PPSP
Internet-Draft
Intended status: Informational
Expires: December 22, 2012

A. Bakker
R. Petrocco
Technische Universiteit Delft

June 20, 2012

Peer-to-Peer Streaming Peer Protocol (PPSPP)
draft-ietf-ppsp-peer-protocol-02

Abstract

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

Table of Contents

1. Introduction

1.1. Purpose

This document describes the Peer-to-Peer Streaming Peer Protocol (PPSPP), designed from the ground up for the task of disseminating the same content to a group of interested parties. PPSPP supports streaming on-demand and live video content, as well as conventional downloading, thus covering today's three major use cases for content distribution. To fulfill this task, clients consuming the content are put on equal footing with the servers initially providing the content to create a peer-to-peer system where everyone can provide data.

PPSPP uses a simple 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 [MERKLE][ABMRKL]. This self-certifying hash tree allows every peer to directly detect when a malicious peer tries to distribute fake content. It also ensures only a small amount of information is needed to start a download (just the root hash and some peer addresses).

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

PPSPP is a generic protocol which can run directly on top of UDP, TCP, or as a layer below RTP, similar to SRTP [RFC3711]. 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 is UDP, PPSPP can also use different congestion control algorithms and facilitate NAT traversal.

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

This documents describes not only the PPSPP protocol but also how it satisfies the requirements for the IETF Peer-to-Peer Streaming

Protocol (PPSP) Working Group's peer protocol [PPSPCHART] [I-D.ietf-ppsp-reqs]. A reference implementation of 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 SHA1 [FIPS180-2], to a piece of data.

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.

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 the recommended method for content integrity protection (Merkle hash trees) is used, and a specific policy for which selecting 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 just a HANDSHAKE and omits HAVE messages as a way of choking A.

2.2. Exchanging Chunks

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

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

2.3. Leaving a Swarm

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

3. Messages

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

For the sake of simplicity, one swarm of peers always deals with one content asset (e.g. file) only. Retrieval of large collections of files is done by retrieving a directory list file and then recursively retrieving files, which might also turn to be directory lists, as described in Section 3.10.

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 roundtrips, 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 3.6).

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. ACK

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 a chunk specification for C. To facilitate delay-based congestion control, an ACK message contains a timestamp (see e.g. [I-D.ietf-ledbat-congestion]).

3.4. 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.

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. 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.

3.7. CANCEL

When downloading on demand or live streaming content, a peer MAY request urgent data from multiple peers to increase the probablity of it is delivered on time. In particular, when the specific chunk picking algorithm (see Section 10.4), 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.

3.8. Peer Address Exchange and NAT Hole Punching

3.8.1. 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_REQ 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.

3.8.2. 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]. To implement this feature, the sending pattern of PEX messages is restricted. In particular, when peer A introduces peer B to peer C by sending a PEX_RES message to C, it SHOULD also send a 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. 300-600ms. 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.

3.9. Keep Alive Signaling

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 mapping they are implemented as simple datagrams consisting of a 4-byte channel number only, see Section 9.1.3 and Section 9.1.4. When PPSPP is used over TCP, each datagram is prefixed with 4 bytes containing its size, the common method of turning TCP's stream of bytes into a sequence of datagrams. In that case, a size of 0 is used as keep alive, as in BitTorrent [BITTORRENT].

3.10. Directory Lists

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

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

3.11. 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.

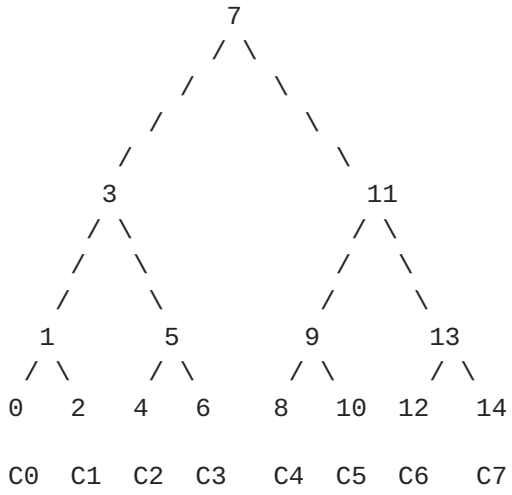
The recommended and mandatory-to-implement scheme of chunk addressing and map compression for PPSPP is to be determined.

4.1. Bin Numbers

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

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

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

4.1.1. 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 bin number of the biggest complete interval of all chunks the receiver has received and checked so far that fully includes the interval of chunks just received. So the bin number MUST denote at least the interval received, but the receiver is supposed to aggregate and

acknowledge bigger bins, when possible.

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

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

4.1.2. 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 bin number of its biggest, complete, interval covering C to the sending peer (see HAVE).

4.2. 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.

4.2.1. Byte Ranges

The start and end specification are byte offsets in the content. Whether or not byte ranges are translatable to bin numbers depends on whether chunks are fixed size or not.

4.2.2. Chunk Ranges

The start and end specification are chunk IDs.

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":

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

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

4.2.3. In Messages

The same rules for sending ACK and HAVE messages as in bin numbering apply in this content addressing scheme. In particular, the receiver is supposed to acknowledge the largest possible super interval that contains the interval of chunks just received.

4.3. Other Addressing Schemes

Note: when introducing other addressing schemes, e.g. BitTorrent BITFIELD messages one must keep in mind that the initial datagrams must not be too larger when the source of the peer's address is not trusted, to prevent DoS attacks via PPSPP. E.g. when the address comes from a PEX_ADD message.

5. Content Integrity Protection

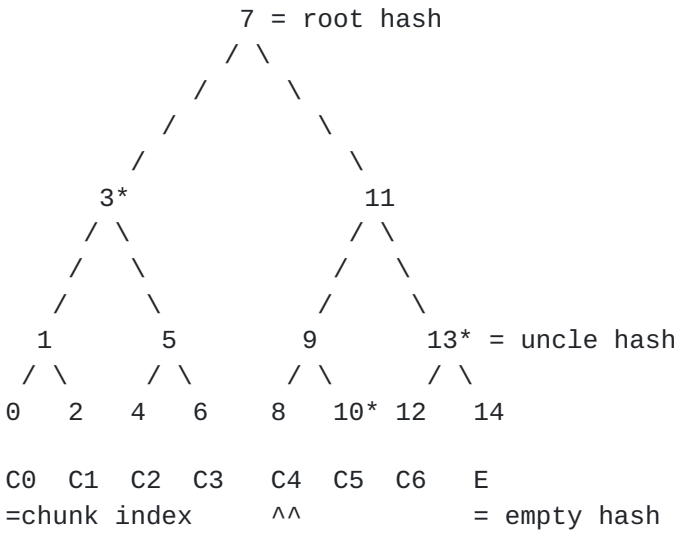
PPSPP can use different methods for protecting the integrity of the content while it is being distributed via the peer-to-peer network. More specifically, PPSPP can use different methods for receiving peers to detect whether a requested chunk has been maliciously modified by the sending peer. The recommended method for bad content detection is the Merkle Hash Tree scheme described below, which is mandatory-to-implement. Another applicable content integrity protection method is providing clients with the hashes of the content's chunks before the download commences by means of metadata files, as with BitTorrent's .torrent files [BITTORRENT].

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 public keys and dynamic trees are used, see below.

The Merkle hash tree of a content asset that is divided into N chunks is constructed as follows. Note the construction does not assume chunks of content to be fixed size. Given a cryptographic hash function, more specifically a modification detection code (MDC) [HAC01] , such as SHA1, the hashes of all the chunks of the content are calculated. Next, a binary tree of sufficient height is created. Sufficient height means that the lowest level in the tree has enough nodes to hold all chunk hashes in the set, as 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 a empty hash value of all zeros. Finally, the hash values of the higher levels in the tree are calculated, by concatenating the hash values of the two children (again left to right) and computing the hash of that aggregate. This process ends in a hash value for the root node, which is called the "root hash". Note the root hash only depends on the content and any modification of the content will result in a different root hash.



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

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 (Section 5.3). As a general rule, if some additional data is still missing to process a message within a datagram, the message SHOULD be dropped.

The hashes necessary to verify a chunk are in principle its sibling's hash and all its uncle hashes, but the set of hashes to sent can be optimized. Before sending a packet of data to the receiver, the sender inspects the receiver's previous acknowledgments (HAVE or ACK) to derive which hashes the receiver already has for sure. Suppose, the receiver had acknowledged 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 denotes chunks 0,2,4,6. The DATA message MUST contain the chunk ID of the chunk and chunk itself. A peer MAY send the required messages for multiple chunks in the same datagram.

5.5. Overhead

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

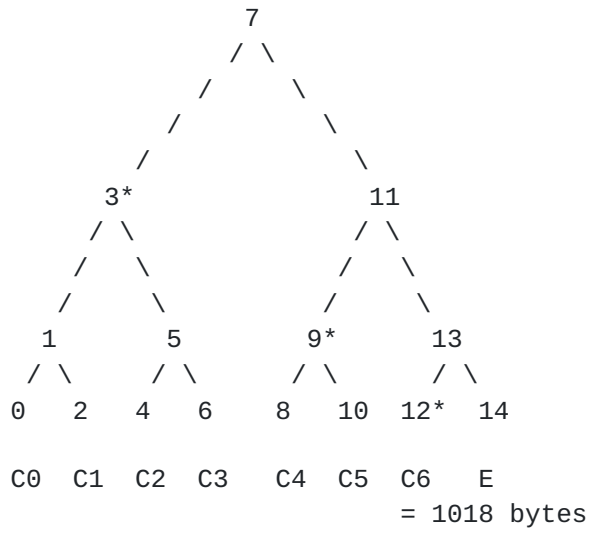
In PPSPP, the root hash of a static content asset, such as a video file, along with some peer addresses is sufficient to start a download. In addition, PPSPP can reliably and automatically derive the size of such content from information received from the network when fixed sized chunks are used. As 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 signaled 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 that receive 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 signatures are sent along with the chunk using a new 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 can also reduce the number of signing and verification operations per second, that is, provide signature amortization following the approach described in [SIGMCAST].

7.1.1. Unified Merkle Tree

In this method, the chunks of content are used as the basis for a Merkle hash tree as before. 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 sign the new peak hashes (see Section 6.1) that are created as the tree grows.

Live/download unification is achieved by sending the signed peak hashes on-demand, ahead of the actual data. As before, the sender might use acknowledgment's to derive which content range the receiver has peak hashes for and to prepend the data hashes with the necessary (signed) peak hashes. Except for the fact that the set of peak hashes changes with time, other parts of the algorithm work as described above.

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

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

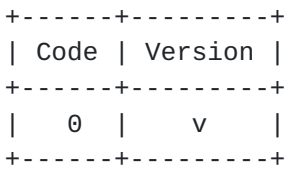
To avoid an increase in signing and verification operations signature amortization via Merkle Tree Chaining can be used [SIGMCAST]. In that case, the live injector creates a number of chunks, which are organized in a small Merkle hash tree and only the root of the (sub)tree is signed. This amortization will increase latency as a receiving peer has to wait for the signature before delivering the chunks to the higher layers responsible for playback [POLLIVE], unless some (optimistic) optimisations are made.

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.

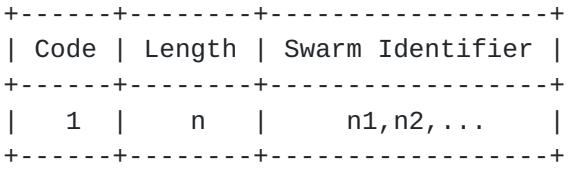
8.1. Version

A peer MUST include the version of the PPSPP protocol it supports.



8.2. 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.

8.3. Content Integrity Protection Method

Code	Method
2	m

Currently one value is defined for the method, 0 = Merkle Hash Trees (see Section 5.1).

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

8.4. 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 one value is defined for the hash function, 0 = SHA1 [FIPS180-2].

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

8.5. Chunk Addressing

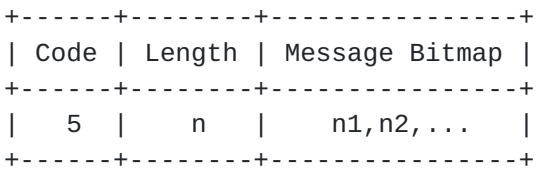
Code	Scheme
4	a

Currently three values are defined for the chunk addressing scheme, 0=bins, 1=byte ranges, and 2=chunk ranges.

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

8.6. 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. Transport Protocols and Encapsulation

9.1. UDP

The following description assumes the use of bin numbers as chunk addressing scheme and Merkle hash trees as content integrity protection. Furthermore it has not yet been updated following the redesign of the HANDSHAKE message.

9.1.1. Chunk Size

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

9.1.2. Datagrams and Messages

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

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

Furthermore, integers are serialized in the network (big-endian) byte order. So consider the example of an ACK message (Section 3.3). It has message type of 0x02 and a payload of a bin number, a four-byte integer (say, 1); hence, its on the wire representation for UDP can be written in hex as: "02 00000001". This hex-like two character-per-byte notation is used to represent message formats in the rest of this section.

9.1.3. 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.

9.1.4. HANDSHAKE and VERSION

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

1. the IP address of a peer

2. peer's UDP port and

3. the root hash of the content (see Section 5.1).

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

On the wire the datagram will look something like this:

00000000 10 01 04 7FFFFFFF 1234123412341234123412341234123412341234
00 00000011

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

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

Peer's response datagram on the wire:

00000011 10 01 00 00000022 03 00000003

(peer to the initiator: use channel number 0x22 for this transfer and proto version 0x01; I also have first 4 chunks of the file, see Section 3.2).

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 3.6). When receiving the third datagram, both peers have the proof they really talk to each other; three-way handshake is complete.

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

On the wire:

00 00000000

9.1.5. HAVE

A HAVE message (type 0x03) states that the sending peer has the complete specified bin and successfully checked its integrity:

03 00000003

(got/checked first four kilobytes of a file/stream)

9.1.6. ACK

An ACK message (type 0x02) acknowledges data that was received from its addressee; to facilitate delay-based congestion control, an ACK message contains a timestamp, in particular, a 64-bit microsecond time.

02 00000002 12345678

(got the second kilobyte of the file from you; my microsecond timer was showing 0x12345678 at that moment)

9.1.7. INTEGRITY

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

04 7FFFFFFF 1234123412341234123412341234123412341234

9.1.8. DATA

A DATA message (type 0x01) consists of a four-byte bin number and the actual chunk. In case a datagram contains a DATA message, a sender MUST always put the data message in the tail of the datagram. For example:

01 00000000 48656c6c6f20776f726c6421

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

9.1.9. KEEPALIVE

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

On the wire:

00000022

9.1.10. 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, in particular, it supports the new IETF Low Extra Delay Background Transport (LEDBAT) congestion control algorithm that ensures that peer-to-peer traffic yields to regular best-effort traffic [I-D.ietf-ledbat-congestion].

9.2. TCP

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

9.3. RTP Profile for PPSP

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

In particular, to encode a PPSPP datagram in an RTP packet all the non-DATA messages of PPSPP such as REQUEST and HAVE are postfixed to the RTP packet using the UDP encoding and the content of DATA messages is sent in the payload field. Implementations MAY omit the RTP header for packets without payload. This construction allows the

streaming application to use of all RTP's current features, and with a modification to the Merkle tree hashing scheme (see below) meets PPSPP's atomic datagram principle. The latter means that a receiving peer can autonomously verify the RTP packet as being correct content, thus preventing the spread of corrupt data (see requirement PPSP.SEC-REQ-4).

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

9.3.1. Design

9.3.1.1. Joining a Swarm

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

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

9.3.1.2. Joining a Swarm

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

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

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

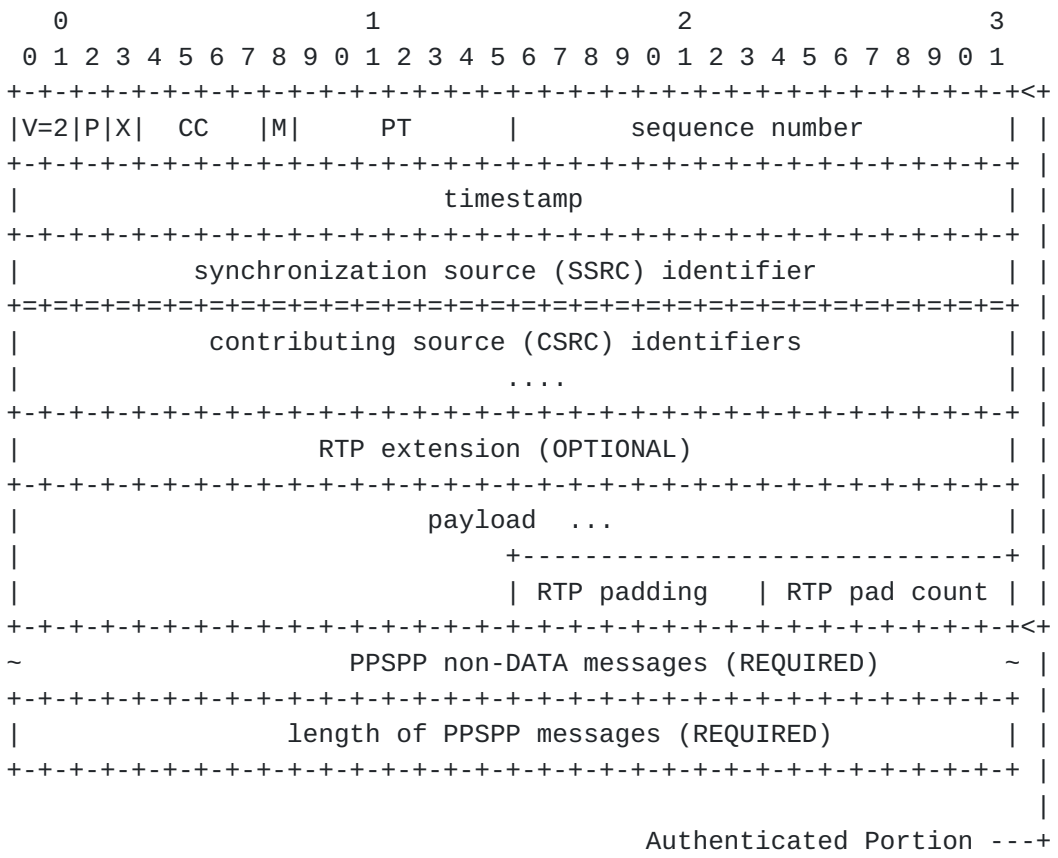
9.3.1.3. Leaving a Swarm

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

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

9.3.1.4. Discussion

Using PPSPP as an RTP profile requires a change to the content integrity protection scheme (see Section 5.1). The fields in the RTP header, such as the timestamp and PT fields, must be protected by the Merkle tree hashing scheme to prevent malicious alterations. Therefore, the Merkle tree is no longer constructed from pure content chunks, but from the complete RTP packet for a chunk as it would be transmitted (minus the non-DATA PPSPP messages). In other words, the hash of the leaves in the tree is the hash over the Authenticated Portion of the RTP packet as defined by SRTP, illustrated in the following figure (extended from [RFC3711]). There is no need for the RTP packets to be fixed size, as the hashing scheme can deal with variable-sized leaves.



The format of an RTP-PPSPP packet.

Figure 4

As a downside, with variable-sized payloads the automatic content size detection of Section 6 no longer works, so content length MUST be explicit in the metadata. In addition, storage on disk is more complex with out-of-order, variable-sized packets. On the upside, carrying RTP over PPSPP allow decryption-less caching.

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

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

9.3.2. PPSP Requirements

9.3.2.1. Basic Requirements

- PPSP.REQ-1: The PPSPP PEX message can also be used as the basis for a tracker protocol, to be discussed elsewhere.
- PPSP.REQ-2: This draft preserves the properties of RTP.
- PPSP.REQ-3: This draft does not place requirements on peer IDs, IP+port is sufficient.
- PPSP.REQ-4: The content is identified by its root hash (video-on-demand) or a public key (live streaming).
- PPSP.REQ-5: The content is partitioned by the streaming application.
- PPSP.REQ-6: Each chunk is identified by a bin number (and its cryptographic hash.)
- PPSP.REQ-7: The protocol is carried over UDP because RTP is.
- PPSP.REQ-8: The protocol has been designed to allow meaningful data transfer between peers as soon as possible and to avoid unnecessary round-trips. It supports small and variable chunk sizes, and its content integrity protection enables wide scale caching.

9.3.2.2. Peer Protocol Requirements

- PPSP.PP.REQ-1: A GET_HAVE would have to be added to request which chunks are available from a peer, if the proposed push-based HAVE mechanism is not sufficient.
- PPSP.PP.REQ-2: A set of HAVE messages satisfies this.
- PPSP.PP.REQ-3: The PEX_REQ message satisfies this. Care should be taken with peer address exchange in general, as the use of such hearsay is a risk for the protocol as it may be exploited by malicious peers (as a DDoS attack mechanism). A secure tracking / peer sampling protocol like [PUPPETCAST] may be needed to make peer-address exchange safe.
- PPSP.PP.REQ-4: HAVE messages convey current availability via a push model.
- PPSP.PP.REQ-5: Bin numbering enables a compact representation of chunk availability.

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

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

9.3.2.2.1. Security Requirements

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

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

- PPSP.SEC.REQ-3: RTP encryption or IPsec [RFC4301] can be used, if the PPSPP messages must also be encrypted.

- PPSP.SEC.REQ-4: The Merkle tree hashing scheme prevents the indirect spread of corrupt content, as peers will only forward chunks to others if their integrity check out. Another protection mechanism is to not depend on hearsay (i.e., do not forward other peers' availability information), or to only use it when the information spread is self-certified by its subjects. Other attacks, such as a malicious peer claiming it has content but not replying, are still possible. Or wasting CPU and bandwidth at a receiving peer by sending packets where the DATA doesn't match the hashes from the INTEGRITY messages.

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

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

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

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

- PPSP.SEC.REQ-9: PPSPP has built-in content integrity protection via self-certified naming of content, see SEC.REQ-5 and Section 5.1.

10. Extensibility

10.1. 32 bit vs 64 bit

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

44 000000000000000E 01234567890ABCDEF1234567890ABCDEF1234567

10.2. IPv6

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

10.3. Congestion Control Algorithms

Congestion control algorithm is left to the implementation and may even vary from peer to peer. Congestion control is entirely implemented by the sending peer, the receiver only provides clues, such as hints, acknowledgments and timestamps. In general, it is expected that servers would use TCP-like congestion control schemes such as classic AIMD or CUBIC [CUBIC]. End-user peers are expected to use weaker-than-TCP (least than best effort) congestion control, such as [I-D.ietf-ledbat-congestion] to minimize seeding counter-

incentives.

10.4. 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 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.

10.5. Reciprocity Algorithms

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

10.6. Different crypto/hashing schemes

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

11. Acknowledgements

Arno Bakker 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 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.

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 3.8, 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. 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.

13.5.4. 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.

13.5.5. 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. PEX_RES

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

13.5.9. Unsollicited 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

[FIPS180-2]
Federal Information Processing Standards, "Secure Hash Standard", Publication 180-2, Aug 2002.

[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.

[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>.

[CUBIC] Rhee, Injong. and Lisong. Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant", International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet'05), Lyon, France, Feb 2005.

[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.

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

[HTTP1MLN] Jones, R., "A Million-user Comet Application with Mochiweb, Part 3", Nov 2008, <http://www.metabrew.com/article/a-million-user-comet-application-with-mochiweb-part-3>.

[I-D.ietf-ledbat-congestion] Hazel, G., Iyengar, J., Kuehlewind, M., and S. Shalunov, "Low Extra Delay Background Transport (LEDBAT)", draft-ietf-ledbat-congestion-09 (work in progress), October 2011.

[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.

[LUCNAT] D'Acunto, L., Meulpolder, M., Rahman, R., Pouwelse, J., and H. Sips, "Modeling and Analyzing the Effects of Firewalls and NATs in P2P Swarming Systems", International Workshop on Hot Topics in Peer-to-Peer Systems (HotP2P'10), Atlanta, USA, Apr 2010.

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

[MOLNAT] Mol, J., Pouwelse, J., Epema, D., and H. Sips, "Free-

riding, Fairness, and Firewalls in P2P File-Sharing", IEEE International Conference on Peer-to-Peer Computing (P2P '08), Aachen, Germany, Sep 2008.

[PERMIDS] Bakker, A. and others, "Next-Share Platform M8-- Specification Part", P2P-Next project deliverable D4.0.1 (revised), App. C., Jun 2009, <http://www.p2p-next.org/download.php?id=E7750C654035D8C2E04D836243E6526E>.

[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.

[PPSPCHART] Stiemerling, M. and others, "Peer to Peer Streaming Protocol (ppsp) Description of Working Group", 2006, <http://datatracker.ietf.org/wg/ppsp/charter/>.

[PUPPETCAST] Bakker, A. and M. van Steen, "PuppetCast: A Secure Peer Sampling Protocol", European Conference on Computer Network Defense (EC2ND'08), pp. 3-10, Dublin, Ireland, Dec 2008.

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

[RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", STD 64, RFC 3550, July 2003.

[RFC3711] Baugher, M., McGrew, D., Naslund, M., Carrara, E., and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)", RFC 3711, March 2004.

[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.

[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. Rationale

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

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

Table 2: Use cases.

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

A.1. Design Goals

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

1. Embeddable kernel-ready protocol.

2. Embrace real-time streaming, in- and out-of-order download.

3. Have short warm-up times.

4. Traverse NATs transparently.

5. Be extensible, allow for multitude of implementation over diverse mediums, allow for drop-in pluggability.

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

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

Otherwise, besides the resource considerations, maintaining and auditing the code might become prohibitively expensive.

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

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

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

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

A.2. Not TCP

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

- in-order delivery is less useful as peer-to-peer protocols often employ out-of-order delivery themselves and in either case out-of-order data can still be stored;

- reliable delivery/retransmissions are not useful because the same data might be requested from different sources; as in-order delivery is not required, packet losses might be patched up lazily, without stopping the flow of data;

- flow control is not necessary as the receiver is much less likely to be saturated with the data and even if so, that situation is perfectly detected by the congestion control;

- TCP congestion control is less useful as custom congestion control is often needed [I-D.ietf-ledbat-congestion].

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

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

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

To avoid the usual layering of positive/negative acknowledgment mechanisms we introduce a scale-invariant acknowledgment system (see Appendix A.3). The system allows for aggregation and variable level of detail in requesting, announcing and acknowledging data, serves in-order and out-of-order retrieval with equal ease. Besides the protocol's footprint, we also aim at lowering the size of a minimal useful interaction. Once a single datagram is received, it must be checked for data integrity, and then either dropped or accepted, consumed and relayed.

A.3. Generic Acknowledgments

Generic acknowledgments came out of the need to simplify the data addressing/requesting/acknowledging mechanics, which tends to become

overly complex and multilayered with the conventional approach. Take the BitTorrent+TCP tandem for example:

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

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

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

Appendix B. 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.

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