PPSP
Internet-Draft
Intended status: Informational
Expires: April 25, 2013

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

October 22, 2012

Peer-to-Peer Streaming Peer Protocol (PPSPP)
draft-ietf-ppsp-peer-protocol-03

Abstract

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

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 pre-recorded (on-demand) and live audio/video content. It is based on the peer-to-peer paradigm where clients consuming the content are put on equal footing with the servers initially providing the content, to create a system where everyone can potentially provide upload bandwidth.

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

PPSPP has also been designed to be extensible for different transports and use cases. Hence, PPSPP is a generic protocol which can run directly on top of UDP, TCP, or other protocols. As such, PPSPP defines a common set of messages that make up the protocol, which can have different representations on the wire depending on the lower-level protocol used. When the lower-level transport allows, PPSPP can also use different congestion control algorithms.

At present, PPSPP is set to run on top of UDP using LEDBAT for congestion control [I-D.ietf-ledbat-congestion]. 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. LEDBAT may be replaced with a different algorithm when the work of the IETF working group on RTP Media Congestion Avoidance Techniques (RMCAT) [RMCATCHART] matures.

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 chunk addressing and content integrity protection, if the defaults are not fit for a particular use case. In addition, it can work with different peer

discovery schemes, such as centralized trackers or fast Distributed Hash Tables [JIM11]. Finally, in this default setup, PPSPP maintains only a small amount of state per peer. A reference implementation of PPSPP over UDP is available [SWIFTIMPL].

1.2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

1.3. Terminology

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

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

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

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

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

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

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

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

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

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

hash
: The result of applying a cryptographic hash function, more specifically a modification detection code (MDC) [HAC01], such as SHA-1 [FIPS180-3], to a piece of data.

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

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

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

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

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

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

seeding/leeching
: When a peer A is seeding it means that A has downloaded a static content asset completely and is now offering it for others to download. If peer A does not yet have all content or is not offering it for download, A is said to be leeching.

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 9).

The overall operation of PPSPP is illustrated in the following examples. The examples assume that UDP is used for transport, the recommended method for content integrity protection (Merkle hash trees) is used, and that a specific policy is used for selecting which chunks to download.

2.1. Joining a Swarm

Consider a peer A that wants to download a certain content asset. To commence a PPSPP download, peer A must have the swarm ID of the content and a list of one or more tracker contact points (e.g. host+port). The list of trackers is optional in the presence of a decentralized tracking mechanism.

Peer A now registers with the tracker following e.g. the PPSP tracker protocol [I-D.ietf-ppsp-reqs] and receives the IP address and port of peers already in the swarm, say B, C, and D. Peer A now sends a datagram containing a HANDSHAKE message to B, C, and D. This message conveys protocol options and may serve as an end-to-end check that the peers are actually in the correct swarm (in which case it contains the ID of the swarm).

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

2.2. Exchanging Chunks

In response to B and C, A sends new datagrams to B and C containing REQUEST messages. A REQUEST message indicates the chunks that a peer wants to download, and thus contains a chunk specification. The REQUEST messages to B and C refer to disjunct sets of chunks. B and C respond with datagrams containing HAVE, DATA and, in this example, INTEGRITY messages. In the Merkle hash tree content protection scheme (see Section 5.1), the INTEGRITY messages contain all cryptographic hashes that peer A needs to verify the integrity of the content chunk sent in the DATA message. Using these hashes peer A verifies that the chunks received from B and C are correct. It also updates the chunk availability of B and C using the information in the received HAVE messages.

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

Peer D also sends HAVE messages to A when it downloads chunks from other peers. When D is willing to accept REQUESTs from A, D sends a datagram with an UNCHOKE messages to inform A. If B or C decide to choke A they sending a CHOKE message and A should then re-request from other peers. B and C may continue to send HAVE, REQUEST, or periodic KEEPALIVE messages such that A keeps sending them HAVE messages.

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

2.3. Leaving a Swarm

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

3. Messages

In general, no error codes or responses are used in the protocol; absence of any response indicates an error. Invalid messages are discarded, and further communication with the peer SHOULD be stopped. The rationale is that it is sufficient to classify peers as either good (i.e., responding) or bad and only use the good ones. This behavior allows a peer to deal with slow, crashed and (silent) malicious peers.

For the sake of simplicity, one swarm of peers generally deals with one content asset (e.g. file) only. Retrieval of a collections of files can be done either by using multiple swarms or by using an external storage mapping from the linear byte space of a single swarm to different files, transparent to the protocol as described in Section 3.12.

3.1. HANDSHAKE

The initiating peer and the addressed peer MUST send a HANDSHAKE message in the first datagrams they exchange. The payload of the HANDSHAKE message is a sequence of protocol options. Example options are the content integrity protection scheme used and an option to specify the swarm identifier. The latter option MAY be used as an end-to-end check that the peers are actually in the correct swarm. Protocol options are specified in Section 8.

After the handshakes are exchanged, the initiator knows that the peer really responds. Hence, the second datagram the initiator sends MAY already contain some heavy payload. To minimize the number of initialization round-trips, the first two datagrams exchanged MAY also contain some minor payload, e.g. HAVE messages to indicate the current progress of a peer or a REQUEST (see Section 3.7).

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 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 has successfully checked the integrity of a chunk or interval of chunks it MUST send a HAVE message to all peers it wants to interact with in the near future. The latter confinement allows the HAVE message to be used as a method of choking. The HAVE message MUST contain the chunk specification of

the received chunks. A receiving peer MUST not send a HAVE message to peers for which the handshake procedure is still incomplete, see Section 13.1.

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 [I-D.ietf-ledbat-congestion] 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

When PPSPP is run over an unreliable transport protocol, an implementation MAY choose to use ACK messages to acknowledge received data. When used, a receiving peer that has successfully checked the integrity of a chunk or interval of chunks C it MUST send an ACK message containing a chunk specification for C. As LEDBAT is used, an ACK message MUST contain the one-way delay, computed from the peer's current system time received in the DATA message. A peer MAY delay sending ACK messages as defined in the LEDBAT specification.

3.5. INTEGRITY

The INTEGRITY message carries information required by the receiver to verify the integrity of a chunk. Its payload depends on the content integrity protection scheme used. When the recommended method of Merkle hash trees is used, the datagram carrying the DATA message MUST include the cryptographic hashes that are necessary for a receiver to check the integrity of the chunk in the form of INTEGRITY messages. What are the necessary hashes is explained in Section 5.3.

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 and a digital signature. Its exact payload depends on the live content integrity protection scheme used, see Section 7.1.

3.7. REQUEST

While bulk download protocols normally do explicit requests for certain ranges of data (i.e., use a pull model, for example,

BitTorrent [BITTORRENT]), live streaming protocols quite often use a request-less push model to save round trips. PPSPP supports both models of operation.

A peer MAY send a REQUEST message that MUST contain the specification of the chunks it wants to download. A peer receiving a REQUEST message MAY send out requested pieces. When peer Q receives multiple REQUESTs from the same peer P peer Q SHOULD process the REQUESTs sequentially. 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, 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 MAY 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 10.1), detects that a request for urgent data might not be served on time, a request for the same data MAY be sent to a different peer. When a peer P decides to request urgent data from a peer Q, peer P SHOULD send a CANCEL message to all the peers to which the data has been previously requested. The CANCEL message contains the specification of the chunks P no longer wants to request. In addition, when peer Q receives a HAVE message for the urgent data from peer P, peer Q MUST also cancel the previous REQUEST(s) from P. In other words, the HAVE message acts as an implicit CANCEL.

3.9. CHOKE and UNCHOKE

Peer A MAY send a CHOKE message to peer B to signal it will no longer be responding to REQUEST messages from B, for example, because A's upload capacity is exhausted. Peer A MAY send a subsequent UNCHOKE message to signal that it will respond to new REQUESTs from B again (A SHOULD discard old requests). When peer B receives a CHOKE message from A it MUST not send new REQUEST messages and should not expect answers to any outstanding ones. The CHOKE and UNCHOKE messages are informational as a peer is not required to respond to REQUESTs.

3.10. Peer Address Exchange and NAT Hole Punching

3.10.1. PEX_REQ and PEX_RES Messages

Peer address exchange messages (or PEX messages for short) are common in many peer-to-peer protocols. By exchanging peer addresses in gossip fashion, peers relieve central coordinating entities (the trackers) from unnecessary work. PPSPP optionally features two types of PEX messages: PEX_REQ and PEX_RES. A peer that wants to retrieve some peer addresses MUST send a PEX_REQ message. The receiving peer MAY respond with a PEX_RES message containing the (potentially signed) addresses of several peers. The addresses MUST be of peers it has exchanged messages with in the last 60 seconds to guarantee liveliness. Alternatively, the receiving peer MAY ignore PEX messages if uninterested in obtaining new peers or because of security considerations (rate limiting) or any other reason. The PEX messages can be used to construct a dedicated tracker peer.

As peer-address exchange enables a number of attacks it should not be used outside a benign environment unless extra security measures are in place. These security measures, which involve exchanging addresses in cryptographically signed swarm-membership certificates, are described in Section 13.2.

3.10.2. Hole Punching via PPSPP Messages

PPSPP can be used in combination with STUN [RFC5389]. In addition, the native PEX messages can be used to do simple NAT hole punching [SNP], as follows. When peer A introduces peer B to peer C by sending a PEX_RES message to C, it SHOULD also send a PEX_RES message to B introducing C. These messages SHOULD be within 2 seconds from each other, but MAY not be simultaneous, instead leaving a gap of twice the "typical" RTT, i.e. 300-600 ms. As a result, the peers are supposed to initiate handshakes to each other thus forming a simple NAT hole punching pattern where the introducing peer effectively acts as a STUN server. Note that the PEX_RES message is sent without a prior PEX_REQ in this case.

3.11. Keep Alive Signalling

A peer MUST send a "keep alive" message periodically to each peer it wants to interact with in the future, but has no other messages to send them at present. PPSPP does not define an explicit message type for "keep alive" messages. In the PPSP-over-UDP encapsulation they are implemented as simple datagrams consisting of a 4-byte channel number only, see Section 9.3 and Section 9.4.

3.12. Storage Independence

Note PPSPP does not prescribe how chunks are stored. This also allows users of PPSPP to map different files into a single swarm as in BitTorrent multi-file torrents [BITTORRENT], and more innovative storage solutions when variable-sized chunks are used.

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.

4.1. Start-End Ranges

A chunk specification consists of a list of (start specification,end specification) pairs. A list MUST contain at least one pair. Each pair identifies a range of chunks. The start and end specifications can use one of multiple addressing schemes. Two schemes are currently defined.

4.1.1. Chunk Ranges

The start and end specification are both chunk identifiers. A PPSPP peer MUST support this scheme.

4.1.2. Byte Ranges

The start and end specification are byte offsets in the content. A PPSPP peer MAY support this scheme.

4.2. Bin Numbers

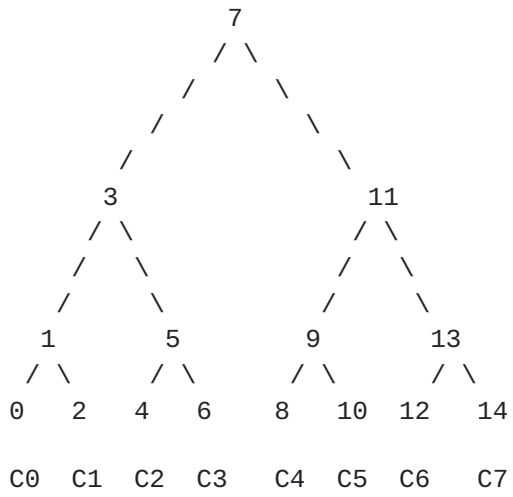
PPSPP introduces a novel method of addressing chunks of content called "bin numbers" (or "bins" for short). Bin numbers allow the addressing of a binary interval of data using a single integer. This reduces the amount of state that needs to be recorded per peer and the space needed to denote intervals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

In bin addressing, the smallest binary interval is a single chunk (e.g. a block of bytes which may be of variable size), the largest interval is a complete range of 2**63 chunks. In a novel addition to the classical scheme, these intervals are numbered in a way which

lays them out into a vector nicely, which is called bin numbering, as follows. Consider an chunk interval of width W. To derive the bin numbers of the complete interval and the subintervals, a minimal balanced binary tree is built that is at least W chunks wide at the base. The leaves from left-to-right correspond to the chunks 0..W-1 in the interval, and have bin number I*2 where I is the index of the chunk (counting beyond W-1 to balance the tree). The bin number of higher level nodes P in the tree is calculated as follows:

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

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



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

Figure 1

So bin 7 represents the complete interval, bin 3 represents the interval of chunk 0..3, bin 1 represents the interval of chunks 0 and 1, and bin 2 represents chunk C1. The special numbers 0xFFFFFFFF (32-bit) or 0xFFFFFFFFFFFFFFFF (64-bit) stands for an empty interval, and 0x7FFF...FFF stands for "everything".

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

4.3. In Messages

4.3.1. In HAVE Messages

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

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

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

4.3.2. In ACK Messages

When PPSPP is run over an unreliable transport protocol, an implementation MAY choose to use ACK messages to acknowledge received data. When a receiving peer has successfully checked the integrity of a chunk or interval of chunks C it MUST send a ACK message containing the chunk specification of its biggest, complete, interval covering C to the sending peer (see HAVE).

4.4. Compatibility

In principle, peers using range addressing and peers using bin numbering can interact, with some limitations. Alternatively, a peer A MAY refuse to interact with a peer B using a different addressing scheme. In that case, A MUST respond to B'S HANDSHAKE message by sending an explicit close (see Section 9.4). PPSPP presently supports only interaction between willing peers when fixed sized chunks are used, as follows:

When a bin peer sends a message containing a chunk specification to a byte-range peer it MUST translate its internal bin numbers to byte ranges. When a byte range peer sends a message with a chunk specification message to a bin peer, it MUST round its internal byte ranges to 1 or more bins. For the latter translation, the byte-range peer MUST know the fixed chunk size used (which it should receive

along with the swarm identifier). When a range translates to multiple bins, the byte-range peer the byte-range peer should send multiple e.g. HAVE messages. Note that the bin peer may not be able to request all content the byte-range peer has if it does not have an integral number of chunks.

Aside: Translation from bytes to bins is possible for variable sized chunks only when the byte-range peer has extra information. In particular, it will need to know the individual sizes of the chunks from the start of the content till the byte range it wants to convey to the bin peer.

A similar translation MUST be done for translating between bins and chunk ranges. Chunk ranges are directly translatable to bins. Assuming ranges are intervals of a list of chunks numbered 0...N, for a given bin number "bin" and bitwise operations AND and OR:

```
startrange = (bin AND (bin + 1))/2

endrange = ((bin OR (bin + 1)) - 1)/2
```

The reverse translation may require a chunk range to be rounded to the largest binary interval it covers, or for a range be translated to a series of bin numbers that should be sent using multiple (e.g. HAVE) messages.

Finally, byte-range peers can interact with chunk-range peers, by using the direct translation from chunks into bytes and by rounding byte ranges into chunk ranges. The latter requires the byte-range peer to know the fixed chunk size.

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.

This section describes the recommended method for bad content detection, the Merkle Hash Tree scheme, which SHOULD be implemented for protecting static content. It can also be efficiently used in protecting live streams, as explained below and in Section 7.1.

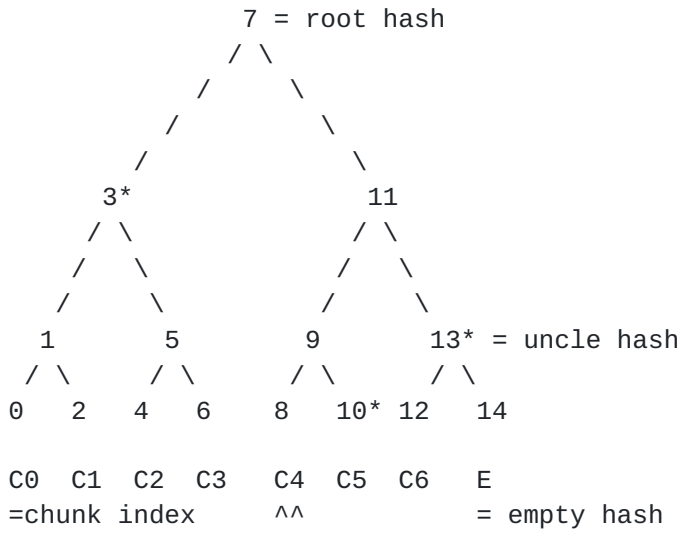
The Merkle hash tree scheme can use different chunk addressing schemes. All it requires is the ability to address a range of chunks. In the following description abstract node IDs are used to

identify nodes in the tree. On the wire these are translated to the corresponding range of chunks in the chosen chunk addressing scheme. When bin numbering is used, node IDs correspond directly to bin numbers in the INTEGRITY message, see below.

5.1. Merkle Hash Tree Scheme

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

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



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

Figure 2

5.2. Content Integrity Verification

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

In the case of live streaming the tree of chunks grows dynamically and the root hash is undefined or, more precisely, transient, as long as new data is generated by the live source. Section 7.1 defines a method for content integrity verification for live streams that works with such a dynamic tree. Although the tree is dynamic, content verification works the same for both live and predefined content, resulting in a unified method for both types of streaming.

5.3. The Atomic Datagram Principle

As explained above, a datagram consists of a sequence of messages. Ideally, every datagram sent must be independent of other datagrams, so each datagram SHOULD be processed separately and a loss of one datagram MUST NOT disrupt the flow. Thus, as a datagram carries zero or more messages, neither messages nor message interdependencies should span over multiple datagrams.

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

The hashes necessary to verify a chunk are in principle its sibling's hash and all its uncle hashes, but the set of hashes to send can be optimized. Before sending a packet of data to the receiver, the sender inspects the receiver's previous acknowledgments (HAVE or ACK) to derive which hashes the receiver already has for sure. Suppose, the receiver had acknowledged chunks C0 and C1 (first two chunks of the file), then it must already have uncle hashes 5, 11 and so on. That is because those hashes are necessary to check C0 and C1 against the root hash. Then, hashes 3, 7 and so on must be also known as they are calculated in the process of checking the uncle hash chain. Hence, to send chunk C7, the sender needs to include just the hashes for nodes 14 and 9, which let the data be checked against hash 11 which is already known to the receiver.

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

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

5.4. INTEGRITY Messages

Concretely, a peer that wants to send a chunk of content creates a datagram that MUST consist of one or more INTEGRITY messages and a DATA message. The datagram MUST contain a INTEGRITY message for each hash the receiver misses for integrity checking. A INTEGRITY message for a hash MUST contain the chunk specification corresponding to the

node ID of the hash and the hash data itself. The chunk specification corresponding to a node ID is defined as the range of chunks formed by the leaves of the subtree rooted at the node. For example, node 3 in Figure 2 denotes chunks 0,2,4,6, so the chunk specification should denote that interval. The DATA message MUST contain the chunk specification of the chunk and chunk itself. A peer MAY send the required messages for multiple chunks in the same datagram.

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 PPSP this implies having to download a large number of hashes before content download can begin. This, in turn, increases time-till-playback for end users, making this method unsuited for streaming.

The overhead of using Merkle hash trees is limited. The size of the hash tree expressed as the total number of nodes depends on the number of chunks the content is divided (and hence the size of chunks) following this formula:

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

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

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

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

Table 1: Overhead for the example tree

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

6. Merkle Hash Trees and The Automatic Detection of Content Size

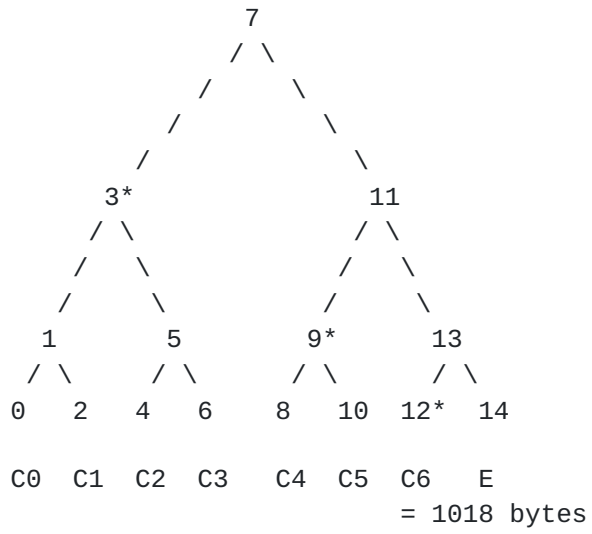
In PPSPP, the root hash of a static content asset, such as a video file, along with some peer addresses is sufficient to start a download. In addition, PPSPP can reliably and automatically derive the size of such content from information received from the network when fixed sized chunks are used. As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

6.1. Peak Hashes

The ability for a newcomer peer to detect the size of the content depends heavily on the concept of peak hashes. Peak hashes, in general, enable two cornerstone features of PPSPP: reliable file size detection and download/live streaming unification (see Section 7). The concept of peak hashes depends on the concepts of filled and incomplete nodes. Recall that when constructing the binary trees for content verification and addressing the base of the tree may have more leaves than the number of chunks in the content. In the Merkle hash tree these leaves were assigned empty all-zero hashes to be able to calculate the higher level hashes. A filled node is now defined as a node that corresponds to an interval of leaves that consists only of hashes of content chunks, not empty hashes. Reversely, an incomplete (not filled) node corresponds to an interval that contains

also empty hashes, typically an interval that extends past the end of the file. In the following figure nodes 7, 11, 13 and 14 are incomplete the rest is filled.

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



Peak hashes in a Merkle hash tree.

Figure 3

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

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

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

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

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

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

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

6.2. Procedure

A PPSPP implementation that wants to use automatic size detection MUST operate as follows. When a peer B sends a DATA message for the first time to a peer A, B MUST include all the peak hashes for the content in the same datagram, unless A has already signalled earlier in the exchange that it knows the peak hashes by having acknowledged any chunk. The receiver A MUST check the peak hashes against the root hash to determine the approximate content size. To obtain the definite content size peer A MUST download the last chunk of the content from any peer that offers it.

7. 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).

7.1. Content Authentication

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

In the "Sign All" method, the live injector signs each chunk of content using a private key and peers, upon receiving the chunk, check the signature using the corresponding public key obtained from a trusted source. In particular, in PPSP, the swarm ID of the live stream is that public key. The signature is sent along with the DATA message containing the relevant chunk using the SIGNED_INTEGRITY message.

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 case. The use of Merkle hash trees reduces the number of signing and verification operations per second, that is, provide signature amortization similar to the approach described in [SIGMCAST].

7.1.1. Sign All

Even with "Sign All", the number of cryptographic operations will be limited. For example, consider a 25 frame/second video transmitted over UDP. When each frame is transmitted in its own chunk, only 25 signature verification operations per second are required, for the receiving peer, for bitrates up to ~12.8 megabit/second over UDP. For higher bitrates multiple UDP packets per frame are needed.

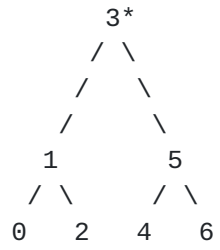
7.1.2. Unified Merkle Tree

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

The live injector creates a number of chunks N, a fixed power of 2 (N>=2), which are added as new leaves to the existing hash tree, expanding the tree as required. As a result of this expansion, the tree will have gotten a set of new peak hashes (see Section 6.1).

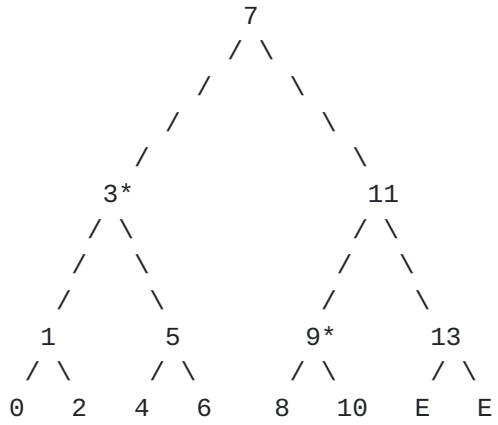
The injector now signs only the peak hashes in this set that are not in the old set of peak hashes. For N being a power of 2 there will just be one new peak hash (see below). This complementary signed peak is distributed to the peers. Receiving peers will verify the signature on the signed peak against the swarm ID, update their tree and request the new chunks.

To illustrate this procedure, consider the injector has generated the tree shown in Figure 4 and it is connected to several peers that currently have the same tree and all chunks. In this tree the root node 3 is also the peak node for this tree. Now the injector generates N=2 new chunks. As a result the tree expands as shown in Figure 5. The two new pieces 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 peak nodes in this tree are 3 and 9.



Current live tree

Figure 4



Next current live tree

Figure 5

The injector now needs to inform its peers of the changed tree, in particular the addition of the new complementary peak hash 9. To this extent, it sends a HASH message with the hash of node 9, a SIGNED_INTEGRITY message with the signature of the hash of node 9 and a HAVE message for node 9. The receiving peers now expand their view of the tree. Next, the peers will request e.g. chunk 8 from the injector by sending a REQUEST message. The injector responds by sending the requester the HASH of node 10, and a DATA message with chunk 8. This allows the peer to verify the chunk against peak hash 9 which is signed by the trusted injector.

The injector MAY send HAVE messages for the chunks it creates immediately, and allow peers to retrieve them. This optimizes the use of the injector's bandwidth. Peers MUST NOT forward these chunks to others until they have received and checked the peak hash signature and the necessary hashes.

This procedure generates just 1 new peak hash for every N blocks, so it requires just one signature on each iteration, making it N times cheaper than "Sign All". To see why just 1 new peak hash is generated each iteration let's return to the definition of a peak hash in a tree, from Section 6.1. A peak hash is the hash of a filled node in the Merkle tree, whose sibling is an incomplete node. Now consider the above procedure where N chunks (with N a power of 2) are added to a tree at each iteration. In the first iteration, the tree consists of just N leaves, therefore the only peak is the root of the tree. In the second iteration, the tree consists of 2N peaks and the only peak is the root of that bigger tree (depicted in Figure 4 for N=2). In the third iteration, we have 3N chunks as leaves and a tree that is 4N wide (to span the 3N chunks) and hence has N empty leaves (depicted in Figure 5 for N=2). This implies that the tree has 2 peaks, notably the peak from the previous iteration (node 3 in the figure) and the top of the subtree of the N chunks that were added last (node 9 in the figure). Although this iteration has two peaks, there is only one new peak as the expanded tree overlaps with the tree from the previous iteration. In the fourth iteration, we have a complete balanced tree again, and just a single new peak. It is now easy to see that this process in which previous peaks are either consumed into a single new peak, or peak sets overlap with just 1 new addition yields a single new peak per N chunks.

From this we can conclude that the injector has to sign less hashes than in the "Sign All" method. A receiving peer therefore also has to verify less signatures. It does additionally need to check one or more hashes per chunk via the Merkle Tree scheme, but this requires

less CPU than a signature verification for every chunk. This approach is similar to signature amortization via Merkle Tree Chaining [SIGMCAST]. The downside of this amortization of signature costs over several chunks is that latency will increase. A receiving peer now has to wait for the signature before delivering the chunks to the higher layers responsible for playback [POLLIVE], unless some (optimistic) optimisations are made. It MUST check the signature before forwarding the chunks to other peers.

The number of chunks per signature N MUST be a fixed power of 2 (N>=2). The procedure does not preclude using variable-sized chunks. Using a variable number N, however, is not allowed as this breaks the property of generating just 1 new peak per iteration.

Unification of static content checking and live content checking is achieved by sending the signed peak hashes on-demand, ahead of the actual data. As before, the sender SHOULD use acknowledgments to derive which content range the receiver has peak hashes for, and to prepend the data hashes with the necessary (signed) peak hashes. Except for the fact that the set of peak hashes changes with time, other parts of the protocol work as described in Section 5.1.

This method of integrity verification has an added benefit if the system includes some peers that saved the complete broadcast. The benefit is that 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 have saved all chunks can now announce this root hash to the tracking infrastructure and instantly seed it.

7.2. Forgetting Chunks

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

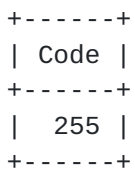
PPSPP provides a simple solution for peers to stay up-to-date with the chunk availability of a discarding peer. A discarding peer in a live stream MUST enable the Live Discard Window protocol option, specifying how many chunks/bytes it caches before the last chunk/byte it advertised as being available (see Section 8.8). Its peers SHOULD apply this number as a sliding window filter over the peer's chunk availability as conveyed via its HAVE messages.

8. Protocol Options

The HANDSHAKE message in PPSPP can contain the following protocol options (cf. [RFC2132] (DHCP options)). Each element in a protocol option is 8 bits wide, unless stated otherwise.

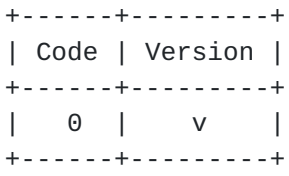
8.1. End Option

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 its length is 1 octet.



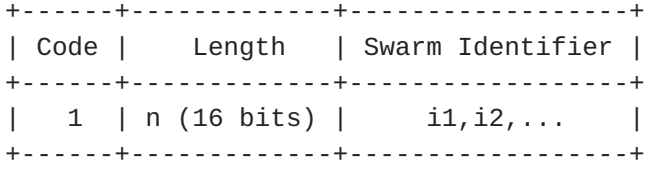
8.2. Version

A peer MUST include the version of the PPSPP protocol it supports as the first protocol option in the list.



8.3. Swarm Identifier

To enable end-to-end checking of any peer discovery process a peer MAY include a swarm identifier option.



Each PPSPP peer knows the IDs of the swarms it joins so this information can be immediately verified upon receipt. The length field is 2 octets to allow for large public keys as identifiers in live streaming.

8.4. Content Integrity Protection Method

A peer MUST include the content integrity method used by a swarm, unless it uses the default, in which case it MAY include the method.

Code	Method
2	m

Currently three values are defined for the method, 0 = No integrity protection, 1 = Merkle Hash Trees (for static content, see Section 5.1), 2 = Sign All, and 3 = Unified Merkle Tree (for live content, see Section 7.1).

The veracity of this information will come out when the receiver successfully verifies the first chunk from any peer.

8.5. Merkle Tree Hash Function

When the content integrity protection method is Merkle Hash Trees this option MUST also be defined.

Code	Hash Func
3	h

Currently the following values are defined for the hash function, 0 = SHA1, 1 = SHA-224, 2 = SHA-256, 3 = SHA-384, and 4 = SHA-512 [FIPS180-3].

The veracity of this information will come out when the receiver successfully verifies the first chunk from any peer.

8.6. Live Signature Algorithm

When the content integrity protection method is "Sign All" or "Unified Merkle Tree" this option MUST also be defined.

Code	Sig Alg
4	s

The value of this option is one of the Domain Name System Security (DNSSEC) Algorithm Numbers [IANADNSSECALGNUM]. If necessary, the key size that impacts signature length can be derived from the swarm identifier which is the signing public key in live streaming.

The veracity of this information will come out when the receiver successfully verifies the first chunk from any peer.

8.7. Chunk Addressing Method

A peer MUST include the chunk addressing method it uses, unless it uses the default, in which case it MAY include the method.

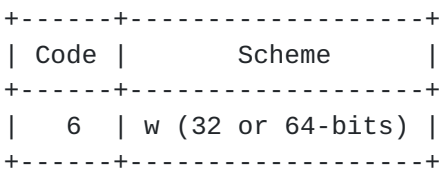
Code	Scheme
5	a

Currently six values are defined for the chunk addressing scheme, 0=32-bit bins, 1=64-bit byte ranges, and 2=32-bit chunk ranges, 3=64-bit bins, 4=64-bit chunk ranges.

The veracity of this information will come out when the receiver parses the first message containing a chunk specification from any peer.

8.8. Live Discard Window

A peer in a live swarm MUST include the discard window it uses. The unit of the discard window depends on the chunk addressing method used. For bins and chunk ranges it is a number of chunks, for byte ranges it is a number of bytes. Its data type is the same as for a bin, or one value in a range specification. In other words, a 32-bit or 64-bit integer in big endian format. This option MUST therefore appear after the Chunk Addressing option, if present in the list of protocol options.

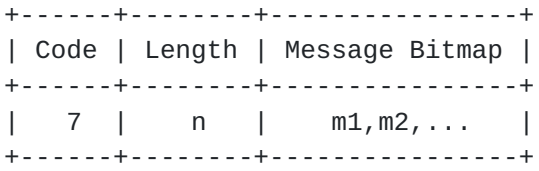


A peer that does not, under normal circumstances, discard chunks MUST set this option to the special value 0xFFFFFFFF (32-bit) or 0xFFFFFFFFFFFFFFFF (64-bit). For example, peers that record a complete broadcast to offer it directly as a static asset after the broadcast ends (see Section 7.1.2).

The veracity of this information does not impact a receiving peer more than when a sender peer just does not respond to REQUEST messages.

8.9. 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 value of this option is a 256-bit bitmap where each bit represents a message type. The bitmap may be truncated to the last non-zero byte.



9. UDP Encapsulation

Currently, PPSPP-over-UDP is the preferred deployment option. Effectively, UDP allows the use of IP with minimal overhead and it also allows userspace implementations. LEDBAT is used for congestion control [I-D.ietf-ledbat-congestion]. 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. LEDBAT may be replaced with a different algorithm when the work of the IETF working group on RTP Media Congestion Avoidance Techniques (RMCAT) [RMCATCHART] matures.

9.1. Chunk Size

The default is to use fixed-sized chunks of 1 kilobyte such that a UDP datagram with a DATA message can be transmitted as a single IP packet over an Ethernet network with 1500-byte frames. PPSP implementations can use larger chunk sizes. For example, on CPU-limited hardware 8 kilobyte chunks may be used, transported as a single UDP datagram fragmented over multiple IP packets (with the increased chance of that UDP datagram getting lost). The chunk addressing schemes can all work with different chunk sizes, see Section 4.

The chunk size used for a particular swarm MUST be part of the swarm's metadata (which is then the swarm ID and the chunk size).

9.2. Datagrams and Messages

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

- HANDSHAKE = 0x00
- DATA = 0x01
- ACK = 0x02
- HAVE = 0x03
- INTEGRITY = 0x04
- PEX_RES = 0x05
- PEX_REQ = 0x06
- SIGNED_INTEGRITY = 0x07
- REQUEST = 0x08
- CANCEL = 0x09
- CHOKE = 0x0a
- UNCHOKE = 0x0b

- PEX_RESv6 = 0x0c

Furthermore, integers are serialized in the network (big-endian) byte order. So consider the example of a HAVE message (Section 3.2) using bin chunk addressing. It has message type of 0x02 and a payload of a bin number, a four-byte integer (say, 1); hence, its on the wire representation for UDP can be written in hex as: "0200000001".

All messages are idempotent or recognizable as duplicates. 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 and be dealt with.

9.3. Channels

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

9.4. HANDSHAKE

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

1. the IP address of a peer

2. peer's UDP port and

3. the swarm id of the content (see Section 5.1 and Section 7).

4. the chunk size used, unless the 1 KB default

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

On the wire the datagram will look something like this:

(CHANNEL) 00000000 HANDSHAKE 00000011 v=01 si=123...1234 ca=0 end

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

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

Peer's response datagram on the wire:

(CHANNEL) 00000011 HANDSHAKE 00000022 v=01 protocol options end

(peer to the initiator: use channel number 0x22 for this transfer and proto version 0x01.)

At this point, the initiator knows that the peer really responds; for that purpose channel ids MUST be random enough to prevent easy guessing. So, the third datagram of a handshake MAY already contain some heavy payload. To minimize the number of initialization roundtrips, the first two datagrams MAY also contain some minor payload, e.g. a couple of HAVE messages roughly indicating the current progress of a peer or a REQUEST (see Section 3.7). When receiving the third datagram, both peers have the proof they really talk to each other; the three-way handshake is complete.

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

On the wire:

(CHANNEL) 00000022 HANDSHAKE 00000000

9.5. HAVE

A HAVE message (type 0x03) consist of a chunk specification that states that the sending peer has those chunks and successfully checked their integrity. 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 for bin 3 on the wire:

HAVE 00000003

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

9.6. DATA

A DATA message (type 0x01) consists of a chunk specification and the actual chunk. In case a datagram contains a DATA message, a sender MUST always put the data message in the tail of the datagram. As the LEDBAT congestion control is used, a sender MUST include a timestamp, in particular, a 64-bit integer representing the current system time with microsecond accuracy. The timestamp MUST be included between chunk specification and the actual chunk.

A DATA message for bin 0, with timestamp 12345678, and some data on the wire:

DATA 00000000 12345678 48656c6c6f20776f726c6421

(This message accommodates an entire file: "Hello world!")

9.7. 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 is computed from the timestamp received from the previous DATA message containing the chunk being acknowledged following the LEDBAT specification.

An ACK message for bin 2 with one-way delay 12345678 on the wire:

ACK 00000002 12345678

9.8. 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 for bin 0 with a SHA1 hash on the wire:

HASH 00000000 1234123412341234123412341234123412341234

9.9. SIGNED_INTEGRITY

A SIGNED_INTEGRITY message (type 0x07) consists of a chunk specification and a digital signature encoded as in DNSSEC without the BASE-64 encoding [RFC4034]. The signature algorithm is defined by the Live Signature Algorithm protocol option, see Section 8.6.

9.10. REQUEST

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

9.11. CANCEL

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

9.12. CHOKE and UNCHOKE

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

9.13. PEX_REQ, PEX_RES and PEX_RESv6

A PEX_REQ (0x06) message has no payload. A PEX_RES (0x05) message consists of a 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.

9.14. KEEPALIVE

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

On the wire:

    (CHANNEL) 00000022

9.15. Flow and Congestion Control

Explicit flow control is not necessary in PPSPP-over-UDP. In the case of video-on-demand the receiver will request data explicitly from peers and is therefore in control of how much data is coming towards it. In the case of live streaming, where a push-model may be used, the amount of data incoming is limited to the bitrate, which the receiver must be able to process otherwise it cannot play the stream. Should, for any reason, the receiver get saturated with data that situation is perfectly detected by the congestion control.

PPSPP-over-UDP can support different congestion control algorithms. At present, it uses the LEDBAT congestion control algorithm [I-D.ietf-ledbat-congestion].

10. Extensibility

10.1. Chunk Picking Algorithms

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

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

10.2. 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 upload to others. Examples of reciprocity algorithms are tit-for-tat as used in BitTorrent ([TIT4TAT]) and Give-to-Get ([GIVE2GET]). In PPSP, reciprocity enforcement is the sole responsibility of the sender peer.

11. Acknowledgements

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

The PPSPP protocol was designed by Victor Grishchenko at Technische Universiteit Delft. The authors would like to thank the following people for their contributions to this draft: the chairs and members of the IETF PPSP working group, and Mihai Capota, Raul Jimenez, Flutra Osmani, Johan Pouwelse, and Raynor Vliegendhart.

12. IANA Considerations

To be determined.

13. Security Considerations

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

13.1. Security of the Handshake Procedure

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

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

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

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

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

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

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

When peer B receives this datagram, it creates some state for peer A,

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

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

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

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

13.1.1. Protection against attack 1

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

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

13.1.2. Protection against attack 2

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

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

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

13.1.3. Protection against attack 3

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

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

13.2. Secure Peer Address Exchange

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

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

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

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

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

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

13.2.1. Protection against the Amplification Attack

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

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

13.2.2. Example: Tracker as Certification Authority

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

Peer A then includes this certificate when it sends a PEX_RES to peer B. Receiver B verifies it against the tracker public key. This tracker public key should be part of the swarm's metadata, which B received from a trusted source. Subsequently, peer B can send the member certificate of A to other peers in PEX_RES 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 certs is to prevent that malicious peer T can cause good peer A to contact *random* hosts. The freshness of the timestamp just adds extra protection in addition to achieving that goal. It protects against malicious hosts causing a good peer A to contact hosts that previously participated in the swarm.

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

13.2.3. Protection Against Eclipse Attacks

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

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

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

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

The Closed Swarms [CLOSED] and Enhanced Closed Swarms [ECS] mechanisms provide swarm-level access control. The basic idea is

that a peer cannot download from another peer unless it shows a Proof-of-Access. Enhanced Closed Swarms improve on the original Closed Swarms by adding on-the-wire encryption against man-in-the-middle attacks and more flexible access control rules.

The exact mapping of ECS to PPSPP is work in progress.

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

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

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

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

13.5.1. HANDSHAKE

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

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

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

13.5.2. HAVE

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

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

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

  Solution: Peer and chunk selection algorithms external to the

protocol will implement fairness and provide sharing incentives.

13.5.3. DATA

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

  Solution: The content integrity protection schemes defend against this.

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

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

13.5.4. ACK

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

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

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

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

13.5.5. INTEGRITY and SIGNED_INTEGRITY

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

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

13.5.6. REQUEST

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

  Solution: A limit is imposed on the upload capacity a single peer

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

13.5.7. CANCEL

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

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

13.5.8. CHOKE

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

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

13.5.9. UNCHOKE

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

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

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

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

13.5.10. PEX_RES

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

13.5.11. Unsolicited Messages in General

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

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

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

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

14. References

14.1. Normative References

[FIPS180-3] Information Technology Laboratory, National Institute of Standards and Technology, "Federal Information Processing Standards: Secure Hash Standard (SHS)", Publication 180-3, Oct 2008.

[I-D.ietf-ledbat-congestion] Shalunov, S., Hazel, G., Iyengar, J., and M. Kuehlewind, "Low Extra Delay Background Transport (LEDBAT)", draft-ietf-ledbat-congestion-10 (work in progress), September 2012.

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

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

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

14.2. Informative References

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

[BINMAP] Grishchenko, V. and J. Pouwelse, "Binmaps: hybridizing bitmaps and binary trees", Technical Report PDS-2011-005, Parallel and Distributed Systems Group, Fac. of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, The Netherlands, Apr 2009.

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

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

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

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

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

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

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

[GIVE2GET] Mol, J., Pouwelse, J., Meulpolder, M., Epema, D., and H. Sips, "Give-to-Get: Free-riding Resilient Video-on-demand in P2P Systems", Proceedings Multimedia Computing and Networking conference (Proceedings of SPIE Vol. 6818) San Jose, California, USA, Jan 2008.

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

[I-D.ietf-ppsp-reqs] Williams, C., Xiao, L., Zong, N., Pascual, V., and Y. Zhang, "P2P Streaming Protocol (PPSP) Requirements", draft-ietf-ppsp-reqs-05 (work in progress), October 2011.

[I-D.narten-iana-considerations-rfc2434bis] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", draft-narten-iana-considerations-rfc2434bis-09 (work in progress), March 2008.

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

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

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

[RFC2132] Alexander, S. and R. Droms, "DHCP Options and BOOTP Vendor Extensions", RFC 2132, March 1997.

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

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

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

October 2008.

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

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

[RMCATCHART] Eggert, L. and others, "RTP Media Congestion Avoidance Techniques (rmcat) Description of Working Group", 2012, <http://datatracker.ietf.org/wg/rmcat/charter/>.

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

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

[SNP] Ford, B., Srisuresh, P., and D. Kegel, "Peer-to-Peer Communication Across Network Address Translators", Feb 2005, <http://www.brynosaurus.com/pub/net/p2pnat/>.

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

[SWIFTIMPL] Grishchenko, V., Paananen, J., Pronchenkov, A., Bakker, A., and R. Petrocco, "Swift reference implementation", 2012, <https://github.com/triblerteam/libswift/>.

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

Appendix A. Revision History

-00 2011-12-19 Initial version.

-01 2012-01-30 Minor text revision:

* Changed heading to "A. Bakker"

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

* Replaced swift with PPSPP.

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

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

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

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

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

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

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

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

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

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

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

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

-02 2012-04-17 Major revision

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

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

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

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

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

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

* Renamed HASH message to INTEGRITY.

* Renamed HINT to REQUEST.

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

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

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

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

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

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

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

* Moved Rationale to Appendix.

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

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

-03 2012-10-22 Major revision

* Updated Abstract and Introduction, removing download case.

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

* Removed directory lists unused in streaming.

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

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

* Added a section on compatibility between chunk addressing schemes.

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

* Added a section on forgetting chunks in live streaming.

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

* Added SHA-2 support for Merkle Hash functions.

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

* Added a Live Signature Algorithm protocol option.

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

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

* Removed appendix with Rationale.

* Resolved Ticket #21+25: PPSPP currently uses LEDBAT and the DATA and ACK messages now contain the time fields it requires. Should other congestion control algorithms be supported in the future, a protocol option will be added.

Authors' Addresses

Arno Bakker
Technische Universiteit Delft
Mekelweg 4
Delft, 2628CD
The Netherlands

Phone:
Email: arno@cs.vu.nl

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

Phone:
Email: r.petrocco@gmail.com

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

Phone:
Email: victor.grishchenko@gmail.com