Internet Engineering Task Force (IETF)
Request for Comments: 7574
Category: Standards Track
ISSN: 2070-1721

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

July 2015

Peer-to-Peer Streaming Peer Protocol (PPSPP)

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 prerecorded (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 Low Extra Delay Background Transport (LEDBAT) for congestion control.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7574.

Table of Contents

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 prerecorded (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 that 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].

The protocol defined in this document assumes that a peer has already discovered a list of (initial) peers using, for example, a centralized tracker [PPSP-TP]. Once a peer has this list of peers, PPSPP allows the peer to connect to other peers, request chunks of content, and discover other peers disseminating the same content.

The design of PPSPP is based on our research into making BitTorrent [BITTORRENT] suitable for streaming content [P2PWIKI]. Most PPSPP messages have corresponding BitTorrent messages and vice versa. However, PPSPP is specifically targeted towards streaming audio/video content and optimizes time-till-playback. It was also designed to be more flexible and extensible.

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 or a prerecorded multimedia file.

chunk
: The basic unit in which the content is divided. For example, a block of N kilobytes. A chunk may be of variable size.

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. That is, methods for receiving peers to detect whether a requested chunk has been modified, either maliciously by the sending peer or accidentally in transit.

hash
: The result of applying a cryptographic hash function, more specifically a Modification Detection Code (MDC) [HAC01], such as SHA-256 [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).

munro hash
: The hash of a subtree that is the unit of signing in the Unified Merkle Tree content authentication scheme for live streaming (see Section 6.1.2.1).

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. For video on demand with content integrity protection enabled, the identifier is the so-called root hash of a Merkle hash tree over the content. 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 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 file 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 file 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 that channel.

heavy payload
: A datagram has a heavy payload when it contains DATA messages, SIGNED_INTEGRITY messages, or a large number of smaller messages.

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 the content distributed is static, 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 its source. The browser passes this URL to its peer-to-peer streaming protocol handler. Let's call this protocol handler Peer A. Peer A parses the URL to retrieve the transport address of a peer-to-peer streaming protocol tracker and swarm metadata of the content. The tracker address may be optional in the presence of a decentralized tracking mechanism. The mechanisms for tracking peers are outside of the scope of this document.

Peer A now registers with the tracker following the peer-to-peer streaming protocol tracker specification [PPSP-TP] and receives the IP address and port of peers already in the swarm, say, Peers B, C, and D. At this point, the PPSPP starts operating. Peer A now sends a datagram containing a PPSPP HANDSHAKE message to Peers B, C, and D. This message conveys protocol options. In particular, Peer A includes the ID of the swarm (part of the swarm metadata) as a protocol option because the destination peers can listen for multiple swarms on the same transport address.

Peers B and C respond with datagrams containing a PPSPP HANDSHAKE message and one or more HAVE messages. A HAVE message conveys (part of) the chunk availability of a peer; thus, it contains a chunk specification that denotes what chunks of the content Peers B and C have, respectively. Peer D sends a datagram with a HANDSHAKE and HAVE messages, but also with a CHOKE message. The latter indicates that Peer D is not willing to upload chunks to Peer A at present.

2.2. Example: Exchanging Chunks

In response to Peers B and C, Peer A sends new datagrams to Peers B and C containing REQUEST messages. A REQUEST message indicates the chunks that a peer wants to download; thus, it contains a chunk specification. The REQUEST messages to Peers B and C refer to disjoint sets of chunks. Peers 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 Peers B and C are correct against the trusted swarm ID. Peer A also updates the chunk availability of Peers 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, Peer A sends a datagram containing HAVE messages for the chunks it just received to all its peers. In the datagram to Peers 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, for example, Peer C finds that Peer A obtained a chunk (from Peer B) that Peer C did not yet have, Peer C's next datagram includes a REQUEST for that chunk.

Peer D also sends HAVE messages to Peer A when it downloads chunks from other peers. When Peer D is willing to accept REQUESTs from Peer A, Peer D sends a datagram with an UNCHOKE message to inform Peer A. If Peer B or C decides to choke Peer A, they send a CHOKE message and Peer A should then re-request from other peers. Peers B and C may continue to send HAVE, REQUEST, or periodic keep-alive messages such that Peer 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 tracker specification [PPSP-TP]. Peers receiving the datagram should remove Peer A from their current peer list. If Peer A crashes ungracefully, peers should remove Peer A from their peer list when they detect it no longer sends messages (see Section 3.12).

3. Messages

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 or bad and only use the good ones. A good peer is a peer that responds with chunks; a peer that does not respond, or does not respond in time is classified as bad. The idea is that, in PPSPP, the content is available from multiple sources (unlike HTTP), so a peer should not invest too much effort in trying to obtain it from a particular source. This classification in good or bad allows a peer to deal with slow, crashed, and (silent) malicious peers.

Multiple messages MUST be 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. If an invalid message is found in a datagram, the remaining messages MUST be discarded.

For the sake of simplicity, one swarm of peers deals with one content file or stream only. There is a single division of the content into chunks that all peers in the swarm adhere to, determined by the content publisher. Distribution of a collection 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. In other words, the audio/video container format used is outside the scope of this document.

3.1. HANDSHAKE

For Peer P to establish communication with Peer Q in Swarm S, the peers must first exchange HANDSHAKE messages by means of a handshake procedure. The initiating Peer P needs to know the metadata of Swarm S, which consists of:

(a) the swarm ID of the content (see Sections 5.1 and 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 (for static content.)

This document assumes the swarm metadata is obtained from a trusted source. In addition, Peer P needs to know a transport address for Peer Q, obtained from a peer discovery/tracking protocol.

The payload of the HANDSHAKE message contains a sequence of protocol options. The protocol options encode the swarm metadata just described to enable an end-to-end check to see whether the peers are in the right swarm. Additionally, the options encode a number of per-peer configuration parameters. The complete set of protocol options are specified in Section 7. The HANDSHAKE message also contains a channel ID for multiplexing communication and security (see Sections 3.11 and 12.1). A HANDSHAKE message MUST always be the first message in a datagram.

3.1.1. Handshake Procedure

The handshake procedure for a peer, Peer P, to start communication with another peer, Peer Q, in Swarm S is now as follows.

1. The first datagram the initiating Peer P sends to Peer Q MUST start with a HANDSHAKE message. This HANDSHAKE message MUST contain:

   * A channel ID, chanP, randomly chosen as specified in Section 12.1.

   * The metadata of Swarm S, encoded as protocol options, as specified in Section 7. In particular, the initiating Peer P MUST include the swarm ID.

   * The capabilities of Peer P, in particular, its supported protocol versions, "Live Discard Window" (in case of a live swarm) and "Supported Messages", encoded as protocol options.

   This first datagram MUST be prefixed with the (destination) channel ID 0; see Section 3.11. Hence, the datagram contains two channel IDs: the destination channel ID prefixed to the datagram and the channel ID chanP included in the HANDSHAKE message inside the datagram. This datagram MAY also contain some minor additional payload, e.g., HAVE messages to indicate Peer P's current progress, but it MUST NOT include any heavy payload (defined in Section 1.3), such as a DATA message. Allowing minor

payload minimizes the number of initialization round trips, thus improving time-till-playback. Forbidding heavy payload prevents an amplification attack (see Section 12.1).

2. The receiving Peer Q checks the HANDSHAKE message from Peer P. If any check by Peer Q fails, or if Peers P and Q are not in the same swarm, Peer Q MUST NOT send a HANDSHAKE (or any other) message back, as the message from Peer P may have been spoofed (see Section 12.1). Otherwise, if Peer Q is interested in communicating with Peer P, Peer Q MUST send a datagram to Peer P that starts with a HANDSHAKE message. This reply HANDSHAKE MUST contain:

   * A channel ID, chanQ, randomly chosen as specified in Section 12.1.

   * The metadata of Swarm S, encoded as protocol options, as specified in Section 7. In particular, the responding Peer Q MAY include the swarm ID.

   * The capabilities of Peer Q, in particular, its supported protocol versions, its "Live Discard Window" (in case of a live swarm) and "Supported Messages", encoded as protocol options.

   This reply datagram MUST be prefixed with the channel ID chanP sent by Peer P in the first HANDSHAKE message (see Section 3.11). This reply datagram MAY also contain some minor additional payload, e.g., HAVE messages to indicate Peer Q's current progress, or REQUEST messages (see Section 3.7), but it MUST NOT include any heavy payload.

3. The initiating Peer P checks the reply datagram from Peer Q. If the reply datagram is not prefixed with (destination) channel ID chanP, Peer P MUST discard the datagram. Peer P SHOULD continue to process datagrams from Peer Q that do meet this requirement. This check prevents interference by spoofing, see Section 12.1. If Peer P's channel ID is echoed correctly, the initiator Peer P knows that the addressed Peer Q really responds.

4. Next, Peer P checks the HANDSHAKE message in the datagram from Peer Q. If any check by Peer P fails, or Peer P is no longer interested in communicating with Peer Q, Peer P MAY send a HANDSHAKE message to inform Peer Q it will cease communication. This closing HANDSHAKE message MUST contain an all zeros channel ID and a list of protocol options. The list MUST either be empty or contain the maximum version number Peer P supports, following the min/max versioning scheme defined in [RFC6709], Section 4.1.

The datagram containing this closing HANDSHAKE message MUST be prefixed with the (destination) channel ID chanQ. Peer P MAY also simply cease communication.

5. If the addressed peer, Peer Q, does not respond to initiating Peer P's first datagram, Peer P MAY resend that datagram until Peer Q is considered dead, according to the rules specified in Section 3.12.

6. If the reply datagram by Peer Q does pass the checks by Peer P, and Peer P wants to continue interacting with Peer Q, Peer P can now send REQUEST, PEX_REQ, and other messages to Peer Q. Datagrams carrying these messages MUST be prefixed with the channel ID chanQ sent by Peer Q. More specifically, because Peer P knows that Peer Q really responds, Peer P MAY start sending Peer Q messages with heavy payload. That means that Peer P MAY start responding to any REQUEST messages that Peer Q may have sent in this first reply datagram with DATA messages. Hence, transfer of chunks can start soon in PPSPP.

7. If Peer Q receives any datagram (apparently) from Peer P that does not contain channel ID chanQ, Peer Q MUST discard the datagram but SHOULD continue to process datagrams from Peer P that do meet this requirement. Once Peer Q receives a datagram from Peer P that does contain the channel ID chanQ, Peer Q knows that Peer P really received its reply datagram, and the three-way handshake and channel establishment is complete. Peer Q MAY now also start sending messages with heavy payload to Peer P.

8. If Peer P decides it no longer wants to communicate with Peer Q, or vice versa, the peer SHOULD send a closing HANDSHAKE message to the other, as described above.

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 MUST send a HAVE message to all peers Q1..Qn it wants to allow to download those chunks. A policy in Peer P determines when the HAVE is sent. Peer P may send it directly, or Peer P may wait either until it has other data to send to Peer 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 12.1. 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 [RFC6817].

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 Sections 5.3, and 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 push model without requests 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 Peer 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 Peer 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 Peer B, for example, because Peer A's upload capacity is exhausted. Peer A MAY send a subsequent UNCHOKE message to signal that it will respond to new REQUESTs from Peer B again (Peer A SHOULD discard old requests). When Peer B receives a CHOKE message from Peer A, it MUST NOT send new REQUEST messages and it cannot expect answers to any outstanding ones, as the transfer of chunks is choked. When Peer B is choked but receives a HAVE message from Peer A, it is not automatically unchoked and MUST NOT send any new REQUEST messages. 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 Exchange (PEX) messages 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 Distributed Hash Table) to discovery new peers. The strength of this mechanism is therefore that it enables decentralized peer discovery: after an initial bootstrap, a central tracker is no longer needed. Its weakness is that it enables a number of attacks, so it should not be used on the Internet unless extra security measures are in place.

PPSPP supports peer-address exchange on the Internet and in benign private networks as an OPTIONAL feature (not mandatory to implement) under certain conditions. 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_REScert messages. Logically, a PEX_REScert reply message contains the address of a single peer Ci. Peer B MUST have exchanged messages with Peer Ci 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_REScert 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.

To use PEX in PPSPP on the Internet, two conditions must be met:

1. Peer transport addresses must be relatively stable.

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

The full security analysis for PEX messages can be found in Section 12.2. Physically, 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 Peer 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 12.2.2 describes an example where a central tracker acts as the Certification Authority.

In a hostile environment, such as the Internet, 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 12.2.3).

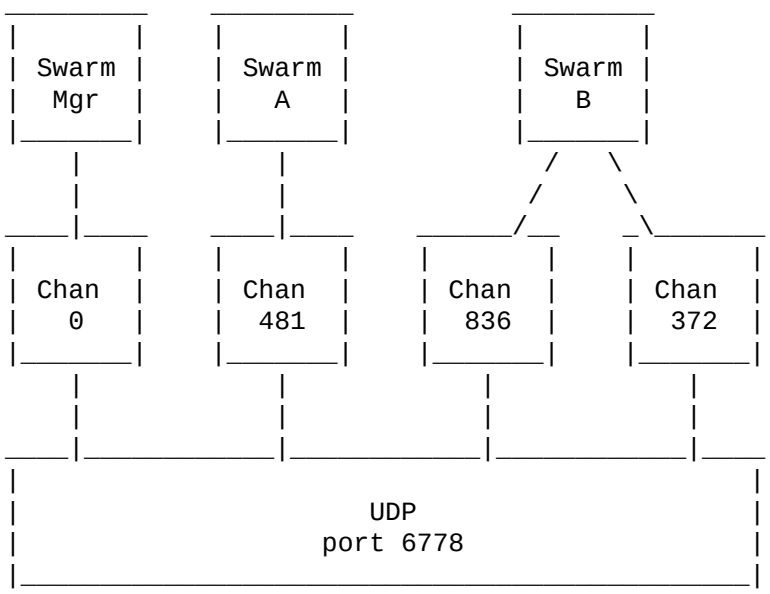
In addition to the PEX_REScert, there are two other PEX reply messages. The PEX_RESv4 message contains a single IPv4 address and port. The PEX_RESv6 message contains a single IPv6 address and port. They MUST only be used in a benign environment, such as a private network, as they provide no guarantees that the host addressed actually participates in a PPSPP swarm.

Once a PPSPP implementation has obtained a list of peers (either via PEX, from a central tracker, or via a Distributed Hash Table (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 peer-to-peer (P2P) system like PPSPP can perform these optimizations using the Application-Layer Traffic Optimization (ALTO) protocol is described in detail in [RFC7285], 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 12.1.

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 Peer A to Peer B are prefixed with the four-byte channel ID allocated by Peer B, and vice versa for datagrams from Peer 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 Session Traversal Utilities for NAT (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 Signaling

A peer SHOULD send a "keep alive" message periodically to each peer it is interested in, but has no other messages to send to them at present. The goal of the keep alives is to keep a signaling channel open to peers that are of interest. Which peers those are is determined by a policy that decides which peers are of interest now and in the near future. This document does not prescribe a policy, but examples of interesting peers are (a) peers that have chunks on offer that this client needs or (b) peers that currently do not have interesting chunks on offer (because they are still downloading themselves, or in live streaming) but gave good performance in the past. When these peers have new chunks to offer, the peer that kept a signaling channel open can use them again. Periodically sending "keep alive" messages prevents other peers declaring the peer dead. A guideline for declaring a peer dead when using UDP consists of a three minute delay since that last packet has been received from that peer and at least three datagrams having been sent to that peer during the same period. When a peer is declared dead, the channel to it is closed, no more messages will be sent to that peer and the

local administration about the peer is discarded. Busy servers can force idle clients to disconnect by not sending keep alives. 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 four-byte channel ID only, see Sections 8.3 and 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.

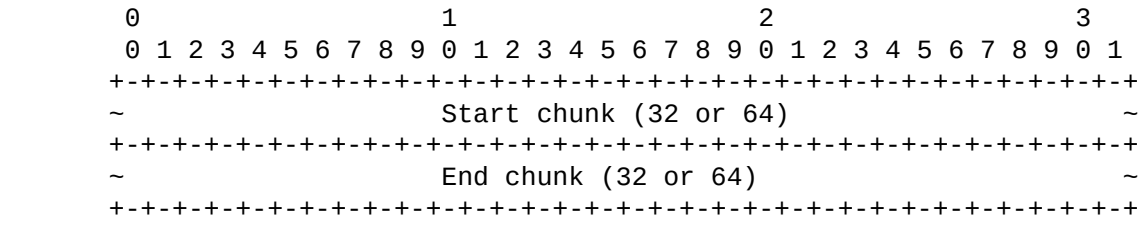
All peers in a swarm MUST use the same chunk addressing method.

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.

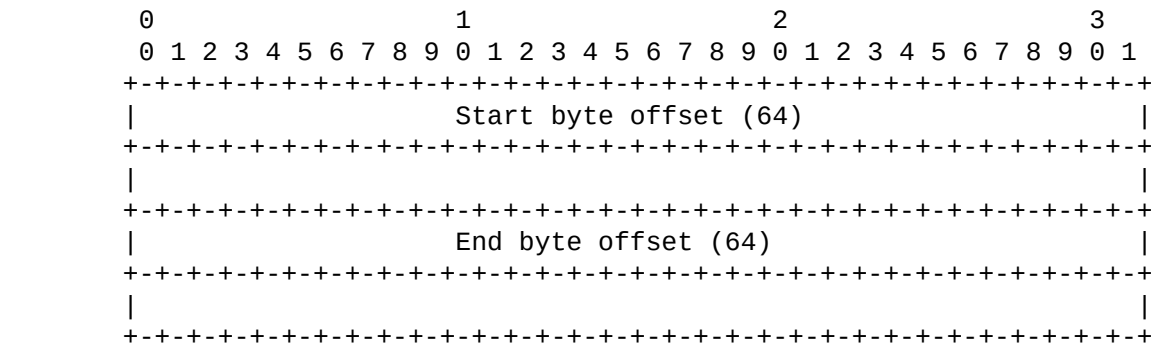
4.1.1. Chunk Ranges

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.



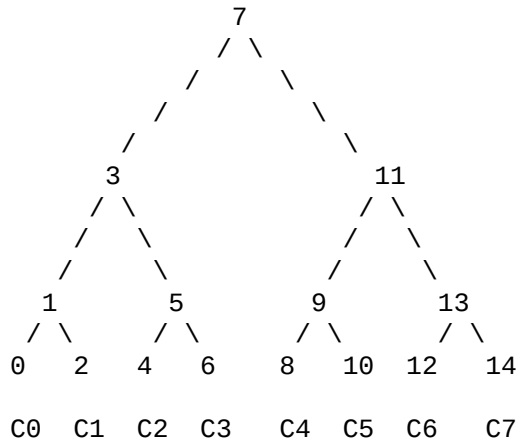
4.2. Bin Numbers

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 lightweight. 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 that 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 that lays them out into a vector nicely, which is called bin numbering, as follows. Consider a 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 node 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:

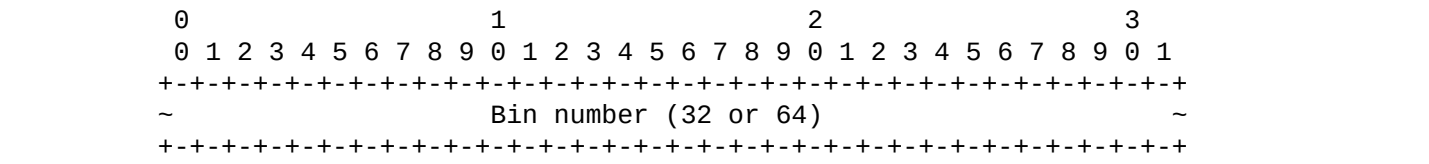


The bin number tree of an interval of width W=8

Figure 2

So bin 7 represents the complete interval, bin 3 represents the interval of chunk C0..C3, bin 1 represents the interval of chunks C0 and C1, 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 allow download of those chunk(s) from. The ability to withhold HAVE messages allows them 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 acknowledgements 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 an 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.

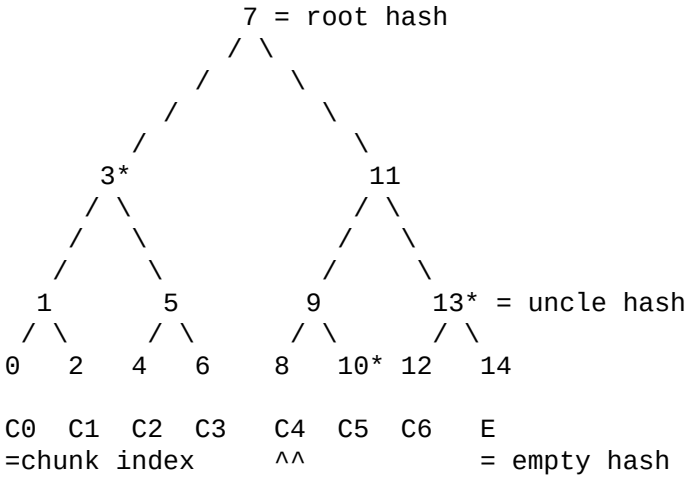
For static content, PPSPP currently defines one method for protecting integrity, called the Merkle Hash Tree scheme. If PPSPP operates over the Internet, this scheme MUST be used. If PPSPP operates in a benign environment, this scheme MAY be used. So the scheme is mandatory to implement, to satisfy the 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 file that is divided into N chunks is constructed as follows. Note the construction does not assume chunks of content to be of a fixed size. Given a cryptographic hash function, more specifically an MDC [HAC01], such as SHA-256, 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 file consisting of 7 chunks. As 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 leftmost 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.



Merkle hash tree of a content file with N=7 chunks

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, the uncles are nodes 13 and 3 and the sibling is 10; all marked with a * 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 sent either the wrong content or the wrong sibling or uncle hashes. For simplicity, the set of sibling and uncle 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: 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, both messages and message interdependencies SHOULD NOT 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 acknowledgements (HAVE or ACK) to derive which hashes the receiver already has for sure. Suppose the receiver had acknowledged chunks C0 and C1 (the 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 also be 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 that were sent out in previous, still-unacknowledged datagrams. It is an optimization trade-off between redundant hash transmission and the possibility of collateral data loss in the case in which some necessary hashes were lost in the network so some delivered data cannot be verified and thus had to be dropped. In either case, the receiver builds the Merkle hash 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 hashes he believes are 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 an INTEGRITY message for each hash the receiver misses for integrity checking. An 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, and 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 leaves are at height 0). The DATA message MUST contain the chunk specification of the chunk and the 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 of the 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 the 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

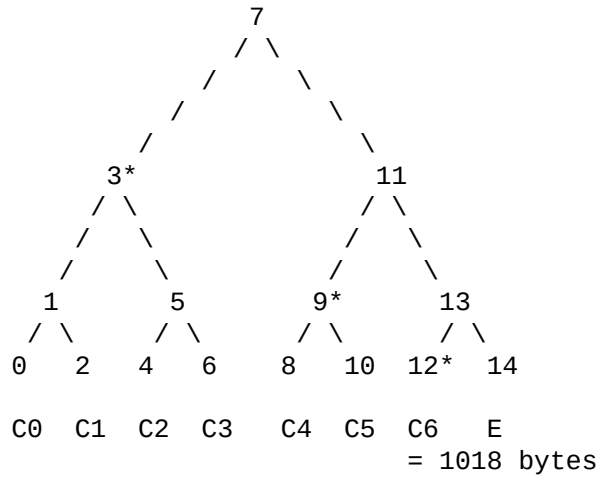
In PPSPP, the size of a static content file, such as a video file, can be reliably and automatically derived from information received from the network when fixed-size chunks are used. As a result, it is not necessary to include the size of the content file as the metadata of the content for such files. Implementations of PPSPP MAY use this automatic detection feature. Note this feature is the only feature of PPSPP that requires that a fixed-size chunk is used. This feature builds on the Merkle hash tree and the trusted root hash as swarm ID as follows.

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. 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 also contains 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 hash 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 hash 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 an *. 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, and 12. Therefore, the number of peak hashes for a file is 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; thus, it gets an estimate of the file size (given all chunks but the last are of a fixed size). Which nodes are the peaks can be securely communicated from one (untrusted) peer, Peer A, to another peer, Peer B, by letting Peer A send the peak hashes and their node IDs to Peer

B. It can be shown that the root hash that Peer 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, if it is the right sibling, its left neighbor/sibling must also be a filled node, because of the way chunks are laid out in the leaves, which contradicts the definition of a peak hash. (b) For the rightmost peak hash, its right sibling is zero. (c) For any peak hash, the 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 of a fixed size, Peer B just needs to know the size of the last chunk. Knowing the number of chunks, Peer B can calculate the node ID of the last chunk and download it. As always, Peer 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. Peer 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 Peer A sends a DATA message for the first time to Peer B, Peer A MUST first send all the peak hashes for the content, in INTEGRITY messages, unless Peer B has already signaled 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, Peer 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-byte file, which fits in 7 chunks of 1 kilobyte, distributed by Peer A. Figure 4 shows the relevant Merkle hash tree. Peer B, which only knows the root hash of the file after successfully connecting to Peer A, requests the first chunk of data, C0 in Figure 4. Peer A replies to Peer 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

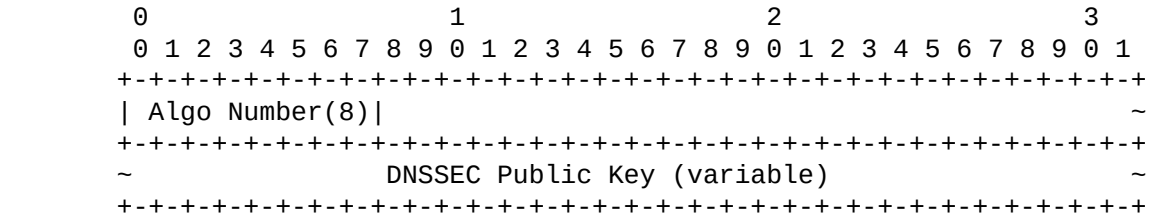
For live streaming, PPSPP supports two methods for a peer to authenticate the content it receives from another peer, called "Sign All" and "Unified Merkle Tree".

In the "Sign All" method, the live injector signs each chunk of content using a private key. Upon receiving the chunk, peers 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 "Unified Merkle Tree" 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 Base64 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 values in the "Domain Name System Security (DNSSEC) Algorithm Numbers" registry [IANADNSSECALGNUM]. The RSASHA1 [RFC4034], RSASHA256 [RFC5702], 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 that 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 timestamp in NTP Timestamp format [RFC5905], and the chunk, in that order. The timestamp is the time signature that was made at the injector in UTC.

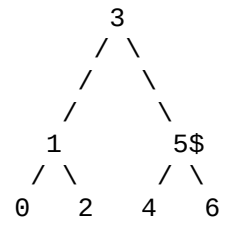
6.1.2. Unified Merkle Tree

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, it has a transient root hash. Therefore, a public key 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.



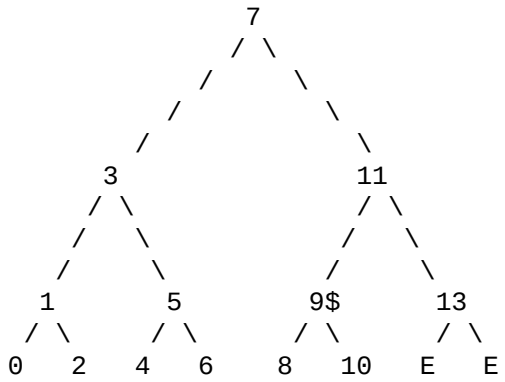
Expanded live tree. With NCHUNKS_PER_SIG=2, node 5 is the munro for the new subtree spanning 4 and 6. Node 1 is the munro for the subtree spanning chunks 0 and 2, created in the previous iteration.

Figure 5

Informally, the process now proceeds as follows. The injector 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, 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 comparing the computed hash of node 9 with the munro hash of the same node he just received; hence, Peer 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 Hash 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 this 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-size 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 that 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 node (see Section 6.1.2.1), a timestamp in 64-bit NTP Timestamp format [RFC5905], and the hash associated with the munro node, 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 Peer 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 in to 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 Peer B responds with a second datagram also containing a HANDSHAKE message (see Section 3.1). When Peer A sends a third datagram to Peer B, and it is received by Peer B, both peers know that the other is listening on its stated transport address. Peer 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 rightmost signed munro hash to Peer B in the third datagram, and in any subsequent datagrams to Peer B, until Peer 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). Peer B may already have indicated this fact by means of HAVE messages in the second datagram. Conversely, when Peer B sends the fourth datagram or any subsequent datagram to Peer A, Peer B MUST send its rightmost signed munro hash, unless Peer A indicated knowledge of it or more recent munros. The rightmost signed munro hash of a peer is defined as the munro hash signed by the injector of the rightmost subtree of width NCHUNKS_PER_SIG chunks in the peer's Merkle hash tree. Peer A MUST

NOT send the signed munro hash in the first datagram of the HANDSHAKE procedure and Peer B MUST NOT send it in the 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 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 so 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 when it actually 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 is also related to the (average) bitrate of the stream. Finally, if a peer has few resources to store chunks and metadata, it should choose 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

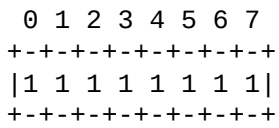
big-endian. Each protocol option is explained in the following subsections. The list of protocol options MUST be sorted on code value (ascending) in a HANDSHAKE message.

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	Chunk Size
10-254	Unassigned
255	End Option

Table 2: PPSPP 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.

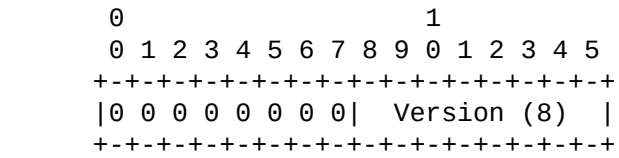


7.2. Version

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

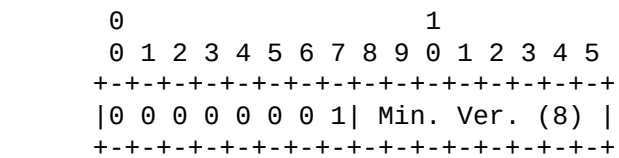
Version	Description
0	Reserved
1	Protocol as described in this document
2-255	Unassigned

Table 3: PPSPP Version Numbers



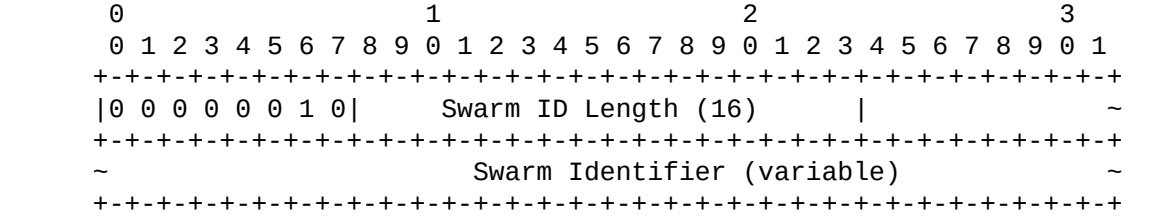
7.3. Minimum Version

When a peer initiates the handshake, it MUST include the minimum version of the PPSPP 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. If the peer is not the initiator, it 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.

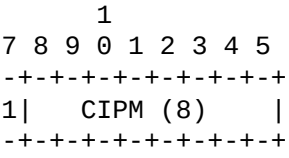
7.5. Content Integrity Protection Method

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: PPSPP 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.



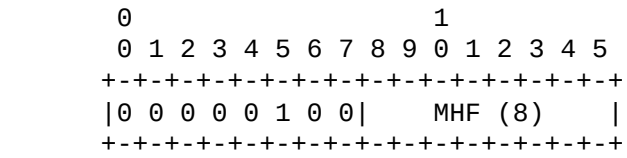
7.6. Merkle Tree Hash Function

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	SHA-1
1	SHA-224
2	SHA-256
3	SHA-384
4	SHA-512
5-255	Unassigned

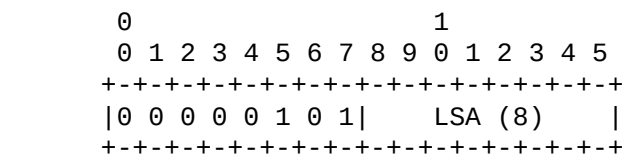
Table 5: PPSPP Merkle Hash Functions

Implementations MUST support SHA-1 (see Section 12.5) and SHA-256. SHA-256 is 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 values listed in the "Domain Name System Security (DNSSEC) Algorithm Numbers" registry [IANADNSSECALGNUM]. The RSASHA1 [RFC4034], RSASHA256 [RFC5702], ECDSAP256SHA256 and ECDSAP384SHA384 [RFC6605] algorithms are mandatory to implement. Default is ECDSAP256SHA256.



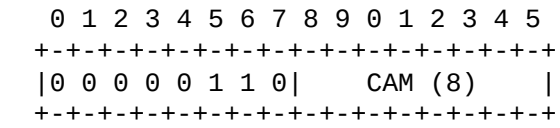
7.8. Chunk Addressing Method

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

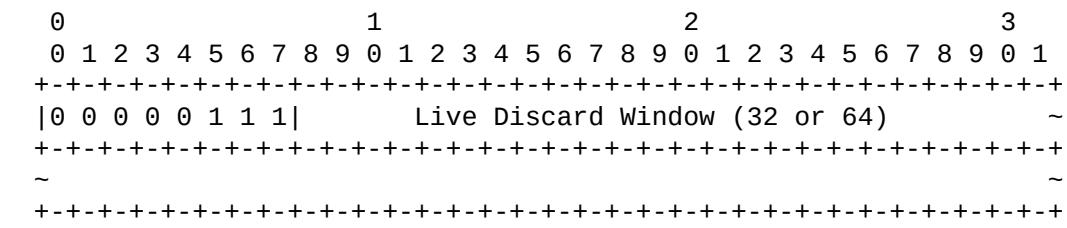
Table 6: PPSPP Chunk Addressing Methods

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, see Table 6. 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:

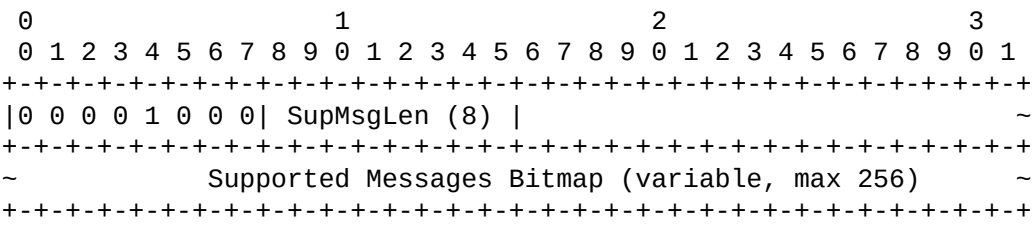


A peer that does not, under normal circumstances, discard chunks MUST set this option to the special value 0xFFFFFFFF (32-bit) or 0xFFFFFFFFFFFFFFFF (64-bit). For example, peers that record a complete broadcast to offer it directly as a static file after the broadcast ends use these values (see Section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

7.10. Supported Messages

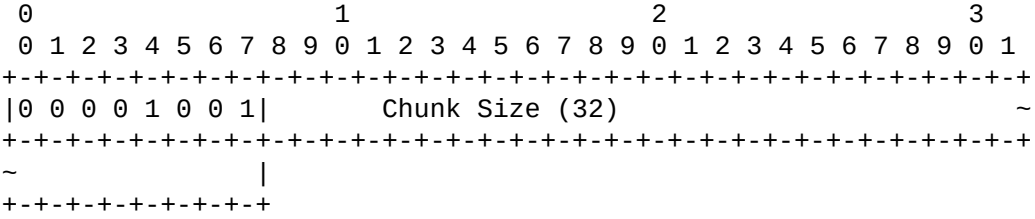
Peers may support just a subset of the PPSPP messages. For example, peers running over TCP may not accept ACK messages or peers used with a centralized tracking infrastructure may not accept PEX messages. For these reasons, peers who support only a proper subset of the PPSPP messages MUST signal which 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. An example of the first 16 bits of the compressed bitmap for a peer supporting every message except ACKs and PEXs is 11011001 11110000.



7.11. Chunk Size

A peer in a swarm MUST include the chunk size the swarm uses. The code for this option is 9. Its value is a 32-bit integer denoting the size of the chunks in bytes in big-endian format. When variable chunk sizes are used, this option MUST be set to the special value 0xFFFFFFFF. Section 8.1 explains how content publishers can determine a value for this option.



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, a 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 megabyte 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-size chunks of 1024 bytes, as this size has a high likelihood of traveling 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 for a given path. 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 PPSPP 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 the 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: PPSPP Message Types

Furthermore, integers are serialized in network (big-endian) byte order. So, consider the example of a HAVE message (Section 3.2) using bin chunk addressing. It has a message type of 0x03 and a payload of a bin number, a 4-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

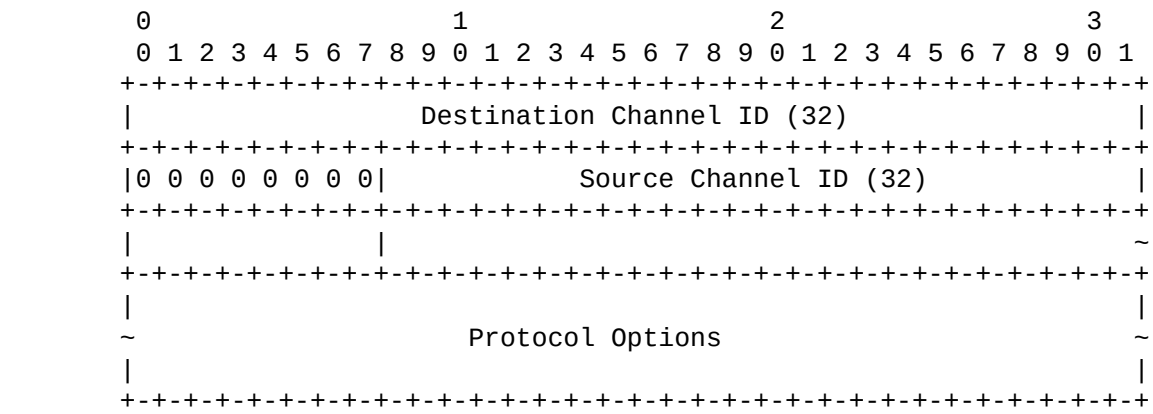
HANDSHAKE message, it can be recognized as duplicate by the receiver, because it already recorded the first connection attempt, and be dealt with.

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 each other's channel ID during the handshake as explained in Section 3.1.1. A channel ID consists of 4 bytes and MUST be generated following the requirements in [RFC4960] (Section 5.1.3).

8.4. HANDSHAKE

A channel is established with a handshake. To start a handshake, the initiating peer needs to know the swarm metadata, defined in Section 3.1 and the IP address and UDP port of a peer. A datagram containing a HANDSHAKE message then looks as follows:



where:

Destination Channel ID:

If the datagram is sent by the initiating peer, then it MUST be an all-zeros channel ID.

If the datagram is sent by the responding peer, then it MUST consist of the Source Channel ID from the sender's HANDSHAKE message.

The octet 0x00: The HANDSHAKE message type

Source Channel ID: A locally unused channel ID

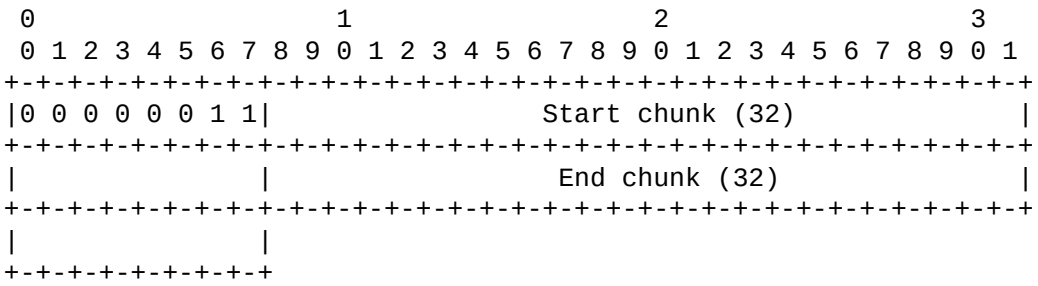
Protocol Options: A list of protocol options encoding the swarm's metadata, as defined in Section 7.

A peer SHOULD explicitly close a channel by sending a HANDSHAKE message that MUST contain an all zeros Source Channel ID and a list of protocol options. The list MUST either be 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:



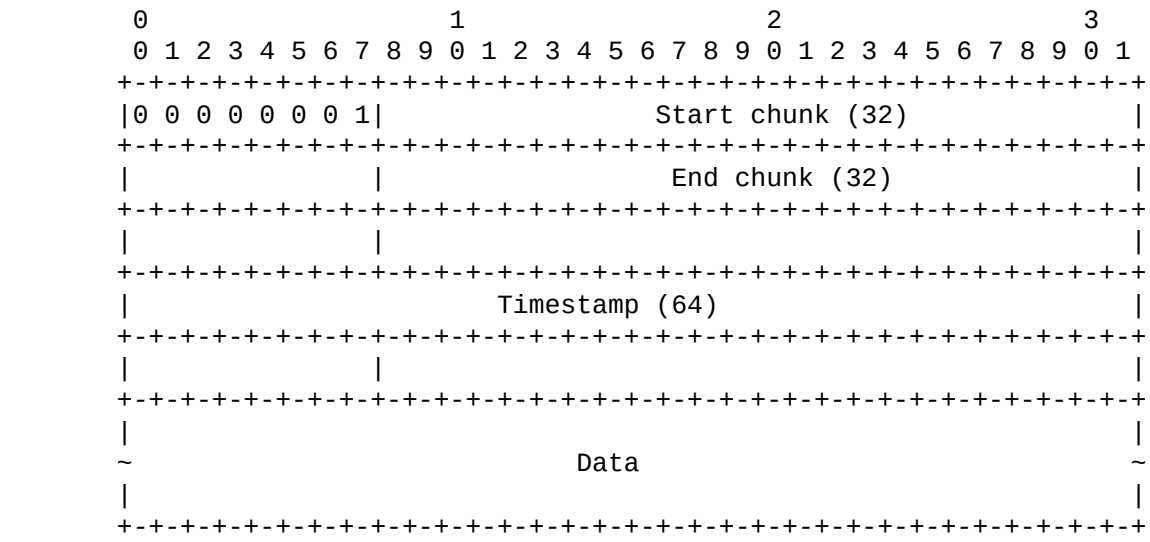
where the first octet is the HAVE message (0x03) followed by the start chunk and the end chunk describing the chunk range. Note this diagram shows a message and not a datagram, so it is not prefixed by the destination Channel ID. This holds for all subsequent message diagrams.

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 the DATA messages carry the last chunk, if that is smaller than the chunk size. As 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:

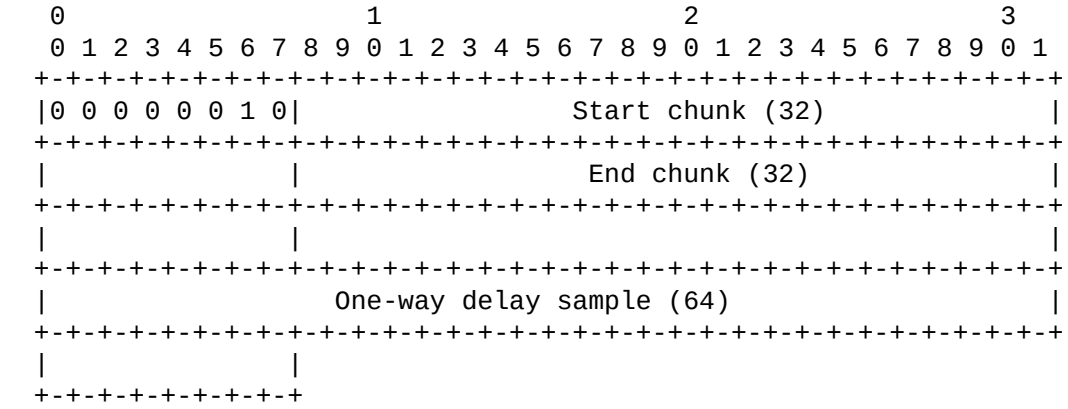


where the first octet is the DATA message (0x01) followed by the start chunk and the end chunk describing the single chunk, the timestamp, and the actual data.

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 a one-way delay sample. The one-way delay sample is a 64-bit integer with microsecond accuracy, and it 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:

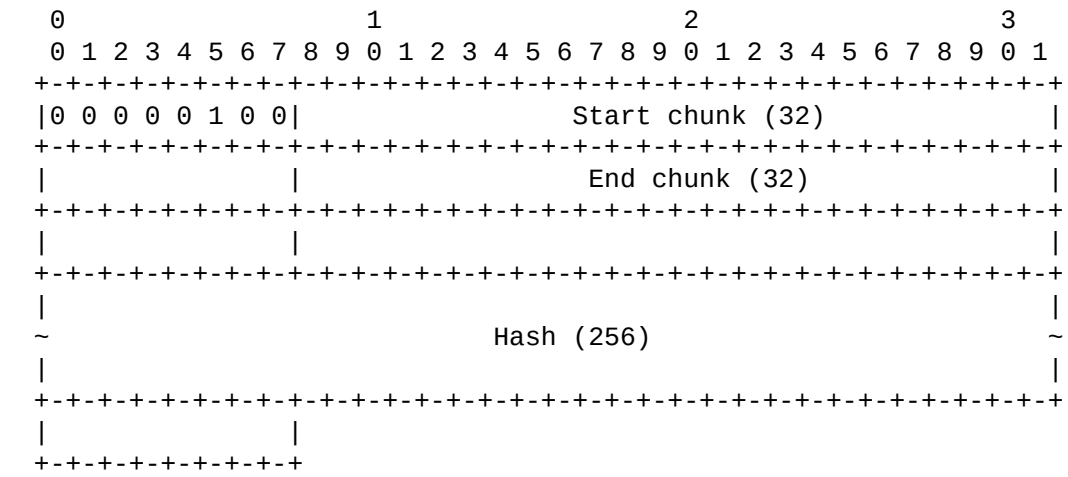


where the first octet is the ACK message (0x02) followed by the start chunk and the end chunk describing the chunk range and the one-way delay sample.

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 a SHA-256 hash:

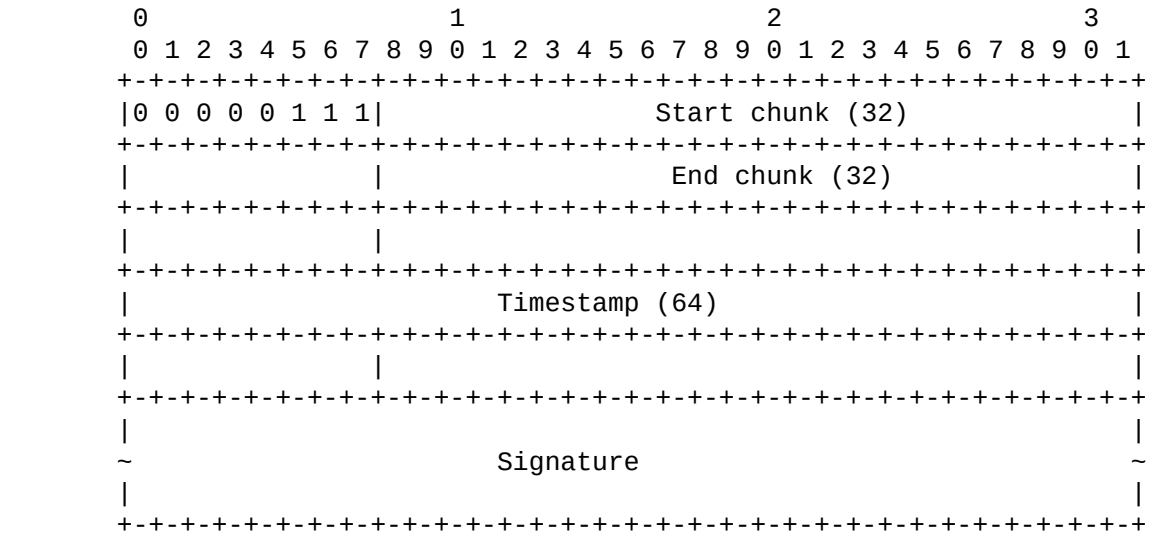


where the first octet is the INTEGRITY message (0x04) followed by the start chunk and the end chunk describing the chunk range and the hash.

8.9. SIGNED_INTEGRITY

A SIGNED_INTEGRITY message (type 0x07) consists of a chunk specification, a 64-bit timestamp in NTP Timestamp format [RFC5905] and a digital signature encoded as a Signature field would be in an RRSIG record in DNSSEC without the Base64 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.

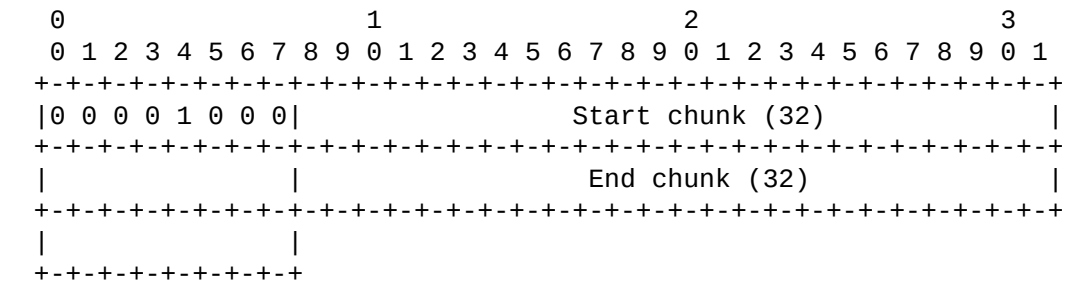
The length of the digital signature can be derived from the Live Signature Algorithm protocol option and the swarm ID as follows. The first mandatory algorithms are RSASHA1 and RSASHA256. For those algorithms, the swarm ID consists of a 1-byte Algorithm field followed by an RSA public key stored as a 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:

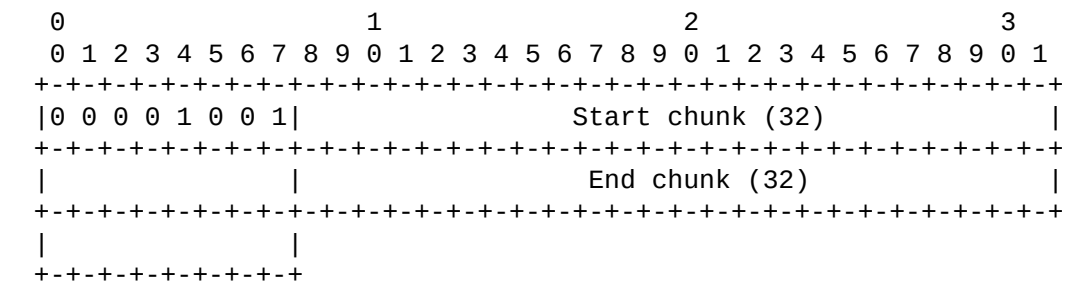


where the first octet is the REQUEST message (0x08) followed by the start chunk and the end chunk describing the chunk range.

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:

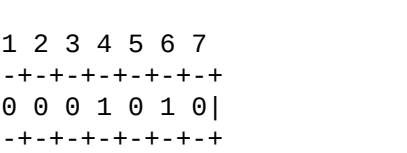


where the first octet is the CANCEL message (0x09) followed by the start chunk and the end chunk describing the chunk range.

8.12. CHOKE and UNCHOKE

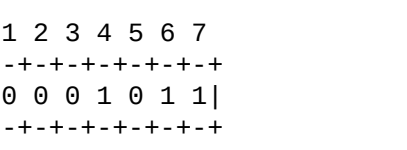
Both CHOKE and UNCHOKE messages (types 0x0a and 0x0b, respectively) carry no payload.

A CHOKE message:



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

An UNCHOKE message:

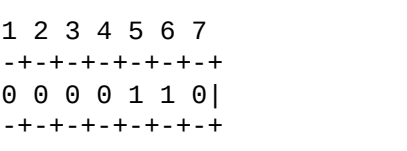


where the first octet is the UNCHOKE message (0x0b).

8.13. PEX_REQ, PEX_RESv4, PEX_RESv6, and PEX_REScert

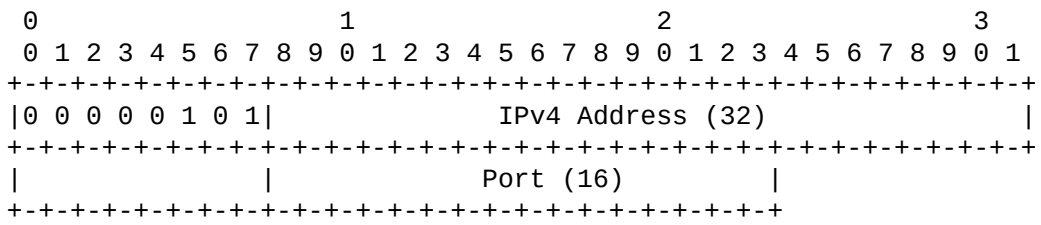
A PEX_REQ (0x06) message has no payload. A PEX_RESv4 (0x05) message consists of an IPv4 address in big-endian format followed by a UDP port number in big-endian format. A PEX_RESv6 (0x0c) message contains a 128-bit IPv6 address instead of an IPv4 one. If a PEX_REQ message does not originate from a private, unique-local, link-local, or multicast address [RFC1918] [RFC4193] [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:



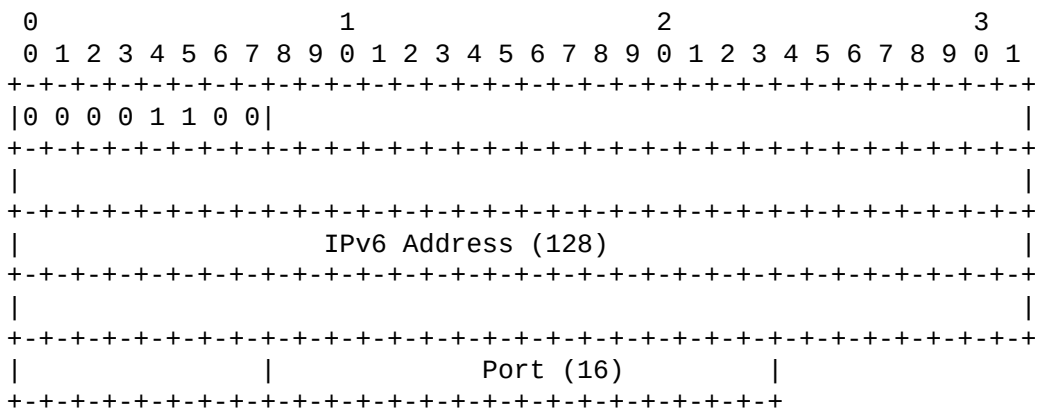
where the first octet is the PEX_REQ message (0x06).

A PEX_RESv4 message:



where the first octet is the PEX_RESv4 message (0x05) followed by the IPv4 address and the port number.

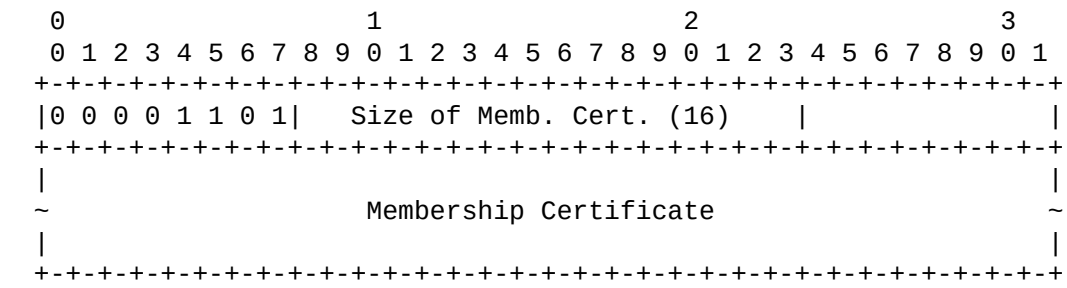
A PEX_RESv6 message:



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 12.2.1. 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.X690.2002]. 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.

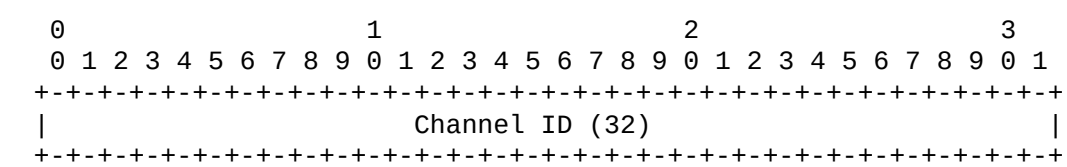
The URL contained in the name extension MUST follow the generic syntax for URLs [RFC3986], where its scheme component is "file", the host in the authority component is the DNS name or IP address of Peer P, the port in the authority component is the port of Peer P, and the path contains the swarm identifier 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:

```
file://192.0.2.0:6778/e5a12c7ad2d8fab33c699d1e198d66f79fa610c3
```

8.14. KEEPALIVE

Keep alives 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 keep-alive datagram:



8.15. Flow and Congestion Control

Explicit flow control is not required for PPSPP over UDP. In the case of video on demand, the receiver explicitly requests the content 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 stream bitrate, which the receiver must be able to process for a continuous playback. Should, for any reason, the receiver get saturated with data, the congestion control at the sender side will detect the situation and adjust the sending rate accordingly.

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 every day by millions of users as part of the uTP transmission protocol of BitTorrent [LBT] [LCOMPL] and is suitable for P2P streaming [PPSPPERF].

LEDBAT monitors the delay of the packets on the data path. It uses the one-way delay variations to react early and limit the congestion that the stream may induce in the network [RFC6817]. Using LEDBAT enables PPSPP to serve the content to other interested peers after the playback has finished (seeding), without disrupting the user. After the playback, the user might move to different tasks that use its network link, which are prioritized over PPSPP traffic. Hence, the user does not notice the background PPSPP traffic, which in turn increases the chances of seeding the content for a longer period of time.

The property of reacting early is not a problem in a peer-to-peer system where multiple sources offer the content. Considering the case of congestion near the sender, LEDBAT's early reaction impacts the transmission of chunks to the receiver. However, for the receiver, it is actually beneficial to learn early that the transmission from a particular source is impacted. The receiver can then choose to download time-critical chunks from other sources during its chunk picking phase.

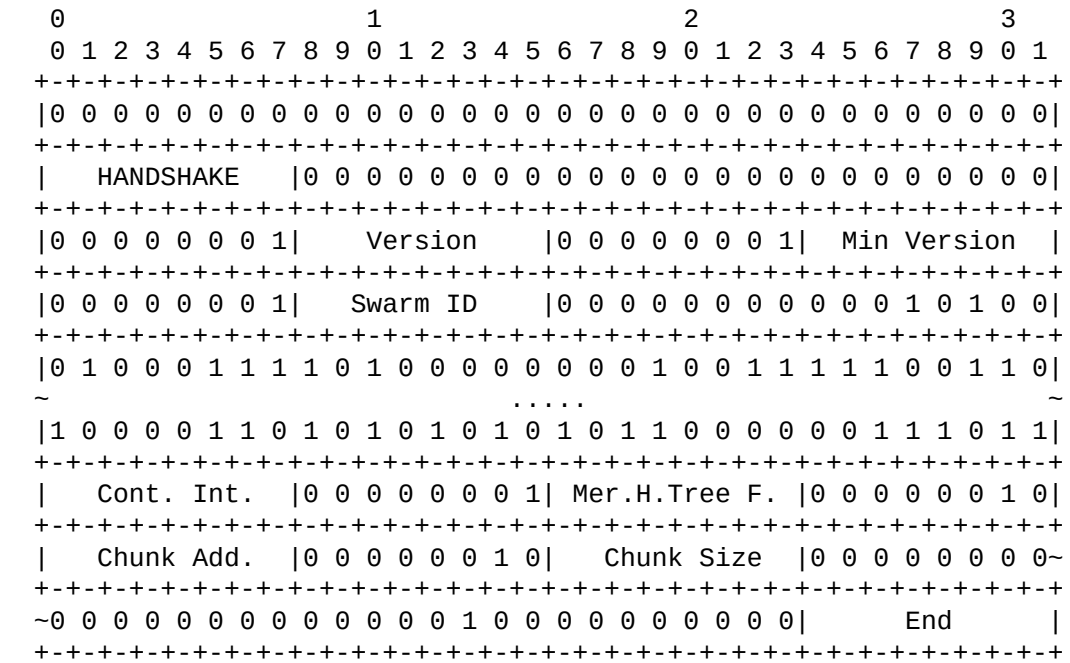
If the bottleneck is near the receiver, the receiver is indeed unlucky that transmissions from any source that runs through this bottleneck will back off quite fast due to LEDBAT. However, for the rest of the network (and the network operator), this is beneficial as the video-streaming system will back off early enough and not contribute too much to the congestion.

The power of LEDBAT is that its behavior can be configured. In the case of live streaming, a PPSPP deployer may want a more aggressive behavior to ensure quality of service. In that case, LEDBAT can be configured to be more aggressive. In particular, LEDBAT's queuing target delay value (TARGET in [RFC6817]) and other parameters can be adjusted such that it acts as aggressive as TCP (or even more). Hence, LEDBAT is an algorithm that works for many scenarios in a peer-to-peer context.

8.16. 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-zeros channel ID (0x00000000); followed by a HANDSHAKE message, whose payload is a locally unused; a random channel ID (in this case 0x00000001); and a list of protocol options. Channel IDs MUST be randomly chosen, as described in Section 12.1.



The protocol options are:

Version: 1

Minimum supported Version: 1

Swarm Identifier: A 32-byte root hash (47a0...b03b) identifying the content

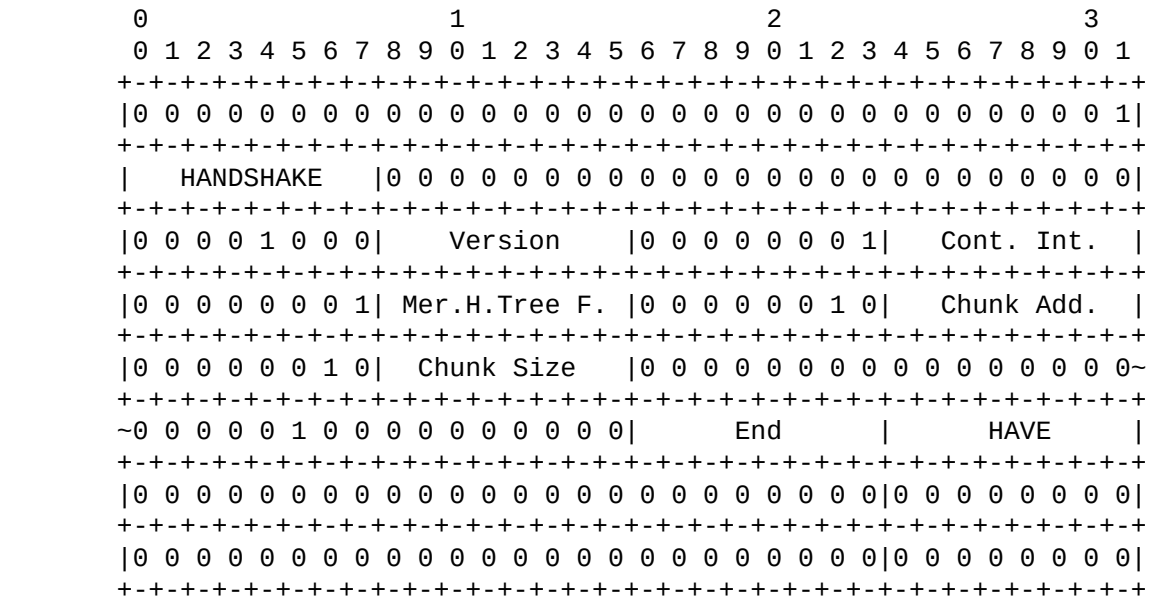
Content Integrity Protection Method: Merkle Hash Tree

Merkle Tree Hash Function: SHA-256

Chunk Addressing Method: 32-bit chunk ranges

Chunk Size: 1024

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; a random channel ID (0x00000008); and a list of protocol options; followed by any other messages it wants to send.

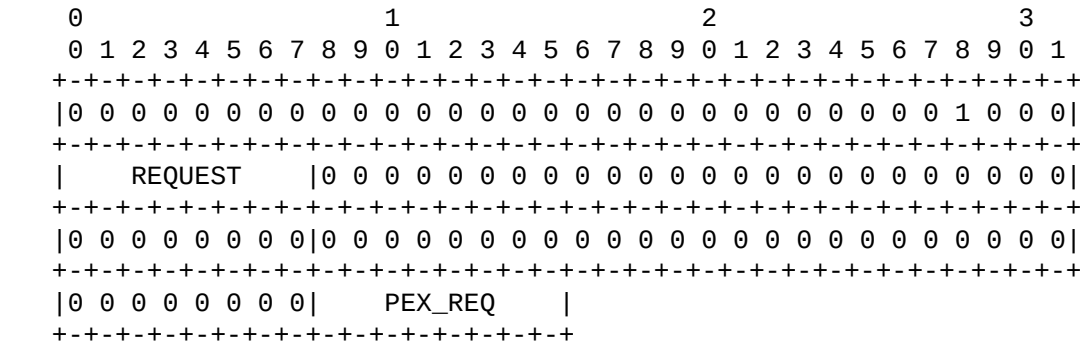


With the protocol options, the receiving peer agrees on speaking protocol version 1, on using the Merkle Hash Tree as the Content Integrity Protection Method, SHA-256 hash as the Merkle Tree Hash Function, 32-bit chunk ranges as the Chunk Addressing Method, and

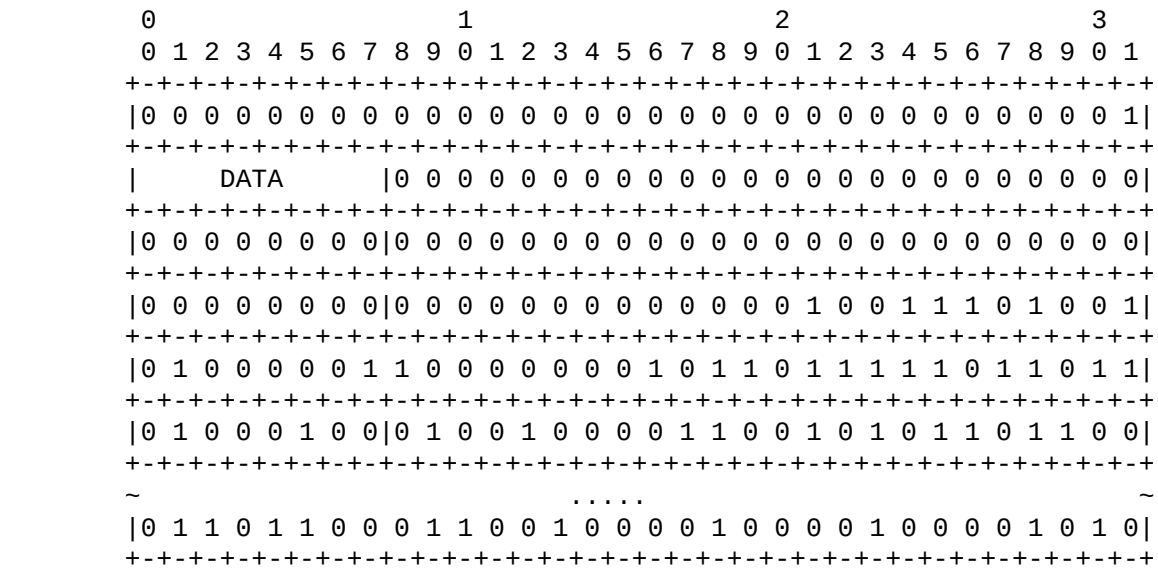
Chunk Size 1024. Furthermore, it sends a HAVE message within the same datagram, announcing that it has locally available the first chunk of content.

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 (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 proof that 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:

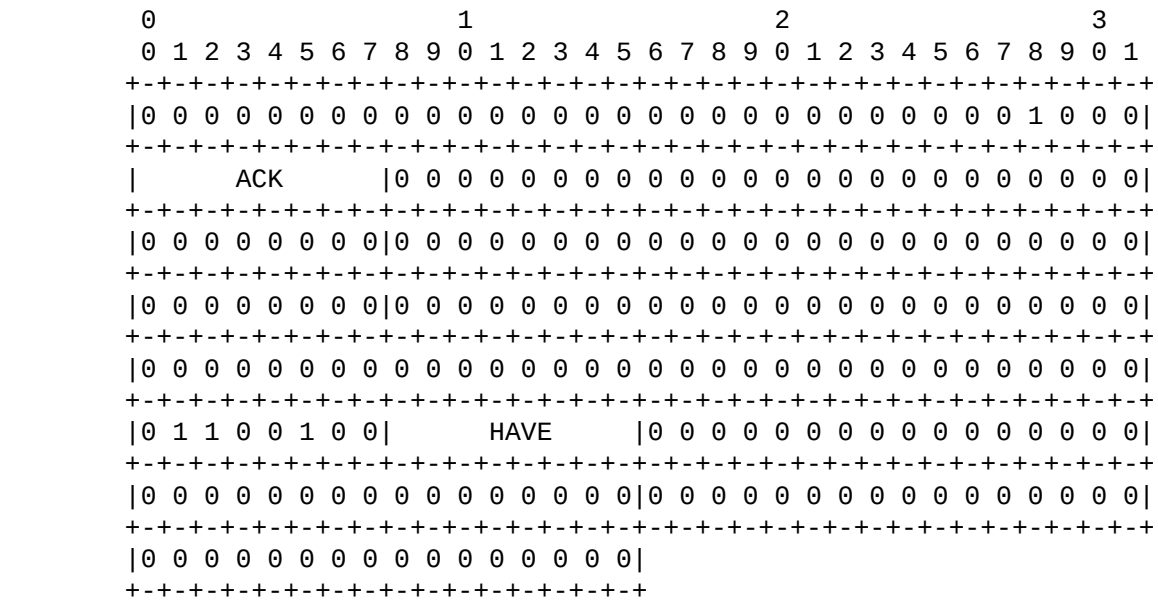
The 32-bit chunk range: 0,0 (the first chunk)

The timestamp value: 0004e94180b7db44

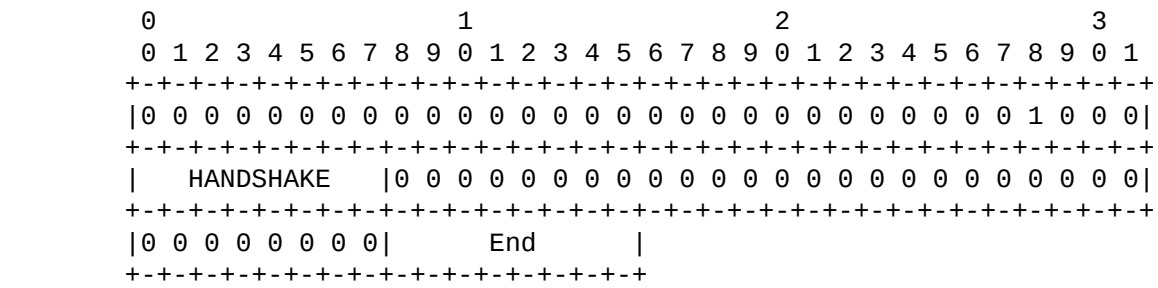
The data: 48656c6c6f20776f726c6421 (the "Hello world!" file)

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 ACK message for the first chunk, containing a one-way delay sample (100 ms). Furthermore, it also adds a HAVE message for the chunk.



At this point, the initiating peer has successfully retrieved the entire file. Then, it explicitly closes the connection by sending a HANDSHAKE message that contains an all-zeros Source Channel ID.



9. Extensibility

9.1. Chunk Picking Algorithms

Chunk (or piece) picking entirely depends on the receiving peer. The sending 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 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 or 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 sending peer.

10. IANA Considerations

IANA has created a new top-level registry called "Peer-to-Peer Streaming Peer Protocol (PPSPP)", which hosts the six new sub-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].

10.1. PPSPP Message Type Registry

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

10.2. PPSPP Option Registry

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

10.3. PPSPP Version Number Registry

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

10.4. PPSPP Content Integrity Protection Method Registry

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

10.5. PPSPP Merkle Hash Tree Function Registry

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

10.6. PPSPP Chunk Addressing Method Registry

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

11. 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.

11.1. Operations

11.1.1. Installation and Initial Setup

A content provider wishing to use PPSPP to distribute content should set up at least one PPSPP server. PPSPP servers need to have access to either some static content or 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 set up a tracking facility for the content by configuring, for example, a peer-to-peer streaming protocol tracker [PPSP-TP] 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 website 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 requester 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.

11.1.2. Migration Path

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

11.1.3. Requirements on Other Protocols and Functional Components

When using the peer-to-peer streaming protocol tracker, PPSPP requires a specific behavior from this protocol for security reasons, as detailed in Section 12.2.

11.1.4. 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.

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

Basic operation of the protocol can be easily verified when a tracker and swarm metadata 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.

11.1.6. 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	2 (SHA-256)	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 3.12

Table 8: PPSPP Defaults

11.2. 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 are faults detected and how are servers and end-user performance measured? 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.

11.2.1. 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. Therefore, PPSPP servers may 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 limiting 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].

11.2.2. 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.

11.2.3. 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.

11.2.4. 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.

11.2.5. 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 PPSP.OAM.REQ-3 in [RFC6972].

11.2.6. 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 12.7 describes a procedure for long-term exclusion.

12. Security Considerations

As any other network protocol, 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.

12.1. Security of the Handshake Procedure

Borrowing from the analysis in [RFC5971], the PPSPP may be attacked with three 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 Peer 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] (Section 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 zeros. 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 Peer 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 Peer A sends may already contain heavy payload, i.e., a chunk. This next datagram to Peer B will be prefixed with the chanB channel ID. When Peer 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] (Section 5.1)).

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

12.1.1. 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 Peer B pretending to be Peer A now need to intercept the message

from Peer B to Peer A to get Peer 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.

12.1.2. 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 Peer A included some HAVE messages, Peer B may add a chunk availability map to Peer A's state. In addition, Peer B may request some chunks from Peer A in the second datagram, and Peer 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 Stream Control Transmission Protocol (SCTP) [RFC4960] (Section 5.1) is quicker response time for the user. In the SCTP procedure, Peers 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 less time in PPSPP between starting the video stream and seeing the first images.

12.1.3. 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 Peer B assigned to Peer A, chanB. Furthermore, attacker Peer T would need to be able to spoof, e.g., REQUEST and HAVE messages from Peer A to cause Peer B to send heavy DATA messages to Peer A, or prevent Peer 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.

12.2. Secure Peer Address Exchange

As described in Section 3.10, Peer A can send Peer-Exchange messages PEX_RES to 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. 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..Bn; 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 Peer A's behavior. For example, when Peer 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.

12.2.1. 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 Peer A can check the certificate for a valid signature, the right swarm and liveliness, and only then consider contacting Peer 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 peer-to-peer streaming protocol tracker acts as certification authority.

12.2.2. Example: Tracker as Certification Authority

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 Peer A.

Peer A then includes this certificate when it sends a PEX_REScert to Peer B. Receiver Peer B verifies it against the tracker public key. This tracker public key should be part of the swarm's metadata, which Peer B received from a trusted source. Subsequently, Peer B can send the member certificate of Peer 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 Peer T sends. To counter this, Peer 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.

12.2.3. 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 Peer B with the tracker, and many peers joining the swarm will contact Peer 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 peer-to-peer streaming protocol tracker specification [PPSP-TP], 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 are 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, letting 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, Peer A can contact the tracker to find new candidates.

12.3. Support for Closed Swarms

Regarding PPSP.SEC.REQ-1 in [RFC6972], 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 [ECS-protocol].

12.4. Confidentiality of Streamed Content

Regarding PPSP.SEC.REQ-1 in [RFC6972], no extra mechanism is needed to support confidentiality in PPSPP. A content publisher wishing confidentiality should just distribute content in ciphertext and/or in a format to which Digital Rights Management (DRM) techniques have been applied. 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, by DTLS [RFC6347], or by IPsec [RFC4301].

When transmitting over DTLS, PPSPP can obtain the PMTU estimate maintained by the IP layer to determine how much payload can be put in a single datagram without fragmentation ([RFC6347], Section 4.1.1.1). If PMTU changes and the chunk size becomes too large to fit into a single datagram, PPSPP can choose to allow fragmentation by clearing the Don't Fragment (DF) bit. Alternatively, the content publisher can decide to use smaller chunks and transmit multiple in the same datagram when the MTU allows.

12.5. Strength of the Hash Function for Merkle Hash Trees

Implementations MUST support SHA-1 as the hash function for content integrity protection via Merkle hash trees. SHA-1 may be preferred over stronger hash functions by content providers because it reduces on-the-wire overhead. As such, it presents a trade-off between performance and security. The security considerations for SHA-1 are discussed in [RFC6194].

In general, note that the hash function is used in a hash tree, which makes it more complex to create collisions. 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-size chunks are used, the collision even 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 collisions.

12.6. Limit Potential Damage and Resource Exhaustion by Bad or Broken Peers

Regarding PPSP.SEC.REQ-2 in [RFC6972], this section provides an analysis of the potential damage a malicious peer can do with each message in the protocol, and how it is prevented by the protocol (implementation).

12.6.1. HANDSHAKE

- Secured against DoS Amplification attacks as described in Section 12.1.

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

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

12.6.2. HAVE

- Threat HAVE.1: Malicious Peer T can claim to have content that it does not. Subsequently, Peer T won't respond to requests.

  Solution: Peer A will consider Peer 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.

12.6.3. 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.

12.6.4. ACK

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

  Solution: Peer A will detect inconsistencies with the data it sent to Peer 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 that there is no congestion when in fact there is, causing Peer 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.

12.6.5. 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, Peer A will stop asking Peer T because of the atomic datagram principle and the content integrity protection. Subsequent unsolicited traffic from Peer 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 Peer T will be ignored.

12.6.6. REQUEST

- Threat REQUEST.1: Peer T could request lots from Peer A, leaving Peer 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.

12.6.7. 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.

12.6.8. CHOKE

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

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

12.6.9. 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 Peer T to be a slow peer and not ask it again.

12.6.10. PEX_RES

- Secured against amplification and Eclipse attacks as described in Section 12.2.

12.6.11. Unsolicited Messages in General

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

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

12.7. Exclude Bad or Broken Peers

This section is regarding PPSP.SEC.REQ-2 in [RFC6972]. 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.

13. References

13.1. Normative References

[CCITT.X690.2002]
International Telephone and Telegraph Consultative Committee, "ASN.1 encoding rules: Specification of basic encoding Rules (BER), Canonical encoding rules (CER) and Distinguished encoding rules (DER)", CCITT Recommendation X.690, July 2002.

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

[IANADNSSECALGNUM]
IANA, "Domain Name System Security (DNSSEC) Algorithm Numbers", March 2014, <http://www.iana.org/assignments/dns-sec-alg-numbers>.

[RFC1918] Rekhter, Y., Moskowitz, B., Karrenberg, D., J. de Groot, G., and E. Lear, "Address Allocation for Private Internets", BCP 5, RFC 1918, DOI 10.17487/RFC1918, February 1996, <http://www.rfc-editor.org/info/rfc1918>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC3110] Eastlake 3rd, D., "RSA/SHA-1 SIGs and RSA KEYs in the Domain Name System (DNS)", RFC 3110, DOI 10.17487/RFC3110, May 2001, <http://www.rfc-editor.org/info/rfc3110>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <http://www.rfc-editor.org/info/rfc3986>.

[RFC4034] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Resource Records for the DNS Security Extensions", RFC 4034, DOI 10.17487/RFC4034, March 2005, <http://www.rfc-editor.org/info/rfc4034>.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <http://www.rfc-editor.org/info/rfc4291>.

[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, DOI 10.17487/RFC5280, May 2008, <http://www.rfc-editor.org/info/rfc5280>.

[RFC5702] Jansen, J., "Use of SHA-2 Algorithms with RSA in DNSKEY and RRSIG Resource Records for DNSSEC", RFC 5702, DOI 10.17487/RFC5702, October 2009, <http://www.rfc-editor.org/info/rfc5702>.

[RFC5905] Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, DOI 10.17487/RFC5905, June 2010, <http://www.rfc-editor.org/info/rfc5905>.

[RFC6605] Hoffman, P. and W. Wijngaards, "Elliptic Curve Digital Signature Algorithm (DSA) for DNSSEC", RFC 6605, DOI 10.17487/RFC6605, April 2012, <http://www.rfc-editor.org/info/rfc6605>.

[RFC6817] Shalunov, S., Hazel, G., Iyengar, J., and M. Kuehlewind, "Low Extra Delay Background Transport (LEDBAT)", RFC 6817, DOI 10.17487/RFC6817, December 2012, <http://www.rfc-editor.org/info/rfc6817>.

13.2. Informative References

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

[BINMAP] Grishchenko, V. and J. Pouwelse, "Binmaps: Hybridizing Bitmaps and Binary Trees", Delft University of Technology Parallel and Distributed Systems Report Series, Report number PDS-2011-005, ISSN 1387-2109, April 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, April 2006.

[BITTORRENT]
Cohen, B., "The BitTorrent Protocol Specification", BitTorrent Enhancement Proposal 3, February 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, October 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, November 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, August 2011.

[ECS-protocol]
Gabrijelcic, D., "Enhanced Closed Swarm protocol", Work in Progress, draft-ppsp-gabrijelcic-ecs-01, June 2013.

[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, June 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, CA, USA, January 2008.

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

[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, August 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, August 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, August 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.

[P2PWIKI] Bakker, A., Petrocco, R., Dale, M., Gerber, J., Grishchenko, V., Rabaioli, D., and J. Pouwelse, "Online video using BitTorrent and HTML5 applied to Wikipedia", IEEE International Conference on Peer-to-Peer Computing (P2P'10), Delft, The Netherlands, August 2010.

[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.

[PPSP-TP] Cruz, R., Nunes, M., Yingjie, G., Xia, J., Huang, R., Taveira, J., and D. Lingli, "PPSP Tracker Protocol-Base Protocol (PPSP-TP/1.0)", Work in Progress, draft-ietf-ppsp-base-tracker-protocol-09, March 2015.

[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, September 2012.

[RFC2564] Kalbfleisch, C., Krupczak, C., Presuhn, R., and J. Saperia, "Application Management MIB", RFC 2564, DOI 10.17487/RFC2564, May 1999, <http://www.rfc-editor.org/info/rfc2564>.

[RFC2790] Waldbusser, S. and P. Grillo, "Host Resources MIB", RFC 2790, DOI 10.17487/RFC2790, March 2000, <http://www.rfc-editor.org/info/rfc2790>.

[RFC2975] Aboba, B., Arkko, J., and D. Harrington, "Introduction to Accounting Management", RFC 2975, DOI 10.17487/RFC2975, October 2000, <http://www.rfc-editor.org/info/rfc2975>.

[RFC3365] Schiller, J., "Strong Security Requirements for Internet Engineering Task Force Standard Protocols", BCP 61, RFC 3365, DOI 10.17487/RFC3365, August 2002, <http://www.rfc-editor.org/info/rfc3365>.

[RFC3729] Waldbusser, S., "Application Performance Measurement MIB", RFC 3729, DOI 10.17487/RFC3729, March 2004, <http://www.rfc-editor.org/info/rfc3729>.

[RFC4113] Fenner, B. and J. Flick, "Management Information Base for the User Datagram Protocol (UDP)", RFC 4113, DOI 10.17487/RFC4113, June 2005, <http://www.rfc-editor.org/info/rfc4113>.

[RFC4150] Dietz, R. and R. Cole, "Transport Performance Metrics MIB", RFC 4150, DOI 10.17487/RFC4150, August 2005, <http://www.rfc-editor.org/info/rfc4150>.

[RFC4193] Hinden, R. and B. Haberman, "Unique Local IPv6 Unicast Addresses", RFC 4193, DOI 10.17487/RFC4193, October 2005, <http://www.rfc-editor.org/info/rfc4193>.

[RFC4301] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, DOI 10.17487/RFC4301, December 2005, <http://www.rfc-editor.org/info/rfc4301>.

[RFC4821] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, DOI 10.17487/RFC4821, March 2007, <http://www.rfc-editor.org/info/rfc4821>.

[RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007, <http://www.rfc-editor.org/info/rfc4960>.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <http://www.rfc-editor.org/info/rfc5226>.

[RFC5389] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing, "Session Traversal Utilities for NAT (STUN)", RFC 5389, DOI 10.17487/RFC5389, October 2008, <http://www.rfc-editor.org/info/rfc5389>.

[RFC5424] Gerhards, R., "The Syslog Protocol", RFC 5424, DOI 10.17487/RFC5424, March 2009, <http://www.rfc-editor.org/info/rfc5424>.

[RFC5706] Harrington, D., "Guidelines for Considering Operations and Management of New Protocols and Protocol Extensions", RFC 5706, DOI 10.17487/RFC5706, November 2009, <http://www.rfc-editor.org/info/rfc5706>.

[RFC5971] Schulzrinne, H. and R. Hancock, "GIST: General Internet Signalling Transport", RFC 5971, DOI 10.17487/RFC5971, October 2010, <http://www.rfc-editor.org/info/rfc5971>.

[RFC6194] Polk, T., Chen, L., Turner, S., and P. Hoffman, "Security Considerations for the SHA-0 and SHA-1 Message-Digest Algorithms", RFC 6194, DOI 10.17487/RFC6194, March 2011, <http://www.rfc-editor.org/info/rfc6194>.

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, DOI 10.17487/RFC6241, June 2011, <http://www.rfc-editor.org/info/rfc6241>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <http://www.rfc-editor.org/info/rfc6347>.

[RFC6709] Carpenter, B., Aboba, B., Ed., and S. Cheshire, "Design Considerations for Protocol Extensions", RFC 6709, DOI 10.17487/RFC6709, September 2012, <http://www.rfc-editor.org/info/rfc6709>.

[RFC6972] Zhang, Y. and N. Zong, "Problem Statement and Requirements of the Peer-to-Peer Streaming Protocol (PPSP)", RFC 6972, DOI 10.17487/RFC6972, July 2013, <http://www.rfc-editor.org/info/rfc6972>.

[RFC7285] Alimi, R., Ed., Penno, R., Ed., Yang, Y., Ed., Kiesel, S., Previdi, S., Roome, W., Shalunov, S., and R. Woundy, "Application-Layer Traffic Optimization (ALTO) Protocol", RFC 7285, DOI 10.17487/RFC7285, September 2014, <http://www.rfc-editor.org/info/rfc7285>.

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

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

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

[SWIFTIMPL] Grishchenko, V., Paananen, J., Pronchenkov, A., Bakker, A., and R. Petrocco, "Swift reference implementation", 2015, <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, May 2003.

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.

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

Authors' Addresses

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

Email: arno@cs.vu.nl

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

Email: r.petrocco@gmail.com

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

Email: victor.grishchenko@gmail.com