PPSP
Internet-Draft
Intended status: Informational
Expires: April 28, 2012

V. Grishchenko
A. Bakker
TU Delft

October 26, 2011

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

Abstract

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

Status of this memo

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

Table of Contents

1. Introduction

1.1. Purpose

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

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

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

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

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

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

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

1.2. Conventions Used in This Document

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

1.3. Terminology

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

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

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

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

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

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

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

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

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

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

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

2. Overall Operation

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

2.1. Joining a Swarm

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

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

2.2. Exchanging Chunks

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

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

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

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

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

2.3. Leaving a Swarm

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

3. Messages

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

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

3.1. HANDSHAKE

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

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

3.3. HAVE

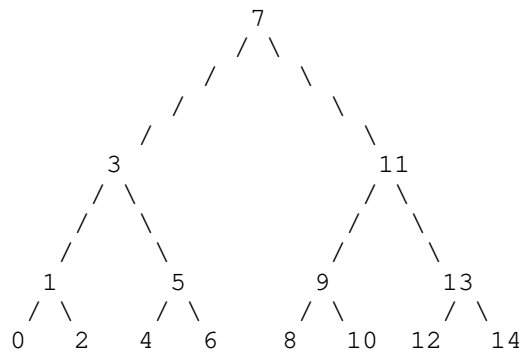
The HAVE message is used to convey which chunks a peers has available, expressed in a new content addressing scheme called "bin numbers".

3.3.1. Bin Numbers

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

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

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



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

3.3.2. HAVE Message

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

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

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

3.4. ACK

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

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

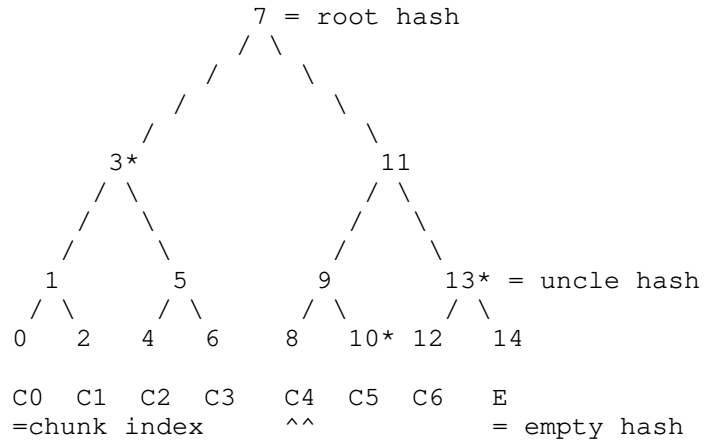
3.5. DATA and HASH

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

3.5.1. Merkle Hash Tree

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

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



3.5.2. Content Integrity Verification

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

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

3.5.3. The Atomic Datagram Principle

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

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

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

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

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

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

3.5.4. DATA and HASH Messages

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

3.6. HINT

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

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

3.7. Peer Address Exchange and NAT Hole Punching

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

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

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

3.8. KEEPALIVE

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

3.9. VERSION

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

3.10. Conveying Peer Capabilities

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

3.11. Directory Lists

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

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

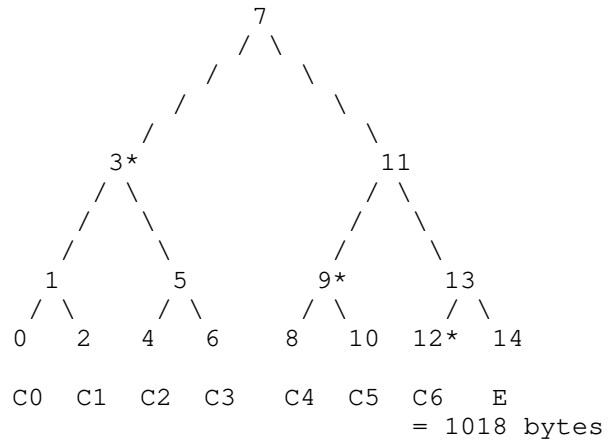
4. Automatic Detection of Content Size

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

4.1. Peak Hashes

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

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



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

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

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

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

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

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

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

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

4.2. Procedure

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

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

5. Live streaming

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

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

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

6. Transport Protocols and Encapsulation

6.1. UDP

6.1.1. Chunk Size

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

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

6.1.2. Datagrams and Messages

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

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

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

6.1.3. Channels

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

6.1.4. HANDSHAKE and VERSION

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

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

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

On the wire the datagram will look something like this:

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

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

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

Peer's response datagram on the wire:

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

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

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

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

On the wire:
00 00000000

6.1.5. HAVE

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

6.1.6. ACK

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

6.1.7. HASH

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

6.1.8. DATA

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

6.1.9. KEEPALIVE

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

On the wire:
00000022

6.1.10. Flow and Congestion Control

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

6.2. TCP

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

6.3. RTP Profile for PPSP

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

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

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

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

6.3.1. Design

6.3.1.1. Joining a Swarm

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

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

6.3.1.2. Exchanging Chunks

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

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

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

6.3.1.3. Leaving a Swarm

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

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

6.3.1.4. Discussion

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

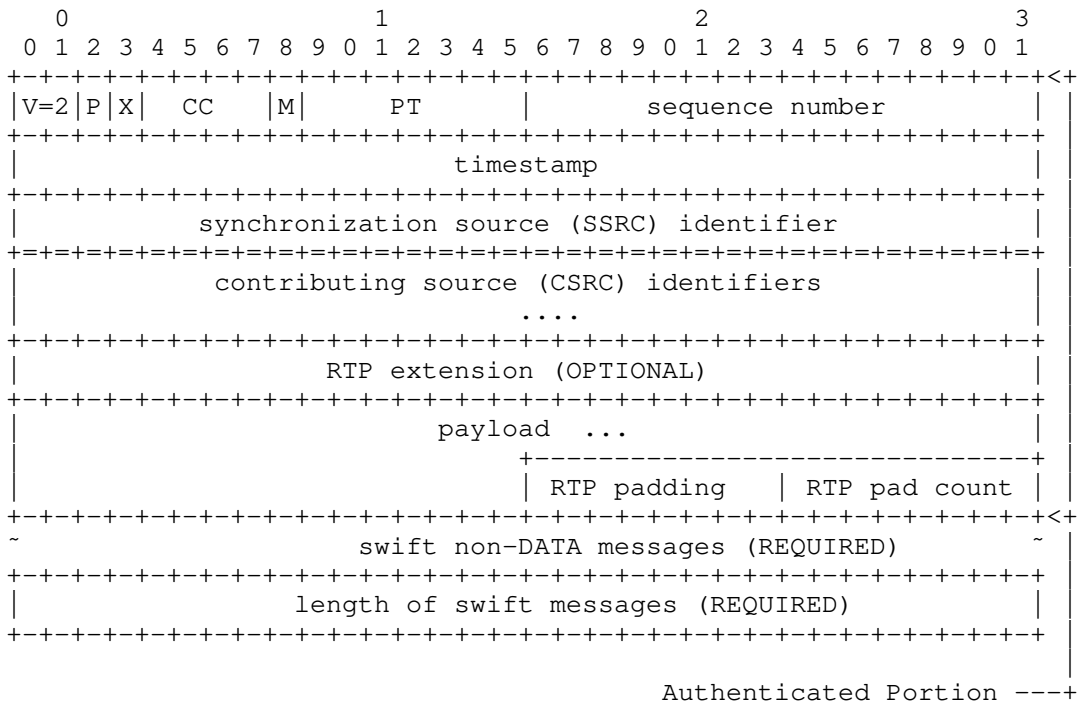


Figure: The format of an RTP-Swift packet.

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

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

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

6.3.2. PPSP Requirements

6.3.2.1. Basic Requirements

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

a tracker protocol, to be discussed elsewhere.

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

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

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

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

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

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

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

6.3.2.2. Peer Protocol Requirements

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

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

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

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

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

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

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

6.3.2.3. Security Requirements

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

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

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

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

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

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

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

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

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

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

6.4. HTTP (as PPSP)

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

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

6.4.1. Design

6.4.1.1. Joining a Swarm

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

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

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

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

6.4.1.2. Exchanging Chunks

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

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

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

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

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

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

6.4.1.3. Leaving a Swarm

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

6.4.1.4. Discussion

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

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

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

6.4.2. PPSP Requirements

6.4.2.1. Basic Requirements

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

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

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

IP+port is sufficient.

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

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

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

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

6.4.2.2. Peer Protocol Requirements

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

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

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

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

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

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

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

6.4.2.3. Security Requirements

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

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

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

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

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

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

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

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

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

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

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

7. Security Considerations

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

8. Extensibility

8.1. 32 bit vs 64 bit

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

```
44 000000000000000E 01234567890ABCDEF1234567890ABCDEF1234567
```

8.2. IPv6

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

8.3. Congestion Control Algorithms

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

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

8.4. Piece Picking Algorithms

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

8.5. Reciprocity Algorithms

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

8.6. Different crypto/hashing schemes

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

9. Rationale

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

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

TABLE 1. Use cases.

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

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

9.1. Design Goals

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

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

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

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

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

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

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

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

9.2. Not TCP

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

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

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

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

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

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

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

9.3. Generic Acknowledgments

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

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

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

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

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

Acknowledgements

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

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

References

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

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

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

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

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

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

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

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

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

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

Authors' addresses

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

Email: arno@cs.vu.nl

PPSP
Internet-Draft
Intended status: Standards Track
Expires: May 3, 2012

Y. Gu
J. Xia
Huawei
R. Cruz
M. Nunes
IST/INESC-ID/INOV
David A. Bryan
Polycom
J. Taveira
ID/INOV

Oct 31, 2011

Peer Protocol
draft-gu-ppsp-peer-protocol-03

Abstract

This document presents the architecture of the PPSP Peer protocol outlining the functional entities, message flows and message processing instructions, with the respective parameters. The PPSP Peer Protocol proposed in this document extends the capabilities of PPSP to support adaptive and scalable video and 3D video, for Video On Demand (VoD) and Live video services. The protocol messages formal syntax and semantics, methods, and formats are presented for both Binary and HTTP/XML encoded formats.

Status of this Memo

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

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

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

This Internet-Draft will expire on May 3, 2012.

Table of Contents

1. Introduction

The P2P Streaming Protocol (PPSP) is composed of two protocols: the PPSP Tracker Protocol and the PPSP Peer Protocol [I-D.ietf-ppsp-problem-statement].

The PPSP architecture requires PPSP peers able to communicate with a Tracker in order to participate in a particular swarm. This centralized Tracker service is used for peer and content registration and location. Content indexes (Media Presentation Descriptions) are also stored in the Tracker system allowing the association of content location information to the active peers in the swarm sharing the content.

The PPSP Tracker Protocol provides communication between Trackers and Peers and outlines how a peer is able to communicate with a tracker in order to exchange meta information about the location of other peers contributing with a specific stream (swarm) the peer interested in, as well as to report streaming status. The Peer can also apply to be a contributor for several streams (swarms), periodically reporting its status to the Tracker, allow it to estimate whether the peer is a competent contributor.

The PPSP Peer protocol outlines how a peer is able to communicate with other peers in order to control the advertising and exchange of media data, directly between peers, for a specific stream (swarm), as described in [I-D.ietf-ppsp-problem-statement].

The process used for media streaming distribution assumes a segment transfer scheme whereby the original content (that can be encoded using adaptive or scalable techniques) is chopped into small segments (and subsegments). For simplicity, in this document the segments (and subsegments) of media are named Chunks. The media streaming process has the following representations:

1. Adaptive - alternate representations with different qualities and bitrates; a single represention is non-adaptive;

2. Scalable description levels - multiple additive descriptions (i.e., addition of descriptions refine the quality of the video);

3. Scalable layered levels - nested dependent layers corresponding to several hierarchical levels of quality, i.e., higher enhancement layers refine the quality of the video of lower layers.

4. Scalable multiple views - views correspond to mono and stereoscopic 3D videos, with several hierarchical levels of

quality.

These streaming distribution techniques support dynamic variations in video streaming quality while ensuring support for a plethora of end user devices and network connections.

The information that should be exchanged between peers using this Peer Protocol includes:

1. ChunkMap indicating which chunks a peer possesses.

2. Required ChunkIDs

3. Peer preferences and status information

4. Signalling and Data Transport protocol negotiation

5. Information that can help improve the performance of PPSP.

In this document, a set of concrete information that needs to be exchanged between peers is introduced, together with the messages to convey such information.

This documents describes the PPSP Peer protocol and how it satisfies the requirements for the IETF Peer-to-Peer Streaming Protocol (PPSP), in order to derive the implications for the standardization of the PPSP streaming protocols and to identify open issues and promote further discussion.

This PPSP Peer Protocol proposal presents an early sketch for an extensible protocol that extends the capabilities of PPSP to support adaptive and scalable video.

2. Document Conventions

2.1. Notational Conventions

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

2.2. Terminology

The draft uses the terms defined in [I-D.ietf-ppsp-problem-statement], [I-D.gu-ppsp-tracker-protocol] and [I-D.cruz-ppsp-http-peer-protocol]. Additionally, This document uses the following acronyms and definitions frequently in itself:

Peer-Peer Messages

The Peer Protocol messages enable each Peer to exchange content availability with other Peers and request other Peers for content.

Tracker-Peer Messages

The Tracker Protocol messages provide communication between Peers and Trackers, by which Peers provide content availability, report streaming status and request candidate Peer lists from Trackers.

Connection Tracker

The Tracker Node to which the PPSP Peer will connect when it wants to join the PPSP system.

Sender Peer

A peer that contains the corresponding chunk files requested by leech peer is the Sender peer. Many peers can contain the content, but only one who is contributing the content to the leech peer can be named as Sender peer.

Leech Peer

A peer that requests the specific media content from other peers. Note that the leech peer can also contribute the downloaded media content (i.e., chunks) even the swarm is not completed, in such case, the leech peer will take on the role of sender peer for downloaded chunks.

Chunk Map

A peer list that indicates which chunks can be available for leech peer to playback smoothly.

Live Streaming

The scenario where all clients receive streaming content for the same ongoing event. The lags between the play points of the clients and that of the streaming source are small.

Video-on-demand (VoD)

The scenario where all clients are allowed to select and watch video content on demand.

Adaptive Streaming

Multiple alternate versions (different qualities and bitrates) of the same media content co-exist for the same streaming session; each alternate version corresponds to a different media quality level; peers can choose among the alternate versions for decode and playback.

Scalable Streaming

With Multiple Description Coding (MDC), multiple additive descriptions (that can be independently played-out) to refine the quality of the video when combined together. With Scalable Video Coding (SVC), nested dependent enhancement layers (hierarchical levels of quality), refine the quality of lower layers, from the lowest level (the playable Base Layer). With Multiple View Coding (MVC), multiple views allow the video to be played in 3D when the views are combined together.

Base Layer

The playable level in Scalable Video Coding (SVC) required by all upper level Enhancements Layers for proper decoding of the video.

Enhancement Layer

Enhancement differential quality level in Scalable Video Coding (SVC) used to produce a higher quality, higher definition video in terms of space (i.e., image resolution), time (i.e., frame rate) or Signal-to-Noise Ratio (SNR) when combined with the playable Base Layer.

Continuous Media

Media with an inherent notion of time, for example, speech, audio, video, timed text or timed metadata.

Media Component

An encoded version of one individual media type such as audio, video or timed text with specific attributes, e.g., bandwidth, language, or resolution.

3. Protocol Overview

3.1. Protocol Architecture

The functional entities involved in the PPSP Peer Protocol are Peers, which may support different capabilities.

Peers correspond to devices that actually participate in sharing a media content and are organized in (various) swarms corresponding each swarm to the group of peers streaming that content at any given time.

Each peer contacts a Tracker to advertise which information it has available. When a peer wishes to obtain information about the swarm, it contacts the Tracker to find other peers participating in that specific swarm.

The tracker is a logical entity that maintains the lists of peers storing/exchanging chunks for a specific Live media channel or VoD media streaming content, answers queries from peers and collects information on the activity of peers. A simplified network diagram showing this interaction of tracker and peers is depicted in Figure 1.

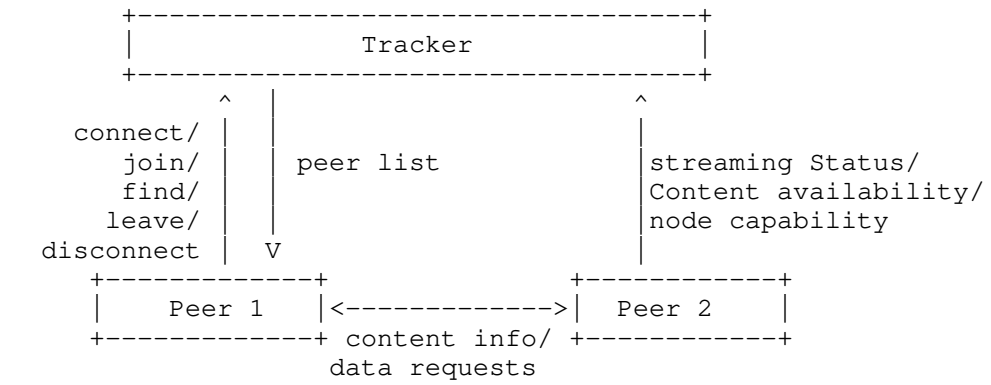


Figure 1: A PPSP streaming process

The signaling between PPSP Peers and trackers is done using a request/reply mechanism as defined in PPSP Tracker protocol [I-D.gu-ppsp-tracker-protocol].

This protocol can be used to connect peers that are sharing real-time streams of video or offline video, segmented in chunks. As for the streams of video, they can correspond to Live or Video on Demand streaming modes.

There are some significant differences between the details of these

scenarios, i.e., Live streaming, VoD and offline video. From a high level perspective the overall structure is quite similar. The optimal signaling flow for the different scenarios could also be different, but it depends on the real situation and on the implementer's choice

This draft defines a PULL based streaming signaling, as mandatory. However, a PUSH based or hybrid streaming signaling can optionally be considered.

For a PULL based Peer Protocol, the steps of signaling for a peer wishing to participate either in a Live streaming or a VoD or offline video is as follows (assuming the leech peer has already obtained from the Tracker a list of peers) and that, in case of traversing a NAT, performed ICE connectivity checks [I-D.li-ppsp-nat-traversal] with candidate peers using PPSP's own authentication method, as described in [I-D.gu-ppsp-tracker-protocol]:

1. The leech peer using PPSP Peer Protocol messages, establishes a connection to at least one of the peers in the Peerlist, based on the known PeerID and Peer IP address.

2. The peer sends request to candidate peers and the request could include one or more of the information described in below:

    * Request for the data availability of the candidate peer;

    * Notify its data availability to the candidate peer;

    * Request for the peer status of the candidate peer;

    * Notify its peer status to the candidate peer;

    * Request for additional peerlist;

    * Transport negotiation, wherein the requesting peer can have two choices:

        + Only support Mandatory Tranport Protocol;

        + Providing a list of supported Transport protocol.

3. Finally, the peers exchange the actual chunks of data, using the mechanism/protocol negotiated in the previous step.

In terms of Data Transport protocol negotiation, the leech peer can either inform the candidate that it supports a Mandatory Tranport Protocol or provides a list of supported Transport protocols. That

there are several options here to negotiate the connection model. The PPSP Peer Protocol may include new mechanisms to negotiate the protocol used to exchange data, or the offer-answer mechanism in SIP [RFC3261] (the IETF protocol for session establishment) along with SDP [RFC4566].

Note also that these mechanisms are not new protocols defined in PPSP, but existing protocols, and would eventually differ between an offline and a Live streaming scenario. Mechanisms such as flow control are handled in the negotiated Data Transport mechanism, not in the Peer Protocol itself.

3.2. Example Call Flow

This is a very high-level example of a session in which a leech peer joins a swarm, and retrieves some data (either via blocks or by streaming). The protocol used is indicated for each transaction. Note that not all of the communication shown in this figure are in scope of Peer Protocol, only those request/response followed by Peer Protocol are in scope.

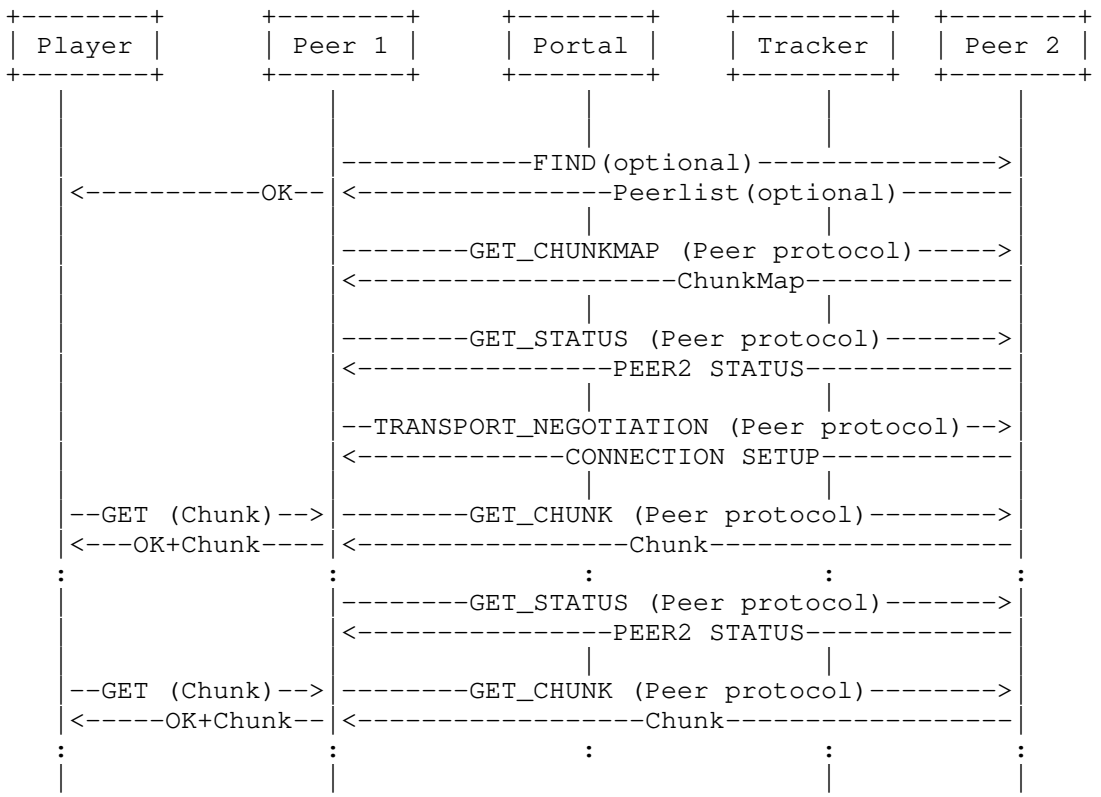


Figure 2: Example Call Flow

3.3. Chunk Scheduling

The goal of chunk trading is receiving the stream smoothly (and with small delay) and to cooperate in the distribution procedure. Peers need to exchange information about their current status to enable scheduling decisions. The information exchanged refers to the state of the peer with respect to the flow, i.e., a map of which chunks are needed by a peer to smoothly playback the stream.

This task means:

1. sending chunk maps to other nodes with the proper timing,

2. receiving chunk maps from other nodes and merging the information in the local buffer map.

3. besides chunk map exchange, the signaling includes Status/ Request/Select primitives used to trade chunks.

The core of the scheduler, not described in this specification, is the algorithm used to choose the chunks to be exchanged and the peers to communicate with.

4. Protocol Architecture

The PPSP Peer Protocol is a request-response protocol. Requests are sent, and responses returned to these requests. A single request generates a single response (neglecting fragmentation of messages).

As shown in example call flow depicted in Figure 2, the Peer protocol only provides signaling messages for obtaining additional peerlist (optionally), query for content availability and negotiation for transfer protocol. Peer protocol may also provide communication for peers to exchange information that can improve system performance.

The encoding for the signaling messages is not yet decided. Two encodings are proposed, a Text-based (HTTP/XML) and a Binary-based, described in Appendixes A and B. The authors will raise more discussion on the encoding, and will move the one that gets rough consensus of the PPSP WG to the draft text. In the Appendixes, some considerations are provided on each encoding based on the Mail List discussions.

The specific PPSP signaling messages are listed as following:

GET_PEERLIST:

The GET_PEERLIST message is sent from a leech peer to one or more remote peers in order for a peer to refresh/update the list of active peers in the swarm.

When receiving the GET_PEERLIST message, and if the message is well formed and accepted, the peer will search for the requested data and will respond to the leech peer with the peer list with PeerIDs (and respective IP Addresses) of sender peers that can provide the specific content.

GET_CHUNKMAP:

The GET_CHUNKMAP message is sent from a leech peer to one or more remote peers in order to receive the map of chunks of a content (of a swarm identified by SwarmID) the other peer presently stores. The chunk map returned by the other peer lists ranges of chunks.

When receiving the GET_CHUNKMAP message, and if the message is well formed and accepted, the peer will search for the requested data and will respond to the leech peer with the map of chunks it currently stores of the specific content.

GET_CHUNK:

The GET_CHUNK message is sent from a leech peer to sender peer in order to request the delivery of media content chunks.

When receiving the GET_CHUNK message, and if the message is well formed and accepted, the peer will search for the requested data and will respond to the leech peer with the specific chunks the leech peer requested.

GET_STATUS:

The GET_ STATUS message is sent from a leech peer to one or more remote peers in order to request the corresponding properties of the sender peers. The corresponding properties are enumerated in [draft-gu-ppsp-tracker-protocol], e.g., Caching Size, Bandwidth etc.

When receiving the GET_STATUS message, and if the message is well formed and accepted, the peer will search for the requested data and will respond to the leech peer with the specific parameters to the properties the leech peer requested.

TRANSPORT_NEGOTIATION:

The TRANSPORT_NEGOTIATION message is sent from a leech peer to a sender peer in order to negotiate the underlying transport protocol. Leech peer provide a set of transport protocols it supported to sender peer, and leave send peer to choose its preference. Reusing existing transport protocol to transfer data is recommended.

When receiving the TRANSPORT_NEGOTIATION message, and if the message is well formed and accepted, the sender peer will decide the transport protocol and will respond to the leech peer with the specific transport protocol the sender peer preferred.

5. Security Consideration

P2P streaming systems are subject to attacks by malicious/unfriendly peers/trackers that may eavesdrop on signaling, forge/deny information/knowledge about streaming content and/or its availability, impersonating to be another valid participant, or launch DoS attacks to a chosen victim.

No security system can guarantees complete security in an open P2P streaming system where participants may be malicious or uncooperative. The goal of security considerations described here is to provide sufficient protection for maintaining some security properties during the peer-peer communication even in the face of a large number of malicious peers.

As in typical Peer to Peer network, the most significant security issue is that the peers are untrusted. A peer may announce that it has a specific content, but the content might be just noise or it could be poisoned. A peer could also download a large number of chunks but upload very few of them. This problem can be alleviated by incentive mechanism, the goal of which is to reward honest peers and degrade dishonest peers.

5.1. Authentication

To protect the PPSP signaling from attackers pretending to be valid peers (or peers other than themselves) all messages received in the Tracker are required to be received from authorized peers.

For that purpose a peer must enroll in the system via a centralized enrollment server. The enrollment server is expected to provide a proper PeerID for the peer and information about the authentication mechanisms. The specification of the enrollment method and the provision of identifiers and authentication tokens is out of scope of this draft.

The authetication mechanism MUST allow the means for negotiating data security layer mechanisms to provide data integrity, data confidentiality, and other services, subject to local policies and security requirements.

5.2. Content Integrity Protection Against Polluting Peers/Trackers

Malicious peers may declaim ownership of popular content to the Tracker but serve polluted (i.e., decoy content or even virus/trojan infected contents) to other peers. This kind of pollution can be detected by incorporating a checksum distribution scheme for published sharing content. As content chunks of the same content are

transferred independently and concurrently, correspondent chunk-level checksums MUST be distributed from an authentic origin.

5.3. Residual Attacks and Mitigation

To mitigate the impact of sybil attackers, impersonating a large number of valid participants by repeatedly acquiring different peer identities, the enrollment server SHOULD carefully regulate the rate of peer/tracker admission.

There is no guarantee that a peer honestly report its status to the Tracker, or server authentic content to other peers as it claims to the Tracker. It is expected that a global trust mechanism, where the credit of each peer is accumulated from evaluations for previous transactions, may be taken into account by other peers when selecting partner for future transactions, helping to mitigate the impact of such malicious behaviors. A globally trusted Tracker MAY also take part of the trust mechanism by collecting evaluations, computing credit values and providing them to joining peers.

5.4. Pro-incentive Parameter Trustfulness

Properties for PEER_STATUS messages will consider pro-incentive parameters, which can enable the improvement of the performance of the whole P2P streaming system. Trustworthiness of these pro-incentive parameters is critical to the effectiveness of the incentive mechanisms. For example, ChunkMap is essential, and needs to be accurate. The P2P system should be designed in a way such that a peer will have the incentive to report truthfully its ChunkMap (otherwise it may penalize itself).

Furthermore, both the amount of upload and download should be reported to the Tracker to allow checking if there is any inconsistency between the upload and download report, and establish an appropriate credit/trust system.

6. References

6.1. Normative References

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

[RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002.

[RFC4566] Handley, M., Jacobson, V., and C. Perkins, "SDP: Session Description Protocol", RFC 4566, July 2006.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[draft-ietf-p2psip-base]
Jennings, C., Lowekamp, B., Ed., Rescorla, E., and H. Schulzrinne, "draft-ietf-p2psip-base-07", February 2010, <draft-ietf-p2psip-base>.

6.2. Informative References

[I-D.ietf-ppsp-problem-statement]
Zhang, Y., Zong, N., Camarillo, V., Seng, J., and R. Yang, "Problem Statement of P2P Streaming Protocol (PPSP)", Januray 2011, <I-D.ietf-ppsp-problem-statement>.

[I-D.ietf-ppsp-reqs]
Zong, N., Zhang, Y., Pascual, V., and C. Williams, "P2P Streaming Protocol (PPSP) Requirements", February 2011, <I-D.ietf-ppsp-reqs>.

[I-D.ietf-ppsp-survey]
Gu, Y., Zong, N., Zhang, H., Zhang, Y., Camarillo, G., and Y. Liu, "Survey of P2P Streaming Applications", March 2011, <I-D.ietf-ppsp-survey>.

[I-D.gu-ppsp-tracker-protocol]
Cruz, R., Nunes, M., Gu, Y., Xia, J., Bryan, D., Taveira, J., and D. Deng, "PPSP Tracker Protocol", March 2011, <I-D.gu-ppsp-tracker-protocol>.

[I-D.cruz-ppsp-http-peer-protocol]
Gu, Y., Xia, J., Cruz, R., Nunes, M., Bryan, D., and J. Taveira, "PPSP HTTP-Based Peer Protocol", March 2011, <I-D.cruz-ppsp-http-peer-protocol>.

[I-D.li-ppsp-nat-traversal]
Li , L., Wang , J., and W. Chen , "PPSP NAT Traversal", March 2011, <I-D.li-ppsp-nat-traversal>.

[BittorrentSpecification]
"Bittorrent Protocol Specification v1.0", February 2010, <Bittorrent Specification>.

Appendix A. Binary Encoding

Binary Encoding is an encoding of data in plain text. More precisely, it is an encoding of binary data in a sequence of ASCII-printable characters. Binary Encoding is necessary for transmission of data when the channel or the protocol only allows ASCII-printable characters.

The PPSP Peer protocol can be carried on top of IP, UDP, RTP or TCP. But using which layer to carry peer protocol is out of scope in current stage.

The peer message header has the following format:

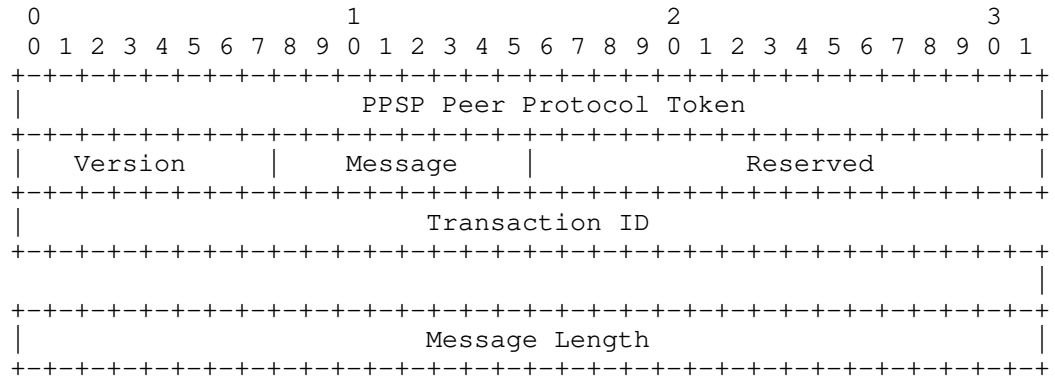


Figure 3: PPSP Peer message header

The fields have the following meaning:

PPSP Peer Protocol Token: 32 bits

A fixed token indicating to the receiver this message is a PPSP Peer Protocol message. The token field is four bytes long. This value MUST be set to 0x50505350, the string "PPSP".

Version: 8 bits The version of the PPSP peer protocol being used in the form of a fixed point integer between 0.1 and 25.4. For the version of the protocol described in this document, this field MUST be set to 0.1. The version field is one byte long.

Message Types: 8 bits

Message types currently have two kinds of value: Request and Response.

Reserved: 16 bits

Not to be assigned. Reserved values are held for special uses, such as to extend the namespace when it becomes exhausted. Reserved values are not available for general assignment.

Transaction ID: 64 bits

Identifies the transaction and also allows receivers to disambiguate transactions which are otherwise identical. Responses use the same Transaction ID as the request they correspond to. Transaction IDs are also used for fragment reassembly.

Message Length: 32 bits:

The length of the message, including header, in bytes. Note if the message is fragmented, this is the length of this message, not the total length of all assembled fragments.

A.1. Methods in Peer messages

To improve the compatibility of the peer methods, the method fields in message extension MUST be encoded as TLV elements as described below and sketched in Figure 4:

To improve the compatibility of the peer methods, the method fields in message extension MUST be encoded as TLV elements as described below and sketched in Figure 4:

- Type: A single-octet identifier that defines the type of the parameter represented in this TLV element.
- Length: A two-octet field that indicates the length (in octets) of the TLV element excluding the Type and Length fields, and the 8-bit Reserved field between them. Note that this length does not include any padding that is required for alignment.

o Value: Variable-size set of octets that contains the specific value for the parameter.

In the extensions, the Reserved field SHALL be set to zero and ignored. If a TLV element does not fall on a 32-bit boundary, the last word MUST be padded to the boundary using further bits set to zero.

In a peer message, any method extension MUST be placed after the mandatory message header. The extensions MAY be placed in any order.

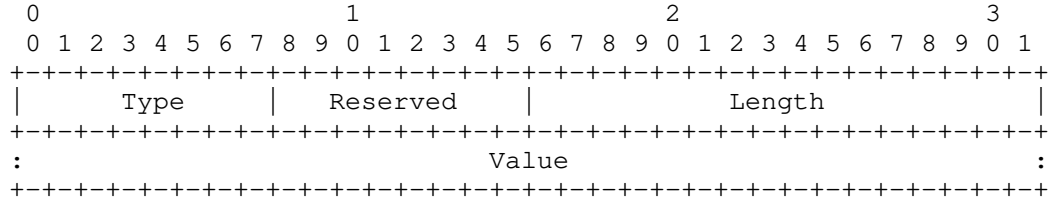


Figure 4: Structure of a TLV element

Method Type: 8 bits

Indicates the method type for the message. There are five method types: GET_PEERLIST, GET_CHUNKMAP, GET_CHUNK, GET_ PROPERTY and TRANSPORT_NEGOTIATION. They are counted from 1 to 5.

Method Body Length: 24 bits

The length of the method body in bytes.

A.1.1. GET_PEERLIST

Peerlist is composed of several pairs of Peer ID and Peer IP. Peer ID is a 128 bit integer that is unique in the P2P streaming system. That’s no matter there is a centralized tracker or several distributed trackers in the streaming system, a peer ID should be unique.

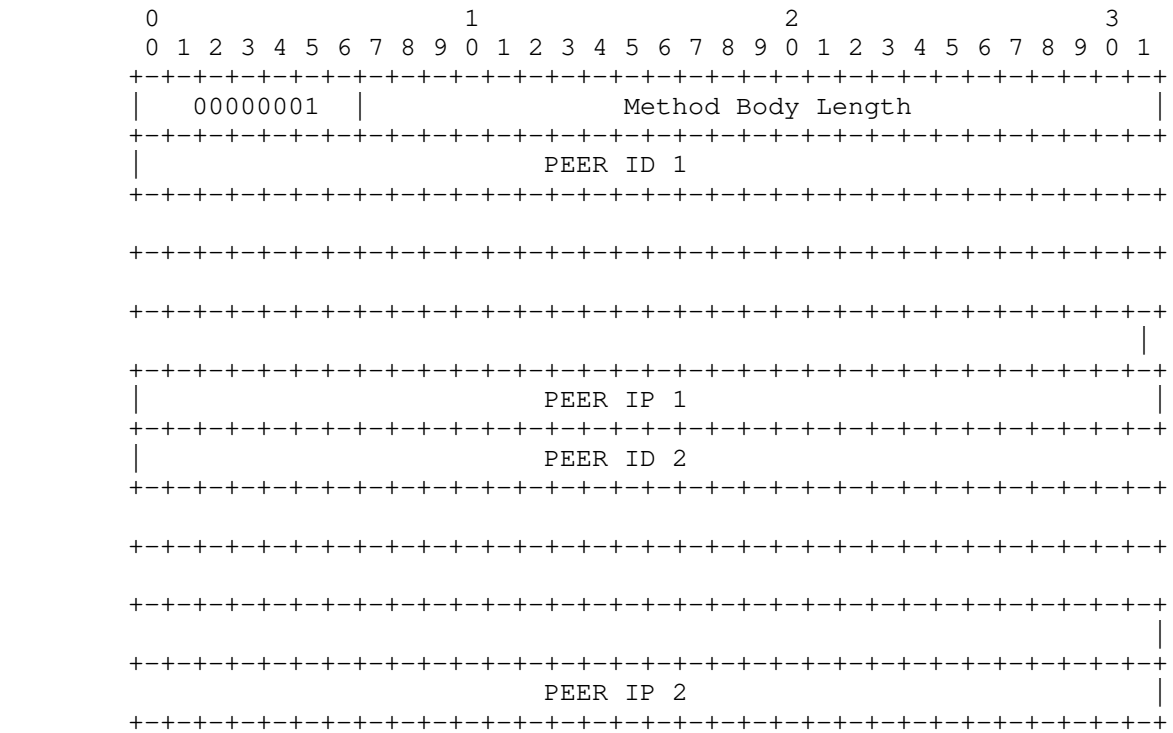


Figure 5: GET_PEERLIST Method Body

A.1.2. GET_CHUNKMAP

Chunkmap of a content (a swarm identified by SwarmID)

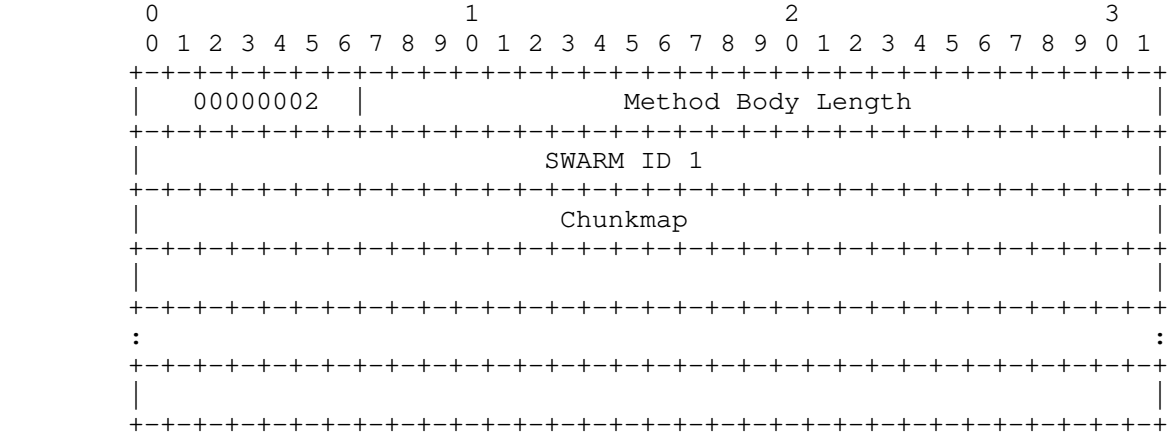


Figure 6: GET_CHUNKMAP Method Body

A.1.3. GET_CHUNK

[TBD]

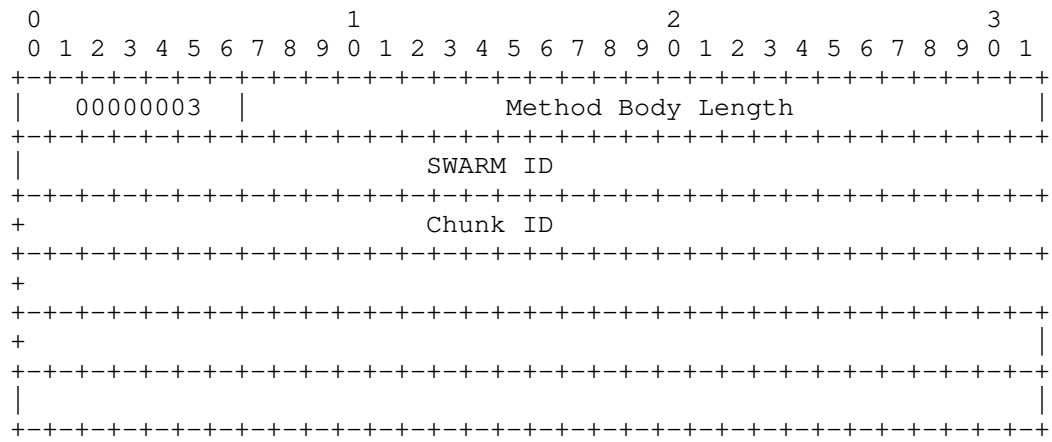


Figure 6: GET_CHUNKMAP Method Body

A.1.4. GET_STATUS

Several property types are defined in I-D.gu-ppsp-tracker-protocol. But not all of the property types are reasonable to be used in peer protocol. So we just list the following property types. New types can be easily added.

PROPERTY	Description	Code
CachingSize	Caching size: available size for caching	0x01
Bandwidth	Bandwidth: available bandwidth	0x02
LinkNumber	Link number: acceptable links for remote peer	0x03
Certificate	Certificate: certificate of the peer	0x04

Table 1: Status changed between peers

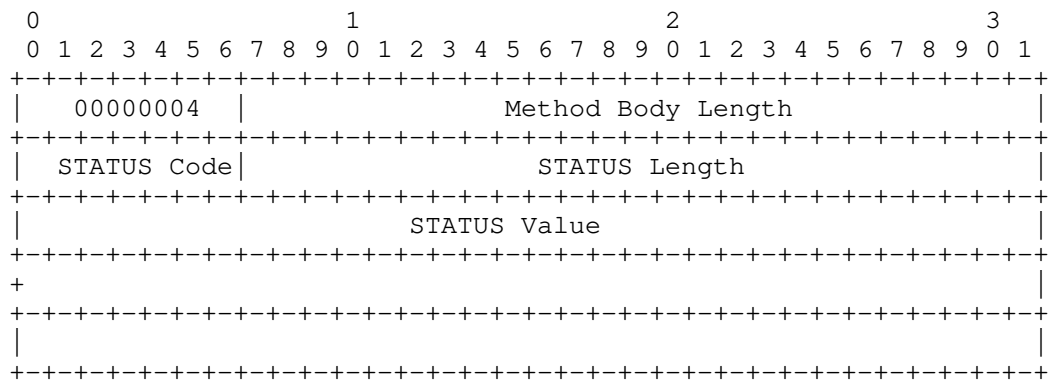


Figure 6: GET_STATUS Method Body

A.1.5. TRANSPORT_NEGOTIATION

To Do: Define mandatory transport protocol and some optional transport protocol.

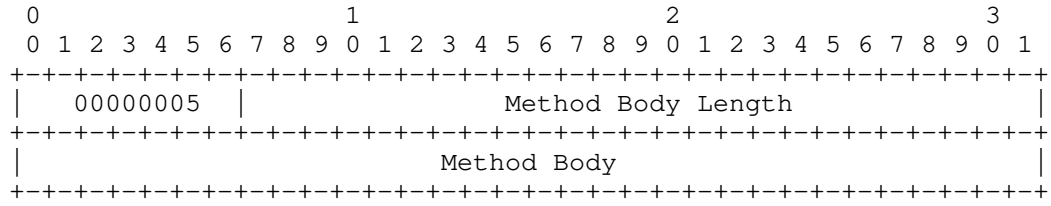


Figure 7: TRANSPORT_NEGOTIATION Method Body

Appendix B. HTTP/XML Encoding

The PPSP Peer Protocol HTTP/XML encoding messages follow the request and response standard formats for HTTP Request and Response messages [RFC2616].

B.1. HTTP/XML Encoding

A Request message is a standard HTTP Request generated by the HTTP Client Peer with the following syntax:

```
<Method> /<Resource> HTTP/1.1
Host: <Host>
```

The HTTP Method and URI path (the Resource) indicates the operation requested. The current proposal uses only HTTP POST as a mechanism for the request messages.

The Host header follows the standard rules for the HTTP 1.1 Host Header.

The Response message is also a standard HTTP Response generated by the HTTP Serving Peer with the following syntax:

```
HTTP/1.1 <StatusCode> <StatusMsg>
Content-Lenght: <ContentLenght>
Content-Type: <ContentType>
Content-Encoding: <ContentCoding>
<Response_Body>
```

The body for both Request and Response messages are encoded in XML for all the PPSP Peer Protocols messages, with the following schema (the XML information being method specific):

```
<?xml version="1.0" encoding="utf-8"?>
<ProtocolName version="#.#">
   <Method>***</Method>     <!-- for the Request method -->
   <Response>***</Response> <!-- for the Response method -->
   <TransactionID>###</TransactionID>
     ...XML information specific of the Method...
</ProtocolName>
```

In the XML body, the *** represents alphanumeric data and ### represents numeric data to be inserted. The <Method> corresponds to the method type for the message, the <Response> corresponds to the response method type of the message and the element <TransactionID> uniquely identifies the transaction.

The Response message MAY use Content-Encoding entity-header with "gzip" compression scheme [RFC2616] for faster transmission times and less network bandwidth usage.

B.2. Method Fields

Table B 1 and Table B 2 define the valid string representations for the requests and responses, respectively. These values MUST be treated as case-insensitive.

PPSP Request	XML Request Value String
GET_PEERLIST	GET_PEERLIST
GET_CHUNKMAP	GET_CHUNKMAP
GET_CHUNK	GET_CHUNK
PEER_STATUS	PEER_STATUS
TRANSPORT_NEGOTIATION	TRANSP_NEGO

Table B 1: Valid Strings for Requests

Response Method Name	HTTP Response Mechanism	XML Response Value
SUCCESSFUL (OK)	200 OK	OK
INVALID SYNTAX	400 Bad Request	INVALID SYNTAX
VERSION NOT SUPPORTED	400 Bad Request	VERSION NOT SUPPORTED
AUTHENTICATION REQUIRED	401 Unauthorized	AUTHENTICATION REQUIRED
MESSAGE FORBIDDEN	403 Forbidden	MESSAGE FORBIDDEN
OBJECT NOT FOUND	404 Not Found	OBJECT NOT FOUND
INTERNAL ERROR	500 Internal Server Error	INTERNAL ERROR
TEMPORARILY OVERLOADED	503 Service Unavailable	TEMPORARILY OVERLOADED

Table B 2: Valid Strings for Responses

B.3. Message Processing

When a PPSP Peer Protocol message is received in a peer, some basic processing is performed, regardless of the message type. Upon reception, a message is examined to ensure that it is properly formed. The receiver MUST check that the HTTP message itself is properly formed, and if not appropriate standard HTTP errors MUST be generated. The receiver must also verify that the XML body is properly formed. If the message is found to be incorrectly formed or the length does not match the length encoded in the header, the receiver MUST reply with an HTTP 400 response with a PPSP XML body with the Response method set to INVALID SYNTAX.

B.4. GET_PEERLIST Message

The GET_PEERLIST message is sent from a client peer to a selected serving peer in order for a peer to refresh/update the list of active peers in the swarm.

The Request message uses a HTTP POST method with the following body:

```
<?xml version="1.0" encoding="utf-8"?>
<PPSPPeerProtocol version="#.#">
   <Method>GET_PEERLIST</Method>
   <PeerID>***</PeerID>
   <SwarmID>***</SwarmID>
   <TransactionID>###</TransactionID>
</PPSPPeerProtocol>
```

The sender MUST properly form the XML body, MUST set the Method string to GET_PEERLIST, MUST set the PeerID to the PeerID of the peer, MUST set the SwarmID to the joined swarm identifier and randomly generate and set the TransactionID value.

When receiving the GET_PEERLIST message, and if the message is well formed and accepted, the peer will search for the requested data and will respond to the requesting peer with an HTTP 200 OK message response with a PPSP XML payload SUCCESSFUL, as well as the peer list with PeerIDs (and respective IP Addresses) of peers that can provide the specific content.

The response MUST have the same TransactionID value as the request.

An example of the Response message structure is the following:

```
<?xml version="1.0" encoding="utf-8"?>
<PPSPPeerProtocol version="#.#">
   <Response>OK</Response>
   <SwarmID>***</SwarmID>
   <TransactionID>###</TransactionID>
   <PeerInfoList>
      <PeerInfo>
         <PeerID>***</PeerID>
         <PeerType>***</PeerType>
         <PeerAddresses>
            <PeerAddress ip="##.##.##.##"
                         port="###" />
            <PeerAddress ip="hh:hh:hh:hh:hh:hh:hh:hh"
                         port="###" />
         </PeerAddresses>
         <PeerLocation>****</PeerLocation>
         <ConnectionType>***</ConnectionType>
         <EndPointRankCost>###</EndPointRankCost>
      </PeerInfo>
      <PeerInfo>
         <PeerID>***</PeerID>
         <PeerType>***</PeerType>
         <PeerAddresses>
            <PeerAddress ip="##.##.##.##"
                         port="###" />
            <PeerAddress ip="hh:hh:hh:hh:hh:hh:hh:hh"
                         port="###" />
         </PeerAddresses>
         <PeerLocation>****</PeerLocation>
         <ConnectionType>***</ConnectionType>
         <EndPointRankCost>###</EndPointRankCost>
      </PeerInfo>
   </PeerInfoList>
</PPSPPeerProtocol>
```

The element <PeerInfoList> MAY contain multiple <PeerInfo> child elements.

The element <PeerAddresses> MAY contain multiple <PeerAddress> child elements with attributes "ip" and "port" corresponding to each of the network interfaces of the peer. The "ip" attribute can be expressed in dotted decimal format for IPv4 or 16-bit hexadecimal values (hh) separated by colons (:) for IPv6.

The elements <PeerLocation> and <ConnectionType> have a string format, and together with the element <EndPointRankCost> of numerical integer format, form a set of information related to peer location.

B.5. GET_CHUNKMAP Message

The GET_CHUNKMAP message is sent from a client peer to a selected serving peer in order to receive the map of chunks of a content (of a swarm identified by SwarmID) the other peer presently stores. The chunk map returned by the other peer lists ranges of chunks. The Request message uses a HTTP POST method with the following body:

```
<?xml version="1.0" encoding="utf-8"?>
<PPSPPeerProtocol version="#.#">
   <Method>GET_CHUNKMAP</Method>
   <PeerID>***</PeerID>
   <SwarmID>***</SwarmID>
   <TransactionID>###</TransactionID>
</PPSPPeerProtocol>
```

The sender MUST properly form the XML body, MUST set the Method string to GET_CHUNKMAP, MUST set the PeerID to the PeerID of the peer, MUST set the SwarmID to the joined swarm identifier and randomly generate and set the TransactionID value.

When receiving the GET_CHUNKMAP message, and if the message is well formed and accepted, the peer will search for the requested data and will respond to the requesting peer with an HTTP 200 OK message response with a PPSP XML payload SUCCESSFUL, as well as the map of chunks it currently stores of the specific content.

The response MUST have the same TransactionID value as the request.

The Response message is an HTTP 200 OK message with the following body, exemplified for a video component of a media clip:

```
<?xml version="1.0" encoding="utf-8"?>
<PPSPPeerProtocol version="#.#">
   <Response>OK</Response>
   <TransactionID>###</TransactionID>
   <StreamInfo>
      <SwarmID>***</SwarmID>
      <Clip>
         <Name>***</Name>
         <ChunkSegments type="video/audio/etc">
            <ChunkSegment from="###" to="###"
                          bitmapSize="###">
               ...(base64 string)...
           </ChunkSegment>
         </ChunkSegments>
      </Clip>
   </StreamInfo>
```

```
</PPSPPeerProtocol>
```

The element <StreamInfo> MAY contain multiple <Clip> child elements.

The element <ChunkSegments> has an attribute "type" that indicates the type of media of the corresponding chunks.

A <ChunkSegments> element MAY contain multiple <ChunkSegment> child elements with attributes "from" and "to" corresponding to ranges of contiguous chunks. The "from", "to", and "bitmapSize" attributes are expressed as integer number string format. The <ChunkSegment> content corresponds to the chunk map, and is represented as base64 encoded string.

B.6. GET_CHUNK Message

The GET_CHUNK message is sent from a client peer to a serving peer in order to request the delivery of media content chunks. The Request message uses a HTTP POST method with the following body:

```
<?xml version="1.0" encoding="utf-8"?>
<PPSPPeerProtocol version="#.#">
   <Method>GET_CHUNK</Method>
   <PeerID>***</PeerID>
   <SwarmID>***</SwarmID>
   <TransactionID>###</TransactionID>
</PPSPPeerProtocol>
```

The sender MUST properly for the HTTP request for a POST method including the URI path (the Resource) of the chunk. The sender MUST also properly form the XML body, MUST set the Method string to GET_CHUNK, MUST set the PeerID to the PeerID of the peer, MUST set the SwarmID to the joined swarm identifier and randomly generate and set the TransactionID value.

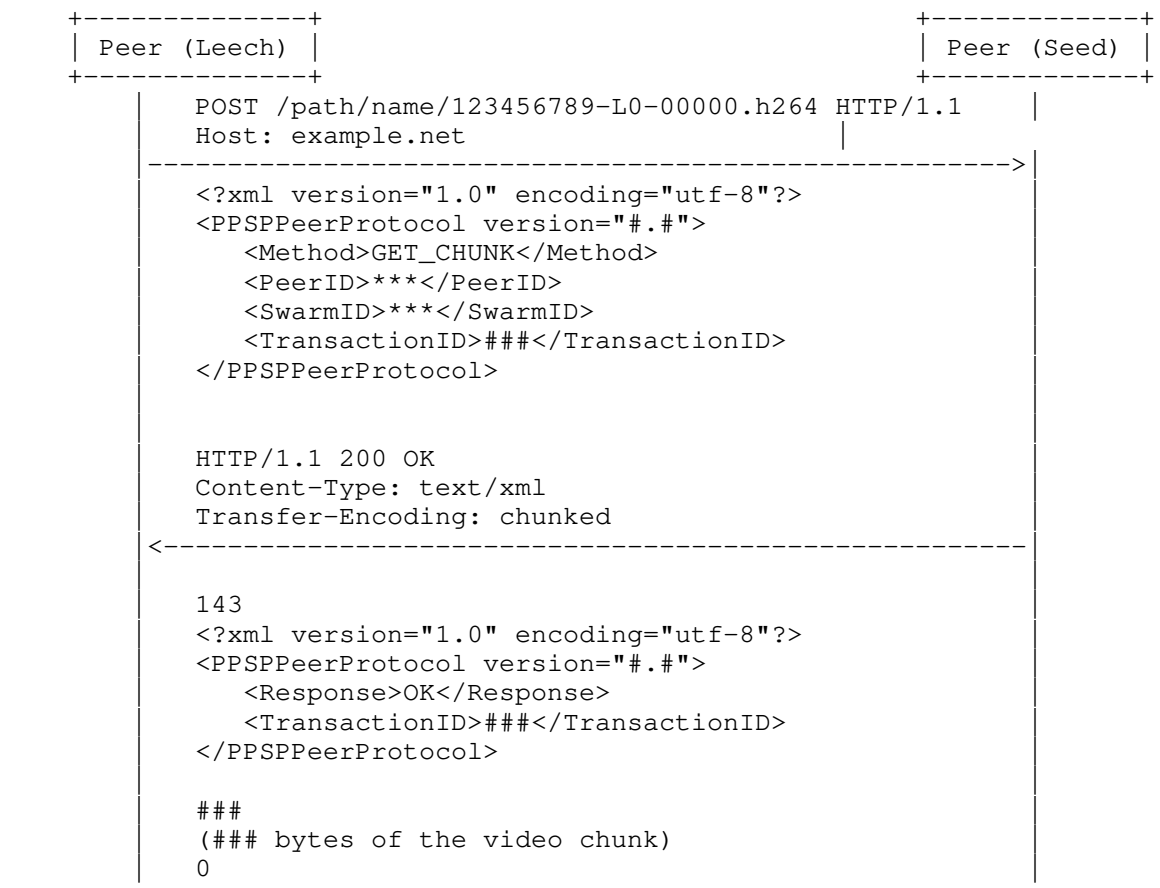


Figure B 1: Example of GET_CHUNK message sequence (simplified)

When receiving the GET_CHUNK message, and if the message is well formed and accepted, the peer will search for the requested data and will respond to the requesting peer with an HTTP 200 OK message response with a PPSP XML payload SUCCESSFUL.

The Response message is an HTTP 200 OK message. If The Data Transport Protocol negotiated is also HTTP/XML, the body of the response to GET_CHUNK can be immediately followed by the chunk data transfer, as shown in Figure B 1.

The response MUST have the same TransactionID as the request.

B.7. PEER_STATUS Message

The PEER_STATUS message is sent from a serving peer to a client peer to indicate its participation status. The information conveyed may include information related to chunk trading like "choke" (to inform the other peer of the intention to stop sending data to it) and "unchoke" (to inform the other peer of the intention to start/re-start sending data to it).

The Request message uses a HTTP POST method with the following body:

```
<?xml version="1.0" encoding="utf-8"?>
<PPSPPeerProtocol version="#.#">
   <Method>PEER_STATUS</Method>
   <PeerID>***</PeerID>
   <SwarmID>***</SwarmID>
   <TransactionID>###</TransactionID>
   <Status>(choke/unchoke)</Status>
</PPSPPeerProtocol>
```

The sender MUST properly form the XML body, MUST set the Method string to PEER_STATUS, MUST set the PeerID to the PeerID of the peer, MUST set the SwarmID to the joined swarm identifier and randomly generate and set the TransactionID value.

When receiving the PEER_STATUS message, and if the message is well formed and accepted, the peer will respond to the requesting peer with an HTTP 200 OK message response with a PPSP XML payload SUCCESSFUL.

If the status signal received is "choke" the peer will stop requesting chunks from the other peer until receiving an "unchoke" status signal.

The only element currently defined in the request message is <Status>, assuming values of "choke" and "unchoke", but, in future, other values may be added.

The Response message is an HTTP 200 OK message with the following body.

```
<?xml version="1.0" encoding="utf-8"?>
<PPSPPeerProtocol version="#.#">
   <Response>OK</Response>
   <TransactionID>###</TransactionID>
</PPSPPeerProtocol>
```

The response MUST have the same TransactionID value as the request.

The only element currently defined in the response message is the <TransactionID>, but, in future, other elements may be added, for example, containing statistical data or other primitives for chunk trading negotiation.

B.8. TRANSPORT_NEGOTIATION Message

To Do: Define message format, mandatory transport protocol and some optional transport protocols.

Authors' Addresses

Yingjie Gu
Huawei
No.101 Software Avenue
Nanjing, Jiangsu Province 210012
P.R.China

Phone: +86-25-56622638
Email: guyingjie@huawei.com

Jinwei Xia
Huawei
Software No.101
Nanjing, Yuhuatai District 210012
China

Phone: +86-025-86622310
Email: xiajinwei@huawei.com

Rui Santos Cruz
IST/INESC-ID/INOV

Phone: +351.939060939
Email: rui.cruz@ieee.org

Mario Serafim Nunes
IST/INESC-ID/INOV
Rua Alves Redol, n.9
1000-029 LISBOA
Portugal

Phone: +351.213100256
Email: mario.nunes@inov.pt

David A. Bryan
Polycom
San Jose, CA, USA,
USA

Phone:
Email: dbryan@ethernot.org

Joao P. Taveira
ID/INOV

Email: joao.silva@inov.pt

PPSP
INTERNET-DRAFT
Intended Status: Standards Track
Expires: August 27, 2012

Rui S. Cruz
Mario S. Nunes
IST/INESC-ID/INOV
Yingjie Gu
Jinwei Xia
Huawei
David A. Bryan
Polycom
Joao P. Taveira
IST/INOV
Deng Lingli
China Mobile

February 24, 2012

PPSP Tracker Protocol (PPSP-TP)
draft-gu-ppsp-tracker-protocol-07

Abstract

This document specifies the Peer-to-Peer Streaming Protocol--Tracker Protocol (PPSP-TP), an application-layer control (signaling) protocol for the exchange of meta information between trackers and peers, such as initial offer/request of participation in multimedia content streaming, content information, peer lists and reports of activity and status. The specification outlines the architecture of the protocol and its functionality, and describes message flows, message processing instructions, message formats, formal syntax and semantics. The PPSP Tracker Protocol enables cooperating peers to form content streaming overlay networks to support near real-time Structured Media content (audio, video, associated text/metadata) delivery, such as adaptive multi-rate, layered (scalable) and multi-view (3D), in live, time-shifted and on-demand modes.

Status of this Memo

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

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

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

material or to cite them other than as "work in progress."

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

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html

Table of Contents

1. Introduction

The Peer-to-Peer Streaming Protocol (PPSP) is composed of two protocols: the PPSP Tracker Protocol and the PPSP Peer Protocol. [I-D.ietf-ppsp-problem-statement] specifies that the Tracker Protocol should standardize format/encoding of information and messages between PPSP peers and PPSP trackers and [I-D.ietf-ppsp-reqs] defines the requirements.

The PPSP Tracker Protocol provides communication between trackers and peers, by which peers send meta information to trackers, report streaming status and obtain peer lists from trackers.

The PPSP architecture requires PPSP peers able to communicate with a tracker in order to participate in a particular streaming content swarm. This centralized tracker service is used by PPSP peers for content registration and location. Content information metafiles (Media Presentation Descriptions) are also stored in the tracker system allowing active peers in the swarm to interpret content structure.

The signaling and the media data transfer between PPSP peers is not in the scope of this specification.

This document describes the PPSP Tracker protocol and how it satisfies the requirements for the IETF Peer-to-Peer Streaming Protocol (PPSP-TP), in order to derive the implications for the standardization of the PPSP streaming protocols and to identify open issues and promote further discussion.

1.1. Use Scenarios and Streaming Modes

This section is tutorial in nature and does not contain any normative statements.

This section describes some aspects of the use of PPSP-TP. The examples were chosen to illustrate the basic operation, but not to limit what PPSP-TP may be used for.

The functional entities related to PPSP protocols are the Client Media Player, the service Portal, the tracker and the peers. The complete description of these entities is not discussed here, as not in the scope the specification.

The Client Media Player is a logical entity providing direct interface to the end user at the client device, and includes the functions to select, request, decode and render contents. The Client Media Player may interface with the local peer application using

request and response standard formats for HTTP Request and Response messages [RFC2616].

The service Portal is a logical entity typically used for client enrollment and content information publishing, searching and retrieval.

The tracker is a logical entity that maintains the lists of PPSP active peers storing and exchanging a specific media content. The tracker also stores the status of active peers in swarms, to help in the selection of appropriate peers for a requesting peer. The tracker can be realized by geographically distributed tracker nodes or multiple server nodes in a data center, increasing the content availability, the service robustness and the network scalability or reliability. The management and locating of content index information are totally internal behaviors of the tracker system, which is invisible to the PPSP Peer [I-D.xiao-ppsp-reload-distributed-tracker].

The peer is also a logical entity embedding the P2P core engine, with a client serving side interface to respond to Client Media Player requests and a network side interface to exchange data and PPSP signaling with trackers and other peers.

The streaming technique is chunk-based, i.e., client peers obtain media chunks from serving peers and handle the buffering that is necessary for the playback processes during the download of the media chunks.

In Live streaming, all end users are interested in a specific media coming from an ongoing program, which means that all respective peers share nearly the same streaming content at a given point of time. Peers may store the live media for further distribution (known as time-shift TV), where the stored media is distributed in a VoD-like manner.

In VoD, different end users watch different parts of the recorded media content during a past event. In this case, each respective peer obtains from other peers the information on media chunks they store and then gets the required media from a selected set of those peers. While watching VoD, an end user can also switch to any place of the content, e.g., skip the credits part, or skip the part that it is not interested in. In this case the respective participating peer may not store all the content segments. From the whole swarm point of view, the participating peers typically store different parts of content.

1.2. Assumptions

This section is tutorial in nature and does not contain any normative statements.

The process used for media streaming distribution assumes a segment (chunk) transfer scheme whereby the original content (that can be encoded using adaptive or scalable techniques) is chopped into small segments (and subsegments) having the following representations:

1. Adaptive - alternate representations with different qualities and bitrates; a single represention is non-adaptive;
2. Scalable description levels - multiple additive descriptions (i.e., addition of descriptions refine the quality of the video);
3. Scalable layered levels - nested dependent layers corresponding to several hierarchical levels of quality, i.e., higher enhancement layers refine the quality of the video of lower layers.
4. Scalable multiple views - views correspond to mono (2D) and stereoscopic (3D) videos, with several hierarchical levels of quality.

These streaming distribution techniques support dynamic variations in video streaming quality while ensuring support for a plethora of end user devices and network connections.

1.2.1. Enrollment and Bootstrap

In order to join an existing P2P streaming service and to participate in content sharing, any peer must first locate a tracker, using for example, the following method (as illustrated in Figure 1):

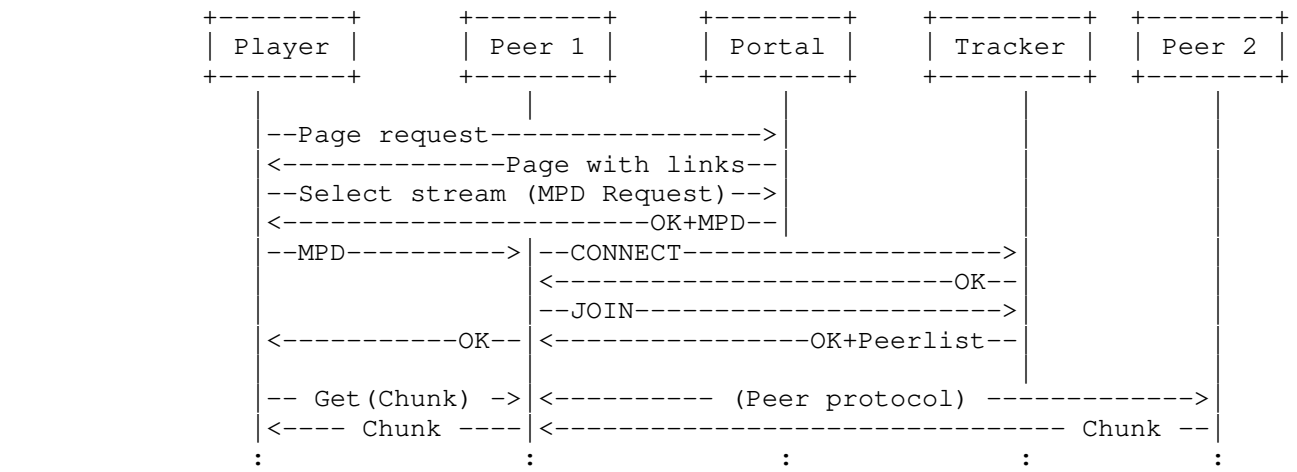


Figure 1: A typical PPSP session

1. From a service provider provisioning mechanism: this is a typical case used on the provider Super-Seeders (edge caches and/or Media Servers).
2. From a web page: a Publishing and Searching Portal may provide tracker location information to end users.
3. From the MPD file of a content: this metainfo file must contain information about the address of one or more trackers (that can be grouped by tiers of priority) which are controlling the swarm for that media content.

In order to be able to bootstrap, a peer must first obtain a Peer-ID (identity associated with the end user authentication credentials) and any required security certificates or authorization tokens from an enrollment service (end user registration).

The specification of the mechanism used to obtain a Peer-ID, certificates or tokens is not in the scope of this document.

1.2.2. NAT Traversal

It is assumed that all trackers must be in the public Internet and have been placed there deliberately. This document will not describe NAT Traversal mechanisms but the protocol facilitates flexible NAT Traversal techniques, such as those based on ICE [RFC5245], considering that the tracker node may provide NAT traversal services, as a STUN-like tracker. Being a STUN-like tracker, it can discover the reflexive candidate addresses of a peer and make them available in responses to requesting peers, a mechanism named PPSP-ICE in [I-D.li-ppsp-nat-traversal-02].

1.2.3. Content Information Metadata

Multimedia contents may consist of several media components (for example, audio, video, and text), each of which might have different characteristics.

The representations of a media content correspond to encoded alternative of the same media component, varying from other representations by bitrate, resolution, number of channels, or other characteristics. Each representation consists of one or multiple segments. Segments are the media stream chunks in temporal sequence.

These characteristics may be described in a Media Presentation Description (MPD). Examples of MPD for on-demand and Live programs are illustrated in Appendix B. It is envisioned that the content information metadata used in PPSP may align with the MPD format of ISO/IEC 23009-1 [ISO.IEC.23009-1].

1.2.4. Authentication, Confidentiality, Integrity

Channel-oriented security should be used in the communication between peers and tracker, such as the Transport Layer Security (TLS) to provide privacy and data integrity. HTTP/1.1 over TLS (HTTPS) [RFC2818] is the preferred approach for preventing disclosure of peer critical information via the communication channel.

Due to the transactional nature of the communication between peers and tracker a method for adding authentication and data security services via replaceable mechanisms should be employed. One such method is the OAuth 2.0 Authorization [I-D.ietf-oauth-v2] with bearer token, providing the peer with the information required to successfully utilize the access token to make protected requests to the tracker [I-D.ietf-oauth-v2-bearer].

2. Terminology

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].

This draft uses terms defined in [I-D.ietf-ppsp-problem-statement] and in [I-D.xiao-ppsp-reload-distributed-tracker].

Absolute Time: Absolute time is expressed as ISO 8601 [ISO.8601.2004] timestamps, using zero UTC offset (GMT). Fractions of a second may be indicated. Example for December 25, 2010 at 14h56 and 20.25 seconds: basic format 20101225T145620.25Z or extended format 2010-12-25T14:56:20.25Z.

Adaptive Streaming: Multiple alternate representations (different qualities and bitrates) of the same media content co-exist for the same streaming session; each alternate representation corresponds to a different media quality level; peers can choose among the alternate representations for decode and playback.

Base Layer: The playable representation level in Scalable Video Coding (SVC) required by all upper level Enhancements Layers for proper decoding of the video.

Chunk: A chunk is a generic term used whenever no ambiguity is raised, to refer to a segment or a subsegment of partitioned streaming media.

Complementary Representation: Representation in a set of representations which have inter-representation dependencies and which when combined result in a single representation for decoding

and presentation.

Connection Tracker: The tracker node to which the PPSP peer will connect when it wants to get registered and join the PPSP system.

Continuous media: Media with an inherent notion of time, for example, speech, audio, video, timed text or timed metadata.

Enhancement Layer: Enhancement differential quality level (complementary representation) in Scalable Video Coding (SVC) used to produce a higher quality, higher definition video in terms of space (i.e., image resolution), time (i.e., frame rate) or Signal-to-Noise Ratio (SNR) when combined with the playable Base Layer [ITU-T.H.264].

Join Time: Join time is the absolute time when a peer registers on a tracker. This value is recorded by the tracker and is used to calculate Online Time.

Live streaming: The scenario where all clients receive streaming content for the same ongoing event. The lags between the play points of the clients and that of the streaming source are small.

Media Component: An encoded version of one individual media type such as audio, video or timed text with specific attributes, e.g., bandwidth, language, or resolution.

Media Presentation Description (MPD): Formalized description for a media presentation, i.e., describes the structure of the media, namely, the representations, the codecs used, the segments (chunks), and the corresponding addressing scheme.

Method: The method is the primary function that a request from a peer is meant to invoke on a tracker. The method is carried in the request message itself.

Online Time: Online Time shows how long the peer has been in the P2P streaming system since it joins. This value indicates the stability of a peer, and it is calculated by tracker when necessary.

Peer: A peer refers to a participant in a P2P streaming system that not only receives streaming content, but also stores and uploads streaming content to other participants.

Peer-ID: Unique identifier for the peer. The Peer-ID and any required security certificates are obtained from an offline enrollment server.

Peer-Peer Messages (i.e., Peer Protocol): The Peer Protocol messages

enable each peer to exchange content availability with other peers and request other peers for content.

PPSP: The abbreviation of Peer-to-Peer Streaming Protocols. PPSP protocols refer to the key signaling protocols among various P2P streaming system components, including the tracker and peers.

Representation: Structured collection of one or more media components.

Request: A message sent from a peer to a tracker, for the purpose of invoking a particular operation.

Response: A message sent from a tracker to a peer, for indicating the status of a request sent from the peer to the tracker.

Scalable Streaming: With Multiple Description Coding (MDC), multiple additive descriptions (that can be independently played-out) to refine the quality of the video when combined together. With Scalable Video Coding (SVC), nested dependent enhancement layers (hierarchical levels of quality), refine the quality of lower layers, from the lowest level (the playable Base Layer). With Multiple View Coding (MVC), multiple views allow the video to be played in 3D when the views are combined together.

Segment: A segment is a resource that can be identified, by an ID or an HTTP-URL and possibly a byte-range, and is included in the MPD. The segment is a basic unit of partitioned streaming media, which is used by a peer for the purpose of storage, advertisement and exchange among peers.

Subsegment: Smallest unit within segments which may be indexed at the segment level.

Swarm: A swarm refers to a group of peers sharing the same content (e.g., video/audio program, digital file, etc.) at a given time.

Swarm-ID: Unique identifier for a swarm. It is used to describe a specific resource shared among peers.

Tracker: A tracker refers to a centralized logical directory service used to communicate with PPSP Peers for content registration and location, which maintains the lists of PPSP peers storing segments for a specific live content channel or streaming media and answers queries from PPSP peers.

Tracker-Peer Messages (i.e., Tracker Protocol): The Tracker Protocol messages provide communication between peers and trackers, by which

peers provide content availability, report streaming status and request peer lists from trackers.

Video-on-demand (VoD): A kind of application that allows users to select and watch video content on demand.

3. Architectural and Functional View

The PPSP Tracker Protocol architecture uses a two-layer approach i.e., a PPSP-TP messaging layer and a PPSP-TP request/response layer.

The PPSP-TP messaging layer deals with the underlying transport protocol and the asynchronous nature of the interactions between tracker and peers.

The PPSP-TP request/response layer deals with the interactions between tracker and peers using Method and Response codes (see Figure 2).

The transport layer is responsible for the actual transmission of requests and responses over network transports, including the determination of the connection to use for a request or response when using a connection-oriented transport like TCP, or TLS [RFC5246] over it.

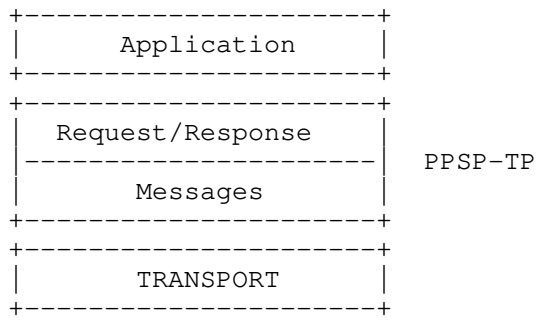


Figure 2: Abstract layering of PPSP-TP

The functional entities involved in the PPSP Tracker Protocol are trackers and peers (which may support different capabilities).

Peers correspond to devices that actually participate in sharing a media content and are organized in (various) swarms corresponding each swarm to the group of peers streaming that content at any given time. Each peer stores chunks of the media content, called segments (or subsegments), and contacts the tracker to advertise which information it has available. When a peer wishes to obtain information about the swarm, it contacts the tracker to find other

peers participating in that specific swarm.

The tracker is a logical entity that maintains the lists of peers storing chunks for a specific Live media channel or media streaming content, answers queries from peers and collects information on the activity of peers. While a tracker may have an underlying implementation consisting of more than one physical node, logically the tracker can most simply be thought of as a single element, and in this document, it will be treated as a single logical entity.

The Tracker Protocol is not used to exchange actual content data (either VoD or Live streaming) with peers, but information about which peers can provide which pieces of content.

When a peer wants to receive streaming of a selected content:

1. Peer connects to a local connection tracker and joins a swarm.
2. Peer acquires a list of peers from the connection tracker.
3. Peer exchanges its content availability with the peers on the obtained peer list.
4. Peer identifies the peers with desired content.
5. Peer requests for the content from the identified peers.

When a peer wants to share streaming of certain content with other peers:

1. Peer connects to the connection tracker.
2. Peer sends information to the connection tracker about the swarm it belongs to (joins), plus streaming status and/or content availability.

A P2P streaming process is summarized in Figure 3.

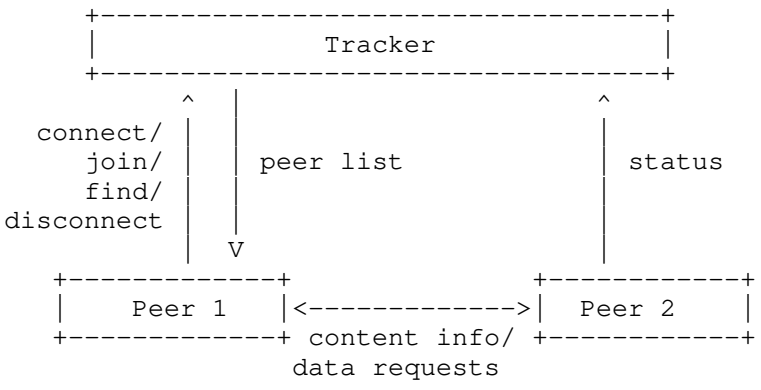


Figure 3: A PPSP streaming process

4. Messaging Model

The messaging model of PPSP-TP is based on the exchange of messages that follow the syntax and semantics of the current HTTP/1.1 specification [RFC2616]. The exchange of messages is envisioned to be performed over a stream-oriented reliable transport protocol, like TCP.

PPSP-TP is a text-based protocol, uses the UTF-8 character set [RFC3629] and the protocol messages are either requests from client peers to a tracker server, or responses from a tracker server to client peers.

5. Request/Response model

PPSP-TP request and response semantics are carried as entities (header and body) in PPSP-TP messages which correspond to either HTTP/1.1 request methods or HTTP/1.1 response codes, respectively.

Requests are sent, and responses returned to these requests. A single request generates a single response (neglecting fragmentation of messages in transport).

The response codes are consistent with HTTP/1.1 response codes, however, not all HTTP/1.1 response codes are used for the PPSP-TP (section 7).

The Request Messages of the protocol, are listed in Table 1:

PPSP Tracker Req. Messages
CONNECT
DISCONNECT
JOIN
FIND
STAT_REPORT

Table 1: Request Messages

CONNECT: This request message is used when a peer registers in the tracker. The tracker records the Peer-ID, connect-time (referenced to the absolute time), peer IP addresses and link status.

DISCONNECT: This request message is used when the peer intends to no longer participate in a specific swarm, or in all swarms. The

tracker deletes the corresponding activity records related to the peer (including its status and all content status for the corresponding swarms).

JOIN: This request message is used for a peer to notify the tracker that it wishes to participate in a swarm. The tracker records the content availability, i.e., adds the peer to the peers list for the swarm. On receiving a JOIN message, the tracker first checks the PeerMode type and then decides the next step (more details are referred in section 8.3).

FIND: This request message allows a peer to request to the tracker the peer list for a specific content representation or specific chunks of a media component in a swarm, before it can request the content from the peers. On receiving a FIND message, the tracker finds the peers, listed in content status of the specified swarm, that can satisfy the requesting peer's requirements, returning the list to the requesting peer. To create the peer list, the tracker may take peer status, capabilities and peers priority into consideration. Peer priority may be determined by network topology preference, operator policy preference, etc.

STAT_REPORT: This request message allows the exchange of statistic and status data between an active peer and a tracker to improve system performance. This request message is sent periodically to the tracker.

6. The Tracker State Machine

The state machine for the tracker is very simple, as shown in Figure 4.
Peer-ID registrations represent a dynamic piece of state maintained by the network.

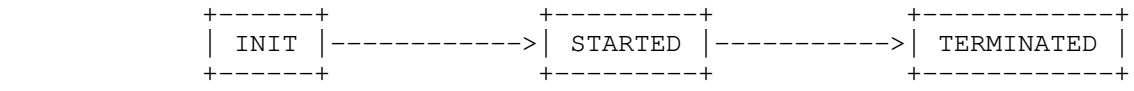


Figure 4: Tracker State Machine

When there are no peers registered in the tracker, the state machine is in the INIT state. When the first peer is registered with its Peer-ID, the state machine moves from INIT to STARTED.

As long as there is at least one active registration of a Peer-ID, the state machine remains in the STARTED state. When the last Peer-ID is removed, the state machine transitions to TERMINATED. From there, it immediately transitions back to the INIT state. Because of

that, the TERMINATED state here is transient.

In addition to this state machine, each registered Peer-ID is modeled with its own transaction state machine (Figure 5), instantiated per Peer-ID registered in the tracker, and deleted when it is removed. Unlike the state machine for the Peer-ID registration, which exists even when no Peer-IDs are registered, the per-Peer-ID transaction state machine is instantiated when the Peer-ID is registered, and deleted when the Peer-ID is removed.

This allows for an implementation optimization whereby the tracker can destroy the objects associated with the per-Peer-ID transaction state machine once it enters the TERMINATE state (Figure 5).

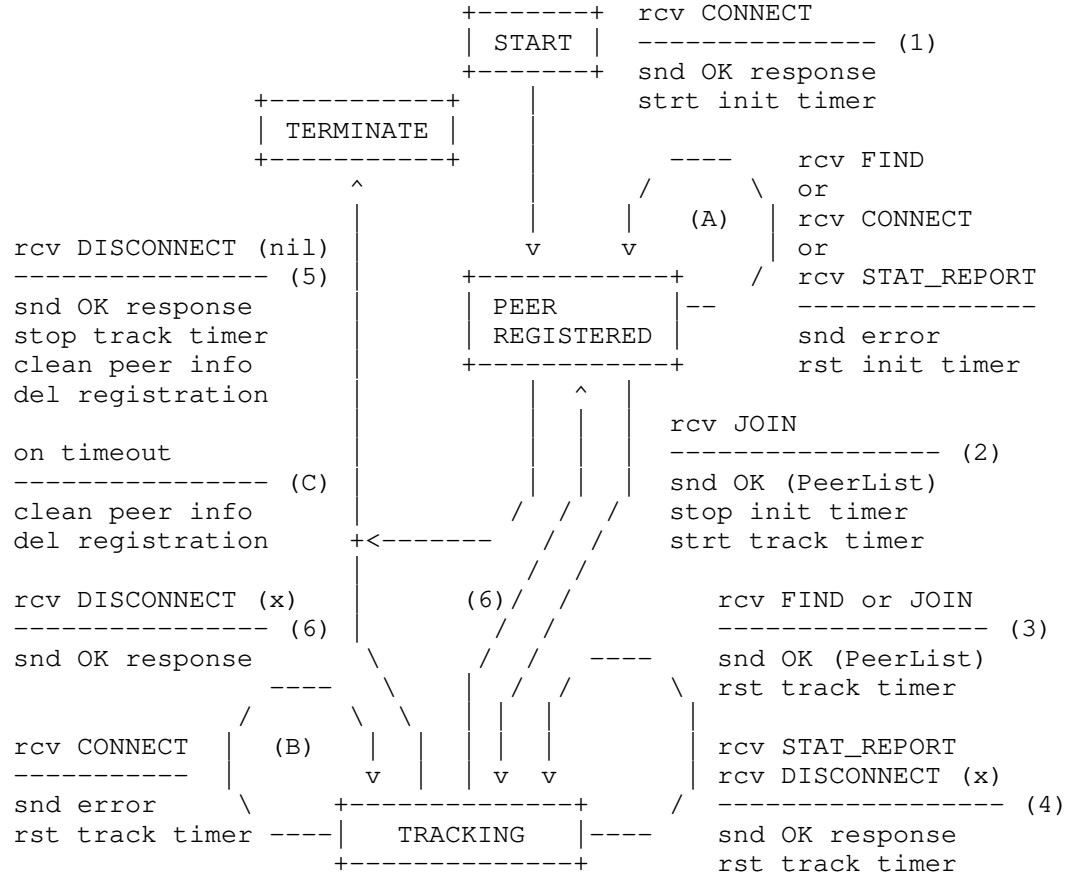


Figure 5: Per-Peer-ID Transaction State Machine

When a new Peer-ID is added, the per-Peer-ID state machine for it is

instantiated, and it moves into the PEER REGISTERED state. Because of that, the START state here is transient.

When the Peer-ID is no longer bound to a registration, the per-Peer-ID state machine moves to the TERMINATE state, and the state machine is destroyed.

During the life time of streaming activity of a peer, the per-Peer-ID transaction state machine progresses from one state to another in response to various events. The events that may potentially advance the state include:

- o Reception of CONNECT, JOIN, FIND, DISCONNECT and STAT_REPORT messages, or
- o Timeout events.

The state diagram in Figure 5 illustrates state changes, together with the causing events and resulting actions. Specific error conditions are not shown in the state diagram.

6.1. Normal Operation

On normal operation the process consists of the following steps:

1) When a CONNECT message is received from a peer, if successfully authenticated and validated, the tracker registers the Peer-ID and associated information (IP addresses), sends the response of successful registration to peer and starts the "init timer" waiting for a new message from the peer.

2) While PEER REGISTERED, when a JOIN message is received with valid swarm information, the tracker stops the "init timer", starts the "track timer" and sends the response of successful join to the peer. The response MAY contain the appropriate list of peers in the swarm, depending on PeerMode (section 8.3). A successful first JOIN starts the TRACKING state associated with the peer-ID for the requested swarm.

3) While TRACKING, a JOIN or FIND message received with valid swarm information from the peer resets the "track timer" and is responded with a successful condition, either for the JOIN to (an additional) swarm or for including the appropriate list of peers for the scope in the FIND request.

4) While TRACKING, a DISCONNECT(x) message received from the peer, containing a valid x=Swarm-ID resets the "track timer" and is responded with a successful condition. The tracker cleans the information associated with the participation of the Peer-ID in

the specified swarm(s).

In TRACKING state a STAT_REPORT message received from the peer resets the "track timer" and is responded with a successful condition. The STAT_REPORT message MAY contain information related with Swarm-IDs to which the peer is joined.

5) From either PEER REGISTERED or TRACKING states a DISCONNECT(x) message received from the peer, where x=nil, the tracker stops the "track timer", cleans the information associated with the participation of the Peer-ID in the the swarm(s) joined, responds with a successful condition, deletes the registration of the Peer-ID and transitions to TERMINATED state for that Peer-ID.

6) From TRACKING state a DISCONNECT(x) message received from the peer, where x=ALL or x=Swarm-ID is the last swarm, the tracker stops the "track timer", cleans the information associated with the participation of the Peer-ID in the the swarm(s) joined, responds with a successful condition and transitions to PEER REGISTERED state.

6.2. Error Conditions

Peers MUST NOT generate protocol elements that are invalid. However, several situations of a peer may lead to abnormal conditions in the interaction with the tracker. The situations may be related with peer malfunction or communications errors. The tracker reacts to the abnormal situations depending on its current state related to a peer-ID, as follows:

A) At the PEER REGISTERED state (while the "init timer" has not expired) receiving FIND, CONNECT or STAT_REPORT messages from the peer is considered an error condition. The tracker responds with error code 403 Forbidden (described in section 7), and resets the "init timer" one last time.

B) At the TRACKING state (while the "track timer" has not expired) receiving a CONNECT message from the peer is considered an error condition. The tracker responds with error code 403 Forbidden (described in section 7), and resets the "track timer".

NOTE: This situation may correspond to a malfunction at the peer or to malicious conditions. A preventive measure would be to reset the "track timer" one last time and if no valid message is received proceed to TERMINATE state for the Peer-ID by de-registering the peer and cleaning all peer information.

C) Without receiving messages from the peer, either from PEER

REGISTERED sate (init timer) or TRACKING state (track timer), on timeout the tracker cleans all the information associated with the Peer-ID in all swarms it was joined, deletes the registration, and transitions to TERMINATE state for that Peer-ID. The same action is taken if no valid message is received at the PEER REGISTERED state after the last "init timer" expires.

7. Protocol Specification

7.1. Messages Syntax

PPSP-TP messages use the generic message format of RFC 5322 [RFC5322] for transferring the payload of the message (Requests and Responses).

PPSP-TP messages consist of a start-line, one or more header fields, an empty line indicating the end of the header fields, and, when applicable, a message-body.

The start-line, each message-header line, and the empty line MUST be terminated by a carriage-return line-feed sequence (CRLF). Note that the empty line MUST be present even if the message-body is not.

The PPSP-TP message and header field syntax is identical to HTTP/1.1 [RFC2616].

A Request message is a standard HTTP/1.1 message starting with a Request-Line generated by the HTTP client peer. The Request-Line contains a method name, a Request-URI, and the protocol version separated by a single space (SP) character.

```
Request-Line =
    Method SP Request-URI SP HTTP-Version CRLF
```

A Request message example is the following:

```
<Method> /<Resource> HTTP/1.1
Host: <Host>
Content-Lenght: <ContentLenght>
Content-Type: <ContentType>
Authorization: <AuthToken>

[Request_Body]
```

The HTTP Method token and Request-URI (the Resource) identifies the resource upon which to apply the operation requested.

The Response message is also a standard HTTP/1.1 message starting with a Status-Line generated by the tracker. The Status-Line consists of the protocol version followed by a numeric Status-Code and its associated Reason-Phrase, with each element separated by a single SP character.

Status-Line =
HTTP-Version SP Status-Code SP Reason-Phrase CRLF

A Response message example is the following:

```
HTTP/1.1 <Status-Code> <Reason-Phrase>
Content-Lenght: <ContentLenght>
Content-Type: <ContentType>
Content-Encoding: <ContentCoding>

[Response_Body]
```

The Status-Code element is a 3-digit integer result code that indicates the outcome of an attempt to understand and satisfy a request.

The Reason-Phrase element is intended to give a short textual description of the Status-Code.

7.1.1. Header Fields

The header fields are identical to HTTP/1.1 header fields in both syntax and semantics.

Some header fields only make sense in requests or responses. If a header field appears in a message not matching its category (such as a request header field in a response), it MUST be ignored.

The Host request-header field in the request message follows the standard rules for the HTTP/1.1 Host header.

The Content-Type entity-header field MUST be used in requests and responses containing message-bodies to define the Internet media type of the message-body.

The Content-Encoding entity-header field MAY be used in response messages with "gzip" compression scheme [RFC2616] for faster transmission times and less network bandwidth usage.

The Content-Length entity-header field MUST be used in messages containing message-bodies to locate the end of each message in a stream.

The Authorization header field in the request message allows a peer to authenticate itself with a tracker, containing authentication information.

7.1.2. Methods

PPSP-TP uses HTTP/1.1 POST method token for all request messages.

7.1.3. Message Bodies

PPSP-TP requests MUST contain message-bodies.

PPSP-TP responses MAY include a message-body.

If the message-body has undergone any encoding such as compression, then this MUST be indicated by the Content-Encoding header field; otherwise, Content-Encoding MUST be omitted.

If applicable, the character set of the message body is indicated as part of the Content-Type header-field, and the default value for PPSP-TP messages is "UTF-8".

7.1.4. Message Response Codes

The response codes in PPSP-TP response messages are consistent with HTTP/1.1 response status-codes. However, not all HTTP/1.1 response status-codes are appropriate for PPSP-TP, and only those that are appropriate are given here. Other HTTP/1.1 response codes SHOULD NOT be used in PPSP-TP.

The class of the response is defined by the first digit of the Status-Code. The last two digits do not have any categorization role. For this reason, any response with a Status-Code between 200 and 299 is referred to as a "2xx response", and similarly to the other supported classes:

2xx: Success -- the action was successfully received, understood, and accepted;
4xx: Peer Error -- the request contains bad syntax or cannot be fulfilled at this tracker;
5xx: Tracker Error -- the tracker failed to fulfill an apparently valid request;

The valid response codes are the following (Status-Code Reason-Phrase):

200 OK -- The request has succeeded. The information returned with the response describes or contains the result of the action;

400 Bad Request -- The request could not be understood due to malformed syntax.

401 Unauthorized -- The request requires authentication.

403 Forbidden -- The tracker understood the request, but is refusing to fulfill it. The request SHOULD NOT be repeated.

404 Not Found -- This status is returned if the tracker did not find anything matching the Request-URI.

408 Request Timeout -- The peer did not produce a request within the time that the tracker was prepared to wait.

411 Length Required -- The tracker refuses to accept the request without a defined Content-Length. The peer MAY repeat the request if it adds a valid Content-Length header field containing the length of the message-body in the request message.

414 Request-URI Too Long -- The tracker is refusing to service the request because the Request-URI is longer than the tracker is willing to interpret. This rare condition is likely to occur when the tracker is under attack by a client attempting to exploit security holes.

500 Internal Server Error -- The tracker encountered an unexpected condition which prevented it from fulfilling the request.

503 Service Unavailable -- The tracker is currently unable to handle the request due to a temporary overloading or maintenance condition.

7.2. Request/Response Syntax and Format

The message-body for Requests and Responses requiring it, is encoded in XML.

The XML message-body MUST begin with an XML declaration line specifying the version of XML being used and indicating the character encoding, that SHOULD be "UTF-8". The root element MUST be PPSPTrackerProtocol.

The generic format of a Request is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
    <PPSPTrackerProtocol version="1.0">
        <Request></Request>
        <TransactionID></TransactionID>
        <PeerID></PeerID>
        <SwarmID></SwarmID>
        <PeerNum></PeerNum>
        <PeerMode></PeerMode>
        <PeerGroup></PeerGroup>
        <ContentGroup></ContentGroup>
        <StatisticsGroup></StatisticsGroup>
    </PPSPTrackerProtocol>
```

The generic format of a Response is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
    <PPSPTrackerProtocol version="1.0">
        <Response></Response>
        <TransactionID></TransactionID>
        <SwarmID></SwarmID>
        <PeerGroup></PeerGroup>
    </PPSPTrackerProtocol>
```

The Request element MUST be present in requests and corresponds to the request method type for the message.

The Response element MUST be present in responses and corresponds to the response method type of the message.

The element TransactionID MUST be present in requests to uniquely identify the transaction. Responses to completed transactions use the same TransactionID as the request they correspond to.

The version of PPSP-TP being used is indicated by the attribute @version of the root element.

All Request messages MUST contain a PeerID element to uniquely identify the peer (Peer-ID) in the network.

The PeerID information may be present on the following levels:

- On PPSPTrackerProtocol level in PPSPTrackerProtocol.PeerID element. For details refer to 7.2.1 Table 2.

- On PeerGroup level in PeerGroup.PeerInfo.PeerID element. For details refer to 7.2.1 Table 3.

The SwarmID element MUST be be present in JOIN, FIND and DISCONNECT requests. The SwarmID element MUST be present in JOIN and FIND responses. Details of usage in 8.2, 8.3 and 8.4.

The SwarmID information may be present on the following levels:

- On PPSPTrackerProtocol level in PPSPTrackerProtocol.SwarmID element. For details refer to 7.2.1 Table 2.

- On StatisticsGroup level in StatisticsGroup.Stat.SwarmID element. For details refer to 7.2.1 Table 5.

The PeerMode element MUST be present in JOIN requests. Details of usage in 8.3.

The PeerMode information may be present on the following levels:

- On PPSPTrackerProtocol level in PPSPTrackerProtocol.PeerMode element. For details refer to 7.2.1 Table 2.

- On PeerGroup level in PeerGroup.PeerMode element. For details refer to 7.2.1 Table 5.

The PeerNum element MUST be present in JOIN requests and MAY contain the attribute @abilityNAT to inform the tracker on the preferred type of peers, in what concerns their NAT traversal situation, to be returned in a peer list. Details of usage in 8.2, 8.3 and 8.4.

The PeerGroup element MUST be present in CONNECT requests and responses and MAY be present in responses to JOIN and FIND requests if the corresponding response returns information about peers. Details of usage in 8.1, 8.3 and 8.4.

The ContentGroup element MAY be present in requests referencing content, i.e., JOIN and FIND, if the request includes a content scope. Details of usage in 8.3 and 8.4.

The StatisticsGroup element MAY be present in STAT_REPORT requests. Details of usage in 8.5.

The semantics of the attributes and elements within a PPSPTrackerProtocol root element is described in subsection 7.2.1.

Request and Response processing is provided in section 8 for each message.

The XML-syntax of the of PPSP-TP XML elements for Requests and Responses is provided in the XML-Schema of Appendix A.

7.2.1. Semantics of PPSPTrackerProtocol elements

The semantics of PPSPTrackerProtocol elements and attributes are described in the following tables.

Element Name or Attribute Name	Use	Description
PPSPTrackerProtocol	1	The root element.
@version	M	Provides the version of PPSP-TP.
Request	0...1	Provides the request method and MUST be present in Request.
Response	0...1	Provides the response method and MUST be present in Response.
PeerID	0...1	Peer Identification. MUST be present in Request.
SwarmID	0...1	Swarm Identification. Details in 8.2/8.3/8.4/8.5.
PeerMode	0...1	Mode of Peer participation in a swarm, which can be "LEECH" or "SEED". Details in 8.3/8.4.
PeerNUM	0...1	Maximum peers to be received in with capabilities indicated.
@abilityNAT	CM	Type of NAT traversal peers, as "NoNAT", "STUN","TURN" or "PROXY"
@concurrentLinks	CM	Concurrent connectivity level of peers, "HIGH", "LOW" or "NORMAL"
@onlineTime	CM	Availability or online duration of peers, "HIGH" or "NORNMAL"
@uploadBWlevel	CM	Upload bandwidth capability of peers, "HIGH" or "NORMAL"
PeerGroup	0...1	Provides information on peers. More details in Table 3
ContentGroup	0...1	Provides information on content. More details in Table 4
StatisticsGroup	0...1	Provides statistic data of peer and content. Details in Table 5

Legend:
Use for attributes: M=Mandatory, OP=Optional,
CM=Conditionally Mandatory
Use for elements: minOccurs...maxOccurs (N=unbounded)
Elements are represented by their name (case-sensitive)
Attribute names (case-sensitive) are preceded with an @

Table 2: Semantics of PPSPTrackerProtocol.

Element Name or Attribute Name	Use	Description
PeerGroup	0...1	Contains description of peers.
PeerInfo	1...N	Provides information on a peer.
PeerID	0...1	Peer Identification. MAY be present in JOIN and FIND responses. Details in 8.3/8.4.
PeerMode	0...1	Mode of Peer participation in a swarm, which can be "LEECH" or "SEED". MAY be present in JOIN and FIND responses. Details in 8.3/8.4.
PeerAddress	1...N	IP Address information.
@addrType	M	Type of IP address, which can be "ipv4" or "ipv6"
@priority	CM	The priority of this interface. Used for NAT traversal.
@type	CM	Describes the address for NAT traversal, which can be "HOST" "REFLEXIVE" or "PROXY".
@connection	OP	Access type ("3G", "ADSL", etc.)
@asn	OP	Autonomous System number.
@ip	M	IP address value.
@port	M	IP service port value.

Legend:
Use for attributes: M=Mandatory, OP=Optional,
CM=Conditionally Mandatory
Use for elements: minOccurs...maxOccurs (N=unbounded)
Elements are represented by their name (case-sensitive)
Attribute names (case-sensitive) are preceded with an @

Table 3: Semantics of PeerGroup.

If STUN-like functions are enabled in the tracker and a PPSP-ICE method is used, as described in [I-D.li-ppsp-nat-traversal-02], the attributes @type and @priority MUST be returned with the transport address candidates in responses to CONNECT, JOIN or FIND requests.

The @asn attribute MAY be used to inform about the network location, in terms of Autonomous System, for each of the active public network interfaces of the peer.

The @connection attribute is informative on the type of access network of the respective interface.

Element Name or Attribute Name	Use	Description
ContentGroup	0...1	Provides information on content.
Representation	1...N	Describes a component of content.
@id	M	Unique identifier for this Representation.
SegmentInfo	1	Provides segment information.
@startIndex	M	The index of the first media segment in the request scope for this Representation.
@endIndex	OP	The index of the last media segment in the request scope for this Representation.

Legend:
Use for attributes: M=Mandatory, OP=Optional,
CM=Conditionally Mandatory
Use for elements: minOccurs...maxOccurs (N=unbounded)
Elements are represented by their name (case-sensitive)
Attribute names (case-sensitive) are preceded with an @

Table 4: Semantics of ContentGroup.

The Representation element describes a component of a content identified by its attribute @id in the MPD. This element MAY be present for each component desired in the scope of the JOIN or FIND request. The scope of each Representation is indicated in the SegmentInfo element by the attribute @startIndex and, optionally, @endIndex.

The peer may use this information in JOIN or FIND requests, for example, to join a swarm starting from a specific point (as is the case of a live program, by specifying the adequate @startIndex) and/or find adequate peers in the swarm for that content scope.

An example of on-demand usage is the case of an end-user that previously watched a content with a certain audio language, then interrupted for a while (having disconnected) and later continued by re-joining from that point onwards but selecting a different available audio language. In this case the JOIN request would specify the required Representations and the @startIndex for each, i.e., all the adequate video components and the selected audio component. An example is illustrated in subsection 8.3.

Element Name or Attribute Name	Use	Description
StatisticsGroup	0...1	Provides statistic data on peer and content.
Stat	1...N	Groups statistics property data.
@property	M	The property to be reported. Property values in Table 6.
SwarmID	0...1	Swarm Identification.
UploadedBytes	0...1	Bytes sent to swarm.
DownloadedBytes	0...1	Bytes received from swarm.
AvailBandwidth	0...1	Upstream Bandwidth available.
Representation	0...N	Describes a component of content.
@id	CM	Unique identifier for this Representation.
SegmentInfo	1...N	Provides segment information by segment range. The chunkmap can be encoded in Base64 [RFC4648].
@startIndex	CM	The index of the first media segment in the chunkmap report for this Representation.
@endIndex	CM	The index of the last media segment in the chunkmap report for this Representation.
@chunkmapSize	CM	Size of chunkmap reported.

Legend:
Use for attributes: M=Mandatory, OP=Optional,
CM=Conditionally Mandatory
Use for elements: minOccurs...maxOccurs (N=unbounded)
Elements are represented by their name (case-sensitive)
Attribute names (case-sensitive) are preceded with an @

Table 5: Semantics of StatisticsGroup.

The Stat element is used to describe several properties relevant to the P2P network. These properties can be related with stream statistics, peer status information and content data information, like chunkmaps. Each Stat element will correspond to a @property type and several Stat blocks can be reported in a single STAT_REPORT message, corresponding to some or all the swarms the peer is actively involved.

Other properties may be defined, related, for example, with incentives and reputation mechanisms, like peer online time, or connectivity conditions, like physical link status, etc.

For that purpose, the Stat element may be extended to provide additional scheme specific information for new @property groups, new elements and new attributes.

@property	Description
StreamStatistics	Stream statistic values per SwarmID
ContentMap	Reports map of chunks the peer has per Representation of the content

Table 6: StatisticsGroup default Stat @property values.

An example of a STAT_REPORT for multiple properties is illustrated in subsection 8.5.

7.2.2. Request element in request Messages

Table 7 defines the valid string representations for the requests. These values MUST be treated as case-sensitive.

XML Request Methods String Values
CONNECT
DISCONNECT
JOIN
FIND
STAT_REPORT

Table 7: Valid Strings for Request element of requests.

7.2.3. Response element in response Messages

Table 8 defines the valid string representations for Response messages that require message-body. These values MUST be treated as case-sensitive.

Response messages not requiring message-body only use the standard HTTP/1.1 Status-Code and Reason-Phrase (appended, if appropriate, with detail phrase, as described in section 8.6).

XML Response Method String Values	HTTP Status-Code and Reason-Phrase
SUCCESSFUL	200 OK
AUTHENTICATION REQUIRED	401 Unauthorized

Table 8: Valid Strings for Response element of responses.

SUCCESSFUL: indicates that the request has been processed properly and the desired operation has completed. The body of the response message includes the requested information and MUST include the same TransactionID of the corresponding request.

CONNECT: returns information about the successful registration of the peer.

DISCONNECT and STAT_REPORT: confirms the success of the requested operation.

JOIN and FIND: MAY return the list of peers meeting the desired criteria.

AUTHENTICATION REQUIRED: Authentication is required for the peer to make the request.

8. Request/Response Processing

When a PPSP-TP message is received some basic processing is performed, regardless of the message type.

Upon reception, a message is examined to ensure that it is properly formed. The receiver MUST check that the HTTP message itself is properly formed, and if not, appropriate standard HTTP errors MUST be generated. The receiver must also verify that the XML body is properly formed. In case of error due to malformed messages appropriate responses MUST be returned, as described in 8.6.

8.1. CONNECT Request

This method is used when a peer registers to the system. The tracker records the Peer-ID, connect-time, IP addresses and link status.

The peer MUST properly form the XML message-body, set the Request method to CONNECT, generate and set the TransactionID, and set the PeerID with the identifier of the peer. The peer SHOULD also include

the IP addresses of its network interfaces in the CONNECT message.

An example of the message-body of a CONNECT Request is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol version="1.0">
  <Request>CONNECT</Request>
  <PeerID>656164657221</PeerID>
  <TransactionID>12345</TransactionID>
  <PeerGroup>
    <PeerInfo>
      <PeerAddress addrType="ipv4" ip="192.0.2.1" port="80"
                   priority="1" />
      <PeerAddress addrType="ipv6" ip="2001:db8::1" port="80"
                   priority="2"
                   type="HOST"
                   connection="3G" />
    </PeerInfo>
  </PeerGroup>
</PPSPTrackerProtocol>
```

When receiving a well-formed CONNECT Request message, the tracker will first processes the peer authentication information (provided as Authorization scheme and token in the HTTP message) to check whether it is valid and that it can connect to the service, and then proceed to register the peer in the service. In case of success a Response message with a corresponding response value of SUCCESSFULL will be generated.

The element PeerInfo MAY contain multiple PeerAddress child elements with attributes @addrType, @ip, and @port, and optionally @priority and @type (if PPSP-ICE NAT traversal techniques are used) corresponding to each of the network interfaces of the peer.

If STUN-like function is enabled in the tracker, the response MAY include the peer reflexive address [I-D.li-ppsp-nat-traversal-02].

The response MUST have the same TransactionID value as the request.

An example of a Response message for the CONNECT Request is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol version="1.0">
  <Response>SUCCESSFUL</Response>
  <TransactionID>12345</TransactionID>
  <PeerGroup>
    <PeerInfo>
      <PeerAddress addrType="ipv4" ip="198.51.100.1" port="80"
                   priority="1"
                   type="REFLEXIVE"
                   connection="ADSL"
                   asn="64496" />
    </PeerInfo>
  </PeerGroup>
</PPSPTrackerProtocol>
```

The Response MUST include a PeerGroup with PeerInfo data that includes the peer public IP address. If STUN-like function is enabled in the tracker, the PeerAddress includes the attribute @type with a value of REFLEXIVE, corresponding to the transport address "candidate" of the peer.

The tracker MAY also include the attribute @asn with network location information of the transport address, corresponding to the Autonomous System Number of the access network provider.

8.2. DISCONNECT Request

This method is used when the peer intends to leave a specific swarm, or the system, and no longer participate.

The tracker SHOULD delete the corresponding activity records related with the peer in the corresponding swarms (including its status and all content status).

The peer MUST properly form the XML message-body, set the Request method to DISCONNECT, set the PeerID with the identifier of the peer, randomly generate and set the TransactionID and include the SwarmID information.

The SwarmID value MUST be either a specific Swarm-ID the peer had previously joined, the value "ALL" to designate all joined swarms, or the value "nil" to completely disconnect from the system.

An example of the message-body of a DISCONNECT Request is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol version="1.0">
  <Request>DISCONNECT</Request>
  <PeerID>656164657221</PeerID>
  <SwarmID>ALL</SwarmID>
  <TransactionID>12345</TransactionID>
</PPSPTrackerProtocol>
```

In case of success a Response message with a corresponding response value of SUCCESSFULL will be generated. The response MUST have the same TransactionID value as the request.

Upon receiving a DISCONNECT message, the tracker cleans the information associated with the participation of the Peer-ID in the specified swarm (or in all swarms).

An example of a Response message for the DISCONNECT Request is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol version="1.0">
  <Response>SUCCESSFUL</Response>
  <TransactionID>12345</TransactionID>
</PPSPTrackerProtocol>
```

If the scope of SwarmID in the DISCONNECT request is "nil" the tracker will also delete the registration of the Peer-ID.

8.3. JOIN Request

This method is used for peers to notify the tracker that they wish to participate in a particular swarm.

The JOIN message is used when the peer has none or just some chunks (LEECH), or has all the chunks (SEED) of a content. The JOIN is used for both on-demand or Live streaming modes.

The peer MUST properly form the XML message-body, set the Request method to JOIN, set the PeerID with the identifier of the peer, set the SwarmID with the identifier of the swarm it is interested in, and randomly generate and set the TransactionID.

An example of the message-body of a JOIN Request is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol version="1.0">
  <Request>JOIN</Request>
  <PeerID>656164657221</PeerID>
  <SwarmID>1111</SwarmID>
  <TransactionID>12345</TransactionID>
  <PeerNum abilityNAT="STUN"
           concurrentLinks="HIGH"
           onlineTime="NORMAL"
           uploadBWlevel="NORMAL">5</PeerNum>
  <PeerMode>LEECH</PeerMode>
  <ContentGroup>
    <Representation id="tag0">
      <SegmentInfo startIndex="20" />
    </Representation>
    <Representation id="tag6">
      <SegmentInfo startIndex="20" />
    </Representation>
  </ContentGroup>
</PPSPTrackerProtocol>
```

The JOIN request MAY include a PeerNum element to indicate to the tracker the number of peers to be returned in a list corresponding to the indicated properties, being @abilityNAT for NAT traversal (considering that PPSP-ICE NAT traversal techniques may be used), and optionally @concurrentLinks, @onlineTime and @uploadBWlevel for the preferred capabilities.

The PeerMode element SHOULD be set to the type of participation of the peer in the swarm (SEED or LEECH).

In the case of a JOIN to a specific point in a stream the request SHOULD include a ContentGroup to specify the joining point in terms of content Representations. The above example of a JOIN request would be for the case of an end-user that previously watched a content with a certain audio language, then interrupted for a while (having disconnected) and later continued by re-joining from that point onwards but selecting a different available audio language (Representation with @id="tag6" in the MPD of Appendix B).

When receiving a well-formed JOIN Request the tracker processes the information to check if it is valid and if the peer can join the swarm of interest. In case of success a response message with a Response value of SUCCESSFULL will be generated and the tracker enters the peer information into the corresponding swarm activity.

In case the PeerMode is SEED, the tracker just responds with a SUCCESSFUL response and enters the peer information into the corresponding swarm activity.

In case the PeerMode is LEECH the tracker will search and select an appropriate list of peers satisfying the conditions requested. The peer list MUST contain the Peer-IDs and the corresponding IP Addresses. To create the peer list, the tracker may take peer status and network location information into consideration, to express network topology preferences or Operators' policy preferences, with regard to the possibility of connecting with other IETF efforts such as ALTO [I.D.ietf-alto-protocol].

The response MUST have the same TransactionID value as the request.

An example of a Response message for the JOIN Request is:

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol version="1.0">
  <Response>SUCCESSFUL</Response>
  <TransactionID>12345</TransactionID>
  <PeerGroup>
    <PeerInfo>
      <PeerID>956264622298</PeerID>
      <PeerAddress addrType="ipv4" ip="198.51.100.22" port="80"
                   asn="64496" />
    </PeerInfo>
    <PeerInfo>
      <PeerID>3332001256741</PeerID>
      <PeerAddress addrType="ipv4" ip="198.51.100.201" port="80"
                   asn="64496" />
    </PeerInfo>
  </PeerGroup>
</PPSPTrackerProtocol>
```

The Response MUST include a PeerGroup with PeerInfo data that includes the public IP address of the selected active peers in the swarm.

The tracker MAY also include the attribute @asn with network location information of the transport addresses of the peers, corresponding to the Autonomous System Numbers of the access network provider of each peer in the list.

8.4. FIND Request

This method allows peers to request to the tracker, whenever needed and after being joined to a swarm, a new peer list for the swarm or

for specific scope of chunks of a media content Representation of that swarm.

The peer MUST properly form the XML message-body, set the Request method to FIND, set the PeerID with the identifier of the peer, set the SwarmID with the identifier of the swarm the peer is interested, and optionally, in order to find peers having the specific chunks, include information about the content.

The peer MUST also generate and set the TransactionID for the request.

An example of the message-body of a FIND Request is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol version="1.0">
  <Request>FIND</Request>
  <PeerID>656164657221</PeerID>
  <SwarmID>1111</SwarmID>
  <TransactionID>12345</TransactionID>
  <PeerNum abilityNAT="STUN"
           concurrentLinks="HIGH"
           onlineTime="NORMAL"
           uploadBWlevel="NORMAL">5</PeerNum>
  <ContentGroup>
    <Representation id="tag4">
      <SegmentInfo startIndex="110" endIndex="150" />
    </Representation>
  </ContentGroup>
</PPSPTrackerProtocol>
```

The FIND request MAY include a PeerNum element to indicate to the tracker the number of peers to be returned in a list corresponding to the indicated properties, being @abilityNAT for NAT traversal (considering that PPSP-ICE NAT traversal techniques may be used), and optionally @concurrentLinks, @onlineTime and @uploadBWlevel for the preferred capabilities.

In the case of a FIND with a specific scope of a stream content the request SHOULD include a ContentGroup to specify the content Representations segment range of interest.

When receiving a well-formed FIND Request the tracker processes the information to check if it is valid. In case of success a response message with a Response value of SUCCESSFULL will be generated and the tracker will include the appropriate list of peers satisfying the conditions requested. The peer list returned MUST contain the Peer-IDs and the corresponding IP Addresses.

The tracker may take peer status and network location information into consideration when selecting the peer list to return, to express network topology preferences or Operators' policy preferences, with regard to the possibility of connecting with other IETF efforts such as ALTO [I.D.ietf-alto-protocol].

The response MUST have the same TransactionID value as the request.

An example of a Response message for the FIND Request is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol version="1.0">
  <Response>SUCCESSFUL</Response>
  <TransactionID>12345</TransactionID>
  <PeerGroup>
    <PeerInfo>
      <PeerID>956264622298</PeerID>
      <PeerAddress addrType="ipv4" ip="198.51.100.22" port="80"
                   asn="64496" />
    </PeerInfo>
    <PeerInfo>
      <PeerID>3332001256741</PeerID>
      <PeerAddress addrType="ipv4" ip="198.51.100.201" port="80"
                   asn="64496" />
    </PeerInfo>
  </PeerGroup>
</PPSPTrackerProtocol>
```

The Response MUST include a PeerGroup with PeerInfo data that includes the public IP address of the selected active peers in the swarm.

The tracker MAY also include the attribute @asn with network location information of the transport addresses of the peers, corresponding to the Autonomous System Numbers of the access network provider of each peer in the list.

8.5. STAT_REPORT Request

This method allows the exchange of statistic and status data between peers and trackers to improve system performance. The method is initiated by the peer, periodically while active.

The peer MUST properly form the XML message-body, set the Request method to STAT_REPORT, set the PeerID with the identifier of the peer, and generate and set the TransactionID.

The report MAY include a StatisticsGroup containing multiple Stat elements describing several properties relevant to the P2P network. These properties can be related with stream statistics, peer status information and content data information, like chunkmaps.

Other properties may be defined, related for example, with incentives and reputation mechanisms.

In case no StatisticsGroup is included, the STAT_REPORT may be used as a "keep-alive" message, to prevent the Tracker from de-registering the peer when timer expired.

An example of the message-body of a STAT_REPORT Request is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol version="1.0">
  <Request>STAT_REPORT</Request>
  <PeerID>656164657221</PeerID>
  <TransactionID>12345</TransactionID>
  <StatisticsGroup>
    <Stat property="StreamStatistics">
      <SwarmID>1111</SwarmID>
      <UploadedBytes>512</UploadedBytes>
      <DownloadedBytes>768</DownloadedBytes>
      <AvailBandwidth>1024000</AvailBandwidth>
    </Stat>
    <Stat property="StreamStatistics">
      <SwarmID>2222</SwarmID>
      <UploadedBytes>1024</UploadedBytes>
      <DownloadedBytes>2048</DownloadedBytes>
      <AvailBandwidth>512000</AvailBandwidth>
    </Stat>
    <Stat property="ContentMap">
      <SwarmID>1111</SwarmID>
      <Representation id="tag0">
        <SegmentInfo startIndex="0" endIndex="24"
                     chunkmapSize="25">
          A/8D/wP/A/8D/wP/A/8D/wP/A/8D/wP/....
        </SegmentInfo>
      </Representation>
      <Representation id="tag1">
        <SegmentInfo startIndex="0" endIndex="14"
                     chunkmapSize="15">
          A/8D/wP/A/8D/wP/A/8D/wP/A/8D/wP/....
        </SegmentInfo>
        <SegmentInfo startIndex="20" endIndex="24"
                     chunkmapSize="5">
```

```
            A/8D/wP/A/8D/wP/A/8D/wP/A/8D/wP/....
          </SegmentInfo>
        </Representation>
      </Stat>
      <Stat property="ContentMap">
        <SwarmID>2222</SwarmID>
        <Representation id="tag5">
          <SegmentInfo startIndex="0" endIndex="4"
                     chunkmapSize="5">
            A/8D/wP/A/8D/wP/A/8D/wP/A/8D/wP/....
          </SegmentInfo>
        </Representation>
        <Representation id="tag6">
          <SegmentInfo startIndex="0" endIndex="4"
                      chunkmapSize="5">
            A/8D/wP/A/8D/wP/A/8D/wP/A/8D/wP/....
          </SegmentInfo>
        </Representation>
      </Stat>
    </StatisticsGroup>
  </PPSPTrackerProtocol>
```

If the request is valid the tracker process the received information for future use, and generates a response message with a Response value of SUCCESSFULL.

The response MUST have the same TransactionID value as the request.

An example of a Response message for the START_REPORT Request is the following:

```
  <?xml version="1.0" encoding="UTF-8"?>
  <PPSPTrackerProtocol version="1.0">
    <Response>SUCCESSFUL</Response>
    <TransactionID>12345</TransactionID>
  </PPSPTrackerProtocol>
```

8.6. Error and Recovery conditions

If the peer fails to read the tracker response, the same Request with identical content, including the same TransactionID, SHOULD be repeated, if the condition is transient.

The TransactionID on a Request can be reused if and only if all of the content is identical, including eventual Date/Time information. Details of the retry process (including time intervals to pause, number of retries to attempt, and timeouts for retrying) are implementation dependent.

The tracker SHOULD be prepared to receive a Request with a repeated TransactionID.

Error situations resulting from the Normal Operation or from abnormal conditions (section 6.2) MUST be responded with the adequate response codes, as described here:

If the message is found to be incorrectly formed, the receiver MUST respond with a 400 (Bad Request) response with an empty message-body. The Reason-Phrase SHOULD identify the syntax problem in more detail, for example, "Missing Content-Type header field".

If the version number of the protocol is for a version the receiver does not supports, the receiver MUST respond with a 400 (Bad Request) with an empty message-body. Additional information SHOULD be provided in the Reason-Phrase, for example, "PPSP Version #.#".

If the length of the message does not matches the Content-Length specified in the message header, or the message is received without a defined Content-Length, the receiver MUST respond with a 411 (Length Required) response with an empty message-body.

If the Request-URI in a Request message is longer than the tracker is willing to interpret, the tracker MUST respond with a 414 (Request-URI Too Long) response with an empty message-body.

In the PEER REGISTERED and TRACKING states of the tracker, certain requests are not allowed (section 6.2). The tracker MUST respond with a 403 (Forbidden) response with an empty message-body. The Reason-Phrase SHOULD identify the error condition in more detail, for example, "Already Connected".

If the tracker is unable to process a Request message due to unexpected condition, it SHOULD respond with a 500 (Internal Server Error) response with an empty message-body.

If the tracker is unable to process a Request message for being in an overloaded state, it SHOULD respond with a 503 (Service Unavailable) response with an empty message-body.

9. Security Considerations

P2P streaming systems are subject to attacks by malicious/unfriendly peers/trackers that may eavesdrop on signaling, forge/deny information/knowledge about streaming content and/or its availability, impersonating to be another valid participant, or launch DoS attacks to a chosen victim.

No security system can guarantees complete security in an open P2P streaming system where participants may be malicious or uncooperative. The goal of security considerations described here is to provide sufficient protection for maintaining some security properties during the tracker-peer communication even in the face of a large number of malicious peers and/or eventual distrustful trackers (under the distributed tracker deployment scenario).

Since the protocol uses HTTP to transfer signaling most of the same security considerations described in RFC 2616 also apply [RFC2616].

9.1. Authentication between Tracker and Peers

To protect the PPSP-TP signaling from attackers pretending to be valid peers (or peers other than themselves) all messages received in the tracker are required to be received from authorized peers.

For that purpose a peer must enroll in the system via a centralized enrollment server. The enrollment server is expected to provide a proper Peer-ID for the peer and information about the authentication mechanisms. The specification of the enrollment method and the provision of identifiers and authentication tokens is out of scope of this specification.

A Channel-oriented security mechanism should be used in the communication between peers and tracker, such as the Transport Layer Security (TLS) to provide privacy and data integrity.

Due to the transactional nature of the communication between peers and tracker the method for adding authentication and data security services can be the OAuth 2.0 Authorization [I-D.ietf-oauth-v2] with bearer token, which provides the peer with the information required to successfully utilize an access token to make protected requests to the tracker [I-D.ietf-oauth-v2-bearer].

9.2. Content Integrity protection against polluting peers/trackers

Malicious peers may declaim ownership of popular content to the tracker but try to serve polluted (i.e., decoy content or even virus/trojan infected contents) to other peers.

This kind of pollution can be detected by incorporating integrity verification schemes for published shared contents. As content chunks are transferred independently and concurrently, a correspondent chunk-level integrity verification MUST be used, checked with signed fingerprints received from authentic origin.

9.3. Residual attacks and mitigation

To mitigate the impact of sybil attackers, impersonating a large number of valid participants by repeatedly acquiring different peer identities, the enrollment server SHOULD carefully regulate the rate of peer/tracker admission.

There is no guarantee that peers honestly report their status to the tracker, or serve authentic content to other peers as they claim to the tracker. It is expected that a global trust mechanism, where the credit of each peer is accumulated from evaluations for previous transactions, may be taken into account by other peers when selecting partners for future transactions, helping to mitigate the impact of such malicious behaviors. A globally trusted tracker MAY also take part of the trust mechanism by collecting evaluations, computing credit values and providing them to joining peers.

9.4. Pro-incentive parameter trustfulness

Property types for STAT_REPORT messages may consider pro-incentive parameters, which can enable the tracker to improve the performance of the whole P2P streaming system.

Trustworthiness of these pro-incentive parameters is critical to the effectiveness of the incentive mechanisms. For example, ChunkMaps are essential, and need to be accurate. The P2P system should be designed in a way such that a peer will have the incentive to report truthfully its ChunkMaps (otherwise it may penalize itself, as in the case of under-reporting addressed in [prTorrent]).

Furthermore, both the amount of uploaded and downloaded data should be reported to the tracker to allow checking if there is any inconsistency between the upload and download report, and establish an appropriate credit/trust system. Alternatively, exchange of cryptographic receipts signed by receiving peers can be used to attest to the upload contribution of a peer to the swarm, as

suggested in [Contracts].

10. IANA Considerations

There are presently no IANA considerations with this document.

11. Acknowledgments

The authors would like to thank many people for for their help and comments, particularly: Zhang Yunfei, Liao Hongluan, Roni Even, Bhumip Khasnabish, Wu Yichuan, Peng Jin, Chi Jing, Zong Ning, Song Haibin, Chen Wei, Zhijia Chen, Christian Schmidt, Lars Eggert, David Harrington, Henning Schulzrinne, Kangheng Wu, Martin Stiemerling, Jianyin Zhang, Johan Pouwelse and Arno Bakker.

The authors would also like to thank the people participating in the EU FP7 project SARACEN (contract no. ICT-248474) [refs.saracenwebpage] for contributions and feedback to this document.

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 SARACEN project or the European Commission.

12. References

12.1. Normative References

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

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC5245] Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", RFC 5245, April 2010.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC5322] Resnick, P., Ed., "Internet Message Format", RFC 5322, October 2008.

[ISO.8601.2004] International Organization for Standardization, "Data elements and interchange formats - Information interchange - Representation of dates and times", ISO Standard 8601, December 2004.

12.2. Informative References

[RFC1952] Deutsch, P., "GZIP file format specification version 4.3", RFC 1952, May 1996.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

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

[I-D.ietf-ppsp-problem-statement] Zhang, Y., Zong, N., Camarillo, G., Seng, J., and Y. Yang, "Problem Statement of P2P Streaming Protocol (PPSP)", draft-ietf-ppsp-problem-statement-07

(work in progress), November 2011.

[I-D.li-ppsp-nat-traversal-02] Li, L., Wang, J., Chen, W., "PPSP NAT Traversal", draft-li-ppsp-nat-traversal-02 (work in progress), July 2011.

[I-D.xiao-ppsp-reload-distributed-tracker] Xiao, L., Bryan, D., Gu, Y., Tai, X., "A PPSP Tracker Usage for Reload", draft-xiao-ppsp-reload-distributed-tracker-03 (work in progress), October 2011.

[I.D.ietf-alto-protocol] Alimi, R., Penno, R., Yang, Y., "ALTO Protocol", draft-ietf-alto-protocol-10, (work in progress), October 2011.

[I-D.ietf-oauth-v2] Hammer-Lahav, E., Recordon, D., and D. Hardt, "The OAuth 2.0 Authorization Protocol," draft-ietf-oauth-v2-23 (work in progress), January 2012.

[I-D.ietf-oauth-v2-bearer] Jones, M., Hardt, D., and D. Recordon, "The OAuth 2.0 Authorization Protocol: Bearer Tokens," draft-ietf-oauth-v2-bearer-17 (work in progress), February 2012.

[MP4REG] MP4REG, The MPEG-4 Registration Authority, URL: <http://www.mp4ra.org>.

[ISO.IEC.23009-1] ISO/IEC, "Information technology -- Dynamic adaptive streaming over HTTP (DASH) -- Part 1: Media presentation description and segment formats", ISO/IEC DIS 23009-1, Aug. 2011.

[ITU-T.H.264] ITU-T, "Advanced video coding for generic audiovisual services," Recommendation H.264 (03/2010), International Telecommunication Union - Telecommunication Standardization Sector, Mar. 2010.

[refs.saracenwebpage] "SARACEN Project Website", http://www.saracen-p2p.eu/.

[prTorrent] Roy, S., Zeng, W., "prTorrent: On Establishment of Piece Rarity in the BitTorrent Unchoking Algorithm", in IEEE Ninth International Conference on Peer-to-Peer Computing, September 2009.

[Contracts] Piatek, M., Venkataramani, A., Yang, R., Zhang, D., Jaffe, A., "Contracts: Practical Contribution Incentives for P2P Live Streaming", in NSDI '10: USENIX Symposium on

Networked Systems Design and Implementation, April 2010.

Appendix A. PPSP Tracker Protocol XML-Schema

TO BE ADDED.

Appendix B. Media Presentation Description (MPD)

The MPD file describes a Media Presentation, i.e., a bounded or unbounded presentation of media content. In particular, it defines formats to announce resource identifiers for segments and subsegments (layers in case of SVC, descriptions in case of MDC, or views in case of 3D) and to provide the context for these identified resources within a Media Presentation, i.e., describes the structure of the media, the codecs used (as registered with the MP4 registration authority [MP4REG]), the segments and the corresponding mapping within a container file system.

The MPD contains information about the preferred Connection Trackers, than can be classified in tiers of priority (attribute @tier).

The MPD is a Well-Formed XML Document, encoded as double-byte Unicode. The XML-Schema of the MPD aligns with ISO/IEC 23009-1 [ISO.IEC.23009-1].

The following example of MPD is for an on-demand media program encoded in SVC with two alternative SVC streams, two audio streams and a text stream. The example SVC stream has one base layer representation with two complementary enhancement layers for one video resolution and another SVC stream with a base layer and one complementary enhancement representation for a higher video resolution, an audio stream in English and another in Portuguese, and a timed subtitle file in Portuguese. The contents have protection schemes and include the root fingerprints (attribute @hash of element RootFP) in each video and audio groups (for integrity verification purposes).

```
<?xml version="1.0" encoding="UTF-8"?>
<MPD type="OnDemand">
  <ProgramInformation>
    <Title>Movie in SVC</Title>
  </ProgramInformation>
  <Trackers>
    <Tracker url="http://example.com:80" tier="1" />
    <Tracker url="http://example.net:80" tier="2" />
```

```
    </Trackers>
    <SwarmID>1234</SwarmID>
    <Period>
      <BaseURL>Program01</BaseURL>
      <Group mimeType="video; codecs=h264/SVC" lang="en">
        <Representation frameRate="15" width="1280" height="720"
                        id="tag0" bandwidth="32000">
          <ContentProtection schemeIdUri="urn:uuid:706D6953-656C....">
            <RootFP hash="57438tgfkv...." />
          </ContentProtection>
          <SegmentInfo startIndex="0" endIndex="150"
                       duration="PT2.00S" levels="3" />
        </Representation>
        <Representation frameRate="30" width="1920" height="1080"
                        id="tag3" bandwidth="256000">
          <ContentProtection schemeIdUri="urn:uuid:706D6953-656C....">
            <RootFP hash="95448trf6v...." />
          </ContentProtection>
          <SegmentInfo startIndex="0" endIndex="150"
                       duration="PT2.00S" levels="2" />
        </Representation>
      </Group>
      <Group mimeType="video; codecs=h264/SVC" lang="en">
        <Representation frameRate="30" width="1280" height="720"
                        id="tag1" bandwidth="64000"
                        dependencyId="tag0">
          <ContentProtection schemeIdUri="urn:uuid:706D6953-656C....">
            <RootFP hash="2356ac468k...." />
          </ContentProtection>
          <SegmentInfo startIndex="0" endIndex="150"
                       duration="PT2.00S" />
        </Representation>
        <Representation frameRate="60" width="1920" height="1080"
                        id="tag4" bandwidth="512000"
                        dependencyId="tag3">
          <ContentProtection schemeIdUri="urn:uuid:706D6953-656C....">
            <RootFP hash="98216d99ab...." />
          </ContentProtection>
          <SegmentInfo startIndex="0" endIndex="150"
                       duration="PT2.00S" />
        </Representation>
      </Group>
      <Group mimeType="video; codecs=h264/SVC" lang="en">
        <ContentProtection schemeIdUri="urn:uuid:706D6953-656C....">
          <RootFP hash="364t96au9d...." />
        </ContentProtection>
        <Representation frameRate="60" width="1280" height="720"
                        id="tag2" bandwidth="256000"
```

```
                        dependencyId="tag0 tag1">
          <SegmentInfo startIndex="0" endIndex="150"
                       duration="PT2.00S" />
        </Representation>
      </Group>
      <Group mimeType="audio/mp4; codecs=mp4a" lang="en"
             bandwidth="64000">
        <ContentProtection schemeIdUri="http://example.net/drm">
          <RootFP hash="26ft54zd9a...." />
        </ContentProtection>
        <Representation id="tag5">
          <SegmentInfo startIndex="0" endIndex="150"
                       duration="PT2.00S" />
        </Representation>
      </Group>
      <Group mimeType="audio/mp4; codecs=mp4a" lang="pt"
             bandwidth="64000">
        <ContentProtection schemeIdUri="http://example.net/drm">
          <RootFP hash="64fg53zn53...." />
        </ContentProtection>
        <Representation id="tag6">
          <SegmentInfo startIndex="0" endIndex="150"
                       duration="PT2.00S" />
        </Representation>
      </Group>
      <Group mimeType="application/ttml+xml" lang="pt">
        <Representation id="tag7">
          <SegmentInfo>subtitles/Program01-pt.xml</SegmentInfo>
        </Representation>
      </Group>
    </Period>
  </MPD>
```

The MPD file for P2P Streaming contains tracker information and can be compressed with GZIP file format [RFC1952] in order to be used with HTTP compression [RFC2616] for faster transmission times and less network bandwidth usage.

The Client Media Player parses the downloaded MPD file and, if it includes information for P2P Streaming, sends the information to the peer and waits for the response in order to start requesting media chunks to decode and play-out.

The MPD file for Live Streaming has a similar structure but describes a sliding window of a small range in the SegmentInfo element from the live program stream timeline (typically, 10 seconds of video). The sliding window is updated for every new encoded segments (a range of chunks defined by the attributes @startIndex and @endIndex) of the

program stream.

The following excerpt of MPD is for a Live scalable video content. The MPD is updated every 10 seconds while the content is being encoded in real-time. Note that each segment set defined in the Live MPD is self-contained and the necessary information related to eventual content protection and integrity verification keys for the set is provided:

```
<?xml version="1.0" encoding="UTF-8"?>
<MPD type="Live"
     availabilityStartTime="2001-12-17T09:40Z"
     availabilityEndTime="2001-12-17T09:50Z"
     minBufferTime="PT10.00S"
     minimumUpdatePeriodMPD="PT10S">
  <SwarmID>654321xyz</SwarmID>
  <Period start="PT11S">
    <Group mimeType="video; codecs=h264/SVC" lang="en">
      <Representation frameRate="15" width="1280" height="720"
                      id="tag0" bandwidth="32000">
        <ContentProtection schemeIdUri="urn:uuid:706D6953-656C....">
          <RootFP hash="57438tgfkv...." />
        </ContentProtection>
        <SegmentInfo startIndex="5" endIndex="9"
                     duration="PT2.00S" levels="3" />
      </Representation>
      .... more descriptions ....
    </Group>
      .... more descriptions ....
  </Period>
</MPD>
```

Appendix C. PPSP Requirements Compliance

C.1. PPSP Basic Requirements

PPSP.REQ-1: The design of the Tracker protocol in this document allows the Peer Protocol to be similar in terms of design, message formats and flows.

PPSP.REQ-2: The design of the Tracker protocol in this document enables peers to receive streaming content within required time constraints.

PPSP.REQ-3: Each peer has a unique ID (i.e., Peer-ID) that identifies the peer in all swarms joined.

PPSP.REQ-4: Each streaming content is uniquely identified by a Swarm-ID.

PPSP.REQ-5: The streaming content is partitioned into chunks individually addressable.

PPSP.REQ-6: Each chunk has an unique ID in the swarm and is individually addressable.

PPSP.REQ-7: The Tracker Protocol is carried over TCP.

PPSP.REQ-8: The Tracker Protocol is designed to facilitate acceptable QoS, supporting, without special algorithms, adaptive and scalable video and 3D video, for both Video On Demand (VoD) and Live video services, allowing additionally complementary mechanisms like super peers, in-network storage, alternative peer addresses and usage of QoS information for advanced peer selection.

C.2. PPSP Tracker Protocol Requirements

PPSP.TP.REQ-1: The Tracker Protocol implements the reception of queries from peers, such as those in JOIN and FIND messages and periodical peer status reports (STAT_REPORT), as well as the corresponding replies.

PPSP.TP.REQ-2: The peer MUST implement the Tracker Protocol designed in this draft.

PPSP.TP.REQ-3: The tracker request messages JOIN and FIND allow the requesting of peer list from the tracker with respect to a specific Swarm-ID and include preferred number of peers in the peer list as well as peer properties which enable appropriate candidate peer selections by the tracker.

PPSP.TP.REQ-4: The tracker responses from JOIN and FIND messages allow the tracker to offer the peer list to the requesting peer with respect to a specific Swarm-ID.

PPSP.TP.REQ-5: The Tracker supports generating the peer lists with the help of traffic optimization services like ALTO.

PPSP.TP.REQ-6: The STATUS_REPORT message informs the Tracker about the peer's activity in the swarm.

PPSP.TP.REQ-7: The chunk availability information (ChunkMaps) of the Peer (for all joined swarms) is reported to the tracker in STATUS_REPORT messages.

PPSP.TP.REQ-8: The ChunkMaps exchanged between peer and tracker can be expressed as compact encoded strings.

PPSP.TP.REQ-9: The STATUS_REPORT message informs the tracker about the peer status and capabilities.

C.3. PPSP Security Considerations

PPSP.SEC.REQ-1: The Tracker Protocol supports closed swarms, where the peers are required to be authenticated.

PPSP.SEC.REQ-2: Confidentiality of the streaming content can be supported, and the corresponding key management mechanisms can be negotiated in the authentication and authorization phase (via CONNECT message) before the peer JOINs the swarm.

PPSP.SEC.REQ-3: The Tracker Protocol uses security layers to encrypt the data exchanged among the PPSP entities.

PPSP.SEC.REQ-4: The Tracker Protocol security layer mechanisms help to limit potential damages caused by malfunctioning and badly behaving peers in the P2P streaming system. The streaming mechanisms considered in the PPSP-TP model prevent pollution of contents.

PPSP.SEC.REQ-6: The use of trusted trackers and peer authentication and authorization mechanisms capable to provide additional security and confidentiality, allow to mitigate and prevent peers from DoS attacks.

PPSP.SEC.REQ-7: The Tracker Protocol design supports distributed tracker architectures, providing robustness to the streaming service in case of centralized tracker failure.

PPSP.SEC.REQ-8: The Tracker Protocol use of Transport Layer Security mechanisms avoids the need for developing new security mechanisms.

PPSP.SEC.REQ-9: The Tracker Protocol together with the Media Presentation Description (MPD) allow the use of streaming content integrity mechanisms.

Authors' Addresses

Rui Santos Cruz
IST/INESC-ID/INOV
Phone: +351.939060939
Email: rui.cruz@ieee.org

Gu Yingjie
Huawei
Phone: +86-25-56624760
Fax: +86-25-56624702
Email: guyingjie@huawei.com

Mario Serafim Nunes
IST/INESC-ID/INOV
Rua Alves Redol, n.9
1000-029 LISBOA, Portugal
Phone: +351.213100256
Email: mario.nunes@inov.pt

David A. Bryan
Polycom
P.O. Box 6741
Williamsburg, Virginia 23188
United States of America
Phone: +1.571.314.0256
Email: dbryan@ethernot.org

Jinwei Xia
Huawei
Nanjing, Baixia District 210001
China
Phone: +86-025-86622310
Email: xiajinwei@huawei.com

Joao P. Taveira
IST/INOV
Email: joao.silva@inov.pt

Deng Lingli
China Mobile

PPSP
Internet Draft
Intended status: Informational
Expires: April 26, 2012

L.Xiao
Nokia Siemens Networks
D.Bryan
Cogent Force, LLC/Huawei
Y.Gu
Huawei
X.Tai
China Mobile/BUPT

October 25, 2011

A PPSP Tracker Usage for Reload
draft-xiao-ppsp-reload-distributed-tracker-03

Status of this Memo

Copyright Notice

Abstract

This document defines PPSP tracker usages for REsource LOcation And Discovery (RELOAD). Although PPSP assumes a centralized tracker from peer's point of view, the logical centralized tracker could be realized by a cluster of geographically distributed trackers. In this draft, we design distributed trackers system, which are organized by RELOAD. It provides lookup service for file/channel indexes and Peer Status among the distributed trackers.

Table of Contents

1. Introduction

PPSP assumes that a centralized 'tracker' is used to communicate with the PPSP Peers for content registration and location. The content index is stored in the tracker with location information that which peers have the content.

However, the logically centralized 'tracker' could be also realized by a cluster of geographically distributed trackers or deployed in multiple servers in a data center, which can increase the content availability, the service robustness and the network scalability or reliability. The management and locating of index information are totally internal behaviors of the tracker cluster, which is invisible

to PPSP Peers. The signaling between PPSP Peers and trackers is still the simple request/reply mechanism defined in PPSP tracker protocols. Therefore, the Tracker Nodes themselves construct and maintain a P2P overlay as shown in Figure 1. This document is to define the behaviors of the tracker overlay as a usage for RELOAD. Because all protocols this draft applies to - the PPSP tracker protocol, PPSP peer protocol and RELOAD - are still changing, this draft will certainly change in the future, and is very much a work in progress. The authors encourage list discussion of the draft and any suggestions are welcome.

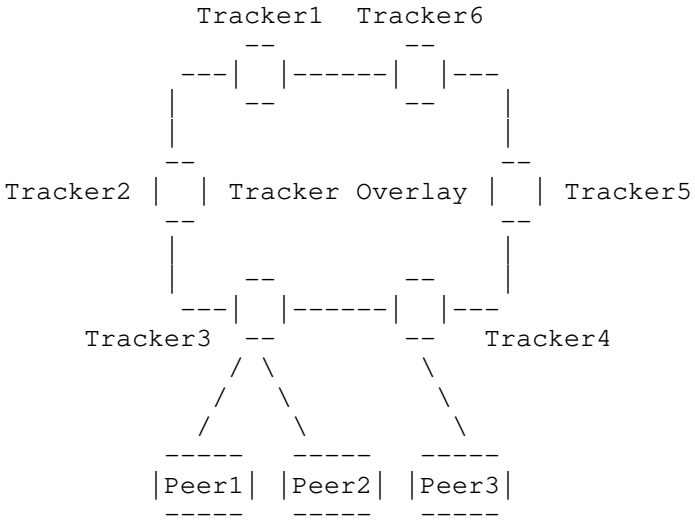


Figure 1 PPSP Peers connect with Tracker Overlay by Tracker Protocol

The document tries to handle the data maintenance for two kinds of information: content index and PPSP peer status, among a cluster of distributed trackers. Four basic functions in tracker overlay are defined as follows:

o Content/ channel index information registration: PPSP Peers registrar/update their contents/channels to a Connection Tracker.(How to find the initial tracker locally is out of scope.) This tracker takes the advantage of the RELOAD data storage functionality to store the index information to tracker nodes in the tracker overly accordingly. At the same time, the local Connection Tracker keeps a copy of local peer's content information for traffic localization.

o Look up a content/channel index: Once a PPSP Peer search for certain content/channel, it makes the request to a local Connection Tracker as defined in PPSP tracker protocol. If the swarm cannot be found or there is not enough peer records for such swarm in the Connection Tracker locally, the tracker will further locate the required index information in the tracker overlay on behalf of the requesting PPSP Peer. Once the full Peer List is fetched, the PPSP Peer will set up communications with the PPSP Peers in the Peer List as defined in PPSP Peer protocol;

o PPSP peer status registration: PPSP Peers registrar/update their status in the tracker overlay. All PPSP peers should firstly register their status to the local Connection Tracker. In order to enable this information being aware globally, the Connection Tracker should then store the position of the PPSP peer's status in the tracker overlay according to RELOAD scheme. The following peer status updates are only sent to the local Connection Tracker, the RELOAD based tracker overlay here only offers a way for remote nodes to find the location of requested peer status.

o Look up status of a certain peer: the tracker overlay can look up the status of a certain PPSP Peer. If the peer status cannot be found in the local Connection Tracker (that means it's not a local peer), the local tracker then searches the Status Position Tracker for the requested peer in the tracker overlay by RELOAD, which gives a route to access the status of the remote peer.

2. Terminology and Conventions

This document makes extensive use of the terminology and definitions from the RELOAD Base Protocol [I-D.ietf-p2psip-base], PPSP Requirements and Problem Statements [I-D.ietf-ppsp-problem-statement][I-D.ietf-ppsp-reqs] and the Gu PPSP Tracker Protocol proposal [I-D.gu-ppsp-tracker-protocol].

This document defines the following additional terminology:

PPSP Peer: The peer in PPSP protocol for content sharing and distribution among swarms.

Tracker Node: The RELOAD Node with PPSP tracker usage. Each Tracker Node takes the responsibility to store and maintain certain content/channel index.

Tracker Overlay: A RELOAD overlay constructed by Tracker Nodes. This Overlay is logically separated with overlay formed by PPSP Peers.

Connection Tracker: The Tracker Node to which the PPSP Peer will connect when it wants to join the PPSP system.

Swarm Tracker: The Tracker Node who is responsible for the swarm in the overlay, and stores the content information (e.g. Peerlist) of the swarm.

Status Position Tracker: A Tracker Node which is responsible to store the Position of certain peers' status of a particular list of Peers.

3. Content Information Registration and Update

To fulfill the functions of content information registration and update mentioned in Section 1, Tracker Node must maintain such resources related to peers;

Content Registration: Information about the content which belongs to a specific swarm. It can be stored in a data structure denoted as ContentRegistration, which primarily includes an identification of the swarm, a name of the content, and a Peer List.

3.1. Data structure of ContentRegistration

Structure The data structure of ContentRegistration uses the RELOAD dictionary kind whereas the DictionaryKey value is the Swarm ID of the content required. The data structure of type ContentRegistration is shown as follows:

```
struct{

        Uint32 index;

        ChunkID chunk_id;
```

```
}ArrayChunkListData;

struct{
       PeerID peer_id;
       ArrayChunkListData  chunklist_data;
}PeerListData;

struct{
       uint16 length;
       PeerListData peerlist_data;
}PeerList;

struct {
       uint16 length;
       opaque content_name<0..2^16-1>;
       PeerList peerlist <0..2^16-1>;
} ContentRegistration;
```

The content of the PeerList structure are as follows:

length

the length of the data structure

content_name

the name of the content

peerlist

the content of Peer List

3.2. Message flows

When a PPSP Peer wishes to share its contents to others, it will inform Tracker Overlay with the swarm information of the contents, then Swarm Tracker need to add this PPSP Peer into the corresponding Peer List to the swarm, or create a new swarm when there is no record of the swarm. A local record of the swarm may also be set up at the Connection tracker. Correspondingly, When a PPSP Peer deletes some old contents locally, it will inform Tracker Overlay that it would like to leave from a particular swarm, then both Connection Tracker and Swarm Tracker need to delete this PPSP Peer from the corresponding Peer List which is defined in the requirement of PPSP [I-D.ietf-ppsp-reqs].

An example is given as the figure has shown below:

1. PPSP Peer wants to join into a swarm to share the content, first it will send a PPSP message "Join" with a Swarm-ID to TrackerA, which is a connection tracker of the Tracker Overlay for PPSP Peer connects to;

2. TrackerA first handles the registration locally, then finds the Swarm Tracker by mapping the swarm ID to node ID of the Swarm Tracker, to forward the request. So TrackerA sends a RELOAD message "StoreReq" to TrackerB who is the Swarm Tracker for the content swarm;

3. When Swarm Tracker (TrackerB) receives the request (or if TrackerA is responsible for the Peer List of the swarm, TrackerB=TrackerA), it searches locally the Peer List of the swarm whose ID is the Swarm-ID, then add the Node-ID of the PPSP Peer into the Peer List or delete it from that, and send the result of the operation (e.g. successful or failed) in a RELOAD message "StoreReqAns" to TrackerA through Tracker Overlay;

4. Finally, TrackerA analyses the received message, and responds to the requesting Peer by a corresponding PPSP message: "Successful (OK)" or some error messages.

Note: When PPSP Peer is the first node of the swarm, which means it is the first one who stores this kind of content in the network, TrackerB doesn’t have records of the new swarm, TrackerB will create a new ContentRegistration for the swarm locally, and put the identification of PPSP Peer into Peer List of this new

ContentRegistration, then send the result of the operation (e.g. successful or failed) in a RELOAD message "StoreReqAns" to TrackerA through Tracker Overlay.

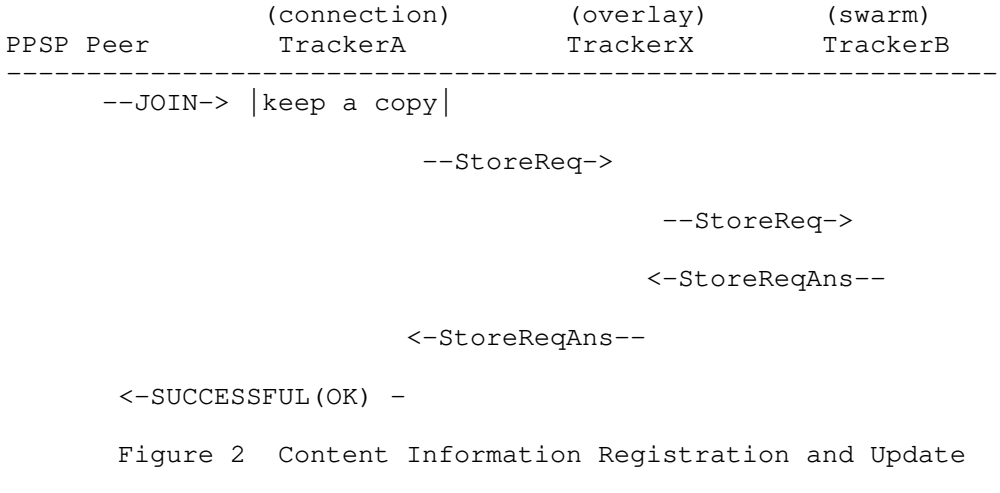


Figure 2 Content Information Registration and Update

If PPSP Peer wishes to update content information, for example, list of chunks it has, it sends a PPSP message "JOIN_CHUNK" to TrackerA. TrackerA makes update in its local table, and then sends the corresponding RELOAD message to TrackerB to update the detailed chunk-IDs in the Swarm according to the request message.

4. Lookup Content Index (a Swarm)

When a PPSP Peer wants to use some streaming service, which means it wants to download some interested contents from the system, it firstly needs to get related Peer List from Tracker Overlay. As the figure has shown below:

1) PPSP Peer wants to watch a video belonging to a swarm with a Swarm-ID, firstly it sends a PPSP message "Find" with the Swarm-ID to Connection TrackerA;

2) If TrackerA has enough local peer record for swarm, it can reply the request directly. Or it maps the Swarm-ID into a Node-ID to identify the Swarm Tracker, TrackerB, which stores the Peer List of the requested swarm. It then sends a RELOAD message "FetchReq" to TrackerB;

3) When Swarm TrackerB receives the request (or if TrackerA is responsible for the Peer List of the swarm, TrackerbB=TrackerA), it searches the Peer List of the swarm locally, then send the Peer List which is organized by the data structure of PeerList in a RELOAD message "FetchReqAns" to TrackerA through Tracker Overlay;

4) Finally, TrackerA analyses the received PeerList structure, and reconstructs it into a PPSP message "Successful(OK)", then forwards it to the PPSP Peer.

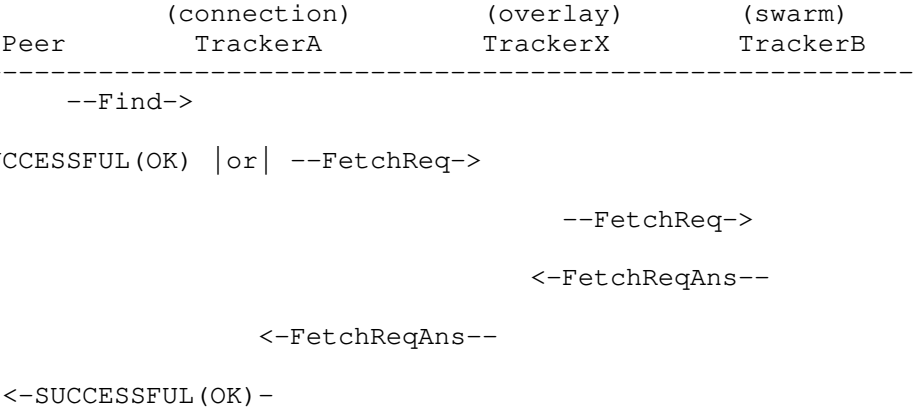


Figure 3 Content Information Lookup

5. Peer Status Registration, Update and Lookup

To fulfill the functions of peer status registration, update and lookup mentioned above, Tracker Node must maintain such resource related to peers:

Information about status of peers: the local Connection Tracker takes the responsibility to maintain the PPSP Peer status locally, including online time, link status, node capability and other streaming parameters, etc. It can be stored in a data structure denoted as PeerStatus.

Position of PPSP peer status: each PPSP Peer can be mapped to a Status Position Tracker in the tracker overlay. The status Position Tracker takes responsibility to only record the route (i.e., the address of the local Connection Tracker of the Peer) to access the PPSP Peer status.

5.1. Data Structure of PeerStatusIndex

The data structure of PeerStatusIndex uses the RELOAD dictionary kind whereas the DictionaryKey value is the Peer ID. The data structure of type PeerStatusIndex is shown as follows:

```
struct{
        TrackerID Connection_Tracker_ID;
}PeerStatusIndex;
```

The content of the PeerStatusIndex structure are as follows:

trackerID the ID of the Peer's Connection Tracker;

6. Kind Definition

6.1. CONTENT-REGISTRATION Kind Definition

This section defines the CONTENT-REGISTRATION kind.

- Name: CONTENT-REGISTRATION

- Kind IDs: The Resource Name for the CONTENT-REGISTRATION Kind-ID is Swarm Name. The data stored is a CONTENT-REGISTRATION, which contains a identification of the swarm, a name of the content, and a list of PPSP Peer-IDs with or not a list of chunk-IDs for each PPSP Peer to show which chunks the PPSP Peer has.

- Data Model: The data model for the CONTENT-REGISTRATION Kind-ID is dictionary. The dictionary key is the Swarm-ID of the peer action as focus.

- Access Control: USER-NODE-MATCH.

6.2. PEER-STATUS Kind Definition

This section defines the PEER-STATUS kind.

- Name: PEER-STATUS

o Kind IDs: The Resource Name for the PEER-STATUS Kind-ID is Peer Status. The data stored is a PEER-STATUS, which contains a identification of the peer and a identification of the peer's connection tracker.

o Data Model: The data model for the PEER-STATUS Kind-ID is dictionary. The dictionary key is the Peer-ID.

o Access Control: USER-NODE-MATCH.

7. Security Considerations

This document does not currently introduce security considerations.

8. IANA Considerations

This document does not specify IANA considerations.

9. Acknowledgments

This document was prepared using 2-Word-v2.0.template.dot.

References

9.1. Normative References

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

[2] [I-D.ietf-ppsp-reqs] Zong, N., Zhang, Y., Avila, V., Williams, C., and L. Xiao, "P2P Streaming Protocol (PPSP) Requirements", draft-ietf-ppsp-reqs-03 (work in progress), July 2011.

[3] [I-D.ietf-ppsp-problem-statement] Zhang, Y., Zong, N., Camarillo, G., Seng, J., and Y. Yang, "Problem Statement of P2P Streaming Protocol (PPSP)", draft-ietf-ppsp-problem-statement-03 (work in progress), August 2011.

[4] [I-D.ietf-p2psip-base] Jennings, C., Lowekamp, B., Rescorla, E., Baset, S., and H. Schulzrinne, "REsource LOcation And Discovery (RELOAD) Base Protocol", draft-ietf-p2psip-base-18 (work in progress), August 2011.

[5] [I-D.gu-ppsp-tracker-protocol] Yingjie, G., Bryan, D., Zhang, Y., and H. liao, "Tracker Protocol", draft-gu-ppsp-tracker-protocol-04 (work in progress), May 2011.

9.2. Informative References

Author's Addresses

Lin Xiao
Nokia Siemens Networks
No.14 Jiuxianqiao Road
Beijing, 100016
P.R.China

Phone: +86-13810361287
Email: lin.xiao@nsn.com

David A. Bryan
Cogent Force, LLC / Huawei

Email: dbryan@ethernot.org

Yingjie Gu
Huawei
No. 101 Software Avenue
Nanjing, Jiangsu Province 210012
P.R.China

Phone: +86-25-56624760
Email: guyingjie@huawei.com

Xuan Tai
China Mobile/BUPT

Phone: +86-13581762082
Email: taixuanyueshi@gmail.com