove A from their peer list when they detect it no longer sends messages (see Section 8.15).

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, and further communication with the peer SHOULD be stopped. The rationale is that it is sufficient to classify peers as either good (i.e., responding with chunks) or bad and only use the good ones. This behavior allows a peer to deal with slow, crashed and (silent) malicious peers.

Multiple messages are multiplexed into a single datagram for transmission. Messages in a single datagram MUST be processed in the strict order in which they appear in the datagram.

For the sake of simplicity, one swarm of peers deals with one content asset (e.g. file) only. Retrieval of a collections of files can be done either by using multiple swarms or by using an external storage mapping from the linear byte space of a single swarm to different files, transparent to the protocol.

3.1. HANDSHAKE

The initiating peer and the addressed peer MUST send a HANDSHAKE message as the first message in the first datagrams they exchange. The payload of the HANDSHAKE message is a channel ID (see Section 3.11) and a sequence of protocol options. Example options are the content integrity protection scheme used and an option to specify the swarm identifier. The complete set of protocol options are specified in Section 7.

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, e.g. DATA messages. To minimize the number of initialization round-trips, 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 REQUEST (see Section 3.7), but MUST NOT include any DATA message.

3.2. HAVE

The HAVE message is used to convey which chunks a peer has available for download. The set of chunks it has available may be expressed using different chunk addressing and availability map compression schemes, described in Section 4. HAVE messages can be used both for sending a complete overview of a peer's chunk availability as well as for updates to that set.

In particular, whenever a receiving peer P has successfully checked the integrity of a chunk, or interval of chunks, it SHOULD send a HAVE message to all peers Q1..Qn it wants to interact with in the near future. A policy in peer P determines when the HAVE is sent. P may sent it directly, or peer P may wait until either it has other data to sent to Qi, or until it has received and checked multiple chunks. The policy will depend on how urgent it is to distribute this information to the other peers. This urgency is generally determined in turn by the chunk picking policy (see Section 9.1). In general, the HAVE messages can be piggybacked onto other messages. Peers that do not receive HAVE messages are effectively prevented from downloading the newly available chunks, hence the HAVE message can be used as a method of choking.

The HAVE message MUST contain the chunk specification of the received and verified chunks. A receiving peer MUST NOT send a HAVE message to peers for which the handshake procedure is still incomplete, see Section 13.1. A peer SHOULD NOT send a HAVE message to peers that have the complete content already (e.g. in video-on-demand scenarios).

3.3. DATA

The DATA message is used to transfer chunks of content. The DATA message MUST contain the chunk ID of the chunk and chunk itself. A peer MAY send the DATA messages for multiple chunks in the same datagram. The DATA message MAY contain additional information if needed by the specific congestion control mechanism used. At present PPSPP uses LEDBAT [RFC6817] for congestion control, which requires the current system time to be sent along with the DATA message, so the current system time MUST be included.

3.4. ACK

ACK messages MUST be sent to acknowledge received chunks if PPSPP is run over an unreliable transport protocol. ACK messages MAY be sent if a reliable transport protocol is used. In the former case, a receiving peer that has successfully checked the integrity of a chunk, or interval of chunks C MUST send an ACK message containing a

chunk specification for C. As LEDBAT is used, an ACK message MUST contain the one-way delay, computed from the peer's current system time received in the DATA message. A peer MAY delay sending ACK messages as defined in the LEDBAT specification.

3.5. INTEGRITY

The INTEGRITY message carries information required by the receiver to verify the integrity of a chunk. Its payload depends on the content integrity protection scheme used. When the Merkle Hash Tree scheme is used, an INTEGRITY message MUST contain a cryptographic hash of a subtree of the Merkle hash tree and the chunk specification that identifies the subtree.

As a typical example, when a peer wants to send a chunk and Merkle hash trees are used, it creates a datagram that consists of several INTEGRITY messages containing the hashes the receiver needs to verify the chunk and the actual chunk itself encoded in a DATA message. What are the necessary hashes and the exact rules for encoding them into datagrams is specified in Section 5.3, and Section 5.4, respectively.

3.6. SIGNED_INTEGRITY

The SIGNED_INTEGRITY message carries digitally signed information required by the receiver to verify the integrity of a chunk in live streaming. It logically contains a chunk specification, a timestamp and a digital signature. Its exact payload depends on the live content integrity protection scheme used, see Section 6.1.

3.7. REQUEST

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. PPSPP supports both models of operation.

The REQUEST message is used to request one or more chunks from another peer. A REQUEST message MUST contain the specification of the chunks the requester wants to download. A peer receiving a REQUEST message MAY send out the requested chunks (by means of DATA messages). When peer Q receives multiple REQUESTs from the same peer P, peer Q SHOULD process the REQUESTs in the order received. Multiple REQUEST messages MAY be sent in one datagram, for example, when a peer wants to request several rare chunks at once.

When live streaming via a push model, a peer receiving REQUESTs 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 REQUEST messages is to provide hints and coordinate peers to avoid unnecessary data retransmission.

3.8. CANCEL

When downloading on demand or live streaming content, a peer can request urgent data from multiple peers to increase the probability of it being delivered on time. In particular, when the specific chunk picking algorithm (see Section 9.1), detects that a request for urgent data might not be served on time, a request for the same data can be sent to a different peer. When a peer P decides to request urgent data from a peer Q, peer P SHOULD send a CANCEL message to all the peers to which the data has been previously requested. The CANCEL message contains the specification of the chunks P no longer wants to request. In addition, when peer Q receives a HAVE message for the urgent data from peer P, peer Q MUST also cancel the previous REQUEST(s) from P. In other words, the HAVE message acts as an implicit CANCEL.

3.9. CHOKE and UNCHOKE

Peer A can send a CHOKE message to peer B to signal it will no longer be responding to REQUEST messages from B, for example, because A's upload capacity is exhausted. Peer A MAY send a subsequent UNCHOKE message to signal that it will respond to new REQUESTs from B again (A SHOULD discard old requests). When peer B receives a CHOKE message from A it MUST NOT send new REQUEST messages and it cannot expect answers to any outstanding ones, as the transfer of chunks is choked. The CHOKE and UNCHOKE messages are informational as responding to REQUESTs is OPTIONAL, see Section 3.7.

3.10. Peer Address Exchange

3.10.1. PEX_REQ and PEX_RES Messages

Peer address exchange messages (or PEX messages for short) are common in many peer-to-peer protocols. They allow peers to exchange the transport addresses of the peers they are currently interacting with, thereby reducing the need to contact a central tracker (or DHT) to discovery new peers. The strength of this mechanism is therefore that it enables decentralized peer discovery: after an initial bootstrap no central tracker is needed anymore. Its weakness is that it enables a number of attacks, so it should not be used outside a benign environment unless extra security measures are in place.

PPSPP supports peer-address exchange in benign and potentially

hostile environments, as an OPTIONAL feature (not mandatory to implement). The general mechanism works as follows. To obtain some peer addresses a peer A MAY send a PEX_REQ message to peer B. Peer B MAY respond with one or more PEX_RES messages. PPSPP supports three types of PEX_RES reply messages, each containing the address of a single peer Ci. The address in the PEX_RES message MUST be of a peer B has exchanged messages with in the last 60 seconds to guarantee liveliness. Upon receipt, peer A may contact any or none of the returned peers Ci. Alternatively, peers MAY ignore PEX_REQ and PEX_RES 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.

As indicated, there are three types of PEX_RES messages: PEX_RESv4 containing a single IPv4 address and port, PEX_RESv6 containing a single IPv6 address and port, and a PEX_REScert message. The PEX_RESv4 and PEX_RESv6 MUST only be used in a benign environment, as they provide no guarantees that the host addressed actually participates in a PPSPP swarm.

To use PEX in PPSPP in a potentially hostile environment, such as the Internet, three conditions must be met:

1. Peer transport addresses must be relatively stable.

2. PEX_REScert messages must be used instead of PEX_RESv4 and PEX_RESv6.

3. A peer must not obtain all its peer addresses through PEX.

The full security analysis for PEX messages can be found in Section 13.2. A PEX_REScert message carries a swarm-membership certificate rather than an IP address and port. A membership certificate for peer C states that peer C at address (ipC,portC) is part of swarm S at time T and is cryptographically signed by an issuer. The receiver A can check the certificate for a valid signature by a trusted issuer, the right swarm and liveliness and only then consider contacting C. These swarm-membership certificates correspond to signed node descriptors in secure decentralized peer sampling services [SPS].

Several designs are possible for the security environment for these membership certificates. That is, there are different designs possible for who signs the membership certificates and how public keys are distributed. Section 13.2.2 describes an example where a central tracker acts as the Certification Authority.

In a potentially hostile environment, peers must also ensure that

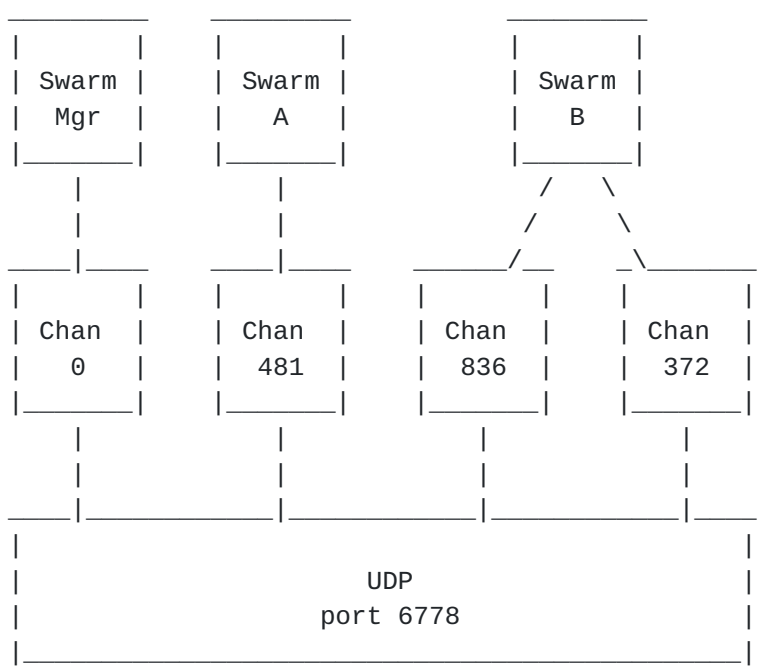
they do not end up interacting only with malicious peers when using the peer-address exchange feature. To this extent, peers MUST ensure that part of their connections are to peers whose addresses came from a trusted and secured tracker (see Section 13.2.3).

Once a PPSPP implementation has obtained a list of peers (either via PEX, from a central tracker or via a DHT), it has to determine which peers to actually contact. In this process, a PPSPP implementation can benefit from information by network or content providers to help improve network usage and boost PPSPP performance. How a P2P system like PPSPP can perform these optimizations using the ALTO protocol is described in detail in [I-D.ietf-alto-protocol], Section 7.

3.11. Channels

It is increasingly complex for peers to enable communication between each other due to NATs and firewalls. Therefore, PPSPP uses a multiplexing scheme, called channels, to allow multiple swarms to use the same transport address. Channels loosely correspond to TCP connections and each channel belongs to a single swarm, as illustrated in Figure 1. As with TCP connections, a channel is identified by a unique identifier local to the peer at each end of the connection (cf. TCP port), which MUST be randomly chosen. In other words, the two peers connected by a channel use different IDs to denote the same channel. The IDs are different and random for security reasons, see Section 13.1.

In the PPSP-over-UDP encapsulation (Section 8.3), when a channel C has been established between peer A and peer B, the datagrams containing messages from A to B are prefixed with the four byte channel ID allocated by peer B, and vice versa for datagrams from B to A. The channel IDs used are exchanged as part of the handshake procedure, see Section 8.4. In that procedure, the channel ID with value 0 is used for the datagram that initiates the handshake. PPSPP can be used in combination with STUN [RFC5389].



Network stack of a PPSPP peer that is reachable on UDP port 6778 and is connected via channel 481 to one peer in swarm A and two peers in swarm B via channels 836 and 372, respectively. Channel ID 0 is special and is used for handshaking.

Figure 1

3.12. Keep Alive Signalling

A peer SHOULD send a "keep alive" message periodically to each peer it wants to interact with in the future, but has no other messages to send them at present. Periodically sending "keep alive" messages prevents other peers from closing the connection after a predefined time interval of 3 minutes, as described in Section 8.15. PPSPP does not define an explicit message type for "keep alive" messages. In the PPSP-over-UDP encapsulation they are implemented as simple datagrams consisting of a 4-byte channel ID only, see Section 8.3 and Section 8.4.

4. Chunk Addressing Schemes

PPSPP can use different methods of chunk addressing, that is, support different ways of identifying chunks and different ways of expressing the chunk availability map of a peer in a compact fashion.

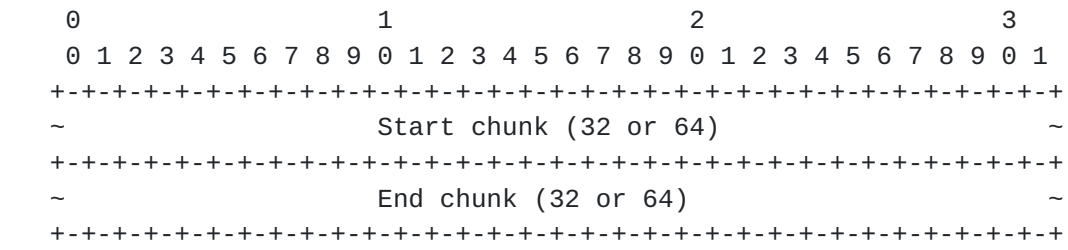
All peers in a swarm MUST use the same chunk addressing method.

4.1. Start-End Ranges

A chunk specification consists of a single (start specification,end specification) pair that identifies a range of chunks (end inclusive). The start and end specifications can use one of multiple addressing schemes. Two schemes are currently defined, chunk ranges and byte ranges.

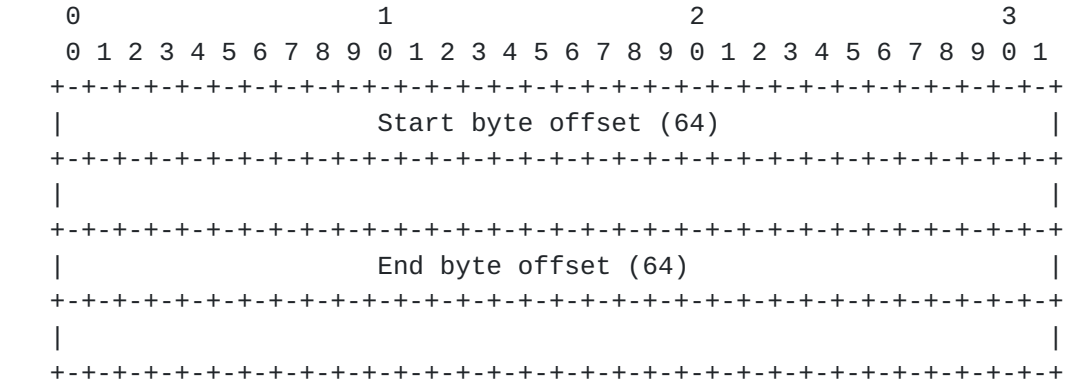
4.1.1. Chunk Ranges

The start and end specification are both chunk identifiers. Chunk identifiers are 32-bit or 64-bit unsigned integers. A PPSPP peer MUST support this scheme.



4.1.2. Byte Ranges

The start and end specification are 64-bit byte offsets in the content. The support for this scheme is OPTIONAL.



4.2. Bin Numbers

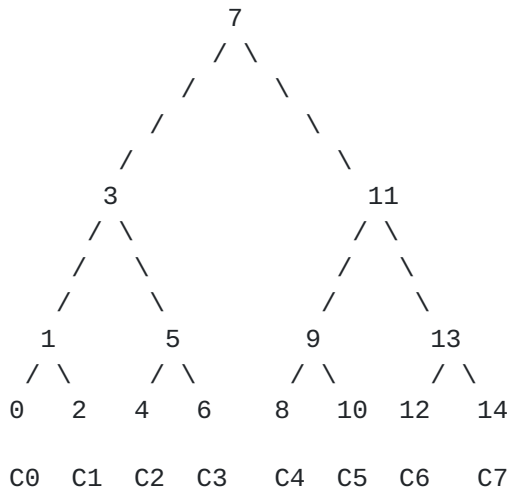
PPSPP introduces a novel method of addressing chunks of content called "bin numbers" (or "bins" for short). Bin numbers allow the addressing of a binary interval of data using a single integer. This reduces the amount of state that needs to be recorded per peer and the space needed to denote intervals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

trees, thus reducing complexity. The support for this scheme is OPTIONAL.

In bin addressing, the smallest binary interval is a single chunk (e.g. a block of bytes which may be of variable size), the largest interval is a complete range of 2**63 chunks. In a novel addition to the classical scheme, these intervals are numbered in a way which lays them out into a vector nicely, which is called bin numbering, as follows. 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-1 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 bin number of higher level nodes P in the tree is calculated as follows:

```
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 an interval of width W=8 looks like this:



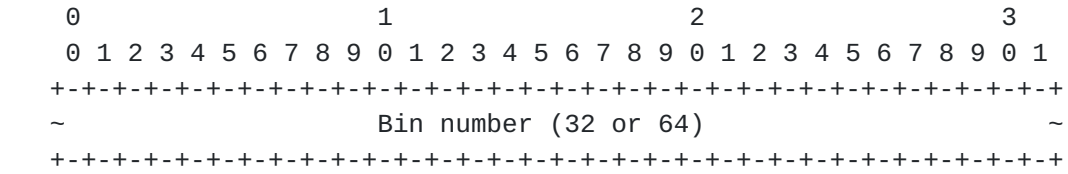
The bin number tree of an interval of width W=8

Figure 2

So bin 7 represents the complete interval, bin 3 represents the interval of chunk 0..3, bin 1 represents the interval of chunks 0 and 1, and bin 2 represents chunk C1. The special numbers 0xFFFFFFFF (32-bit) or 0xFFFFFFFFFFFFFFFF (64-bit) stands for an empty interval,

and 0x7FFF...FFF stands for "everything".

When bin numbering is used, the ID of a chunk is its corresponding (leaf) bin number in the tree and the chunk specification in HAVE and ACK messages is equal to a single bin number (32-bit or 64-bit), as follows.



4.3. In Messages

4.3.1. In HAVE Messages

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 chunk specification 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 chunk specification MUST denote at least the interval received, but the receiver is supposed to aggregate and acknowledge bigger intervals, 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.

Implementation note:

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

4.3.2. In ACK Messages

PPSPP peers MUST use ACK messages to acknowledge received chunks if an unreliable transport protocol is used. 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 chunk specification of its biggest, complete interval covering C to the sending peer (see HAVE).

5. Content Integrity Protection

PPSPP can use different methods for protecting the integrity of the content while it is being distributed via the peer-to-peer network. More specifically, PPSPP can use different methods for receiving peers to detect whether a requested chunk has been maliciously modified by the sending peer. In benign environments, content integrity protection can be disabled.

For static content, PPSPP currently defines one method for protecting integrity, called the Merkle Hash Tree scheme. If PPSPP operates over the Internet, this scheme MUST be used. If PPSPP operates in a benign environment this scheme MAY be used. So the scheme is mandatory-to-implement, to satisfy the requirement of strong security for an IETF protocol [RFC3365]. An extended version of the scheme is used to efficiently protect dynamically generated content (live streams), as explained below and in Section 6.1.

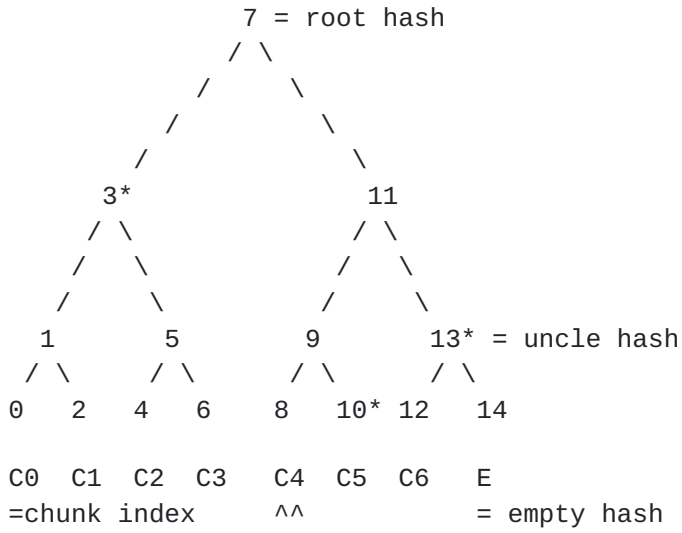
The Merkle Hash Tree scheme can work with different chunk addressing schemes. All it requires is the ability to address a range of chunks. In the following description abstract node IDs are used to identify nodes in the tree. On the wire these are translated to the corresponding range of chunks in the chosen chunk addressing scheme.

5.1. Merkle Hash Tree Scheme

PPSPP uses a method of naming content based on self-certification. In particular, content in PPSPP 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 a dynamic tree and a public key 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 with bin numbering. 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 an 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. If the two children are empty hashes, the parent is an empty all zeros hash as well (to save computation). 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.



The Merkle hash tree of an interval of width W=8

Figure 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 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 nodes 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 the root hash is undefined or, more precisely, transient, as long as new data is generated by the live source. Section 6.1.2 defines a method for content integrity verification for live streams that works with such a dynamic tree. Although the tree is dynamic, content verification works the same for both live and predefined content, resulting in a unified method for both types of streaming.

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 of datagrams between two peers. 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 SHOULD be put into the same datagram as the chunk's data. If this is not possible because of a limitation on datagram size, the necessary hashes MUST be sent first in one or more datagrams. 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 send 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 chunks C0 and C1 (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 C0 and C1 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 chunk C7, the sender needs to include just the hashes for nodes 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 trade-off 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. The receiver MUST remember all the hashes it needs to verify missing chunks that it still wants to download. Note that the latter implies that a hardware-limited receiver MAY forget some hashes if it does not plan to announce possession of these chunks to others (i.e., does not plan to send HAVE messages.)

5.4. INTEGRITY Messages

Concretely, a peer that wants to send a chunk of content creates a datagram that MUST consist of a list of INTEGRITY messages followed by a DATA message. If the INTEGRITY messages and DATA message cannot be put into a single datagram because of a limitation on datagram size, the INTEGRITY messages MUST be sent first in one or more datagrams. The list of INTEGRITY messages sent MUST contain a INTEGRITY message for each hash the receiver misses for integrity checking. A INTEGRITY message for a hash MUST contain the chunk specification corresponding to the node ID of the hash and the hash data itself. The chunk specification corresponding to a node ID is defined as the range of chunks formed by the leaves of the subtree rooted at the node. For example, node 3 in Figure 3 denotes chunks 0,2,4,6, so the chunk specification should denote that interval. The list of INTEGRITY messages MUST be sorted in order of the tree height of the nodes, descending. The DATA message MUST contain the chunk specification of the chunk and chunk itself. A peer MAY send the required messages for multiple chunks in the same datagram, depending on the encapsulation.

5.5. Discussion and Overhead

The current method for protecting content integrity in BitTorrent [BITTORRENT] is not suited for streaming. It involves providing clients with the hashes of the content's chunks before the download commences by means of metadata files (called .torrent files in BitTorrent.) However, when chunks are small as in the current UDP encapsulation of PPSPP this implies having to download a large number of hashes before content download can begin. This, in turn, increases time-till-playback for end users, making this method unsuited for streaming.

The overhead of using Merkle hash trees is limited. The size of the hash tree expressed as the total number of nodes depends on the

number of chunks the content is divided (and hence the size of chunks) following this formula:

```
nnodes = math.pow(2,math.log(nchunks,2)+1)
```

In principle, the hash values of all these nodes will have to be sent to a peer once for it to verify all chunks. Hence the maximum on-the-wire overhead is hashsize * nnodes. However, the actual number of hashes transmitted can be optimized as described in Section 5.3. To see a peer can verify all chunks whilst receiving not all hashes, consider the example tree in Section 5.1.

In case of a simple progressive download, of chunks 0,2,4,6, etc. the sending peer will send the following hashes:

Chunk	Node IDs of hashes sent
0	2,5,11
2	- (receiver already knows all)
4	6
6	-
8	10,13 (hash 3 can be calculated from 0,2,5)
10	-
12	14
14	-
Total	# hashes 7

Table 1: Overhead for the example tree

So the number of hashes sent in total (7) is less than the total number of hashes in the tree (16), as a peer does not need to send hashes that are calculated and verified as part of earlier chunks.

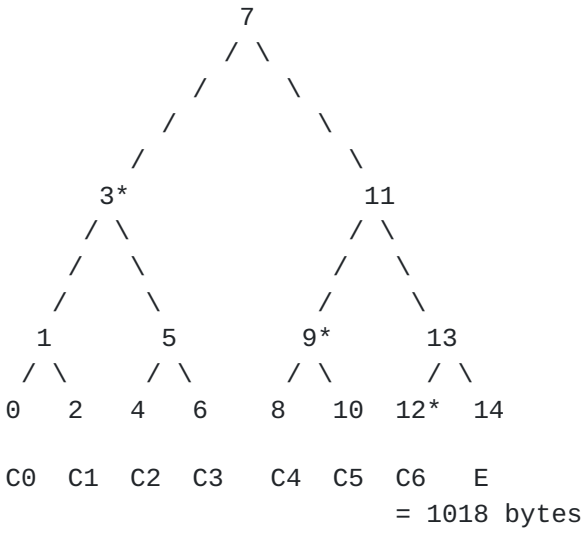
5.6. Automatic Detection of Content Size

In PPSPP, 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, PPSPP 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 PPSPP MAY use this automatic detection feature. Note this feature is the only feature of PPSPP that requires that a fixed-sized chunk is used.

5.6.1. 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 PPSPP: reliable file size detection and download/live streaming unification (see Section 6). The concept of peak hashes depends on the concepts of filled and incomplete nodes. 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 node is now defined as a node that corresponds to an interval of leaves that consists only of hashes of content chunks, not empty hashes. Reversely, an incomplete (not filled) node corresponds to an interval that contains also empty hashes, typically an interval that extends past the end of the file. In the following figure nodes 7, 11, 13 and 14 are incomplete the rest is filled.

Formally, a peak hash is the hash of a filled node in the Merkle tree, whose sibling is an incomplete node. 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 is shown in Figure 4. Following the definition the peak hashes of this file are in nodes 3, 9 and 12, denoted with a *. E denotes an empty hash.



Peak hashes in a Merkle hash tree.

Figure 4

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, nodes 3, 9, 12. The number of peak hashes for a file is therefore also at most logarithmic with its size.

A peer knowing which nodes 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 nodes are the peaks can be securely communicated from one (untrusted) peer A to another B by letting A send the peak hashes and their node IDs 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 a filled node, 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 nodes. (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 node ID 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

5.6.2. Procedure

A PPSPP implementation that wants to use automatic size detection MUST operate as follows. When a peer A sends a DATA message for the first time to a peer B, A MUST first send all the peak hashes for the content, unless B has already signalled earlier in the exchange that it knows the peak hashes by having acknowledged any chunk. If they are needed, the peak hashes MUST be sent as an extra list of uncle hashes for the chunk, before the list of actual uncle hashes of the chunk as described in Section 5.3. The receiver B MUST check the peak hashes against the root hash to determine the approximate content size. To obtain the definite content size peer B MUST download the last chunk of the content from any peer that offers it.

As an example, let's consider a 7162 bytes long file, which fits in 7 chunks of 1 kilobyte, distributed by a peer A. Figure 4 shows the relevant Merkle hash tree. A peer B which only knows the root hash of the file, after successfully connecting to A, requests the first chunk of data, C0 in Figure 4. Peer A replies to B by including in the datagram the following messages in this specific order. First the three peak hashes of this particular file, the hashes of nodes 3, 9 and 12. Second, the uncle hashes of C0, followed by the DATA message containing the actual content of C0. Upon receiving the peak hashes, peer B checks them against the root hash determining that the file is 7 chunks long. To establish the exact size of the file, peer B needs to request and retrieve the last chunk containing data, C6 in Figure 4. Once the last chunk has been retrieved and verified, peer B concludes that it is 1018 bytes long, hence determining that the file is exactly 7162 bytes long.

6. Live Streaming

The set of messages defined above can be used for live streaming as well. In a pull-based model, a live streaming injector can announce the chunks it generates via HAVE messages, and peers can retrieve them via REQUEST messages. Areas that need special attention are content authentication and chunk addressing (to achieve an infinite stream of chunks).

6.1. Content Authentication

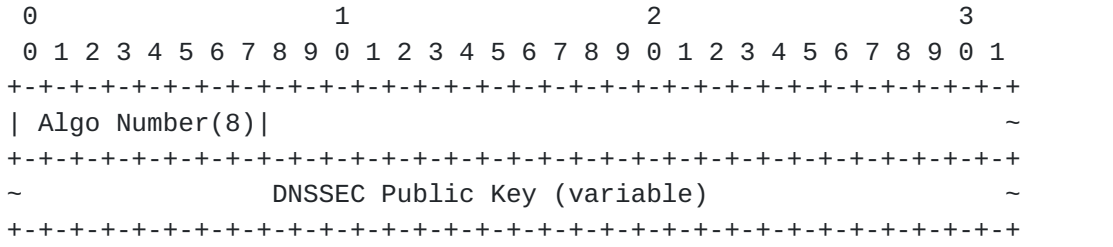
For live streaming, PPSPP supports two methods for a peer to authenticate the content it receives from another peer, called "Sign All" and "Unified Merkle Tree".

In the "Sign All" method, the live injector signs each chunk of content using a private key and peers, upon receiving the chunk,

check the signature using the corresponding public key obtained from a trusted source. Support for this method is OPTIONAL.

In the "Unified Merkle Tree" method, PPSPP combines the Merkle Hash Tree scheme for static content with signatures to unify the video-on-demand and live streaming scenarios. The use of Merkle hash trees reduces the number of signing and verification operations, hence providing a similar signature amortization to the approach described in [SIGMCAST]. If PPSPP operates over the Internet, the "Unified Merkle Tree" method MUST be used. If the protocol operates in a benign environment the method MAY be used. So this method is mandatory-to-implement.

In both methods the swarm ID consists of a public key encoded as in a DNSSEC DNSKEY resource record without BASE-64 encoding [RFC4034]. In particular, the swarm ID consists of a 1 byte Algorithm field that identifies the public key's cryptographic algorithm and determines the format of the Public Key field that follows. The value of this Algorithm field is one of the Domain Name System Security (DNSSEC) Algorithm Numbers [IANADNSSECALGNUM]. The RSASHA1 [RFC4034], ECDSAP256SHA256 and ECDSAP384SHA384 [RFC6605] algorithms are MANDATORY to implement.



6.1.1. Sign All

In the "Sign All" method, the live injector signs each chunk of content using a private key and peers, upon receiving the chunk, check the signature using the corresponding public key obtained from a trusted source. In particular, in PPSPP, the swarm ID of the live stream is that public key.

A peer that wants to send a chunk of content creates a datagram that MUST contain a SIGNED_INTEGRITY message with the chunk's signature, followed by a DATA message with the actual chunk. If the SIGNED_INTEGRITY message and DATA message cannot be contained into a single datagram, because of a limitation on datagram size, the SIGNED_INTEGRITY message MUST be sent first in a separate datagram. The SIGNED_INTEGRITY message consists of the chunk specification, the timestamp, and the digital signature.

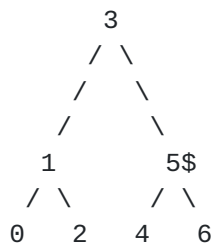
The digital signature algorithm which is used, is determined by the Live Signature Algorithm protocol option, see Section 7.7. The signature is computed over a concatenation of the on-the-wire representation of the chunk specification, a 64-bit NTP timestamp [RFC5905], and the chunk, in that order. The timestamp is the time signature that was made at the injector in UTC.

6.1.2. Unified Merkle Tree

In this method, the chunks of content are used as the basis for a Merkle hash tree as for static content. However, because chunks are continuously generated, this tree is not static, but dynamic. As a result, the tree does not have a root hash, or more precisely has a transient root hash. A public key therefore serves as swarm ID of the content. It is used to digitally sign updates to the tree, allowing peers to expand it based on trusted information using the following process.

6.1.2.1. Signed Munro Hashes

The live injector generates a number of chunks, denoted NCHUNKS_PER_SIG, corresponding to fixed power of 2 (NCHUNKS_PER_SIG>=2), which are added as new leaves to the existing hash tree. As a result of this expansion the hash tree contains a new subtree, that is NCHUNKS_PER_SIG chunks wide at the base. The root of this new subtree is referred to as the munro of that subtree, and its hash as the munro hash of the subtree, illustrated in Figure 5. In this figure, node 5 is the new munro, labeled with a $ sign.



Expanded live tree. With NCHUNKS_PER_SIG=2, node 5 is the munro for the new subtree spanning 4 and 6. Node 1 is the munro for the subtree spanning chunks 0 and 2, created in the previous iteration.

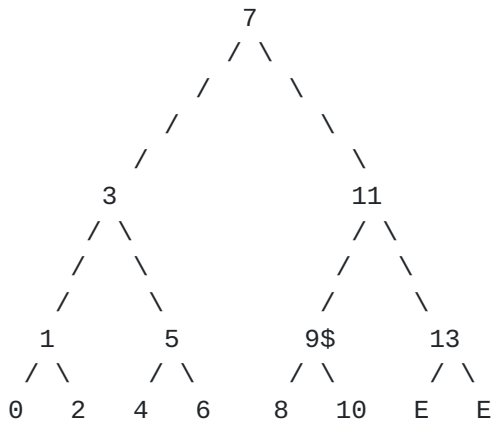
Figure 5

Informally, the process now proceeds as follows. The injector now signs only the munro hash of the new subtree using its private key.

Next, the injector announces the existence of the new subtree to its peers using HAVE messages. When a peer, in response to the HAVE messages, requests a chunk from the new subtree, the injector first sends the signed munro hash corresponding to the requested chunk. Afterwards, similar to static content, the injector sends the uncle hashes necessary to verify that chunk, as in Section 5.1. In particular, the injector sends the uncle hashes necessary to verify the requested chunk against the munro hash. This differs from static content, where the verification takes places against the root hash. Finally, the injector sends the actual chunk.

The receiving peer verifies the signature on the signed munro using the swarm ID (a public key), and updates its hash tree. As the peer now knows the munro hash is trusted, it can verify all chunks in the subtree against this munro hash, using the accompanying uncle hashes as in Section 5.1.

To illustrate this procedure, lets consider the next iteration in the process. The injector has generated the current tree shown in Figure 5 and it is connected to several peers that currently have the same tree and all posses chunks 0, 2, 4 and 6. When the injector generates two new chunks, NCHUNKS_PER_SIG=2, the hash tree expands as shown in Figure 6. The two new chunks, 8 and 10, extend the tree on the right side, and to accommodate them a new root is created, node 7. As this tree is wider at the base than the actual number of chunks, there are currently two empty leaves. The munro node for the new subtree is 9, labeled with a $ sign.



Expanded live tree. With NCHUNKS_PER_SIG=2, node 9 is the munro of the newly added subtree spanning chunks 8 and 10.

Figure 6

The injector now needs to inform its peers of the updated tree, communicating the addition of the new munro hash 9. Hence, it sends a HAVE message with a chunk specification for nodes 8+10 to its peers. As a response, a peer P requests the newly created chunk, e.g. chunk 8, from the injector by sending a REQUEST message. In reply, the injector sends the signed munro hash of node 9 as an INTEGRITY message with the hash of node 9, and a SIGNED_INTEGRITY message with the signature of the hash of node 9. These messages are followed by an INTEGRITY message with the hash of node 10, and a DATA message with chunk 8.

Upon receipt, peer P verifies the signature of the munro and expands its view of the tree. Next, the peer computes the hash of chunk 8 and combines it with the received hash of node 10, computing the expected hash of node 9. He can then verify the content of chunk 8 by computing the computed hash of node 9 with the munro hash of the same node he just received, hence P has successfully verified the integrity of chunk 8.

This procedure requires just one signing operation for every NCHUNKS_PER_SIG chunks created, and one verification operation for every NCHUNKS_PER_SIG received, making it much cheaper than "Sign All". A receiving peer does additionally need to check one or more hashes per chunk via the Merkle Tree scheme, but this has less hardware requirements than a signature verification for every chunk. This approach is similar to signature amortization via Merkle Tree Chaining [SIGMCAST]. The downside of scheme is in an increased latency. A peer cannot download the new chunks until the injector has computed the signature and announced the subtree. A peer MUST check the signature before forwarding the chunks to other peers [POLLIVE].

The number of chunks per signature NCHUNKS_PER_SIG MUST be a fixed power of 2 for simplicity. NCHUNKS_PER_SIG MUST be larger than 1 for performance reasons. There are two related factors to consider when choosing a value for NCHUNKS_PER_SIG. First, the allowed CPU load on clients due to signature verifications, given the expected bitrate of the stream. To achieve a low CPU load in a high bitrate stream, NCHUNKS_PER_SIG should be high. Second, the effect on latency, which increases when NCHUNKS_PER_SIG gets higher, as just discussed. Note how the procedure does not preclude the use of variable-sized chunks.

This method of integrity verification provides an additional benefit. If the system includes some peers that saved the complete broadcast, as soon as the broadcast ends, the content is available as a video-on-demand download using the now stabilized tree and the final root hash as swarm identifier. Peers which saved all the chunks, can now announce the root hash to the tracking infrastructure and instantly

seed the content.

6.1.2.2. Munro Signature Calculation

The digital signature algorithm used is determined by the Live Signature Algorithm protocol option, see Section 7.7. The signature is computed over a concatenation of the on-the-wire representation of the chunk specification of the munro, a 64-bit NTP timestamp [RFC5905], and the munro hash, in that order. The timestamp is the time signature that was made at the injector in UTC.

6.1.2.3. Procedure

Formally, the injector MUST NOT send a HAVE message for chunks in the new subtree until it has computed the signed munro hash for that subtree.

When peer B requests a chunk C from peer A (either the injector or another peer), and peer A decides to reply, it must do so as follows. First, peer A MUST send an INTEGRITY message with the chunk specification for the munro of chunk C and the munro's hash, followed by a SIGNED_INTEGRITY message with the chunk specification for the munro, timestamp and its signature, in a single datagram, unless B indicated earlier in the exchange that it already possess a chunk with the same corresponding munro (by means of HAVE or ACK messages). Following these two messages (if any), peer A MUST send the necessary missing uncles hashes needed for verifying the chunk against its munro hash, and the chunk itself, as described in Section 5.4, sharing datagrams if possible.

6.1.2.4. Secure Tune In

When a peer tunes into a live stream it has to determine what is the last chunk the injector has generated. To facilitate this process in the Unified Merkle Tree scheme, each peer shares its knowledge about the injector's chunks with the others by exchanging their latest signed munro hashes, as follows.

Recall that in PPSPP, when peer A initiates a channel with peer B, peer A sends a first datagram with a HANDSHAKE message, and B responds with a second datagram also containing a HANDSHAKE message (see Section 3.1). When A sends a third datagram to B, and it is received by B both peers know that the other is listening on its stated transport address. B is then allowed to send heavy payload like DATA messages in the fourth datagram. Peer A can already safely do that in the third datagram.

In the Unified Merkle Tree scheme, peer A MUST send its right-most

signed munro hash to B in the third datagram, and in any subsequent datagrams to B, until B indicates that it possess a chunk with the same corresponding munro or a more recent munro (by means of a HAVE or ACK message). B may already have indicated this fact by means of HAVE messages in the second datagram. Conversely, when B sends the fourth datagram or any subsequent datagram to A, B MUST send its right-most signed munro hash, unless A indicated knowledge of it or more recent munros. The right-most signed munro hash of a peer is defined as the munro hash signed by the injector of the right-most subtree of width NCHUNKS_PER_SIG chunks in the peer's Merkle hash tree. Peer A and B MUST NOT send the signed munro hash in the first, respectively, second datagram as it is considered heavy payload.

When a peer receives a SIGNED_INTEGRITY message with a signed munro hash but the timestamp is too old, the peer MUST discard the message. Otherwise it SHOULD use the signed munro to update its hash tree and pick a tune-in point in the live stream. A peer may use the information from multiple peers to pick the tune-in point.

6.2. Forgetting Chunks

As a live broadcast progresses a peer may want to discard the chunks that it already played out. Ideally, other peers should be aware of this fact such that they will not try to request these chunks from this peer. This could happen in scenarios where live streams may be paused by viewers, or viewers are allowed to start late in a live broadcast (e.g., start watching a broadcast at 20:35 whereas it began at 20:30).

PPSPP provides a simple solution for peers to stay up-to-date with the chunk availability of a discarding peer. A discarding peer in a live stream MUST enable the Live Discard Window protocol option, specifying how many chunks/bytes it caches before the last chunk/byte it advertised as being available (see Section 7.9). Its peers SHOULD apply this number as a sliding window filter over the peer's chunk availability as conveyed via its HAVE messages.

Three factors are important when deciding for an appropriate value for this option: the desired amount of playback buffer for peers, the bitrate of the stream and the available resources of the peer. Consider the case of a fresh peer joining the stream. The size of the discard window of the peers it connects to influences how much data it can directly download to establish its prebuffer. If the window is smaller than the desired buffer, the fresh peer has to wait until the peers downloaded more of the stream before it can start playback. As media buffers are generally specified in terms of a number of seconds, the size of the discard window also related to the (average) bitrate of the stream. Finally, if a peer has little

resources to store chunks and metadata it should chose a small discard window.

7. Protocol Options

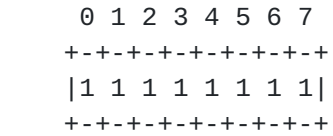
The HANDSHAKE message in PPSPP can contain the following protocol options. Unless stated otherwise, a protocol option consists of an 8-bit code followed by an 8-bit value. Larger values are all encoded big-endian. Each protocol option is explained in the following subsections.

Code	Description
0	Version
1	Minimum Version
2	Swarm Identifier
3	Content Integrity Protection Method
4	Merkle Hash Tree Function
5	Live Signature Algorithm
6	Chunk Addressing Method
7	Live Discard Window
8	Supported Messages
9-254	Unassigned
255	End Option

Table 2: PPSP Peer Protocol Options

7.1. End Option

A peer MUST conclude the list of protocol options with the end option. Subsequent octets should be considered protocol messages. The code for the end option is 255, and unlike others it has no value octet, so the option's length is 1 octet.

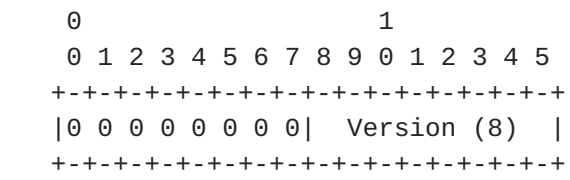


7.2. Version

A peer MUST include the maximum version of the PPSPP protocol it supports as the first protocol option in the list. The code for this option is 0. Defined values are listed in Table 3.

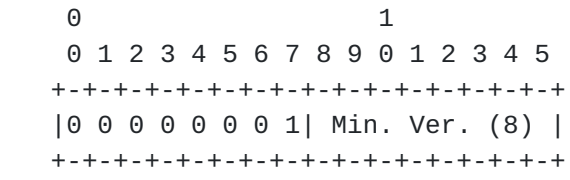
Version	Description	
1	Protocol as described in this document	
2-255	Unassigned	

Table 3: PPSP Peer Protocol Version Numbers



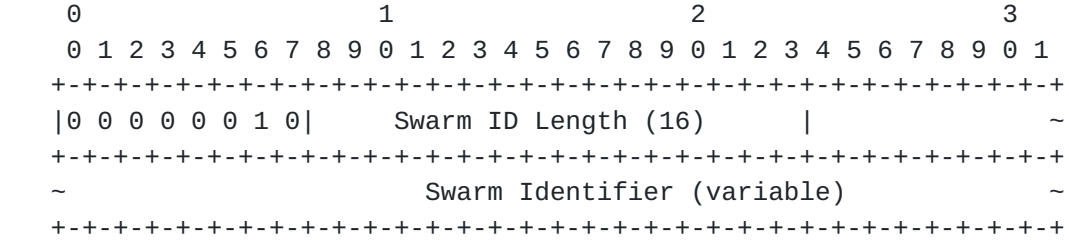
7.3. Minimum Version

When a peer initiates the handshake it MUST include the minimum version of the PPSPP protocol it supports in the list of protocol options, following the Min/max versioning scheme defined in [RFC6709], Section 4.1, strategy 5. The code for this option is 1. Defined values are listed in Table 3.



7.4. Swarm Identifier

When a peer initiates the handshake it MUST include a single swarm identifier option. In other cases a peer MAY include a swarm identifier option, as an end-to-end check. This option has the following structure:



The Swarm ID Length field contains the length of the single Swarm Identifier that follows in bytes. The Length field is 16 bits wide to allow for large public keys as identifiers in live streaming.

Each PPSPP peer knows the IDs of the swarms it joins so this information can be immediately verified upon receipt.

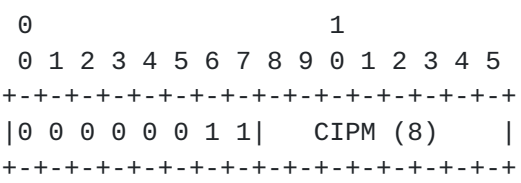
7.5. Content Integrity Protection Method

A peer MUST include the content integrity method used by a swarm. The code for this option is 3. Defined values are listed in Table 4.

Method	Description
0	No integrity protection
1	Merkle Hash Tree
2	Sign All
3	Unified Merkle Tree
4-255	Unassigned

Table 4: PPSP Peer Content Integrity Protection Methods

The "Merkle Hash Tree" method is the default for static content, see Section 5.1. "Sign All", and "Unified Merkle Tree" are for live content, see Section 6.1, with "Unified Merkle Tree" being the default.



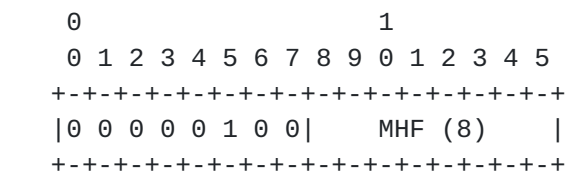
7.6. Merkle Tree Hash Function

When the content integrity protection method is "Merkle Hash Tree" this option defining which hash function is used for the tree MUST be included. The code for this option is 4. Defined values are listed in Table 5 (see [FIPS180-4] for the function semantics).

Function	Description
0	SHA1
1	SHA-224
2	SHA-256
3	SHA-384
4	SHA-512
5-255	Unassigned

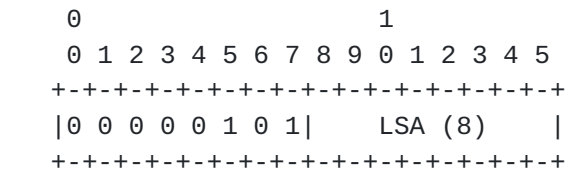
Table 5: PPSP Peer Protocol Merkle Hash Functions

Implementations MUST support SHA1, see Section 13.5, which is also the default.



7.7. Live Signature Algorithm

When the content integrity protection method is "Sign All" or "Unified Merkle Tree" this option MUST be defined. The code for this option is 5. The 8-bit value of this option is one of the Domain Name System Security (DNSSEC) Algorithm Numbers [IANADNSSECALGNUM]. The RSASHA1 [RFC4034], ECDSAP256SHA256 and ECDSAP384SHA384 [RFC6605] algorithms are MANDATORY to implement. Default is ECDSAP256SHA256.



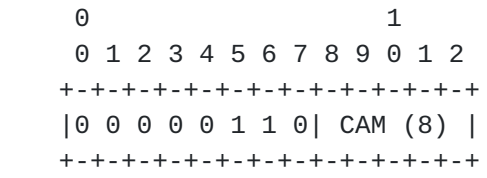
7.8. Chunk Addressing Method

A peer MUST include the chunk addressing method it uses. The code for this option is 6. Defined values are listed in Table 6.

Method	Description
0	32-bit bins
1	64-bit byte ranges
2	32-bit chunk ranges
3	64-bit bins
4	64-bit chunk ranges
5-255	Unassigned

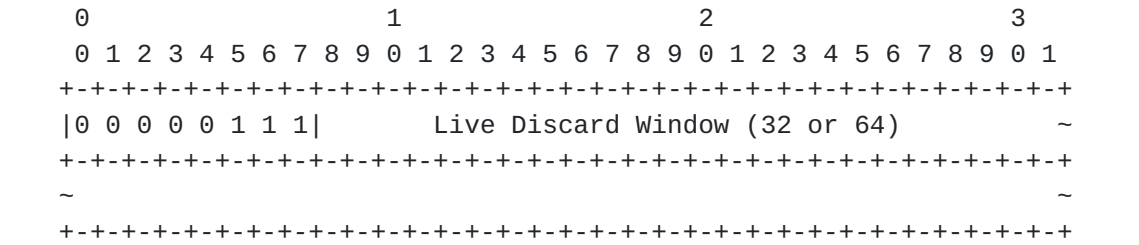
Table 6: PPSP Peer Chunk Addressing Methods

Implementations MUST support "32-bit chunk ranges" and "64-bit chunk ranges". Default is "32-bit chunk ranges".



7.9. Live Discard Window

A peer in a live swarm MUST include the discard window it uses. The code for this option is 7. The unit of the discard window depends on the chunk addressing method used. For bins and chunk ranges it is a number of chunks, for byte ranges it is a number of bytes. Its data type is the same as for a bin, or one value in a range specification. In other words, its value is a 32-bit or 64-bit integer in big endian format. If this option is used, the Chunk Addressing Method MUST appear before it in the list. This option has the following structure:

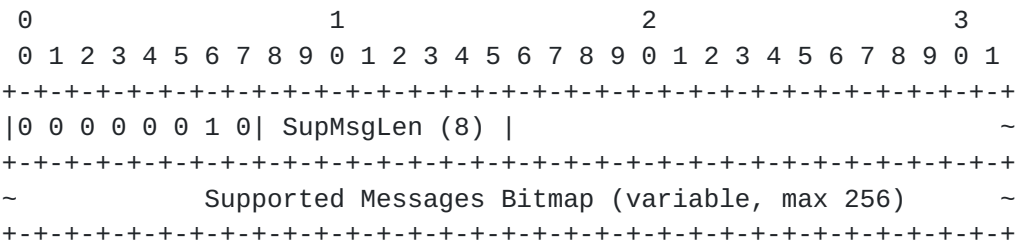


A peer that does not, under normal circumstances, discard chunks MUST set this option to the special value 0xFFFFFFFF (32-bit) or 0xFFFFFFFFFFFFFFFF (64-bit). For example, peers that record a complete broadcast to offer it directly as a static asset after the broadcast ends use these values (see Section 6.1.2). Section 6.2 explains how to determine a value for this option.

7.10. Supported Messages

Peers may support just a subset of the PPSPP 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 who support only a proper subset of the PPSPP messages MUST signal which subset they support by means of this protocol option. The code for this option is 8. The value of this option is a length octet (SupMsgLen) indicating the length in bytes of the compressed bitmap that follows.

The set of messages supported can be derived from the compressed bitmap by padding it with bytes of value 0 until it is 256 bits in length. Then a 1 bit in the resulting bitmap at position X (numbering left to right) corresponds to support for message type X, see Table 7. In other words, to construct the compressed bitmap, create a bitmap with a 1 for each message type supported and a 0 for a message type that is not, store it as an array of bytes and truncate it to the last non-zero byte.



8. UDP Encapsulation

PPSPP implementations MUST use UDP as transport protocol and MUST use LEDBAT for congestion control [RFC6817]. Using LEDBAT enables PPSPP to serve the content after playback (seeding) without disrupting the user who may have moved to different tasks that use its network connection. Future PPSPP versions can also run over other transport protocols, or use different congestion control algorithms.

8.1. Chunk Size

In general, an UDP datagram containing PPSPP messages SHOULD fit inside a single IP packet, so its maximum size depends on the MTU of the network. If the UDP datagram does not fit, its chance of getting lost in the network increases as the loss of a single fragment of the datagram causes the loss of the complete datagram.

The largest message in a PPSPP datagram is the DATA message carrying a chunk of content. So the (maximum) size of a chunk to choose for a

particular swarm depends primarily on the expected MTU. The chunk size should be chosen such that a chunk and its required INTEGRITY messages can generally be carried inside a single datagram, following the Atomic Datagram Principle (Section 5.3). Other considerations are the hardware capabilities of the peers. Having large chunks and therefore less chunks per mebibyte of content reduces processing costs. The chunk addressing schemes can all work with different chunk sizes, see Section 4.

The RECOMMENDED approach is to use fixed-sized chunks of 1024 bytes, as this size has a high likelihood of travelling end-to-end across the Internet without any fragmentation. In particular, with this size a UDP datagram with a DATA message can be transmitted as a single IP packet over an Ethernet network with 1500-byte frames.

A PPSPP implementation MAY use a variant of the Packetization Layer Path MTU Discovery (PLPMTUD), described in [RFC4821], for discovering the optimal MTU between sender and destination. As in PLPMTUD, progressively larger probing packets are used to detect the optimal MTU among a link. However, in PPSPP, probe packets SHOULD contain actual messages, in particular, multiple DATA messages. By using actual DATA messages as probe packets, the returning ACK messages will confirm the probe delivery, effectively updating the MTU estimate on both ends of the link. To be able to scale up probe packets with sensible increments, a minimum chunk size of 512 bytes SHOULD be used. Smaller chunk sizes lead to an inefficient protocol. An implication is that PPSP supports datagrams over IPv4 of 576 bytes or more only. This variant is not mandatory to implement.

The chunk size used for a particular swarm, or that fact that it is variable MUST be part of the swarm's metadata (which then minimally consists of the swarm ID and the chunk nature and size).

8.2. Datagrams and Messages

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

Msg Type	Description
0	HANDSHAKE
1	DATA
2	ACK
3	HAVE
4	INTEGRITY
5	PEX_RESv4
6	PEX_REQ
7	SIGNED_INTEGRITY
8	REQUEST
9	CANCEL
10	CHOKE
11	UNCHOKE
12	PEX_RESv6
13	PEX_REScert
14-254	Unassigned
255	Reserved

Table 7: PPSP Peer Protocol Message Types

Furthermore, integers are serialized in the network (big-endian) byte order. So consider the example of a HAVE message (Section 3.2) using bin chunk addressing. It has message type of 0x03 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: "0300000001".

All messages are idempotent or recognizable as duplicates. Idempotent means that processing a message more than once does not lead to a different state from if it was processed just once. In particular, a peer MAY resend DATA, ACK, HAVE, INTEGRITY, PEX_*, SIGNED_INTEGRITY, REQUEST, CANCEL, CHOKE and UNCHOKE messages without problems when loss is suspected. When a peer resends a HANDSHAKE message it can be recognized as duplicate by the receiver, because it already recorded the first connection attempt, and be dealt with.

8.3. Channels

As described in Section 3.11 PPSPP uses a multiplexing scheme, called channels, to allow multiple swarms to use the same UDP port. In the UDP encapsulation, each datagram from peer A to peer B is prefixed with the channel ID allocated by peer B. The peers learn about eachother's channel ID during the handshake as explained in a moment. A channel ID consists of 4 bytes and MUST be generated following the requirements in [RFC4960] (Sec. 5.1.3).

8.4. HANDSHAKE

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 swarm's metadata record which consists of:

   (a) the swarm ID of the content (see Section 5.1 and Section 6),

   (b) the chunk size used,

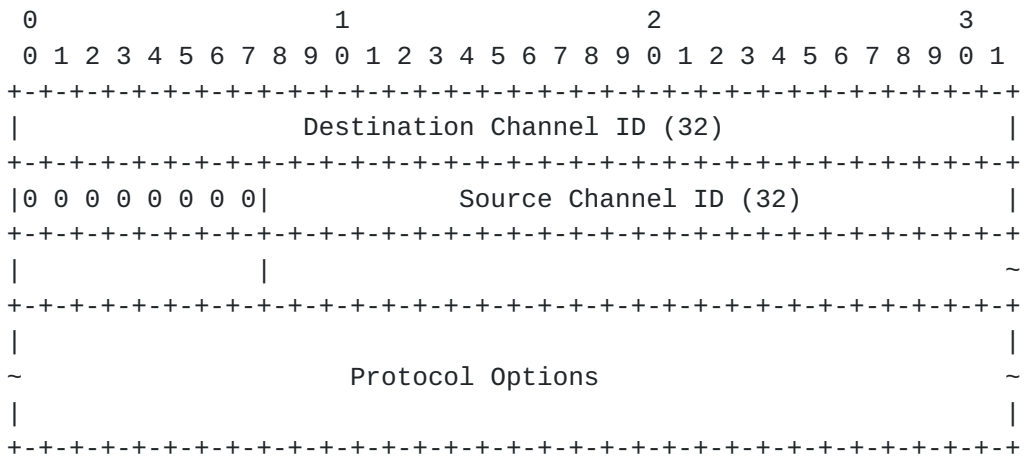
   (c) the chunk addressing method used,

   (d) the content integrity protection method used, and

   (e) the Merkle hash tree function used (if applicable).

   (f) If automatic content size detection (see Section 5.6) is not used, the content length is also part of the metadata record for static content.

A datagram containing a HANDSHAKE message:



where:

Destination Channel ID:

If the message is sent by the initiating peer than it MUST be an all 0-zeros channel ID.

If the message sent by the receiving peer than it MUST consist of the Source Channel ID from the sender's HANDSHAKE message

The octect 0x00: The HANDSHAKE message: 0x00

The Source Channel ID: A locally unused channel ID

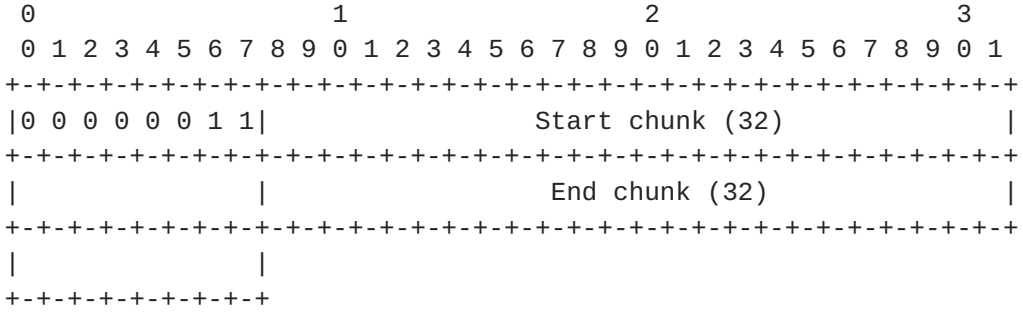
Protocol Options: A list of protocol options encoding the swarm's metadata, as defined in Section 7.

A peer SHOULD explicitly close a channel by sending a HANDSHAKE message that MUST contain an all 0-zeros channel ID and a list of protocol options. The list MUST be either empty or contain the maximum version number the sender supports, following the Min/max versioning scheme defined in [RFC6709], Section 4.1.

8.5. HAVE

A HAVE message (type 0x03) consists of a single chunk specification that states that the sending peer has those chunks and successfully checked their integrity. The single chunk specification represents a consecutive range of verified chunks. A bin consists of a single integer, and a chunk or byte range of two integers, of the width specified by the Chunk Addressing protocol options, encoded big endian.

A HAVE message using 32-bit chunk ranges as Chunk Addressing method:



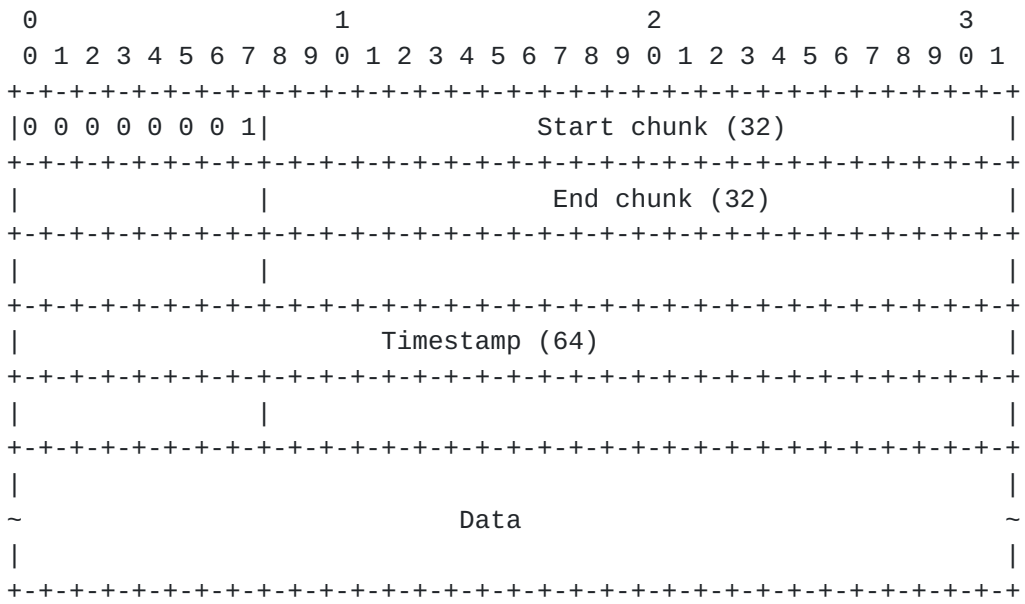
where the first octet is the HAVE message (0x03), followed by the start chunk and the end chunk describing the chunk range.

(received and checked first four kilobytes of a file/stream)

8.6. DATA

A DATA message (type 0x01) consists of a chunk specification, a timestamp and the actual chunk. In case a datagram contains one DATA message, a sender MUST always put the DATA message in the tail of the datagram. A datagram MAY contain multiple DATA messages when the chunk size is fixed and when none of DATA messages carry the last chunk if that is smaller than the chunk size. As the LEDBAT congestion control is used, a sender MUST include a timestamp, in particular, a 64-bit integer representing the current system time with microsecond accuracy. The timestamp MUST be included between chunk specification and the actual chunk.

A DATA message using 32-bit chunk ranges as Chunk Addressing method:

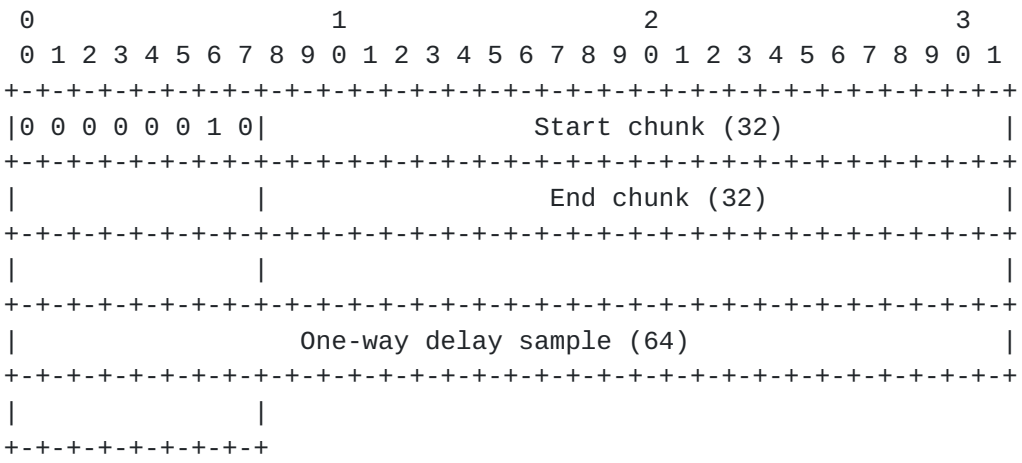


where the first octet is the DATA message (0x01), followed by the start chunk and the end chunk describing the chunk range, the timestamp and the actual data.

8.7. ACK

An ACK message (type 0x02) acknowledges data that was received from its addressee; to comply with the LEDBAT delay-based congestion control an ACK message consists of a chunk specification and a timestamp representing an one-way delay sample. The one-way delay sample is a 64-bit integer with microsecond accuracy, and is computed from the timestamp received from the previous DATA message containing the chunk being acknowledged following the LEDBAT specification.

An ACK message using 32-bit chunk ranges as Chunk Addressing method:

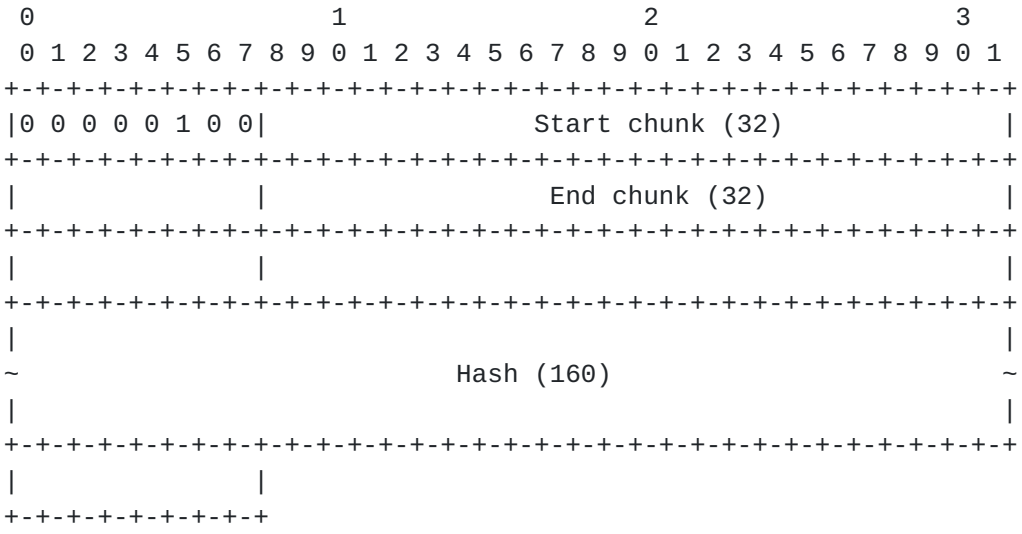


where the first octet is the ACK message (0x02), followed by the start chunk and the end chunk describing the chunk range, and the one-way delay sample.

8.8. INTEGRITY

An INTEGRITY message (type 0x04) consists of a chunk specification and the cryptographic hash for the specified chunk or node. The type and format of the hash depends on the protocol options.

An INTEGRITY message using 32-bit chunk ranges as Chunk Addressing method and SHA1 hash:

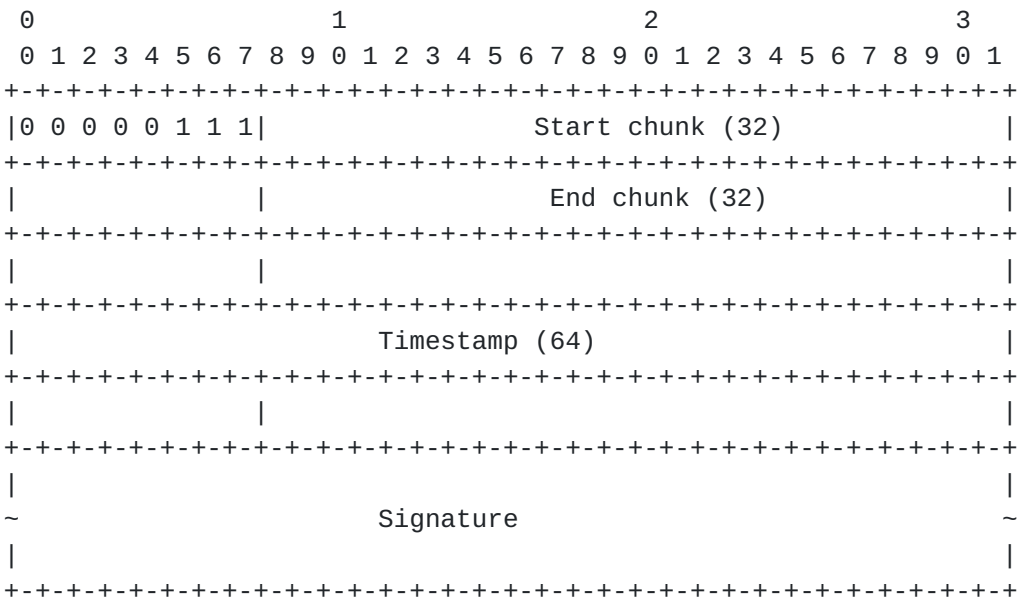


where the first octet is the ACK message (0x02), followed by the start chunk and the end chunk describing the chunk range, and the one-way delay sample.

8.9. SIGNED_INTEGRITY

A SIGNED_INTEGRITY message (type 0x07) consists of a chunk specification, a 64-bit NTP timestamp [RFC5905] and a digital signature encoded as a Signature field in a RRSIG record in DNSSEC without the BASE-64 encoding [RFC4034]. The signature algorithm is defined by the Live Signature Algorithm protocol option, see Section 7.7. The plaintext over which the signature is taken depends on the content integrity protection method used, see Section 6.1.

A SIGNED_INTEGRITY message using 32-bit chunk ranges as Chunk Addressing method:



where the first octet is the SIGNED_INTEGRITY message (0x07), followed by the start chunk and the end chunk describing the chunk range, the timestamp, and the Signature.

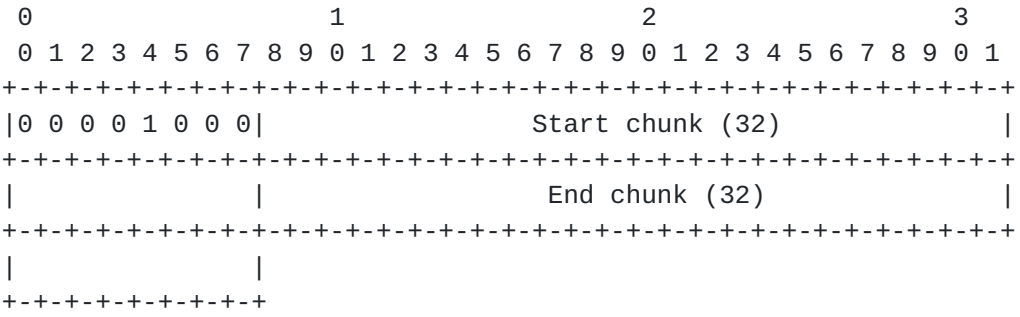
The length of the digital signature can be derived from the Live Signature Algorithm protocol option and the swarm ID as follows. The first MANDATORY algorithm is RSASHA1. In that case, the swarm ID consists of a 1-byte Algorithm field followed by a RSA public key stored as a tuple (exponent length,exponent,modulus) [RFC3110]. Given the exponent length and the length of the public key tuple in the swarm ID, the length of the modulus in bytes can be calculated. This yields the length of the signature as in RSA this is the length of the modulus [HAC01]. The other MANDATORY algorithms are

ECDSAP256SHA256 and ECDSAP384SHA384 [RFC6605]. For these algorithms the length of the digital signature is 64 and 96 bytes, respectively.

8.10. REQUEST

A REQUEST message (type 0x08) consists of a chunk specification for the chunks the requester wants to download.

A REQUEST message using 32-bit chunk ranges as Chunk Addressing method:

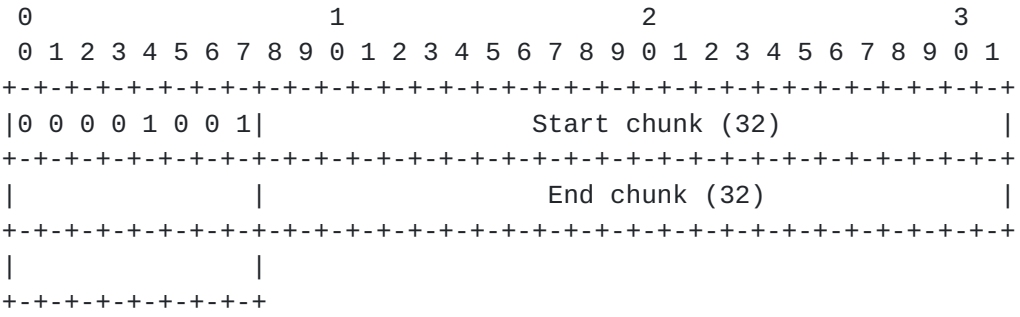


where the first octet is the REQUEST message (0x08), followed by the start chunk and the end chunk describing the chunk range.

8.11. CANCEL

A CANCEL message (type 0x09) consists of a chunk specification for the chunks the requester no longer is interested in.

A CANCEL message using 32-bit chunk ranges as Chunk Addressing method:

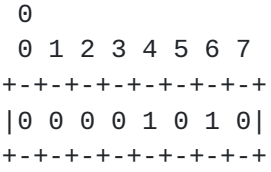


where the first octet is the CANCEL message (0x09), followed by the start chunk and the end chunk describing the chunk range.

8.12. CHOKE and UNCHOKE

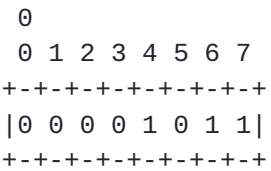
Both CHOKE and UNCHOKE messages (types 0x0a and 0x0b, respectively) carry no payload.

A CHOKE message:



where the first octet is the CHOKE message (0x0a).

An UNCHOKE message:

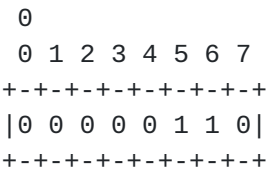


where the first octet is the UNCHOKE message (0x0b).

8.13. PEX_REQ, PEX_RESv4, PEX_RESv6 and PEX_REScert

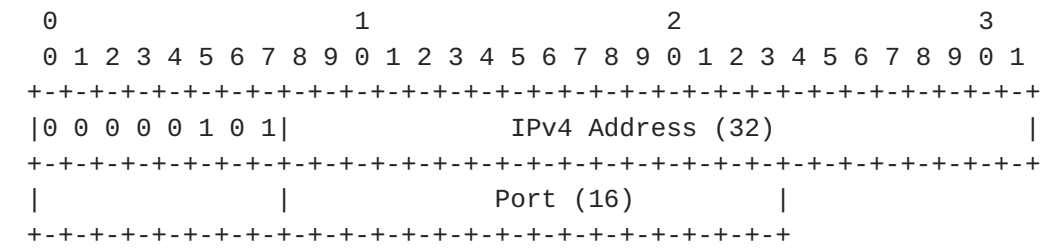
A PEX_REQ (0x06) message has no payload. A PEX_RES (0x05) message consists of an IPv4 address in big endian format followed by a UDP port number in big endian format. A PEX_RESv6 (0x0c) message contains a 128-bit IPv6 address instead of an IPv4 one. If a PEX_REQ message does not originate from a private or link-local address [RFC1918][RFC4291], then the PEX_RES* messages sent in reply MUST NOT contain such addresses. This is to prevent leaking of internal addresses to external peers.

A PEX_REQ message:



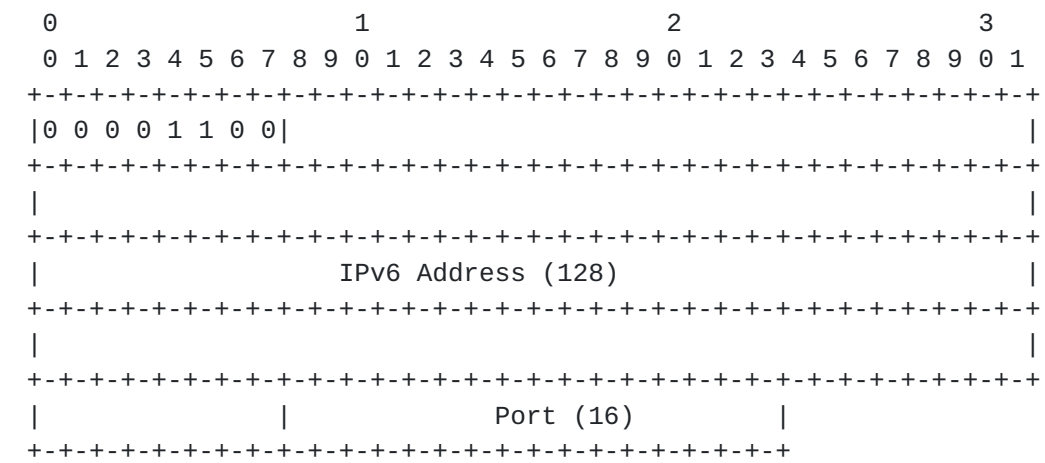
where the first octet is the PEX_REQ message (0x06).

A PEX_RES message:



where the first octet is the PEX_RES message (0x05), followed by the IPv4 address and the port number.

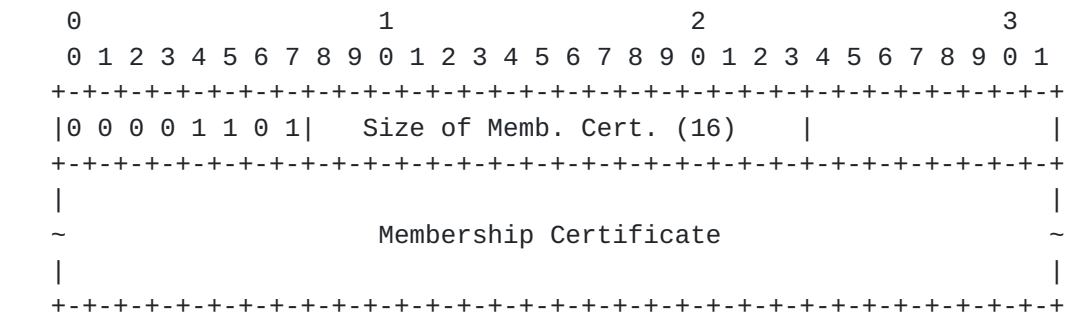
A PEX_RESv6 message:



where the first octet is the PEX_RESv6 message (0x0c), followed by the IPv6 address and the port number.

A PEX_REScert (0x0d) message consists of a 16-bit integer in big endian specifying the size of the membership certificate that follows, see Section 13.2.1. This membership certificate states that peer P at time T is a member of swarm S and is a X.509v3 certificate [RFC5280] that is encoded using the ASN.1 distinguished encoding rules (DER) [CCITT.X208.1988]. The certificate MUST contain a "Subject Alternative Name" extension, marked as critical, of type uniformResourceIdentifier.

A PEX_REScert message:



where the first octet is the PEX_REScert message (0x0d), followed by the size of the membership certificate, and the membership certificate.

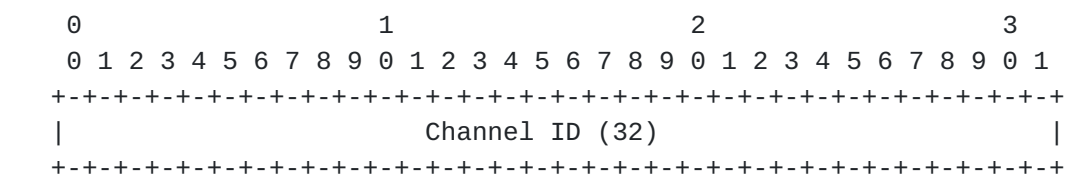
The URL contained in the name extension MUST follow the generic syntax for URLs [RFC3986], where its scheme component is "ppsp", the host in the authority component is the DNS name or IP address of peer P, the port in the authority component is the port of peer P, and the path contains the swarm identifier for swarm S, in hexadecimal form. In particular, the preferred form of the swarm identifier is xxyyzz..., where the 'x's, 'y's and 'z's are 2 hexadecimal digits of the 8-bit pieces of the identifier. The validity time of the certificate is set with notBefore UTCTime set to T and notAfter UTCTime set to T plus some expiry time defined by the issuer. An example URL:

ppsp://192.0.2.0:6778/e5a12c7ad2d8fab33c699d1e198d66f79fa610c3

8.14. KEEPALIVE

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

A keepalive datagram:



8.15. Detecting a Dead Peer

A guideline for declaring a peer dead consist of a 3 minute delay since that last packet has been received from that peer, and at least 3 datagrams were sent to that peer during the same period.

8.16. Flow and Congestion Control

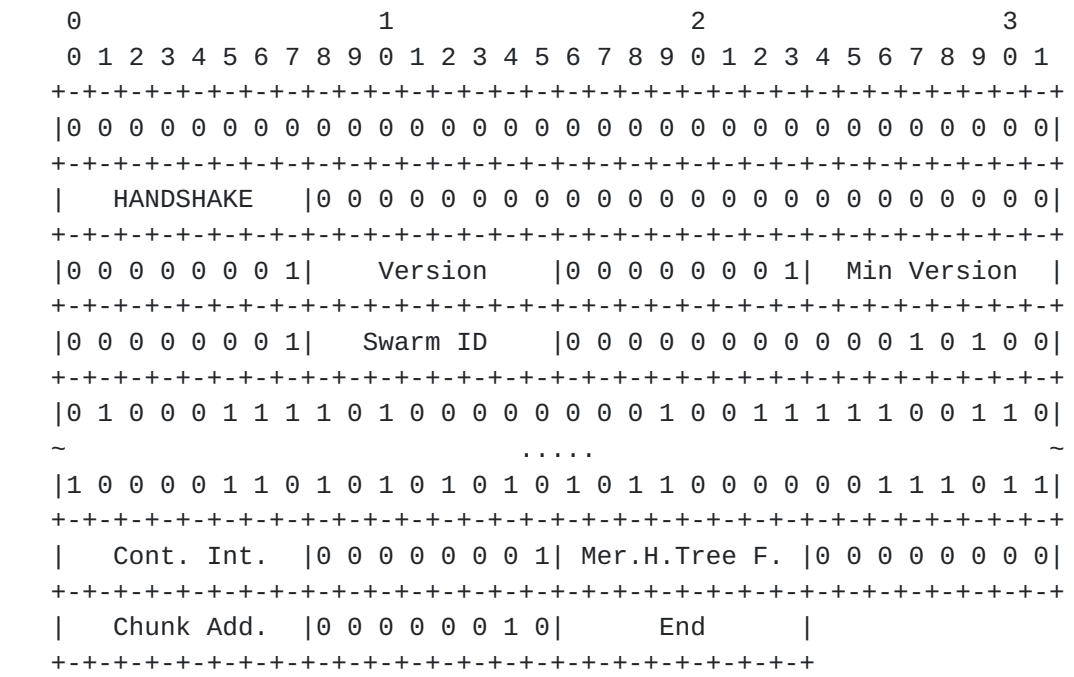
Explicit flow control is not necessary in PPSPP-over-UDP. In the case of video-on-demand the receiver will request data explicitly from peers and is therefore in control of how much data is coming towards it. In the case of live streaming, where a push-model may be used, the amount of data incoming is 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.

PPSPP-over-UDP can support different congestion control algorithms. At present, it uses the LEDBAT congestion control algorithm [RFC6817]. LEDBAT is a delay-based congestion control algorithm that is used by millions of users everyday as part of the uTP transmission protocol of BitTorrent [LBT],[LCOMPL] and is suitable for P2P streaming [PPSPPERF].

8.17. Example of Operation

We present a small example of communication between a leecher and a seeder. The example presents the transmission of the file "Hello World!", which fits within a 1024 byte chunk. For an easy understanding we use the message description names, as listed in Table 7, and the protocol option names as listed in Table 2, rather than the actual binary value.

To do the handshake the initiating peer sends a datagram that MUST start with an all 0-zeros channel ID (0x00000000), followed by a HANDSHAKE message, whose payload is a locally unused channel ID (0x00000001) and a list of protocol options.



The protocol options are:

Version: 1

Minimum supported Version: 1

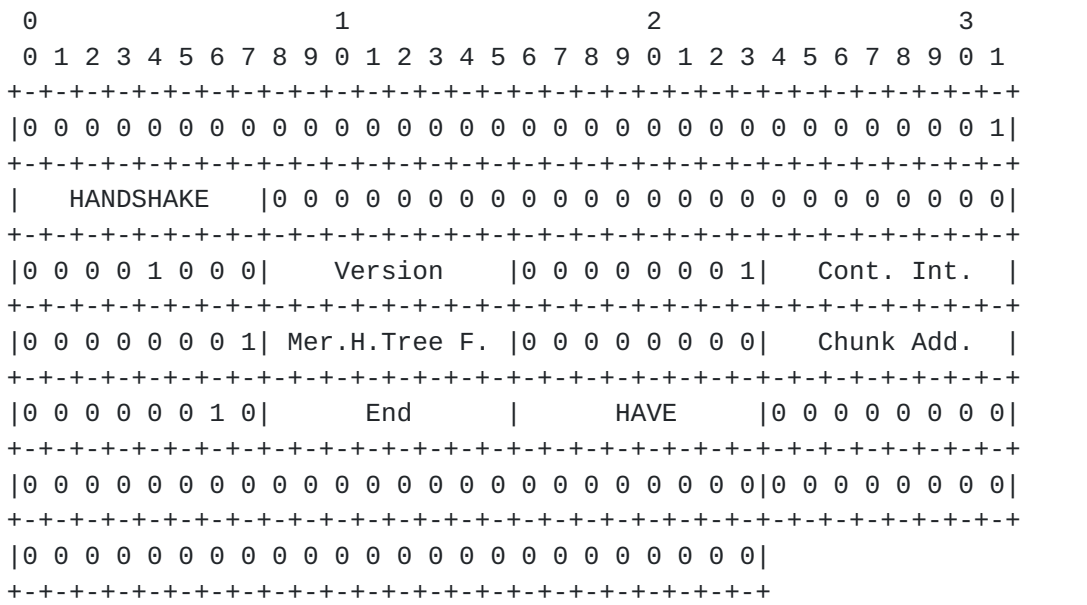
Swarm Identifier: A 20 bytes root hash (47a0...b03b) identifying the content.

Content Integrity Protection Method: Merkle Hash Tree.

Merkle Tree Hash Function: SHA1.

Chunk Addressing Method: 32-bit chunk ranges.

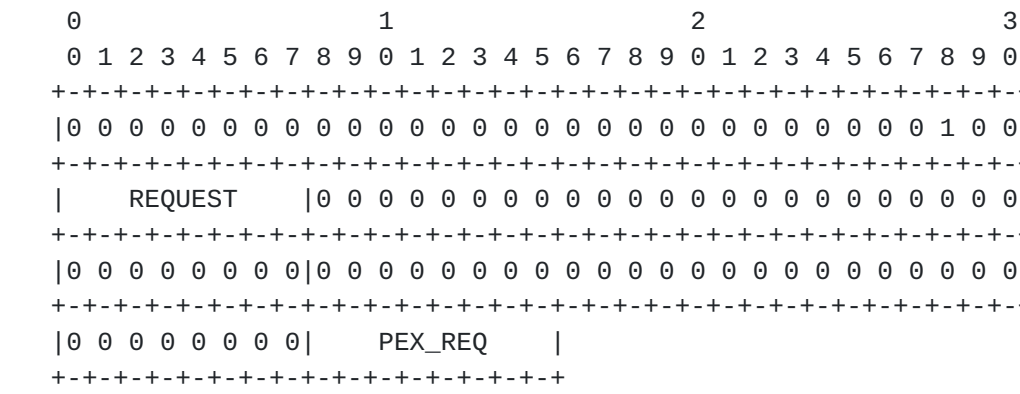
The receiving peer MAY respond, in which case the returned datagram MUST consist of the channel ID from the sender's HANDSHAKE message (0x00000001), a HANDSHAKE message, whose payload is a locally unused channel ID (0x00000008) and a list of protocol options, followed by any other messages it wants to send.



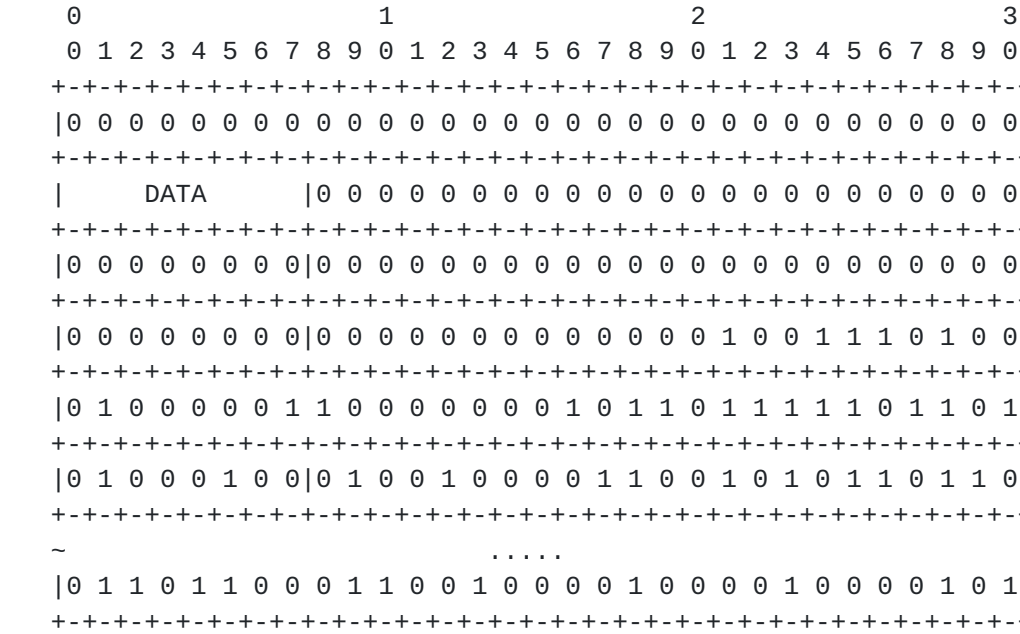
With the protocol options the receiving peer agrees on speaking protocol version 1, on using the Merkle Hash Tree as Content Integrity Protection Method, SHA1 hash as Merkle Tree Hash Function, and 32-bit chunk ranges as Chunk Addressing Method. Furthermore, it sends a HAVE message within the same datagram, announcing that it has locally available the first chunk of content.

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 round trips, the first two datagrams MAY also contain some minor payload, e.g. the HAVE message.

The initiating peer MAY send a request for the chunks of content it wants to retrieve from the receiving peer, e.g. the first chunk announced during the handshake. It always precedes the message with the channel ID of the peer it is communicating with (e.g. 0x00000008 in our example), as described in Section 3.11. Furthermore, it MAY add additional messages such as a PEX_REQ.



When receiving the third datagram, both peers have the proof they really talk to each other; the three-way handshake is complete. The receiving peer responds to the request by sending a DATA message containing the requested content.



The DATA message consists of:

The 32-bit chunk range: 0,0 (the first chunk).

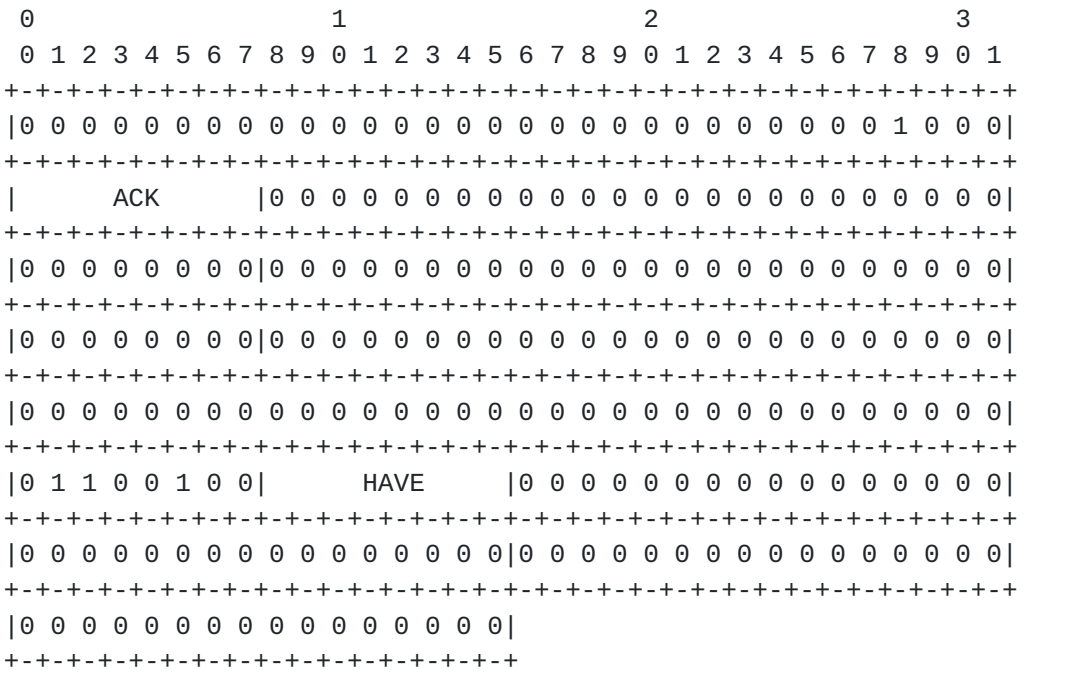
The timestamp value: 0004e94180b7db44

The Data message: 48656c6c6f20776f726c6421 (the "Hello world!" file)

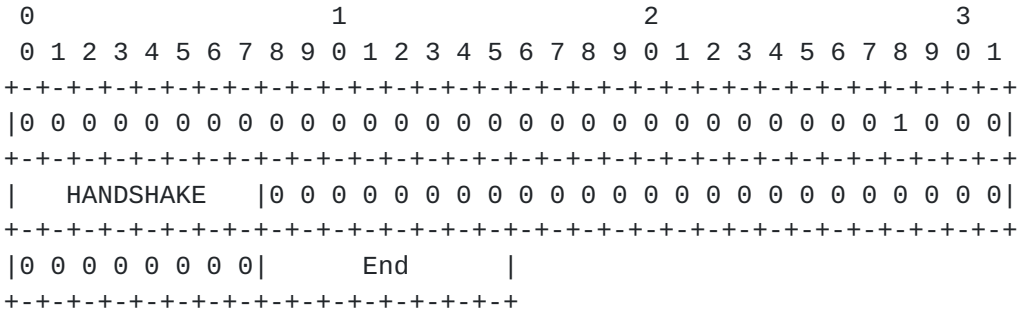
Note that the above datagram does not include the INTEGRITY message, as the entire content can fit into a single message, hence the

initiating peer is able to verify it against the root hash. Also, in this example the peer does not respond to the PEX_REQ as it does not know any third peer participating in the swarm.

Upon receiving the requested data, the initiating peer responds with an acknowledgement message for the first chunk, containing a one way delay sample (100ms). Furthermore it also adds a HAVE message for the chunk.



At this point the initiating peer has successfully retrieved the entire file. It then explicitly closes the connection by sending a HANDSHAKE message that contains an all 0-zeros channel ID.



9. Extensibility

9.1. Chunk Picking Algorithms

Chunk (or piece) picking entirely depends on the receiving peer. The sender peer is made aware of preferred chunks by the means of REQUEST messages. In some (live) scenarios it may be beneficial to allow the sender to ignore those hints and send unrequested data.

The chunk picking algorithm is external to the PPSPP protocol and will generally be a pluggable policy that uses the mechanisms provided by PPSPP. The algorithm will handle the choices made by the user consuming the content, such as seeking, switching audio tracks or subtitles. Example policies for P2P streaming can be found in [BITOS], and [EPLIVEPERF].

9.2. Reciprocity Algorithms

The role of reciprocity algorithms in peer-to-peer systems is to promote client contribution and prevent freeriding. A peer is said to be freeriding if it only downloads content but never uploads to others. Examples of reciprocity algorithms are tit-for-tat as used in BitTorrent [TIT4TAT] and Give-to-Get [GIVE2GET]. In PPSPP, reciprocity enforcement is the sole responsibility of the sender peer.

10. Acknowledgements

Arno Bakker, Riccardo Petrocco and Victor Grishchenko 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 PPSPP protocol was designed by Victor Grishchenko at Technische Universiteit Delft. The authors would like to thank the following people for their contributions to this draft: the chairs (Martin Stiemerling, Yunfei Zhang, Stefano Previdi, Ning Zong) and members of the IETF PPSP working group, and Mihai Capota, Raul Jimenez, Flutra Osmani, Johan Pouwelse, and Raynor Vliegendhart.

11. IANA Considerations

IANA is to create the new registries defined below for the extensibility of the protocol. For all registries, assignments consist of a name and its associated value. Also for all registries, the "Unassigned" ranges designated are governed by the policy 'IETF Review' as described in [RFC5226].

11.1. PPSP Peer Protocol Message Type Registry

Registry name is "PPSP Peer Protocol Message Type Registry". Values are integers in the range 0-255, with initial assignments and reservations given in Table 7.

11.2. PPSP Peer Protocol Option Registry

Registry name is "PPSP Peer Protocol Option Registry". Values are integers in the range 0-255, with initial assignments and reservations given in Table 2.

11.3. PPSP Peer Protocol Version Number Registry

Registry name is "PPSP Peer Protocol Version Number Registry". Values are integers in the range 0-255, with initial assignments and reservations given in Table 3.

11.4. PPSP Peer Protocol Content Integrity Protection Method Registry

Registry name is "PPSP Peer Protocol Content Integrity Protection Method Registry". Values are integers in the range 0-255, with initial assignments and reservations given in Table 4.

11.5. PPSP Peer Protocol Merkle Hash Tree Function Registry

Registry name is "PPSP Peer Protocol Merkle Hash Tree Function Registry". Values are integers in the range 0-255, with initial assignments and reservations given in Table 5.

11.6. PPSP Peer Protocol Chunk Addressing Method Registry

Registry name is "PPSP Peer Protocol Chunk Addressing Method Registry". Values are integers in the range 0-255, with initial assignments and reservations given in Table 6.

12. Manageability Considerations

This section presents operations and management considerations

following the checklist in [RFC5706], Appendix A.

In this section "PPSPP client" is defined as a PPSPP peer acting on behalf of an end user which may not yet have a copy of the content, and "PPSPP server" as a PPSPP peer that provides the initial copies of the content to the swarm on behalf of a content provider.

12.1. Operations

12.1.1. Installation and Initial Setup

A content provider wishing to use PPSPP to distribute content should setup at least one PPSPP server. PPSPP servers need to have access to either some static content or to some live audio/video sources. To provide flexibility for implementors, this configuration process is not standardized. The output of this process will be a list of metadata records, one for each swarm. A metadata record consists of the swarm ID, the chunk size used, the chunk addressing method used, the content integrity protection method used, and the Merkle hash tree function used (if applicable). If automatic content size detection (see Section 5.6) is not used, the content length is also part of the metadata record for static content. Note the swarm ID already contains the Live Signature Algorithm used, in case of a live stream.

In addition, a content provider should setup a tracking facility for the content by configuring, for example, a PPSP tracker [I-D.ietf-ppsp-base-tracker-protocol] or a Distributed Hash Table. The output of the latter process is a list of transport addresses for the tracking facility.

The list of metadata records of available content, and transport address for the tracking facility, can be distributed to users in various ways. Typically, they will be published on a Web site as links. When a user clicks such a link the PPSPP client is launched, either as a standalone application or by invoking the browser's internal PPSPP protocol handler, as exemplified in Section 2. The clients use the tracking facility to obtain the transport address of the PPSPP server(s) and other peers from the swarm, executing the peer protocol to retrieve and redistribute the content. The format of the PPSPP URLs should be defined in an extension document. The default protocol options should be exploited to keep the URLs small.

The minimal information a tracking facility must return when queried for a list of peers for a swarm is as follows. Assuming the communication between tracking facility and requestor is protected, the facility must at least return for each peer in the list its IP address, transport protocol identifier (i.e., UDP), and transport

protocol port number.

12.1.2. Requirements on Other Protocols and Functional Components

When using the PPSP tracker protocol, PPSPP requires a specific behavior from this protocol for security reasons, as detailed in Section 13.2.

12.1.3. Migration Path

This document does not detail a migration path since there is no previous standard protocol providing similar functionality.

12.1.4. Impact on Network Operation

PPSPP is a peer-to-peer protocol that takes advantage of the fact that content is available from multiple sources to improve robustness, scalability and performance. At the same time, poor choices in determining which exact sources to use can lead to bad experience for the end user and high costs for network operators. Hence, PPSPP can benefit from the ALTO protocol to steer peer selection, as described in Section 3.10.1.

12.1.5. Verifying Correct Operation

PPSPP is operating correctly when all peers obtain the desired content on time. Therefore the PPSPP client is the ideal location to verify the protocol's correct operation. However, it is not feasible to mandate logging the behavior of PPSPP peers in all implementations and deployments, for example, due to privacy reasons. There are two alternative options:

- Monitoring the PPSPP servers initially providing the content, using standard metrics such as bandwidth usage, peer connections and activity, can help identify trouble, see next section and [RFC2564].

- The PPSP tracker protocol may be used to gather information about all peers in a swarm, to obtain a global view of operation, according to [RFC6972] (requirement PPSP-TP-REQ-3).

Basic operation of the protocol can be easily verified when a tracker and swarm ID are known by starting a PPSPP download. Deep packet inspection for DATA and ACK messages help to establish that actual content transfer is happening and that the chunk availability signaling and integrity checking are working.

12.1.6. Configuration

Table 8 shows the PPSPP parameters, their defaults and where the parameter is defined. For parameters that have no default, the table row contains the word "var" and refers to the section discussing the considerations to make when choosing a value.

Name	Default	Definition
Chunk Size	var, 1024 bytes recommended	Section 8.1
Static Content Integrity Protection Method	1 (Merkle Hash Tree)	Section 7.5
Live Content Integrity Protection Method	3 (Unified Merkle Tree)	Section 7.5
Merkle Hash Tree Function	0 (SHA1)	Section 7.6
Live Signature Algorithm	13 (ECDSAP256SHA256)	Section 7.7
Chunk Addressing Method	2 (32-bit chunk ranges)	Section 7.8
Live Discard Window	var	Section 6.2, Section 7.9
NCHUNKS_PER_SIG	var	Section 6.1.2.1
Dead peer detection	No reply in 3 minutes + 3 datagrams	Section 8.15

Table 8: PPSPP Defaults

12.2. Management Considerations

The management considerations for PPSPP are very similar to other protocols that are used for large-scale content distribution, in particular HTTP. How does one manage large numbers of servers? How does one push new content out to a server farm and allows staged releases? How to detect faults and how to measure servers and end-user performance? As standard solutions to these challenges are still being developed, this section cannot provide a definitive recommendation on how PPSPP should be managed. Hence, it describes the standard solutions available at this time, and assumes a future extension document will provide more complete guidelines.

12.2.1. Management Interoperability and Information

As just stated, PPSPP servers providing initial copies of the content are akin to WWW and FTP servers. They can also be deployed in large numbers and thus can benefit from standard management facilities. PPSPP servers may therefore implement an SNMP management interface based on the APPLICATION-MIB [RFC2564], where the file object can be used to report on swarms.

What is missing is the ability to remove or rate limit specific PPSPP swarms on a server. This corresponds to removing or limit specific virtual servers on a Web server. In other words, as multiple pieces of content (swarms, virtual WWW servers) are multiplexed onto a single server process, more fine-grained management of that process is required. This functionality is currently missing.

Logging is an important functionality for PPSPP servers and, depending on the deployment, PPSPP clients. Logging should be done via syslog [RFC5424].

12.2.2. Fault Management

The facilities for verifying correct operation and server management (just discussed) appear sufficient for PPSPP fault monitoring. This can be supplemented with host resource [RFC2790] and UDP/IP network monitoring [RFC4113], as PPSPP server failures can generally be attributed directly to conditions on the host or network.

Since PPSPP has been designed to work in a hostile environment, many benign faults will be handled by the mechanisms used for managing attacks. For example, when a malfunctioning peer starts sending the wrong chunks, this is detected by the content integrity protection mechanism and another source is sought.

12.2.3. Configuration Management

Large-scale deployments may benefit from a standard way of replicating a new piece of content on a set of initial PPSPP servers. This functionality may need to include controlled releasing, such that content becomes available only at a specific point in time (e.g. the release of a movie trailer). This functionality could be provided via NETCONF [RFC6241], to enable atomic configuration updates over a set of servers. Uploading the new content could be one configuration change, making the content available for download by the public another.

12.2.4. Accounting Management

Content providers may offer PPSPP hosting for different customers and will want to bill these customers, for example, based on bandwidth usage. This situation is a common accounting scenario, similar to billing per virtual server for Web servers. PPSPP can therefore benefit from general standardization efforts in this area [RFC2975] when they come to fruition.

12.2.5. Performance Management

Depending on the deployment scenarios, the application performance measurement facilities of [RFC3729] and associated [RFC4150] can be used with PPSPP.

In addition, when the PPSPP tracker protocol is used, it provides a built-in, application-level, performance measurement infrastructure for different metrics. See [RFC6972] (requirement PPSP-TP-REQ-3).

12.2.6. Security Management

Malicious peers should ideally be locked out long-term. This is primarily for performance reasons, as the protocol is robust against attacks (see next section). Section 13.7 describes a procedure for long-term exclusion.

13. Security Considerations

As any other network protocol, the PPSPP 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. This section discusses the protocol's security considerations in detail.

13.1. Security of the Handshake Procedure

Borrowing from the analysis in [RFC5971], the PPSP peer protocol may be attacked with 3 types of denial-of-service attacks:

1. DOS amplification attack: attackers try to use a PPSPP peer to generate more traffic to a victim.

2. DOS flood attack: attackers try to deny service to other peers by allocating lots of state at a PPSPP peer.

3. Disrupt service to an individual peer: attackers send bogus e.g. REQUEST and HAVE messages appearing to come from victim peer A to the peers B1..Bn serving that peer. This causes A to receive chunks it did not request or to not receive the chunks it requested.

The basic scheme to protect against these attacks is the use of a secure handshake procedure. In the UDP encapsulation the handshake procedure is secured by the use of randomly chosen channel IDs as follows. The channel IDs must be generated following the requirements in [RFC4960] (Sec. 5.1.3).

When UDP is used, all datagrams carrying PPSPP messages are prefixed with a 4-byte channel ID. These channel IDs are random numbers, established during the handshake phase as follows. Peer A initiates an exchange with peer B by sending a datagram containing a HANDSHAKE message prefixed with the channel ID consisting of all 0s. Peer A's HANDSHAKE contains a randomly chosen channel ID, chanA:

A->B: chan0 + HANDSHAKE(chanA) + ...

When peer B receives this datagram, it creates some state for peer A, that at least contains the channel ID chanA. Next, peer B sends a response to A, consisting of a datagram containing a HANDSHAKE message prefixed with the chanA channel ID. Peer B's HANDSHAKE contains a randomly chosen channel ID, chanB.

B->A: chanA + HANDSHAKE(chanB) + ...

Peer A now knows that peer B really responds, as it echoed chanA. So the next datagram that A sends may already contain heavy payload, i.e., a chunk. This next datagram to B will be prefixed with the chanB channel ID. When B receives this datagram, both peers have the proof they are really talking to each other, the three-way handshake is complete. In other words, the randomly chosen channel IDs act as tags (cf. [RFC4960] (Sec. 5.1)).

A->B: chanB + HAVE + DATA + ...

13.1.1. Protection Against Attack 1

In short, PPSPP does a so-called return routability check before heavy payload is sent. This means that attack 1 is fended off: PPSPP does not send back much more data than it received, unless it knows it is talking to a live peer. Attackers sending a spoofed HANDSHAKE to B pretending to be A now need to intercept the message from B to A to get B to send heavy payload, and ensure that that heavy payload goes to the victim, something assumed too hard to be a practical

attack.

Note the rule is that no heavy payload may be sent until the third datagram. This has implications for PPSPP implementations that use chunk addressing schemes that are verbose. If a PPSPP implementation uses large bitmaps to convey chunk availability these may not be sent by peer B in the second datagram.

13.1.2. Protection Against Attack 2

On receiving the first datagram peer B will record some state about peer A. At present this state consists of the chanA channel ID, and the results of processing the other messages in the first datagram. In particular, if A included some HAVE messages, B may add a chunk availability map to A's state. In addition, B may request some chunks from A in the second datagram, and B will maintain state about these outgoing requests.

So presently, PPSPP is somewhat vulnerable to attack 2. An attacker could send many datagrams with HANDSHAKEs and HAVEs and thus allocate state at the PPSPP peer. Therefore peer A MUST respond immediately to the second datagram, if it is still interested in peer B.

The reason for using this slightly vulnerable three-way handshake instead of the safer handshake procedure of SCTP [RFC4960] (Sec. 5.1) is quicker response time for the user. In the SCTP procedure, peer A and B cannot request chunks until datagrams 3 and 4 respectively, as opposed to 2 and 1 in the proposed procedure. This means that the user has to wait shorter in PPSPP between starting the video stream and seeing the first images.

13.1.3. Protection Against Attack 3

In general, channel IDs serve to authenticate a peer. Hence, to attack, a malicious peer T would need to be able to eavesdrop on conversations between victim A and a benign peer B to obtain the channel ID B assigned to A, chanB. Furthermore, attacker T would need to be able to spoof e.g. REQUEST and HAVE messages from A to cause B to send heavy DATA messages to A, or prevent B from sending them, respectively.

The capability to eavesdrop is not common, so the protection afforded by channel IDs will be sufficient in most cases. If not, point-to-point encryption of traffic should be used, see below.

13.2. Secure Peer Address Exchange

As described in Section 3.10, a peer A can send Peer-Exchange messages PEX_RES to a peer B, which contain the IP address and port of other peers that are supposedly also in the current swarm. The strength of this mechanism is that it allows decentralized tracking: after an initial bootstrap no central tracker is needed anymore. The vulnerability of this mechanism (and DHTs) is that malicious peers can use it for an Amplification attack.

In particular, a malicious peer T could send PEX_RES messages to well-behaved peer A with addresses of peers B1,B2,...,BN and on receipt, peer A could send a HANDSHAKE to all these peers. So in the worst case, a single datagram results in N datagrams. The actual damage depends on A's behavior. E.g. when A already has sufficient connections it may not connect to the offered ones at all, but if it is a fresh peer it may connect to all directly.

In addition, PEX can be used in Eclipse attacks [ECLIPSE] where malicious peers try to isolate a particular peer such that it only interacts with malicious peers. Let us distinguish two specific attacks:

E1. Malicious peers try to eclipse the single injector in live streaming.

E2. Malicious peers try to eclipse a specific consumer peer.

Attack E1 has the most impact on the system as it would disrupt all peers.

13.2.1. Protection against the Amplification Attack

If peer addresses are relatively stable, strong protection against the attack can be provided by using public key cryptography and certification. In particular, a PEX_REScert message will carry swarm-membership certificates rather than IP address and port. A membership certificate for peer B states that peer B at address (ipB,portB) is part of swarm S at time T and is cryptographically signed. The receiver A can check the cert for a valid signature, the right swarm and liveliness and only then consider contacting B. These swarm-membership certificates correspond to signed node descriptors in secure decentralized peer sampling services [SPS].

Several designs are possible for the security environment for these membership certificates. That is, there are different designs possible for who signs the membership certificates and how public keys are distributed. As an example, we describe a design where the

PPSP tracker acts as certification authority.

13.2.2. Example: Tracker as Certification Authority

A peer A wanting to join swarm S sends a certificate request message to a tracker X for that swarm. Upon receipt, the tracker creates a membership certificate from the request with swarm ID S, a timestamp T and the external IP and port it received the message from, signed with the tracker's private key. This certificate is returned to A.

Peer A then includes this certificate when it sends a PEX_REScert to peer B. Receiver B verifies it against the tracker public key. This tracker public key should be part of the swarm's metadata, which B received from a trusted source. Subsequently, peer B can send the member certificate of A to other peers in PEX_REScert messages.

Peer A can send the certification request when it first contacts the tracker, or at a later time. Furthermore, the responses the tracker sends could contain membership certificates instead of plain addresses, such that they can be gossiped securely as well.

We assume the tracker is protected against attacks and does a return routability check. The latter ensures that malicious peers cannot obtain a certificate for a random host, just for hosts where they can eavesdrop on incoming traffic.

The load generated on the tracker depends on churn and the lifetime of a certificate. Certificates can be fairly long lived, given that the main goal of the membership certificates is to prevent that malicious peer T can cause good peer A to contact *random* hosts. The freshness of the timestamp just adds extra protection in addition to achieving that goal. It protects against malicious hosts causing a good peer A to contact hosts that previously participated in the swarm.

The membership certificate mechanism itself can be used for a kind of amplification attack against good peers. Malicious peer T can cause peer A to spend some CPU to verify the signatures on the membership certificates that T sends. To counter this, A SHOULD check a few of the certificates sent and discard the rest if they are defective.

The same membership certificates described above can be registered in a Distributed Hash Table that has been secured against the well-known DHT specific attacks [SECDHTS].

Note that this scheme does not work for peers behind a symmetric Network Address Translator, but neither does normal tracker registration.

13.2.3. Protection Against Eclipse Attacks

Before we can discuss Eclipse attacks we first need to establish the security properties of the central tracker. A tracker is vulnerable to Amplification attacks too. A malicious peer T could register a victim B with the tracker, and many peers joining the swarm will contact B. Trackers can also be used in Eclipse attacks. If many malicious peers register themselves at the tracker, the percentage of bad peers in the returned address list may become high. Leaving the protection of the tracker to the PPSP tracker protocol specification, we assume for the following discussion that it returns a true random sample of the actual swarm membership (achieved via Sybil attack protection). This means that if 50% of the peers is bad, you'll still get 50% good addresses from the tracker.

Attack E1 on PEX can be fended off by letting live injectors disable PEX. Or at least, let live injectors ensure that part of their connections are to peers whose addresses came from the trusted tracker.

The same measures defend against attack E2 on PEX. They can also be employed dynamically. When the current set of peers B that peer A is connected to doesn't provide good quality of service, A can contact the tracker to find new candidates.

13.3. Support for Closed Swarms (PPSP.SEC.REQ-1)

The Closed Swarms [CLOSED] and Enhanced Closed Swarms [ECS] mechanisms provide swarm-level access control. The basic idea is that a peer cannot download from another peer unless it shows a Proof-of-Access. Enhanced Closed Swarms improve on the original Closed Swarms by adding on-the-wire encryption against man-in-the-middle attacks and more flexible access control rules.

The exact mapping of ECS to PPSPP is defined in [I-D.gabrijelcic-ppsp-ecs].

13.4. Confidentiality of Streamed Content (PPSP.SEC.REQ-2+3)

No extra mechanism is needed to support confidentiality in PPSPP. A content publisher wishing confidentiality should just distribute content in cyphertext / DRM-ed format. In that case it is assumed a higher layer handles key management out-of-band. Alternatively, pure point-to-point encryption of content and traffic can be provided by the proposed Closed Swarms access control mechanism, or by DTLS [RFC6347] or IPsec [RFC4301].

13.5. Strength of the Hash Function for Merkle Hash Trees

Implementations MUST support SHA1 as the hash function for content integrity protection via Merkle Hash trees. SHA1 is preferred over stronger hash functions for two reasons. First, it reduces on-the-wire overhead. Second, few implementations need the extra strength of other functions because the function is used in a hash tree. In particular, if attackers manage to find a collision for a hash it can replace just one chunk, so the impact is limited. If fixed sized chunks are used, the collision has to be of the same size as the original chunk. For hashes higher up in the hash tree, a collision must be a concatenation of two hashes. In sum, finding collisions that fit with the hash tree are generally harder to find than regular SHA1 collisions, which are, at the time of writing, still hard to find.

13.6. Limit Potential Damage and Resource Exhaustion by Bad or Broken Peers (PPSP.SEC.REQ-4+6)

In this section an analysis is given of the potential damage a malicious peer can do with each message in the protocol, and how it is prevented by the protocol (implementation).

13.6.1. HANDSHAKE

- Secured against DoS amplification attacks as described in Section 13.1.

- Threat HS.1: An Eclipse attack where peers T1..Tn fill all connection slots of A by initiating the connection to A.

  Solution: Peer A must not let other peers fill all its available connection slots, i.e., A must initiate connections itself too, to prevent isolation.

13.6.2. HAVE

- Threat HAVE.1: Malicious peer T can claim to have content which it hasn't. Subsequently T won't respond to requests.

  Solution: peer A will consider T to be a slow peer and not ask it again.

- Threat HAVE.2: Malicious peer T can claim not to have content. Hence it won't contribute.

  Solution: Peer and chunk selection algorithms external to the protocol will implement fairness and provide sharing incentives.

13.6.3. DATA

- Threat DATA.1: peer T sending bogus chunks.

  Solution: The content integrity protection schemes defend against this.

- Threat DATA.2: peer T sends peer A unrequested chunks.

  To protect against this threat we need network-level DoS prevention.

13.6.4. ACK

- Threat ACK.1: peer T acknowledges wrong chunks.

  Solution: peer A will detect inconsistencies with the data it sent to T.

- Threat ACK.2: peer T modifies timestamp in ACK to peer A used for time-based congestion control.

  Solution: In theory, by decreasing the timestamp peer T could fake there is no congestion when in fact there is, causing A to send more data than it should. [RFC6817] does not list this as a security consideration. Possibly this attack can be detected by the large resulting asymmetry between round-trip time and measured one-way delay.

13.6.5. INTEGRITY and SIGNED_INTEGRITY

- Threat INTEGRITY.1: An amplification attack where peer T sends bogus INTEGRITY or SIGNED_INTEGRITY messages, causing peer A to checks hashes or signatures, thus spending CPU unnecessarily.

  Solution: If the hashes/signatures don't check out A will stop asking T because of the atomic datagram principle and the content integrity protection. Subsequent unsolicited traffic from T will be ignored.

- Threat INTEGRITY.2: An attack where peer T sends old SIGNED_INTEGRITY messages in the Unified Merkle Tree scheme, trying to make peer A tune in at a past point in the live stream.

  Solution: The timestamp in the SIGNED_INTEGRITY message protects against such replays. Subsequent traffic from T will be ignored.

13.6.6. REQUEST

- Threat REQUEST.1: peer T could request lots from A, leaving A without resources for others.

  Solution: A limit is imposed on the upload capacity a single peer can consume, for example, by using an upload bandwidth scheduler that takes into account the need of multiple peers. A natural upper limit of this upload quotum is the bitrate of the content, taking into account that this may be variable.

13.6.7. CANCEL

- Threat CANCEL.1: peer T sends CANCEL messages for content it never requested to peer A.

  Solution: peer A will detect the inconsistency of the messages and ignore them. Note that CANCEL messages may be received unexpectedly when a transport is used where REQUEST messages may be lost or reordered with respect to the subsequent CANCELs.

13.6.8. CHOKE

- Threat CHOKE.1: peer T sends REQUEST messages after peer A sent B a CHOKE message.

  Solution: peer A will just discard the unwanted REQUESTs and resend the CHOKE, assuming it got lost.

13.6.9. UNCHOKE

- Threat UNCHOKE.1: peer T sends an UNCHOKE message to peer A without having sent a CHOKE message before.

  Solution: peer A can easily detect this violation of protocol state, and ignore it. Note this can also happen due to loss of a CHOKE message sent by a benign peer.

- Threat UNCHOKE.2: peer T sends an UNCHOKE message to peer A, but subsequently does not respond to its REQUESTs.

  Solution: peer A will consider T to be a slow peer and not ask it again.

13.6.10. PEX_RES

- Secured against amplification and Eclipse attacks as described in Section 13.2.

13.6.11. Unsolicited Messages in General

- Threat: peer T could send a spoofed PEX_REQ or REQUEST from peer B to peer A, causing A to send a PEX_RES/DATA to B.

  Solution: the message from peer T won't be accepted unless T does a handshake first, in which case the reply goes to T, not victim B.

13.7. Exclude Bad or Broken Peers (PPSP.SEC.REQ-5)

A receiving peer can detect malicious or faulty senders as just described, which it can then subsequently ignore. However, excluding such a bad peer from the system completely is complex. Random monitoring by trusted peers that would blacklist bad peers as described in [DETMAL] is one option. This mechanism does require extra capacity to run such trusted peers, which must be indistinguishable from regular peers, and requires a solution for the timely distribution of this blacklist to peers in a scalable manner.

14. References

14.1. Normative References

[CCITT.X208.1988]
International International Telephone and Telegraph Consultative Committee, "Specification of Abstract Syntax Notation One (ASN.1)", CCITT Recommendation X.208, November 1988.

[FIPS180-4]
Information Technology Laboratory, National Institute of Standards and Technology, "Federal Information Processing Standards: Secure Hash Standard (SHS)", Publication 180-4, Mar 2012.

[IANADNSSECALGNUM]
IANA, "Domain Name System Security (DNSSEC) Algorithm Numbers", <http://www.iana.org/assignments/dns-sec-alg-numbers>.

[RFC1918] Rekhter, Y., Moskowitz, R., Karrenberg, D., Groot, G., and

E. Lear, "Address Allocation for Private Internets", BCP 5, RFC 1918, February 1996.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3110] Eastlake, D., "RSA/SHA-1 SIGs and RSA KEYs in the Domain Name System (DNS)", RFC 3110, May 2001.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC4034] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Resource Records for the DNS Security Extensions", RFC 4034, March 2005.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, February 2006.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.

[RFC5905] Mills, D., Martin, J., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, June 2010.

[RFC6605] Hoffman, P. and W. Wijngaards, "Elliptic Curve Digital Signature Algorithm (DSA) for DNSSEC", RFC 6605, April 2012.

[RFC6817] Shalunov, S., Hazel, G., Iyengar, J., and M. Kuehlewind, "Low Extra Delay Background Transport (LEDBAT)", RFC 6817, December 2012.

14.2. Informative References

[ABMRKL] Bakker, A., "Merkle hash torrent extension", BitTorrent Enhancement Proposal 30, Mar 2009, <http://bittorrent.org/beps/bep_0030.html>.

[BINMAP] Grishchenko, V. and J. Pouwelse, "Binmaps: hybridizing bitmaps and binary trees", Technical Report PDS-2011-005, Parallel and Distributed Systems Group, Fac. of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, The Netherlands,

Apr 2009.

[BITOS] Vlavianos, A., Iliofotou, M., Mathieu, F., and M. Faloutsos, "BiToS: Enhancing BitTorrent for Supporting Streaming Applications", IEEE INFOCOM Global Internet Symposium Barcelona, Spain, Apr 2006.

[BITTORRENT] Cohen, B., "The BitTorrent Protocol Specification", BitTorrent Enhancement Proposal 3, Feb 2008, <http://bittorrent.org/beps/bep_0003.html>.

[CLOSED] Borch, N., Mitchell, K., Arntzen, I., and D. Gabrijelcic, "Access Control to BitTorrent Swarms Using Closed Swarms", ACM workshop on Advanced Video Streaming Techniques for Peer-to-Peer Networks and Social Networking (AVSTP2P '10), Florence, Italy, Oct 2010, <http://doi.acm.org/10.1145/1877891.1877898>.

[DETMAL] Shetty, S., Galdames, P., Tavanapong, W., and Ying. Cai, "Detecting Malicious Peers in Overlay Multicast Streaming", IEEE Conference on Local Computer Networks (LCN'06). Tampa, FL, USA, Nov 2006.

[ECLIPSE] Sit, E. 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, 2002.

[ECS] Jovanovikj, V., Gabrijelcic, D., and T. Klobucar, "Access Control in BitTorrent P2P Networks Using the Enhanced Closed Swarms Protocol", International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2011), pp. 97-102, Nice, France, Aug 2011.

[EPLIVEPERF] Bonald, T., Massoulie, L., Mathieu, F., Perino, D., and A. Twigg, "Epidemic Live Streaming: Optimal Performance Trade-offs", Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems Annapolis, MD, USA, Jun 2008.

[GIVE2GET] Mol, J., Pouwelse, J., Meulpolder, M., Epema, D., and H. Sips, "Give-to-Get: Free-riding Resilient Video-on-demand in P2P Systems", Proceedings Multimedia Computing and Networking conference (Proceedings of SPIE Vol. 6818) San

Jose, California, USA, Jan 2008.

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

[I-D.gabrijelcic-ppsp-ecs] Gabrijelcic, D., "Enhanced Closed Swarm protocol", draft-ppsp-gabrijelcic-ecs (work in progress), November 2012.

[I-D.ietf-alto-protocol] Alimi, R., Penno, R., and Y. Yang, "ALTO Protocol", draft-ietf-alto-protocol-27 (work in progress), March 2014.

[I-D.ietf-ppsp-base-tracker-protocol] Cruz, R., Nunes, M., Yingjie, G., Xia, J., Taveira, J., and D. Lingli, "PPSP Tracker Protocol-Base Protocol (PPSP-TP/1.0)", draft-ietf-ppsp-base-tracker-protocol-03 (work in progress), December 2013.

[JIM11] Jimenez, R., Osmani, F., and B. Knutsson, "Sub-Second Lookups on a Large-Scale Kademlia-Based Overlay", IEEE International Conference on Peer-to-Peer Computing (P2P'11), Kyoto, Japan, Aug 2011.

[LBT] Rossi, D., Testa, C., Valenti, S., and L. Muscariello, "LEDBAT: the new BitTorrent congestion control protocol", Computer Communications and Networks (ICCCN), Zurich, Switzerland, Aug 2010.

[LCOMPL] Testa, C. and D. Rossi, "On the impact of uTP on BitTorrent completion time", IEEE International Conference on Peer-to-Peer Computing (P2P'11), Kyoto, Japan, Aug 2011.

[MERKLE] Merkle, R., "Secrecy, Authentication, and Public Key Systems", Ph.D. thesis Dept. of Electrical Engineering, Stanford University, CA, USA, pp 40-45, 1979.

[POLLIVE] Dhungel, P., Hei, Xiaojun., Ross, K., and N. Saxena, "Pollution in P2P Live Video Streaming", International Journal of Computer Networks & Communications (IJCNC) Vol.1, No.2, Jul 2009.

[PPSPPERF] Petrocco, R., Pouwelse, J., and D. Epema, "Performance

analysis of the Libswift P2P streaming protocol", IEEE International Conference on Peer-to-Peer Computing (P2P'12), Tarragona, Spain, Sept 2012.

[RFC2564] Kalbfleisch, C., Krupczak, C., Presuhn, R., and J. Saperia, "Application Management MIB", RFC 2564, May 1999.

[RFC2790] Waldbusser, S. and P. Grillo, "Host Resources MIB", RFC 2790, March 2000.

[RFC2975] Aboba, B., Arkko, J., and D. Harrington, "Introduction to Accounting Management", RFC 2975, October 2000.

[RFC3365] Schiller, J., "Strong Security Requirements for Internet Engineering Task Force Standard Protocols", BCP 61, RFC 3365, August 2002.

[RFC3729] Waldbusser, S., "Application Performance Measurement MIB", RFC 3729, March 2004.

[RFC4113] Fenner, B. and J. Flick, "Management Information Base for the User Datagram Protocol (UDP)", RFC 4113, June 2005.

[RFC4150] Dietz, R. and R. Cole, "Transport Performance Metrics MIB", RFC 4150, August 2005.

[RFC4301] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.

[RFC4821] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, March 2007.

[RFC4960] Stewart, R., "Stream Control Transmission Protocol", RFC 4960, September 2007.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[RFC5389] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing, "Session Traversal Utilities for NAT (STUN)", RFC 5389, October 2008.

[RFC5424] Gerhards, R., "The Syslog Protocol", RFC 5424, March 2009.

[RFC5706] Harrington, D., "Guidelines for Considering Operations and Management of New Protocols and Protocol Extensions", RFC 5706, November 2009.

[RFC5971] Schulzrinne, H. and R. Hancock, "GIST: General Internet Signalling Transport", RFC 5971, October 2010.

[RFC6241] Enns, R., Bjorklund, M., Schoenwaelder, J., and A. Bierman, "Network Configuration Protocol (NETCONF)", RFC 6241, June 2011.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, January 2012.

[RFC6709] Carpenter, B., Aboba, B., and S. Cheshire, "Design Considerations for Protocol Extensions", RFC 6709, September 2012.

[RFC6972] Zhang, Y. and N. Zong, "Problem Statement and Requirements of the Peer-to-Peer Streaming Protocol (PPSP)", RFC 6972, July 2013.

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

[SIGMCAST] Wong, C. and S. Lam, "Digital Signatures for Flows and Multicasts", IEEE/ACM Transactions on Networking 7(4), pp. 502-513, 1999.

[SPS] Jesi, G., Montresor, A., and M. van Steen, "Secure Peer Sampling", Computer Networks vol. 54(12), pp. 2086-2098, Elsevier, Aug 2010.

[SWIFTIMPL] Grishchenko, V., Paananen, J., Pronchenkov, A., Bakker, A., and R. Petrocco, "Swift reference implementation", 2014, <https://github.com/libswift/libswift>.

[TIT4TAT] Cohen, B., "Incentives Build Robustness in BitTorrent", 1st Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, USA, Jun 2003.

Appendix A. Revision History

-00 2011-12-19 Initial version.

-01 2012-01-30 Minor text revision:

* Changed heading to "A. Bakker"

* Changed title to *Peer* Protocol, and abbreviation PPSPP.

* Replaced swift with PPSPP.

* Removed Sec. 6.4. "HTTP (as PPSP)".

* Renamed Sec. 8.4. to "Chunk Picking Algorithms".

* Resolved Ticket #3: Removed sentence about random set of peers.

* Resolved Ticket #6: Added clarification to "Chunk Picking Algorithms" section.

* Resolved Ticket #11: Added Sec. 3.12 on Storage Independence

* Resolved Ticket #14: Added clarification to "Automatic Size Detection" section.

* Resolved Ticket #15: Operation section now states it shows example behaviour for a specific set of policies and schemes.

* Resolved Ticket #30: Explained why multiple REQUESTs in one datagram.

* Resolved Ticket #31: Renamed PEX_ADD message to PEX_RES.

* Resolved Ticket #32: Renamed Sec 3.8. to "Keep Alive Signaling", and updated explanation.

* Resolved Ticket #33: Explained NAT hole punching via only PPSPP messages.

* Resolved Ticket #34: Added section about limited overhead of the Merkle hash tree scheme.

-02 2012-04-17 Major revision

* Allow different chunk addressing and content integrity protection schemes (ticket #13):

* Added chunk ID, chunk specification, chunk addressing scheme, etc. to terminology.

* Created new Sections 4 and 5 discussing chunk addressing and content integrity protection schemes, respectively and moved relevant sections on bin numbering and Merkle hash trees there.

* Renamed Section 4 to "Merkle Hash Trees and The Automatic Detection of Content Size".

* Reformulated automatic size detection in terms of nodes, not bins.

* Extended HANDSHAKE message to carry protocol options and created Section 8 on Protocol options. VERSION and MSGTYPE_RCVD messages replaced with protocol options.

* Renamed HASH message to INTEGRITY.

* Renamed HINT to REQUEST.

* Added description of chunk addressing via (start,end) ranges.

* Resolved Ticket #26: Extended "Security Considerations" with section on the handshake procedure.

* Resolved Ticket #17: Defined recently as "in last 60 seconds" in PEX.

* Resolved Ticket #20: Extended "Security Considerations" with design to make Peer Address Exchange more secure.

* Resolved Ticket #38+39 / PPSP.SEC.REQ-2+3: Extended "Security Considerations" with a section on confidentiality of content.

* Resolved Ticket #40+42 / PPSP.SEC.REQ-4+6: Extended "Security Considerations" with a per-message analysis of threats and how PPSPP is protected from them.

* Progressed Ticket #41 / PPSP.SEC.REQ-5: Extended "Security Considerations" with a section on possible ways of excluding bad or broken peers from the system.

* Moved Rationale to Appendix.

* Resolved Ticket #43: Updated Live Streaming section to include "Sign All" content authentication, and reference to [SIGMCAST] following discussion with Fabio Picconi.

* Resolved Ticket #12: Added a CANCEL message to cancel REQUESTs for the same data that were sent to multiple peers at the same time in time-critical situations.

-03 2012-10-22 Major revision

* Updated Abstract and Introduction, removing download case.

* Resolved Ticket #4: Added explicit CHOKE/UNCHOKE messages.

* Removed directory lists unused in streaming.

* Resolved Ticket #22, #23, #28: Failure behaviour, error codes and dealing with peer crashes.

* Resolved Ticket #13: Chunk ranges are the default chunk addressing scheme that all peers MUST support.

* Added a section on compatibility between chunk addressing schemes.

* Expanded the explanation of Unified Merkle Trees as a method for content integrity protection for live streams.

* Added a section on forgetting chunks in live streaming.

* Added "End" option to protocol options and corrected bugs in UDP encapsulation, following Karl Knutsson's comments.

* Added SHA-2 support for Merkle Hash functions.

* Added content integrity protection methods for live streaming to the relevant protocol option.

* Added a Live Signature Algorithm protocol option.

* Resolved Ticket #24+27: The choice for UDP + LEDBAT as transport has now been reflected in the draft. TCP and RTP encapsulations have been removed.

* Superfluous parts of Section 10 on extensibility have been removed.

* Removed appendix with Rationale.

* Resolved Ticket #21+25: PPSPP currently uses LEDBAT and the DATA and ACK messages now contain the time fields it requires. Should other congestion control algorithms be

supported in the future, a protocol option will be added.

-04 2012-11-07 Minor revision

* Corrected typos.

* Added empty protocol option list when HANDSHAKE is used for explicitly closing a channel in the UDP encapsulation.

* Corrected definition of a range chunk specification to be a single (start,end) pair. To send multiple disjunct ranges multiple messages should be used.

* Clarified that in a range chunk specification the end is inclusive. I.e., [start,end] not [start,end)

* Added PEX_REScert message to carry a membership certificate. Renamed PEX_RES to PEX_RESv4.

* Added a guideline about private and link-local addresses in PEX_RES messages.

* Defined the format of the public key that is used as swarm ID in live streaming.

* Clarified that a HANDSHAKE message must be the first message in a datagram.

* Clarified sending INTEGRITY messages ahead in a separate datagram if not all necessary hashes that still need to be sent and the chunk fit into a single datagram. Defined an order for the INTEGRITY messages.

* Clarified rare case of sending multiple DATA messages in one datagram.

* Clarified UDP datagrams carrying PPSPP should adhere to the network's MTU to avoid IP fragmentation.

* Defined value for version protocol option.

* Added small clarifications and corrected typos.

* Extended versioning scheme to Min/max versioning scheme defined in [RFC6709], Section 4.1, following Riccardo Bernardini's suggestion.

* Processed comments on unclear phrasing from Riccardo Bernardini.

* Added a guideline on when to declare a peer dead.

* Made sure all essential references are listed as Normative references following RFC3967.

-05 2013-01-23 Minor revision

* Corrected category to Standards Track.

* Clarified that swarm identifier is a required protocol option in an initiating HANDSHAKE in the UDP encapsulation.

* Added IANA considerations and tablised name spaces for registry definition.

-06 2013-02-11 Minor revision

* Updated "Overall Operation" to have more context (HTML5 video).

* Clarified wording on PEX_REQ.

* Clarified wording on SIGNED_INTEGRITY.

* Added a reference on how ALTO can be used with PPSPP.

* Added Manageability Consideration section following RFC5706.

* Clarified that implementations SHOULD implement the "Unified Merkle Tree" content integrity protection method for live, and MAY implement "Sign All".

* Made SHA1 hash function mandatory-to-implement as Merkle Tree Hash function and explained the security considerations.

* Made RSA/SHA1 mandatory-to-implement as Live Signature Algorithm for integrity protection while live streaming.

* Clarified that implementations MUST implement addressing via 32-bit chunk ranges.

* Made LEDBAT an Informational reference to prevent a so-called "down ref".

* Updated reference to PPSP problem statement and requirements document.

* Used kibibyte unit in formal sections.

-07 2013-06-19 Revision following AD Review

Quoting the AD review by Martin Stiemerling: ***High-level issues:

1) Merkle Hash Trees I have found the document very confusing on whether Merkle Hash Trees (MHTs) and the for the MHT required bin numbering scheme are now optional or mandatory. Parts of the draft make the impression that either of them or both or optional (mainly in the beginning of the document), while Section 5 and later Sections are relying heavily on MHTs. My naive reading of the current draft is that you could rely on start-end ranges for chunk addressing and MHTs for content protection. However, I do know that this combination is not working. If MHTs are really optional, including the bin numbering, the document should really state this and make clear what the operations of the protocol are with the mandatory to implement (MTI) mechanisms. The MHT, bins, and all the protocol handling should go in an appendix. There is a call to make for the WG: I do know that MHTs were considered by some as burden and they have called for a leaner way, i.e., the start-end ranges. The call for the leaner way has been implemented in the document but not fully.

+ The text now states that MHTs SHOULD be used unless in benign environments and are mandatory-to-implement. It also states that only start-end chunk range is mandatory-to-implement, and bins are optional.

2) LEDBAT as congestion control vs. PPSPP The PPSP peer protocol is intended for the Standards Track and relies in a normative manner on LEDBAT (RFC 6817). LEDBAT as such is an **experimental** delay-based congestion control algorithm. A Standards Track protocol cannot normatively rely on an Experimental congestion control mechanism (or RFC in general). There are ways out of this situation: i) Do not use ledbat: this would call for another congestion control mechanism to be described in the PPSPP draft. ii) Work on an 'upgrade' of the LEDBAT specification to Standards Track: Possible, but a very long way. iii) Agree on having PPSPP also as Experimental protocol. I'm currently leaning towards option iii), but this is my pure personal opinion as an

individual in the IETF.

+ A new paragraph has been added to Section 8.16 describing the widespread use of LEDBAT in current P2P systems. Hence, aim is a DOWNREF procedure.

3) No formal protocol message definition Section 7 and more specific Section 8 describe the protocol syntax of the protocol options and the messages, though Section 8 is talking about UDP encapsulation. Section 7 is hard to digest if someone should implement the options, see also later, but Section 8 is almost impossible to understand by somebody who has not been involved in the PPSP working group. See also further down for a more detailed review of the sections. To give an example out of Section 8.4: This section describes the HANDSHAKE message and gives examples how such a HANDSHAKE message could look like. But no formal definition of the message is given leaving a number of thins unclear, such as what the local channel number and what's the remote channel number is. This is implicitly defined, but that is not a good way of writing Standards Track drafts.

+ We added the usual bit-based ASCII art representations.

4) Implicit use of default values There are a number of places all over the draft where default values are defined. Many of those default values are used when there are no values explicitly signaled, e.g., the default chunk size of 1 Kbyte in Section 8.4 or Section Section 7.5. with the default for the Content Integrity Protection Method. I have the feeling that the protocol and the surroundings (e.g., what comes in via the 'tracker') are over-optimized, e.g., always providing the Content Integrity Protection Method as part of the Protocol options will not waste more than 2 bytes in a HANDSHAKE message. Further, I do not see the need to define a default chunk size in the base protocol specification, as this default can look very different, depending on who is deploying the protocol and in what context. This calls for a more dynamic way of handling the system chunk size, either as part of an external mechanisms (e.g. via the tracker) or in the HANDSHAKE message.

+ Removed implicit defaults from protocol options. Chunk size is part of the content's metadata and thus configurable. The default 1KiB has been turned into a recommendation.

5) Concept of channels The concept of channels is good but it is introduced too late in the draft, namely in Section 8.3, and it is introduced with very few words. Why isn't this introduced as part of Section 2 or Section 3, also in the relationship to the used transport protocol? I.e., the intention is to keep only one transport 'connection' between two distinct peers and to allow to run multiple swarm instances at the same time over the same transport. And how do swarms and channels correlate?

+ Concept now introduced in Section 3 with a figure.

***Technicals:

- Section 2.1, 2nd paragraph, about the tracker: I haven't seen a single place where the interaction with a tracker is discussed or where the tracker less operation is discussed in contrast. It is further unclear what type of information is really required from a tracker. A tracker (or a resource directory) would need to provide more then IP address & port, e.g., the used transport protocol for the protocol exchange (given that other transports are allowed), used chunk size, chunk addressing scheme, etc

+ Interaction with tracking facilities in general is discussed in the Operations and Management section, Section 12.1.1. This also discusses swarm metadata and information required from tracking facility. Decentralized tracking in PPSPP is discussed in Section 3.10.1.

- Section 2.3, the 1st paragraph, 'close-channel': This has been the first time where I stumbled over the channel without knowing the concept.

+ Rephrased.

- Section 3.1: ordering of messages The 1st sentence implies that ordering of messages in a datagram matters a lot. This is outlined later in the document, but I would add this as part of 3., i.e., the messages are processed in the strict order or something along this line.

+ Phrase added.

- Section 3.1, 1st paragraph, options to include I would not say anything about 'SHOULD include options' here, as this is anyhow described in Section 8.

+ Phrase removed.

- Section 3.1, 2nd paragraph: "Datagrams exchanged MAY also contain some minor payload, e.g. HAVE messages to indicate the current progress of a peer or a REQUEST (see Section 3.7)." to be added, just to make it clear IMHO: ", but MUST NOT include any DATA message".

+ Added.

- Section 3.2, 2nd paragraph: "In particular, whenever 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 in the near future." This looks like a place where a lot of traffic can be send out of a peer, i.e., whenever a chunk arrives a HAVE message must be sent. I don't believe that this should be mandated by the protocol specification, but there should guidance on when to send this, e.g., peers might be also able to wait for a short period of time to gather more chunks to be reported in HAVE. Or should in this case a single UDP datagram contain multiple HAVEs?

+ Clarified that this is indeed controlled by a policy outside the peer protocol that can decide to piggyback onto other traffic or wait till multile chunks are verified.

- Section 3.4 on ACKs This section looks pretty weak, as ACKs may be sent but on the other hand MUST be sent if ledbat is used. I would simply say: - ACK MUST be sent if an unreliable transport protocol is used - ACK MAY be sent if a reliable transport protocol is used - keep clarification about ledbat.

+ DONE.

- Section 3.5: Give text where INTEGERITY is described at least for the MTI scheme.

+ DONE.

- Section 3.7, 2nd paragraph - all 'MAY' are actually not right here. Please remove or replace them with lower letters if appropriate. - It is not clear what the 'sequentially' means exactly. Is it in the received order?

+ Rephrased MAYs. "Sequentially" replaced with "received order".

- Section 3.8: Please replace 'MAY' by can, as those are not normative behaviors but more the fact that peers can, for instance, request urgent data.

+ DONE.

- Section 3.9 Same comment as for the Section 3.8 just above this comment.

+ DONE.

- Section 3.9 waiting for responses OLD " When peer B receives a CHOKE message from A it MUST NOT send new REQUEST messages and SHOULD NOT expect answers to any outstanding ones." NEW " When peer B receives a CHOKE message from A it MUST NOT send new REQUEST messages and it cannot expect answers to any outstanding ones, as the transfer of chunks is choked."

+ DONE.

- Section 3.10.2 This whole section about PEX hole punching reads very, very experimental. The STUN method is ok, but PEX isn't. First of all, the safe behavior for a peer when it receives unsolicited PEX messages, is to discard those messages. Second, this unsolicited PEX messages trigger some behavior which may open an attack vector. The best way, but this needs more discussion, is to include to some token in the messages that are exchanged in order to make avoid any blind attacks here. However, this will need more and detailed discussions of the purpose of this.

+ We moved parts of the security analysis of PEX up, such that all mechanisms are explained in the main text, and the analysis of what attacks there are and how these mechanisms prevent them is in the Sec. Considerations section.

+ The section about hole punching was removed, lacking a reference to the experiments we conducted with this exact variant of the mechanism.

- Section 3.11 I don't see the 'MUST send keep-alive' as a mandatory requirement, as peers might have good reasons not to send any keep alive. Why not saying 'A peer can send a

keep-alive' and it 'MUST use the simple datagram...' as already described. Though there is also no really need to say MUST.

+ Now Section 3.12. Rephrased and clarified the reason and consequences of sending keep-alive msgs.

- Section 4 The syntax definition for each of the chunk addressing schemes is missing. This is not suitable for any specification that aims at interoperable implementations.

+ We added the usual bit-based ASCII art representations.

- Section 4.3.2 PPSPP peers MUST use the ACK message if an unreliable transport protocol is used.

+ DONE.

- Section 4.4 Has been tested in an implementation? I would like to understand the need for such a section, as in my understanding a peer implementation should chose one scheme and support this and there shouldn't be the need to convert between the different schemes.

+ Yes, the reference implementation translates from chunk ranges on the wire to bins internally. However, for simplicity we now state that all peers in a swarm MUST use the same method and the compatibility section has been removed.

- Section 5 This reads that MHTs are mandatory to implement while the document makes the impression that MHTs are optional.

+ Rephrased, see High-level issues.

- Section 5.3 " so each datagram SHOULD be processed separately and a loss of one datagram MUST NOT disrupt the flow" The MUST NOT is not a protocol specification requirement, but more an informative part saying that a lost message shouldn't impact the protocol machinery, but it can impact the overall operation. What is the flow here in that sentence?

+ Rephrased.

- Section 5.6.2. An illustrative example explaining how the automatic size detection works is required here.

+ Added a paragraph with an example that follows the figure used during the explanation. A state diagram could also be added, but might be a bit redundant.

- Section 6.1, 4th paragraph: Where do I find the 1 byte algorithm field in the swarm ID? The swarm ID is not really defined in a single place.

+ Expanded. Added a formal definition.

- Section 7.3 The described min/max versioning relies on the fact that there are major and minor version numbers. I cannot find any major and minor version number scheme in the draft.

+ Actually, it does not. There is a single unstructured version number.

- Section 7.4, Length field It is not clear what the 'Length' field is referring to. Further, it is not clear of the swam IDs are concatenated in one swarm ID option, of each swarm ID must be placed in a separate swam ID option.

+ Clarified.

- Section 7.6 MHTs are mandatory to support though MHTs are optional?

+ Clarified.

- Section 7.7 'key size ... derived from the swarm ID'. This relates to my high level comment no 4. on the use of implicit information. Either it is clearly specified how this information is derived or there is a protocol field/ information about the size.

+ Key size derivation procedure added to description of SIGNED_INTEGRITY in UDP encapsulation.

- Section 7.8 I would recommend to say that the default MUST be supported, but the peer must always signal what method it is supporting or at least using.

+ Corrected, see High-level issues 4.)

- Section 7.10 I have not understood how the 'Lenght' field relates to the message bitmap and how long the message bitmap can grow. The figure looks like a maximum of 16 bits?

+ Clarified.

- Section 8 I do not see the value of the text in the preface of Section 8. I would say that this text should say what is mandatory and what's not, i.e., MUST use UDP and MUST use LEDBAT. Potentially saying that future protocol versions can also run over other transport protocols.

+ Adjusted.

- Section 8.1 about Maximum Transfer Unit (MTU) The text is discussing that a Ethernet can carry 1500 bytes. This is true, but the Ethernet payload is not the normative MTU across all of the Internet. For IPv6 the min MTU is 1280 bytes and for IPv4 it is 576 bytes, though for IPv4 it can be theoretically much lower at 64 bytes. It would move the definition of the default chunk size to a recommendation with text saying that this size has a high likelihood to travel end-to-end in the Internet without any fragmentation. Fragmentation might increase the loss of complete chunks, as one lost fragment will cause the loss of a complete chunk. One way of getting an informed decision on whether chunks can travel in their size is to use the Don't Fragment (DF) bit in IPv4 and also to watch for ICMP error messages. However, ICMP error messages are not a reliable indication, but they can be some indication.

+ 1 KiB chunk size has been made a recommendation.

+ Added a small paragraph discussing the optional integration of MTU path discovery.

- Section 8.1 Definition of the default chunk size There is no need to define a default chunk size, if the chunk size would be always signaled per swarm. This is another default/ implicit value places that is unnecessary.

+ The chunk size is always part of the content's metadata.

- Section 8.3: see also my comment no 3. The concept of channels is introduced very late and with few words. A figure to explain the concept will help a lot and also more formal text on what a channel is and how they are identified. Also what the init channel is.

+ Concept now introduced in Section 3.11.

- Section 8 in general: There is no formal definition of the messages, just bit pattern examples.

+ We added the usual bit-based ASCII art representations.

- Section 8.4 (as example for the other Sections in 8.x): i) What is the '(CHANNEL' paramter? Is it actually a parameter? ii) it is implicit that the first channel no (0000000) is the remote peer's channel and that the second channel no (00000011) is the local peer's channel, right? This isn't clear from the text, but my guess.

+ We added the usual bit-based ASCII art representations.

- Section 8.5 Can HAVE messages multiple bin specs in one message or do I have to make a HAVE message for each bin?

+ Clarified.

- Section 8.6 What is the formal defintion of a DATA message? That's completely missing or I have not understood it.

+ We added the usual bit-based ASCII art representations.

- Section 8.7 looks just underspecified, especially as this is the link to LEDBAT.

+ Implementors will unfortunately need to read the full LEDBAT specification.

- Section 8.11 How are the chunks specified here? The formal syntax definition or reference to one is missing.

We added the usual bit-based ASCII art representations.

- Section 8.13 I'm lost on this section, as I haven't fully understood the concept of the PEX in this document. Especially not why there is the PEX_REScert.

+ We moved parts of the security analysis of PEX up into 3.10, such that all mechanisms are explained in the main text, and the analysis of what attacks there are and how these mechanisms prevent them is in the Sec. Considerations section.

- Section 11 The RFC required for protocol extensions of a standards track protocol looks odd. This must be at least IETF Review or Standards Action.

+ Policy changed to "IETF Review" and the section was extended with information about data types and required information.

***Editorials:

- Abstract (and probably also other places), 1st sentence of, PPSPP is not a transport protocol, just a protocol

+ DONE.

- Section 1.1, 4th paragraph: I would remove the reference to rmcat, as it is not yet clear what the outcome of the rmcat wg will be

+ DONE.

- Section 1.3, on page 8, about seeding/leeching: I would break it in to sub-bullets.

+ DONE.

- Section 2.1 and following: These are examples, isn'it? If so, this should be mentioned or clarified.

+ DONE. All subsections now labeled "Example:".

- Section 2.1: What is the PPSP Url?

+ Reformulated in terms of "Imagine there is a PPSP URL".

- Section 2.3, the 1st paragraph, detection of dead peers: It would be good to say where this detection is described in the remainder of the draft. Just for completeness.

+ DONE. Dead peer detection is now a separate section and referenced here.

- Section 2.2, the very last paragraph, 'Peer A MAY also': This 'MAY' is not useful here. I would just write 'Peer A can also', as there is nothing normative described here.

+ DONE.

- Section 3.2, last paragraph: What is the latter confinement? This is not clear to me.

+ Rephrased.

- Section 3.9, last sentence I am not sure to what the reference to Section 3.7 is pointing in this respect.

+ Rephrased.

- Section 3.10.1 about PEX messages The text says 'PPSPP optionally features...'. I have not understood if this optionally refers to mandatory to implement but optionally to use, or if the PEX messages are optionally to implement.

+ Made it clear that is OPTIONAL and not mandatory-to-implement.

- Section 3.12 I'm not sure what this section is telling exactly. Isn't just saying that PPSPP as such does not care how chunks are stored locally, as this is implementation dependent?

+ Yes. Removed.

- Section 4.2, page 15, 1st paragraph: OLD 'A PPSPP peer MAY support' NEW 'The support for this scheme is OPTIONAL'

+ DONE, for byte ranges as well.

- Section 6.1.1 This section is not describing sign-all, but rather a justification why it may still work. This doesn't help at all.

- Section 7, 1st paragraph Why is there a reference to RFC 2132?

+ Removed, just similarity in format.

- Section 7 in general i) It is common to give bit positions in the figures where the syntax of options is described. This allows to count how many bits are used for a protocol field more easily and also way more reliable. ii) Please add also Figure labels to the syntax definitions of the options. This makes it easier to reference them later on if needed.

- Section 8.1 1 kibibyte is 1 kbyte?

+ Mentioned base 1024 in Terminology. Changed to 1024 bytes where appropriate.

- Section 8.2, last paragraph i) "All messages are idempotent" in what respect? ii) "or recognizable as duplicates" but how are the recognized as duplicates?

+ Idempotent means that processing a message twice does not lead to a different state than processing them once. Resent handshakes can be recognized as duplicates because a peer already recorded the first connection attempt in its state. Updated text.

- Section 8.5, last sentence in brackets: What is this last sentence about?

+ Was explanation of the on-the-wire bytes shown.

- Section 8.13 " If sender of the PEX_REQ message does not have a private or link-local address, then the PEX_RES* messages MUST NOT contain such addresses [RFC1918][RFC4291]." What is this text saying? Do not include what you do not have anyway?

+ Rephrased. It tries to say that internal addresses must not be leaked to external peers.

- Section 8.14 There is no single place where all the constants are collected and also documented what the default values or the recommended values. For instance in this Section 8.14 where the dead peer time out is set to 3 minutes and also the number of datagrams that should have sent. I would make a section or subsection to discuss dead peers and how they are detected and just link to the keep-alive mechanism in Section 8.14.

+ The Section 12.1.6 section was rewritten for this in the Ops & Mgmt part.

- Section 11 This section needs to be overhauled once the document is ready for the IESG. The section is not wrong but can be improved to help IANA.

+ The section was extended with information about data types and required information.

-08 2013-08-8 Continued Revision following AD Review

Please see the -07 entry for our responses to the comments.

Added ECDSAP256SHA256 and ECDSAP384SHA384 as mandatory-to-implement live signature algorithms, as they provide small swarm IDs.

Added line that a peer SHOULD NOT send HAVEs to peers that already have the complete content (e.g. in video-on-demand scenarios).

In response to a remark at WG meeting at IETF 87 we added a paragraph on OPTIONAL MTU discovery using PPSPP messages to Section 8.1.

Authors' Addresses

Arno Bakker
Vrije Universiteit Amsterdam
De Boelelaan 1081
Amsterdam, 1081HV
The Netherlands

Phone:
Email: arno@cs.vu.nl

Riccardo Petrocco
Technische Universiteit Delft
Mekelweg 4
Delft, 2628CD
The Netherlands

Phone:
Email: r.petrocco@gmail.com

Victor Grishchenko
Technische Universiteit Delft
Mekelweg 4
Delft, 2628CD
The Netherlands

Phone:
Email: victor.grishchenko@gmail.com