PPSP
INTERNET-DRAFT
Intended Status: Standards Track
Expires: July 12, 2015

Rui S. Cruz
Mario S. Nunes
IST/INESC-ID/INOV
Yingjie Gu
Jinwei Xia
Rachel Huang
Huawei
Joao P. Taveira
IST/INOV
Deng Lingli
China Mobile
January 8, 2015

PPSP Tracker Protocol-Base Protocol (PPSP-TP/1.0)
draft-ietf-ppsp-base-tracker-protocol-08

Abstract

This document specifies the base Peer-to-Peer Streaming Protocol-Tracker Protocol (PPSP-TP/1.0), an application-layer control (signaling) protocol for the exchange of meta information between trackers and peers. 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 delivery (audio, video, associated timed text and metadata), such as adaptive multi-rate, layered (scalable) and multi-view (3D) videos, in live, time-shifted and on-demand modes.

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. RFC 6972 [RFC6972] specifies that the Tracker Protocol should standardize the messages between PPSP peers and PPSP trackers and also 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.

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

This document describes the base PPSP Tracker protocol and how it satisfies the requirements for the IETF Peer-to-Peer Streaming Protocol, 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 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 [KEYWORDS].

ABSOLUTE TIME: Absolute time is expressed as ISO 8601 timestamps, using zero UTC offset. 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.

CHUNK: A Chunk is a basic unit of data organized in P2P streaming for storage, scheduling, advertisement and exchange among peers.

CHUNK ID: A unique resource identifier for a Chunk. The identifier type depends on the addressing scheme used, i.e., an integer, an HTTP-URL and possibly a byte-range, and is described in the MPD.

CONNECTION TRACKER: The node running the tracker service to which the PPSP peer will connect when it wants to get registered and join the PPSP system.

LEECH: A Peer that has not yet completed the transfer of all Chunks of the media content.

LIVE STREAMING: It refers to a scenario where all the audiences receive streaming content for the same ongoing event. It is desired that the lags between the play points of the audiences and streaming source be small.

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 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 joined. This value indicates the stability of a peer, and can be calculated by the tracker whenever necessary.

PEER: A Peer refers to a participant in a P2P streaming system that not only receives streaming content, but also caches and streams streaming content to other participants.

PEER ID: The identifier of a Peer such that other Peers, or the Tracker, can refer to the Peer by using its ID. The Peer ID is mandatory, can take the form of a universal unique identifier (UUID), defined in [RFC4122], and can be bound to a network address of the Peer, i.e., an IP address, or a uniform resource identifier/locator (URI/URL) that uniquely identifies the corresponding Peer in the network. The Peer ID and any required security certificates are obtained from an offline enrollment server.

PEER LIST: A list of Peers which are in a same SWARM maintained by the Tracker. A Peer can fetch the Peer List of a SWARM from the Tracker or from other Peers in order to know which Peers have the required streaming content.

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

PPSP-TP: The abbreviation of Peer-to-Peer Streaming Protocols - Tracker Protocol.

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.

SEEDER: A Peer that holds and shares the complete media content.

SERVICE PORTAL: A logical entity typically used for client enrollment and content information publishing, searching and retrieval. It is usually located in a server of content provider.

SWARM: A Swarm refers to a group of Peers who exchange data to distribute Chunks of the same content (e.g., video/audio program, digital file, etc.) at a given time.

SWARM ID: The identifier of a Swarm containing a group of Peers sharing a common streaming content. The Swarm-ID may use a universal unique identifier (UUID), e.g., a 64 or 128 bit datum to refer to the content resource being shared among peers.

SUPER-NODE: A Super-Node is a special kind of Peer deployed by ISPs. This kind of Peer is more stable with higher computing, storage and bandwidth capabilities than normal Peers.

TRACKER: A Tracker refers to a directory service that maintains a list of Peers participating in a specific audio/video channel or in the distribution of a streaming file. Also, the Tracker answers Peer List queries received from Peers. The Tracker is a logical component which can be centralized or distributed.

TRANSACTION ID: The identifier of a REQUEST from the Peer to the Tracker. Used to disambiguate RESPONSES that may arrive in a different order of the corresponding REQUESTs.

VIDEO-ON-DEMAND (VoD): It refers to a scenario where different audiences may watch different parts of the same recorded streaming with downloaded content.

1.2 Design Overview

The functional entities related to PPSP protocols are the Client Media Player, the service Portal, the Tracker and the Peers. The complete description of Client Media Player and service Portal is not discussed here, as not in the scope the specification. The functional entities directly involved in the PPSP Tracker Protocol are trackers and peers (which may support different capabilities).

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.

A Peer corresponds to a logical entity (typically in a user device) that actually participates in sharing a media content. Peers are organized in (various) swarms corresponding each swarm to the group of peers streaming a certain content at any given time.

The Tracker is a logical entity that maintains the lists of peers storing Chunks for a specific Live media channel or on-demand 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 on-demand or Live streaming) with peers, but information about which peers can provide the content.

1.2.1 Typical Use Cases

When a peer wants to receive streaming of a selected content (Leech mode):

1. Peer connects to a Connection Tracker and joins a Swarm.
2. Peer acquires a list of other peers in the Swarm from the Connection Tracker.
3. [Peer Protocol] Peer exchanges its content availability with the peers on the obtained peer list.
4. [Peer Protocol] Peer identifies the peers with desired content.
5. [Peer Protocol] Peer requests content from the identified peers.

When a peer wants to share streaming contents (Seeder mode) with other peers:

1. Peer connects to the Connection Tracker.
2. Peer sends information to the Connection Tracker about the swarms it belongs to (joined swarms).

After having been disconnected due to some termination condition, a Peer can resume previous activity by connecting and re-joining the corresponding Swarm(s).

1.2.2 Enrollment and Bootstrap

In order to be able to bootstrap in the P2P network, a peer must first obtain a Peer ID (identifier of the peer) and any required security certificates or authorization tokens from an enrollment service (end-user registration). The specification of the format of the Peer ID is not in the scope of this document.

```
+--------+      +--------+     +--------+    +---------+  +--------+
| Player |      | Peer_1 |     | Portal |    | Tracker |  | Peer_2 |
+--------+      +--------+     +--------+    +---------+  +--------+
    |                |               |              |           |
(a) |--Page request----------------->|              |           |
    |<--------------Page with links--|              |           |
    |--Select stream (MPD Request)-->|              |           |
    |<--------------------OK+MPD(x)--|              |           |
(b) |--Start/Resume->|--CONNECT(join x)------------>|           |
    |<-----------OK--|<----------------OK+Peerlist--|           |
    |                |                              |           |
    |--Get(Chunk)--->|<---------- (Peer protocol) ------------->|
    |<--------Chunk--|<---------------------------------Chunks--|
    :                :               :              :           :
    |                |--STAT_REPORT---------------->|           |
    |                |<-------------------------OK--|           |
    :                :               :              :           :
    |                |--FIND----------------------->|           |
    |                |<----------------OK+Peerlist--|           |
    :                :               :              :           :
    |--Get(Chunk)--->|<---------- (Peer protocol) ------------->|
    |<--------Chunk--|<---------------------------------Chunks--|
    :                :               :              :           :
```

Figure 1: A typical PPSP session for streaming a content.

To join an existing P2P streaming service and to participate in content sharing, any Peer must first locate a Tracker.

As illustrated in Figure 1, a P2P streaming session may be initiated starting at point (a), with the Client Media Player browsing for the desired content in order to request it (to the local Peer_1 in the figure), or resume a previously initiated stream, but starting at point (b). For this example, the Peer_1 is in mode LEECH.

At point (a) in Figure 1, the Client Media Player accesses the Portal and selects the content of interest. The Portal returns the Media Presentation Description (MPD) file that includes information about the address of one or more Trackers (that can be grouped by tiers of priority) which are controlling the Swarm x for that media content (e.g., content x).

With the information from the MPD the Client Media Player is able to trigger the start of the streaming session, requesting to the local Peer_1 the Chunks of interest.

The PPSP streaming session is then started (or resumed) at Peer_1 by sending a PPSP-TP CONNECT message to the Tracker in order to join Swarm x. The Tracker will then return the OK response message containing a peer list, if the CONNECT message is successfully accepted. From that point onwards every Chunk request is addressed by Peer_1 to its neighbors (Peer_2 in Figure 1) using the PPSP Peer Protocol, returning the received Chunks to the Client Media Player.

Once CONNECTed, Peer_1 needs to periodically report its status and statistics data to the Tracker using a PPSP-TP STAT_REPORT message.

If Peer_1 needs to refresh its neighborhood (for example, due to churn) it will send a PPSP-TP FIND message (with the desired scope) to the Tracker.

Peers that are only SEEDERs (i.e., serving contents to other peers), as are the typical cases of service provider P2P edge caches and/or Media Servers, trigger their P2P streaming sessions for contents x, y, z... (Figure 2), not from Media Player signals, but from some "Start" activation signal received from the service provider provisioning mechanism. In this particular case the Peer starts or resumes all its streaming sessions just by sending a PPSP-TP CONNECT message to the Tracker (Figure 2), in order to "join" all the requested swarms.

Periodically, the Peer also report its status and statistics data to the Tracker using a PPSP-TP STAT_REPORT message.

```
                +---------+                      +---------+
                |  Seeder |                      | Tracker |
                +---------+                      +---------+
                     |                                |
              Start->|--CONNECT (join x,y,z)--------->|
                     |<--------------------------OK--|
                     :                                :
                     |                                |
                     |--STAT_REPORT------------------>|
                     |<--------------------------Ok--|
                     :                                :
                     |                                |
                     |--STAT_REPORT------------------>|
                     |<--------------------------Ok--|
                     :                                :
```

Figure 2: A typical PPSP session for a streaming Seeder.

The specification of the mechanisms used by the Client Media Player (or provisioning process) and the Peer to signal start/resume streams or request media chunks, obtain a Peer ID, security certificates or tokens are not in the scope of this document.

2 Protocol Architecture and Functional View

The PPSP Tracker Protocol architecture is intended to be compatible with the web infrastructure. PPSP-TP is designed with a layered approach i.e., a PPSP-TP Request/Response layer, a Message layer and a Transport layer. The PPSP-TP Request/Response layer deals with the interactions between Tracker and Peers using Request and Response codes (see Figure 3).

```
                    +-----------------------+
                    |      Application      |
                    +-----------------------+
                    |  Request/Response     |  PPSP-TP
                    |-----------------------|
                    |   (HTTP) Message      |
                    +-----------------------+
                    |       TRANSPORT       |
                    +-----------------------+
```

Figure 3: Abstract layering of PPSP-TP.

The Message layer deals with the framing format, for encoding and transmitting the data through the underlying transport protocol, as well as the asynchronous nature of the interactions between Tracker and peers.

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 message when using a connection-oriented transport like TCP [RFC0793], or TLS [RFC5246] over it.

2.1 Messaging Model

The messaging model of PPSP-TP aligns with HTTP protocol and the semantics of its messages, currently in version 1.1 [RFC2616], but intended to support future versions of HTTP. The exchange of messages of PPSP-TP is envisioned to be performed over a stream-oriented reliable transport protocol, like TCP [RFC0793].

2.2 Request/Response model

PPSP-TP uses a REST-Like (Representational State Transfer) design [Fielding] with the goal of leveraging current HTTP implementations and infrastructure, as well as familiarity with existing REST-like services in popular use. PPSP-TP messages use the UTF-8 character set [RFC3629] and are either requests from peers to a tracker service, or responses from a tracker service to peers. The Request and Response semantics are carried as entities (header and body) in messages which correspond to either HTTP request methods or HTTP response codes, respectively.

PPSP-TP uses the HTTP POST method to send parameters in requests. PPSP-TP messages use JavaScript Object Notation (JSON) [RFC7159] to encode message bodies.

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

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

PPSP-TP/1.0 Request Messages
CONNECT
FIND
STAT_REPORT

Table 1: Request Messages

CONNECT: This Request message is an "action signal" used when a Peer registers in the Tracker (or if already registered) to notify it about the participation in named swarm(s). The Tracker records the Peer ID, connect-time (referenced to the absolute time), peer IP addresses (and associated location information), link status and Peer Mode for the named swarm(s). The Tracker also changes the content availability of the valid named swarm(s), i.e., changes the peers lists of the corresponding swarm(s) for the requester Peer ID. On receiving a CONNECT message, the Tracker first checks the peer mode type (SEED/LEECH) for the specified swarm(s) and then decides the next steps (more details are referred in section 4.1)

FIND: This Request message is an "action signal" used by peers to request to the Tracker, whenever needed, a list of peers active in the named swarm. 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 is an "information signal" that allows an active Peer to send status (and optionally statistic data) to the Tracker to signal continuing activity. This request message MUST be sent periodically to the Tracker while the Peer is active in the system.

2.3 State Machines and Flows of the Protocol

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.

```
  --------------------------------------------
 /                                            \
|  +------------+    +=========+    +======+   |
 \-| TERMINATED |<---| STARTED |<---| INIT |<-/
   +------------+    +=========+    +======+
    (Transient)                         \- (start tracker)
```

Figure 4: Tracker State Machine

When there are no peers connected in the Tracker, the state machine is in the INIT state.

When the "first" Peer connects for registration 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.

Once in STARTED state, each Peer is instantiated (per Peer ID) in the Tracker state machine with a dedicated transaction state machine (Figure 5), which is deleted when the Peer ID is removed.

```
               --------------------------------------------
              /                                            \
             |  +------------+    +=========+    +======+   |
              \-| TERMINATED |<---| STARTED |<---| INIT |<-/
                +------------+    +=========+    +======+
                 (Transient)           | (1)        \- (start tracker)
                                       V
                   +-----------+   +-------+  rcv CONNECT
       (Transient) | TERMINATE |   | START |  --------------- (1)
                   +-----------+   +-------+  strt init timer
   rcv FIND              ^             |
   rcv STAT_REPORT       |             |
   on registration error |             v
   on action error       |      +------------+
   ---------------- (A)  +<-----| PEER       | (Transient)
   stop init timer       |      | REGISTERED |
   snd error             |      +------------+
                         |            |
                         |            |   process swarm actions
                         |            |   --------------------- (2)
   on CONNECT Error (B)  |            |   snd OK (PeerList)
   on timeout       (C)  |           /    stop init timer
   ----------------      |          /     strt track timer
   stop track timer      |         /
   clean peer info       |        |
   del registration      |        |             rcv FIND
   snd error (B)          \       |     ----    --------------- (3)
                    ----   \      |   /      \  snd OK (PeerList)
                  /      \  \     |  |        | rst track timer
   rcv CONNECT   |  (4)   |  |    |  |        |
   -----------   |        v  |    v  v        | rcv STAT_REPORT
   snd OK         \     +==============+     /  --------------- (3)
   rst track timer  ----|   TRACKING   |----    snd OK response
                        +==============+        rst track timer
```

Figure 5: Per-Peer-ID Transaction State Machine and Flow Diagram

Unlike the Tracker state machine, which exists even when no Peer IDs are registered, the "per-Peer-ID" transaction state machine is instantiated only when the Peer ID starts registration in the tracker, and is deleted when the Peer ID is de-registered/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).

When a new Peer ID is added, the corresponding "per-Peer-ID" state machine 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 lifetime of streaming activity of a peer, the instantiated "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, FIND 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.

2.3.1 Normal Operation

On normal operation the process consists of the following steps:

1) When a Peer wants to access the system it needs to register on a tracker by sending a CONNECT message asking for the swarm(s) it wants to join. This request from a new Peer ID triggers the instantiation in the Tracker of a "per-Peer-ID" State Machine. In the START state of the new "per-Peer-ID" SM, the Tracker registers the Peer ID and associated information (IP addresses), starts the "init timer" and moves to PEER REGISTERED state.

2) In PEER REGISTERED state, if Peer ID is valid, the Tracker either a) processes the requested action(s) for the valid swarm information contained in the CONNECT request and in case of success the tracker stops the "init timer", starts the "track timer" and sends the response to the Peer (the response MAY contain the appropriate list of peers for the joining swarm(s), as detailed in section 4.1, or b) moves the valid FIND request to TRACKING state.

3) In TRACKING state, STAT_REPORT or FIND messages received from that Peer ID will reset the "track timer" and are respectively responded with a) a successful condition, b) a successful condition containing the appropriate list of peers for the named swarm (section 4.2).

4) While TRACKING, a CONNECT message received from that Peer ID with valid swarm actions information (section 4.1.1) resets the "track timer" and is responded with a successful condition.

2.3.2 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 PEER REGISTERED state, when a CONNECT Request only contains invalid swarm actions (section 6.1.1), the Tracker responds with error code 403 Forbidden, deletes the registration, transition to TERMINATE state for that Peer ID and the SM is destroyed.

At the PEER REGISTERED state, if the Peer ID is considered invalid (in the case of a CONNECT request or in the case of FIND or STAT_REPORT requests received from an unregistered Peer ID), the Tracker responds with either error codes authentication required or Forbidden (described in section 4.3), transitions to TERMINATE state for that Peer ID and the SM is destroyed.

B) At the TRACKING state (while the "track timer" has not expired) receiving a CONNECT message from that Peer ID with invalid swarm actions (section 5.1) is considered an error condition. The Tracker responds with error code Forbidden (described in section 4.3), stops the "track timer", deletes the registration, transitions to TERMINATE state for that Peer ID and the SM is destroyed.

C) In TRACKING state, without receiving messages from the Peer, on timeout (track timer) the Tracker cleans all the information associated with the Peer ID in all swarms it was joined, deletes the registration, transitions to TERMINATE state for that Peer ID and the SM is destroyed.

NOTE: These situations may correspond to malfunctions at the Peer or to malicious conditions. As preventive measure, the Tracker proceeds to TERMINATE state for that Peer ID.

3 Protocol Specification

3.1 Presentation Language

PPSP-TP uses a REST-Like design, encoding the requests and responses using JSON [RFC7159]. For a generalization of the definition of protocol elements and fields, their types and structures, this document uses a C-style notation, similar to the presentation language used to define TLS [RFC5246], turning the definitions for JSON objects extensible.

A JSON object consists of name/value pairs. The JSON names of the pairs are indicated with "". In this presentation language, comments begin with "//", and the "ppsp_tp_string_t" and "ppsp_tp_integer_t" types are used to indicate the JSON string and number, respectively. Optional fields are enclosed in "[ ]" brackets. An array is indicated by two numbers in angle brackets, <min..max>, where "min" indicates the minimal number of values and "max" the maximum. An "*" is used to denote a no upper bound value for "max".

3.2 Resource Element Types

This section details the format of PPSP-TP resource element types.

3.2.1 Version

For both requests and responses, the version of PPSP-TP being used MUST be indicated by the attribute version, defined as follows:

```
ppsp_tp_integer_t ppsp_tp_version_t = 1
```

The defined value for ppsp_tp_version_t is listed in Table 2

ppsp_tp_version_t	Description
0	Reserved
1	Protocol specified in this document
2-255	Unassigned

Table 2: PPSP Tracker Protocol Version Numbers

3.2.2 Peer Number Element

The PeerNum element is a scope selector in requests and MAY contain the attribute ability_nat to inform the Tracker on the preferred type of peers to be returned in a peer list, related to their NAT traversal situation.

The PeerNum element is a scope selector that MAY be present in CONNECT and FIND requests.

This element contains the attribute peer_count to indicate the maximum number of peers in the returned peer list. Peer_count should be less than 30 in this specification. The other 4 attributes, i.e., ability_nat, concurrent_links, online_time and upload_bandwidth_level MAY be also contained in this element to inform the Tracker on the preferred type of peers to be returned in a peer list:

- o ability_nat is used to indicate the preferred NAT traversal situation of these peers.

- o concurrent_links means the preferred concurrent connectivity level.

- o online_time represents the preferred availability or online duration degree for the requested peers.

- o upload_bandwidth_level is the preferred upload bandwidth capability of these peers.

The definition of the scope selector element and attributes is defined as follows:

```
Object {
        ppsp_tp_integer_t   peer_count;
        [ppsp_tp_string_t   ability_nat = "NO_NAT"
                                        | "STUN"
                                        | "TURN"
                                        | "PROXY";]
        [ppsp_tp_string_t   concurrent_links = "NORMAL"
                                             | "LOW"
                                             | "HIGH";]
        [ppsp_tp_string_t   online_time = "NORMAL" | "HIGH";]
        [ppsp_tp_string_t   upload_bandwidth_level = "NORMAL"
                                                   | "HIGH";]
} ppsp_tp_peer_num_t;
```

3.2.3 Swarm Action Element

The swarm action element identifies the action(s) to be taken in the named swarm(s) as well as the corresponding Peer Mode (if the peer is LEECH or SEEDER in that swarm).

```
      Object {
              ppsp_tp_string_t  swarm_id;   //Swarm Identifier
              ppsp_tp_string_t  action = "JOIN"
                                        |"LEAVE"; // Action type of
                                                  // the CONNECT
                                                  // message
              ppsp_tp_string_t  peer_mode = "SEED"
                                          | "LEECH"; // Mode of Peer
                                                     // participating
                                                     // in this swarm
      } ppsp_tp_swarm_action_t;
```

3.2.4 Peer Information Elements

The Peer information elements provides network identification information of peers. A Peer information consists of peer identifier and the IP related addressing information.

```
      Object {
              ppsp_tp_string_t    peer_id;
              ppsp_tp_peer_addr_t peer_addr;
      }ppsp_tp_peer_info_t;
```

The ppsp_tp_peer_addr_t element includes the IP address and port, with a few optional attributes related with connection type and network location (in terms of ASN) as well as, optionally, the identifier of the Peer Protocol being used.

```
      Object {
              ppsp_tp_ip_address       ip_address;
              ppsp_tp_integer_t        port;
              ppsp_tp_integer_t        priority;
              ppsp_tp_string_t         type = "HOST"
                                            | "REFLEXIVE"
                                            | "PROXY";
             [ppsp_tp_string_t         connection = "3G"
                                                  | "ADSL"
                                                  | "LTE"
                                                  | "ETHER";]
             [ppsp_tp_string_t         asn;]
             [ppsp_tp_peer_protocol_t  peer_protocol;]
      } ppsp_tp_peer_addr_t;
```

The semantics of ppsp_tp_peer_addr_t attributes are listed in Table 3:

Element or Attribute	Description
ip_address	IP Address information
port	IP service port value
priority	The priority of this interface
type	Describes the address for NAT traversal, which can be HOST REFLEXIVE or PROXY
connection	Access type (3G, ADSL, etc.)
asn	Autonomous System Number
peerProtocol	PPSP Peer Protocol supported

Table 3: Semantics of ppsp_tp_peer_addr_t.

In this document, IP address is specified as ppsp_tp_addr_value. The exact characters and format depend on address_type:

o The IPv4 address is encoded as specified by the IPv4address rule in Section 3.2.2 of [RFC3986].

o The IPv6 address is encoded as specified in section 4 of [RFC5952].

```
Object {
        ppsp_tp_string_t   address_type;
        ppsp_tp_addr_value address;
} ppsp_tp_ip_address;
```

The Peer Information in requests or responses is grouped in a ppsp_tp_peer_group_t element:

```
Object {
        ppsp_tp_peer_info_t peer_info<1..*>;
} ppsp_tp_peer_group_t;
```

3.2.5 Statistics and Status Information Element

The statistics element (stat) is used to describe several properties relevant to the P2P network. These properties can be related with stream statistics and peer status information. 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. This specification only defines the property type "STREAM_STATS".

The definition of the statistic element and attributes is as follows:

```
Object {
       ppsp_tp_string_t  swarm_id;
       ppsp_tp_integer_t uploaded_bytes;
       ppsp_tp_integer_t downloaded_bytes;
       ppsp_tp_integer_t available_bandwidth;
} stream_stats;
```

The semantics of stream_stats attributes are listed in Table 4:

Element or Attribute	Description
swarm_id	Swarm Identifier
uploaded_bytes	Bytes sent to swarm
downloaded_bytes	Bytes received from swarm
available_bandwidth	Upstream Bandwidth available

Table 4: Semantics of stream_stats.

The Stat Information is grouped in the ppsp_tp_stat_group_t element:

```
Object {
   ppsp_tp_string_t     type = "STREAM_STATS"; // property type
   stream_stats         stat<1..*>;
} ppsp_tp_stat_group_t
```

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 specific information for new properties, elements or attributes (guidelines in section 7).

3.3 Requests and Responses

This section defines the structure of PPSP-TP requests and responses.

3.3.1 Request Types

The request type includes CONNECT, FIND and STAT_REPORT, defined as follows:

```
   ppsp_tp_string_t ppsp_tp_request_type_t = "CONNECT"
                                           | "FIND"
                                           | "STAT_REPORT";
```

3.3.2 Response Types

Response type corresponds to the response method type of the message, defined as follows:

```
   JSONValue ppsp_tp_response_type_t = 0x00    // SUCCESSFUL
                                     | 0x01;   // FAILED
```

3.3.3 Request Element

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

The generic definition of a request element is the following:

```
   Object {
           [ppsp_tp_peer_num_t      peer_num;]
           [ppsp_tp_peer_addr_t     peer_addr<1..*>;]
           ppsp_tp_swarm_action_t   swarm_action<1..*>;
   } ppsp_tp_request_connect;

   Object {
           ppsp_tp_string_t         swarm_id;
          [ppsp_tp_peer_num_t       peer_num;]
   } ppsp_tp_request_find;

   Object {
           ppsp_tp_version_t        version;
           ppsp_tp_request_type_t   request_type;
           ppsp_tp_string_t         transaction_id;
           ppsp_tp_string_t         peer_id;
           JSONValue request_data = ppsp_tp_req_connect  connect
                                  | ppsp_tp_req_find     find
                                  | ppsp_tp_stat_group_t stat_report;
   } ppsp_tp_request;
```

A request element consists the version of PPSP tracker protocol, the request type, a transaction identifier and the identifier of the requesting peer, as well as the requesting body, i.e., request_data. The request_data MUST be correctly set to the corresponding element based on the request type (see Table 5).

request_type	request_data
"CONNECT"	"connect"
"FIND"	"find"
"STAT_REPORT"	"stat_report"

Table 5: The relationship between request_type and request_data.

3.3.4 Response Element

The generic definition of a response element is the following:

```
Object {
        ppsp_tp_version_t              version;
        ppsp_tp_response_type_t        response_type;
        ppsp_tp_interger_t             error_code;
        ppsp_tp_string_t               transaction_id;
       [ppsp_tp_peer_addr_t            peer_addr;]
       [ppsp_tp_swarm_action_result_t swarm_result<1..*>;]
} ppsp_tp_response;
```

A response element consists the version of PPSP tracker protocol, the response type, the error code, a transaction identifier, and optionally the public address of the requesting peer and one or multiple swarm action result elements. Normally, swarm action result elements SHOULD be set and error_code MUST be set to 0 when response_type is 0x00. Swarm action result elements SHOULD NOT be set when error_code is 0x01. Detailed selection of error_code is introduced in Section 4.3;

```
Object {
    ppsp_tp_string_t           swarm_id;
    ppsp_tp_response_type_t    result;
    [ppsp_tp_peer_group_t      peer_group;]
}ppsp_tp_swarm_action_result_t;
```

A swarm action result element is the result information for a peer to request the tracker to have some actions towards the swarm. It contains a swarm identifier which globally indicates the swarm, the result for the peer of this action which it could be CONNECT ("JOIN" or "LEAVE"), FIND or STAT_REQPORT, and optionally one peer group element. The attribute result indicates the operation result of the corresponding request. When the response element is corresponding to the STAT_REPORT request, or the result attribute is set to 0x01, the peer group element SHOULD NOT be set.

3.4 PPSP-TP Message Element

PPSP-TP messages (requests or responses) are designed to have a similar structure with a root field named "PPSPTrackerProtocol" containing meta information and data pertaining to a request or a response.

The base type of PPSP-TP message is defined as follows:

```
Object {
        JSONValue PPSPTrackerProtocol = ppsp_tp_request  Request
                                      | ppsp_tp_response Response;
} ppsp_tp_message_root;
```

4 Protocol Specification: Encoding and Operation

PPSP-TP is a message-oriented request/response protocol. PPSP-TP messages use a text type encoding in JSON [RFC7159], which MUST be indicated in the Content-Type field in HTTP/1.1 [RFC2616], specifying the application/ppsp-tracker+json media type for all PPSP-TP request parameters and responses.

Implementations MUST support the "https" URI scheme [RFC2818] and Transport Layer Security (TLS) [RFC5246].

For deployment scenarios where Peer (Client) authentication is desired at the Tracker, HTTP Digest Authentication MUST be supported, with TLS Client Authentication as the preferred mechanism, if available.

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.

PPSP-TP uses the HTTP POST method to send parameters in requests to provide information resources that are the function of one or more of those input parameters. Input parameters are encoded in JSON in the HTTP entity body of the request.

The section describes the operation of the three types of Requests of PPSP-TP and provides some examples of usage.

4.1 Requests and Responses

4.1.1 CONNECT Request

This method is used when a peer registers to the system and/or requests swarm actions. The peer MUST properly set the Request type to CONNECT, generate and set the transaction_ids, set the peer_id and MUST include swarms the peer is interested in, followed by the corresponding action type and peer mode.

- o When a peer already possesses a content and agrees to share it to others, it should set the action type to the value JOIN, as well as set the peer mode to SEED during its start (or re-start) period.

- o When a peer makes a request to join a swarm to consume content, it should set the action type to the value JOIN, as well as set the peer mode to LEECH during its start (or re-start) period.

In the above cases, the peer can provide optional information on the addresses of its network interface(s), for example, the priority, type, connection and ASN.

When a peer plans to leave a previously joined swarm, it should set action type to LEAVE, regardless of the peer mode.

When receiving a well-formed CONNECT Request message, the Tracker MAY, when applicable, start by pre-processing 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, then proceed to register the peer in the service and perform the swarm actions requested. In case of success a Response message with a corresponding response value of SUCCESSFUL will be generated.

The valid sets of number of swarms whose action type is combined with peer mode for the CONNECT Request logic are enumerated in Table 6 (referring to the Tracker "per-Peer-ID" state machine in Section 2.3).

Swarm Number	peer_mode value	action value	Initial State	Final State	Request validity
1	LEECH	JOIN	START	TRACKING	Valid
1	LEECH	LEAVE	START	TERMINATE	Invalid
1	LEECH	LEAVE	TRACKING	TERMINATE	Valid
1 1	LEECH LEECH	JOIN LEAVE	START	TERMINATE	Invalid
1 1	LEECH LEECH	JOIN LEAVE	TRACKING	TRACKING	Valid
N	SEED	JOIN	START	TRACKING	Valid
N	SEED	JOIN	TRACKING	TERMINATE	Invalid
N	SEED	LEAVE	TRACKING	TERMINATE	Valid

Table 6: Validity of action combinations in CONNECT Request.

In the CONNECT Request multiple swarm action elements ppsp_tp_swarm_action_t could be contained. Each contains the request for action and the peer_mode of the peer. The peer_mode attribute MUST be set to the type of participation of the peer in the swarm (SEED or LEECH).

The CONNECT message MAY contain multiple peer_addr elements with attributes ip_address, port, priority and type (if PPSP-ICE [RFC5245] NAT traversal techniques are used), and optionally connection, asn and peer_protocol corresponding to each of the network interfaces the peer wants to advertise.

The element peer_num indicates to the tracker the number of peers to be returned in a list corresponding to the indicated properties, being ability_nat for NAT traversal (considering that PPSP-ICE NAT traversal techniques may be used), and optionally concurrent_links, online_time and upload_bandwidth_level for the preferred capabilities. If STUN-like function is enabled in the tracker, the response MAY include the peer reflexive address.

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

The Response MUST include peer_addr data of the requesting peer public IP address. If STUN-like function is enabled in the tracker, the peer_addr includes the attribute type with a value of REFLEXIVE, corresponding to the transport address "candidate" of the peer. The swarm_result MAY also include peer_addr data corresponding to the Peer IDs and public IP addresses of the selected active peers in the requested swarm. 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 of the referenced peer.

In case the peer_mode is SEED, the tracker responds with a SUCCESSFUL response and enters the peer information into the corresponding swarm activity. In case the peer_mode is LEECH (or if the peer Seeder includes a peer_num element in the request) the tracker will search and select an appropriate list of peers satisfying the conditions set by the requesting peer. The peer list returned 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 [RFC7285].

IMPLEMENTATION NOTE: If no peer_num attributes are present in the request the tracker MAY return a random sample from the peer population.

4.1.1.1 Example

The following example of a CONNECT Request corresponds to a peer that wants to start (or re-start) sharing its previously streamed contents (peerMode is of SEED).

Note for this case that the peer also requests from the Tracker an appropriate list of peers (PeerNum element) already active in the swarm, i.e., a list of 15 peers having STUN capabilities in terms of NAT. In the case of a Super-Node peer of an ISP, the CONNECT request would be similar but, optionally not including the peer_num element:

```
POST / HTTP/1.1
Host: tracker.example.com
Content-Length: 494
Content-Type: application/ppsp-tracker+json
Accept: application/ppsp-tracker+json
```


```
{
  "PPSPTrackerProtocol": {
    "version":              0x01;
    "request_type":         "CONNECT";
    "transaction_id"        "12345";
    "peer_id":              "656164657220";
    "connect":{
        "peer_num": {
            "peer_count":        15;
            "ability_nat":       "STUN";
            "concurrent_links": "NORMAL";
            "online_time": "NORMAL";
            "upload_bandwidth_level": "NORMAL";
        };
        "peer_addr": {
               "ip_address": {
                    "address_type":     "ipv4";
                    "address":          "192.0.2.2";
               };
               "port":          80;
               "priority":      1;
               "type":          "HOST";
               "connection":    "ETHER";
               "asn":           "45645";
        };
        "Swarm_action": {
            "swarm_id":       "1111";
            "action":         "JOIN";
            "peer_mode":      "SEED";
        };
        "Swarm_action": {
            "swarm_id":       "2222";
            "action":         "JOIN";
            "peer_mode":      "SEED";
        };
    };
  };
}
```


Another example of the message-body of a CONNECT Request corresponds to a peer (PeerMode is LEECH, meaning that the peer is not in possession of the content) requesting join to a swarm, in order to start receiving the stream, and providing optional information on the addresses of its network interface(s):


```
{
  "PPSPTrackerProtocol": {
    "version":                0x01;
    "request_type":           "CONNECT";
    "transaction_id"          "12345.0";
    "peer_id":                "656164657221";
    "connect":{
        "peer_num": {
            "peer_count":        5;
            "ability_nat":       "STUN";
            "concurrent_links": "NORMAL";
            "online_time":      "NORMAL";
            "upload_bandwidth_level": "NORMAL";
         };
         "peer_addr": {
               "ip_address": {
                    "address_type":     "ipv4";
                    "address":          "192.0.2.2";
               };
               "port":         80;
               "priority":     1;
               "type":         "HOST";
               "connection":   "ETHER";
               "asn":          "3256546";
         };
         "peer_addr": {
               "ip_address":{
                   "address_type":     "ipv6";
                   "address":          "2001:db8::2";
               };
               "port":         80;
               "priority":     2;
               "type":         "HOST";
               "connection":   "3G";
               "asn":          "34563456";
               "peer_protocol": "PPSP-PP";
         };
         "swarm_action": {
            "swarm_id":       "1111";
            "action":         "JOIN";
            "peer_mode":      "LEECH";
         };
    };
}
```

The next example of a CONNECT Request corresponds to a peer "leaving" a previously joined swarm and requesting join to a new swarm. This is the typical example of a user watching a live channel but then deciding to switch to a different one:

```
{
  "PPSPTrackerProtocol": {
    "version":              0x01;
    "request_type":         "CONNECT";
    "TransactionID":        "12345";
    "peer_id":              "656164657221";
    "connect":{
        "peer_num": {
            "peer_count":         5;
            "ability_nat":        "STUN";
            "concurrent_links":  "NORMAL";
            "online_time":        "NORMAL";
            "upload_bandwidth_level": "NORMAL";
        };
        "swarm_action": {
            "swarm_id":           "1111";
            "action":             "LEAVE";
            "peer_mode":          "LEECH";
        };
        "swarm_action": {
            "swarm_id":           "2222";
            "action":             "JOIN";
            "@peer_mode":         "LEECH";
        };
     };
}
```

The next example illustrates the Response for the previous example of CONNECT Request where the peer requested two swarm actions and not more than 5 other peers, receiving from the Tracker a peer list with only 2 two other peers in the swarm "2222":

```
HTTP/1.1 200 OK
Content-Length: 1342
Content-Type: application/ppsp-tracker+json

{
  "PPSPTrackerProtocol": {
    "version":               0x01;
    "response_type":         0x00;
    "error_code":            0;
    "transaction_id":        "12345";
    "peer_addr": {
        "ip_address": {
            "address_type":     "ipv4";
            "address":          "198.51.100.1";
        };
        "port":          80;
        "priority":      1;
        "asn":           "64496";
     };
     "swarm_result": {
        "swarm_id":        "2222";
        "result":          0x00;
        "peer_group": {
            "peer_info": {
                "peer_id":    "956264622298";
                "peer_addr": {
                    "ip_address": {
                        "address_type":     "ipv4";
                        "address":          "198.51.100.22";
                    };
                    "port":          80;
                    "priority":      2;
                    "type":          "REFLEXIVE";
                    "connection":    "ADSL";
                    "asn":           "64496";
                    "peer_protocol": "PPSP-PP"
                };
            };
            "peer_info": {
                "peer_id":    "3332001256741";
                "peer_addr": {
                    "ip_address": {
                        "address_type":     "ipv4";
                        "address":          "198.51.100.201";
                    };
                    "port":          80;
                    "priority":      2;
                    "type":          "REFLEXIVE";
```

```
                          "connection":    "ADSL";
                          "asn":           "64496";
                          "peer_protocol": "PPSP-PP";
                      };
                  };
             };
         };
      }
```


4.1.2 FIND Request

This method allows peers to request to the tracker, whenever needed, a new peer list for the swarm.

The FIND request MAY include a peer_number element to indicate to the tracker the maximum number of peers to be returned in a list corresponding to the indicated conditions set by the requesting peer, being ability_nat for NAT traversal (considering that PPSP-ICE NAT traversal techniques may be used), and optionally concurrent_links, online_time and upload_bandwidth_level for the preferred capabilities.

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 SUCCESSFUL will be generated and the tracker will search out the list of peers for the swarm and select an appropriate peer list satisfying the conditions set by the requesting peer. The peer list returned MUST contain the Peer IDs and the corresponding IP Addresses.

The tracker may take peers’ ability and popularity of the requested content into consideration. For example, the tracker could select peers with higher ability than the current peers that provide the content if the content is relatively popular (see Section 5.1.1); and the tracker could also select peers with lower ability than the current peers that provide the content when the content is relatively uncommon. The tracker may take network location information into consideration as well, to express network topology preferences or Operators’ policy preferences, with regard to the possibility of connecting with other IETF efforts such as ALTO [RFC7285].

The Response MUST include peer_group element that includes the public IP addresses of the selected active peers in the swarm.

The peer_group list MUST contain the Peer IDs and the corresponding IP Addresses, 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 of the referenced peer.

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.

The response MAY also include peer_addr element that includes the requesting peer public IP address. If STUN-like function is enabled in the tracker, the peer_addr includes the attribute type with a value of REFLEXIVE, corresponding to the transport address "candidate" of the peer.

IMPLEMENTATION NOTE: If no peer_num attributes are present in the request the tracker MAY return a random sample from the peer population.

4.1.2.1 Example

An example of the message-body of a FIND Request, where the peer requests to the Tracker an list of not more than 5 peers in the swarm "1111" conforming to the characteristics expressed (concurrent links, online time, and upload bandwidth level) is the following:


```
{
  "PPSPTrackerProtocol": {
      "version":             0x01;
      "request_type":        "FIND";
      "transaction_id":      "12345";
      "peer_id":             "656164657221";
      "swarm_id":            "1111";
      "peer_num": {
          "peer_count":         5;
          "ability_nat":        "STUN";
          "concurrent_links":  "HIGH";
          "online_time":        "NORMAL";
          "upload_bandwidth_level":    "NORMAL";
      };
  };
}
```


An example of the message-body of a Response for the above FIND Request, including the requesting peer public IP address information, is the following:

```
      {
        "PPSPTrackerProtocol": {
            "version":             0x01;
            "response_type":       0x00;
            "error_code":          0;
            "transaction_id":      "12345";
            "swarm_result": {
                "swarm_id":        "1111";
                "result":          0x00;
                "peer_group": {
                    "peer_info": {
                        "peer_id":    "656164657221";
                        "peer_addr": {
                            "ip_address": {
                                "address_type":     "ipv4";
                                "address":          "198.51.100.1";
                            };
                            "port":          80;
                            "priority":      1;
                            "type":          "REFLEXIVE";
                            "connection":    "3G";
                            "asn":           "64496";
                        };
                    };
                    "peer_info": {
                        "peer_id":    "956264622298";
                        "peer_addr": {
                            "ip_address": {
                                "address_type":     "ipv4";
                                "address":          "198.51.100.22";
                            };
                            "port":          80;
                            "priority":      1;
                            "type":          "REFLEXIVE";
                            "connection":    "3G";
                            "asn":           "64496";
                        };
                    };
                    "peer_info": {
                        "peer_id":    "3332001256741";
                        "peer_addr": {
                            "ip_address": {
                                "address_type":     "ipv4";
                                "address":          "198.51.100.201";
                            };
                            "port":          80;
                            "priority":      1;
                            "type":          "REFLEXIVE";
```

```
                            "connection":    "3G";
                            "asn":           "64496";
                        };
                    };
                };
            };
        };
      }
```

4.1.3 STAT_REPORT Request

This method allows peers to send status and statistic data to trackers. The method is initiated by the peer, periodically while active.

The peer MUST set the request_type to "STAT_REPORT", set the peer_id with the identifier of the peer, and generate and set the transaction_id.

The report MAY include multiple statistics elements describing several properties relevant to a specific swarm. These properties can be related with stream statistics and peer status information, including uploaded_bytes, downloaded_bytes, available_bandwidth and etc.

Other properties may be defined (guidelines in Section 7.1) related for example, with incentives and reputation mechanisms. In case no Statistics Group is included, the STAT_REPORT is used as a "keep-alive" message to prevent the tracker from de-registering the peer when "track timer" expires.

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

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

4.1.3.1 Example

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


```
{
  "PPSPTrackerProtocol": {
      "version":             0x01;
      "request_type":        "STAT_REPORT";
      "transaction_id":      "12345";
      "peer_id":             "656164657221";
      "stat_report": {
          "type":  "STREAM_STATS";
          "Stat": {
                "swarm_id":               "1111";
                "uploaded_bytes":         512;
                "downloaded_bytes":       768;
                "available_bandwidth":    1024000;
          };
      };
  };
}
```


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


```
{
  "PPSPTrackerProtocol": {
      "version":              0x01;
      "response_type":        0x00;
      "error_code":           0;
      "transaction_id":       "12345";
      "swarm_result": {
          "swarm_id":     "1111";
          "result":       0x00;
      };
  };
}
```

4.2 Response element in response Messages

Table 7 indicates the response type and corresponding semantics.

Response Type	Semantics
0x00	SUCCESSFUL
0x01	FAILED

Table 7: Semantics for the Value of Response Type.

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 transaction_id of the corresponding request.

In CONNECT Request: returns information about the successful registration of the peer and/or of each swarm action requested. MAY additionally return the list of peers corresponding to the action attribute requested.

In FIND Request: returns the list of peers corresponding to the requested scope.

In STAT_REPORT Request: confirms the success of the requested operation.

FAILED: indicates that the request has not been processed properly.

4.3 Error and Recovery conditions

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

The transaction_id on a Request can be reused if and only if all of the content is identical, including 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 transaction_id.

Error situations resulting from the Normal Operation or from abnormal conditions (Section 2.3.2) MUST be responded with response_type set to 0x01 and 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) error_code with an empty message-body (no peer_addr and swarm_result attributes).

- If the version number of the protocol is for a version the receiver does not supports, the receiver MUST respond with a 401 (Unsupported Version Number) error_code with an empty message-body (no peer_addr and swarm_result attributes).

- In the PEER REGISTERED and TRACKING states of the tracker, certain requests are not allowed (Section 2.3.2). The tracker MUST respond with a 402 (Forbidden) error_code with an empty message-body (no peer_addr and swarm_result attributes).

- If the tracker is unable to process a Request message due to unexpected condition, it SHOULD respond with a 403 (Internal Server Error) response with an empty message-body (no peer_addr and swarm_result attributes).

- If the tracker is unable to process a Request message for being in an overloaded state, it SHOULD respond with a 404 (Service Unavailable) error_code with an empty message-body (no peer_addr and swarm_result attributes).

- If authentication is required for the peer to make the request, the tracker SHOULD respond with a 405 (Authentication Required) error_code with an empty message-body (no peer_addr and swarm_result attributes).

4.4 Parsing of Unknown Fields in Message-body

This document only details object fields used by this specification. Extensions may include additional fields within JSON objects defined in this document. PPSP-TP implementations MUST ignore unknown fields when processing PPSP-TP messages.

5 Operations and Manageability

This section provides the operational and managements aspects that are required to be considered in implementations of the PPSP Tracker Protocol. These aspects follow the recommendations expressed in RFC 5706 [RFC5706].

5.1 Operational Considerations

The PPSP-TP provides communication between trackers and peers and is conceived as a "client-server" mechanism, allowing the exchange of information about the participant peers sharing multimedia streaming contents.

The "serving" component, i.e., the Tracker, is a logical entity that can be envisioned as a centralized service (implemented in one or more physical nodes), or a fully distributed service.

The "client" component can be implemented at each peer participating in the streaming of contents.

5.1.1 Installation and Initial Setup

Content providers wishing to use PPSP for content distribution should setup at least a PPSP Tracker and a service Portal (public web server) to publish links of the content descriptions, for access to their on-demand or live original contents sources. Content/Service providers should also create conditions to generate Peer IDs and any required security certificates, as well as Chunk IDs and Swarm IDs for each streaming content. The configuration processes for the PPSP Tracking facility, the service Portal and content sources are not standardized, enabling all the flexibility for implementers.

The Swarm IDs of available contents, as well as the addresses of the PPSP Tracking facility, can be distributed to end-users in various ways, but it is common practice to include both the Swarm ID and the corresponding PPSP Tracker addresses (as URLs) in the MPD of the content, which is obtainable (a link) from the service Portal.

The available contents could have different importance attribute values to indicate whether the content is popular or not. However, it is a totally implementation design and outside of this specification. For example, the importance attribute values of the contents could be set by content providers when distributing them or could be determined by the tracker based on the statistics of the requests from the peers that request the content. The tracker could set a upper threshold to decide that the content is popular enough when the importance attribute value is higher than the upper threshold. And the tracker could also set a lower threshold to decide that the content is uncommon enough when the importance attribute value is lower than the lower threshold.

End-users browse and search for the desired contents in the service Portal, selecting by clicking the links of the corresponding MPDs. This action typically launches the Client Media Player (with PPSP awareness) which will then, using PPSP-TP, contact the PPSP Tracker to join the corresponding swarm and obtain the transport addresses of other PPSP peers in order to start streaming the content.

5.1.2 Migration Path

Since there is no previous standard protocol providing similar functionality, this specification does not detail a migration path.

5.1.3 Requirements on Other Protocols and Functional Components

For security reasons, when using PPSP Peer protocol with PPSP-TP, the mechanisms described in Section 6.1 should be observed.

5.1.4 Impact on Network Operation

As the messaging model of PPSP-TP aligns with HTTP protocol and the semantics of its messages, the impact on Network Operation is similar to using HTTP.

5.1.5 Verifying Correct Operation

The correct operation of PPSP-TP can be verified both at the Tracker and at the peer by logging the behavior of PPSP-TP. Additionally, the PPSP Tracker collects the status of the peers including peer’s activity, and such information can be used to monitor and obtain the global view of the operation.

5.2 Management Considerations

The management considerations for PPSP-TP are similar to other solutions using HTTP for large-scale content distribution. The PPSP Tracker can be realized by geographically distributed tracker nodes or multiple server nodes in a data center. As these nodes are akin to WWW nodes, their configuration procedures, detection of faults, measurement of performance, usage accounting and security measures can be achieved by standard solutions and facilities.

5.2.1 Interoperability

Interoperability refers to allowing information sharing and operations between multiple devices and multiple management applications. For PPSP-TP, distinct types of devices host PPSP-TP servers (Trackers) and clients (Peers). Therefore, support for multiple standard schema languages, management protocols and information models, suited to different purposes, was considered in the PPSP-TP design. Specifically, management functionality for PPSP-TP devices can be achieved with Simple Network Management Protocol (SNMP) [RFC3410], syslog [RFC5424] and NETCONF [RFC6241].

5.2.2 Management Information

PPSP Trackers may implement SNMP management interfaces, namely the Application Management MIB [RFC2564] without the need to instrument the Tracker application itself. The channel, connections and transaction objects of the the Application Management MIB can be used to report the basic behavior of the PPSP Tracker service.

The Application Performance Measurement MIB (APM-MIB) [RFC3729] and the Transport Performance Metrics MIB (TPM-MIB) [RFC4150] can be used with PPSP-TP, providing adequate metrics for the analysis of performance for transaction flows in the network, in direct relationship to the transport of PPSP-TP.

The Host Resources MIB [RFC2790] can be used to supply information on the hardware, the operating system, and the installed and running software on a PPSP Tracker host.

The TCP-MIB [RFC4022] can additionally be considered for network monitoring.

Logging is an important functionality for PPSP-TP server (Tracker) and client (Peer), done via syslog [RFC5424].

5.2.3 Fault Management

As PPSP Tracker failures can be mainly attributed to host or network conditions, the facilities previously described for verifying the correct operation of PPSP-TP and the management of PPSP Tracker servers, appear sufficient for PPSP-TP fault monitoring.

5.2.4 Configuration Management

PPSP Tracker deployments, when realized by geographically distributed tracker nodes or multiple server nodes in a data center, may benefit from a standard way of replicating atomic configuration updates over a set of server nodes. This functionality can be provided via NETCONF [RFC6241].

5.2.5 Accounting Management

PPSP-TP implementations, namely for content provider environments, can benefit from accounting standardization efforts as defined in [RFC2975], in terms of resource consumption data, for the purposes of capacity and trend analysis, cost allocation, auditing, and billing.

5.2.6 Performance Management

Being transaction-oriented, PPSP-TP performance, in terms of availability and responsiveness, can be measured with the facilities of the APM-MIB [RFC3729] and the TPM-MIB [RFC4150].

5.2.7 Security Management

Standard SNMP notifications for PPSP Tracker management and syslog messages [RFC5424] can be used, to alert operators to the conditions identified in the security considerations (Section 6).

The statistics collected about the operation of PPSP-TP can be used for detecting attacks, such as the receipt of malformed messages, messages out of order, or messages with invalid timestamps.

6 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 guarantee 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].

6.1 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 SHOULD be received from authorized peers. For that purpose a peer SHOULD 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 [RFC6749] with bearer token, which provides the peer with the information required to successfully utilize an access token to make protected requests to the tracker [RFC6750].

6.2 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.

6.3 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.

6.4 Pro-incentive parameter trustfulness

Property types for STAT_REPORT messages may consider additional pro-incentive parameters (guidelines for extension in Section 7), 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. 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.

One such solution could be a reputation-incentive mechanism, based on the notions of reputation, social awareness and fairness. The mechanism would promote cooperation among participants (via each peer’s reputation) based on the history of past transactions, such as, count of chunk requests (sent, received) in a swarm, contribution time of the peer, cumulative uploaded and downloaded content, JOIN and LEAVE timestamps, attainable rate, etc.

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].

7 Guidelines for Extending PPSP-TP

Extension mechanisms allow designers to add new features or to customize existing features of a protocol for different operating environments [RFC6709].

Extending a protocol implies either the addition of features without changing the protocol itself or the addition of new elements creating new versions of an existing schema and therefore new versions of the protocol.

In PPSP-TP it means that an extension MUST NOT alter an existing protocol schema as the changes would result in a new version of an existing schema, not an extension of an existing schema, typically non-backwards-compatible.

Additionally, a designer MUST remember that extensions themselves MAY also be extensible.

Extensions MUST adhere to the principles described in this section in order to be considered valid.

Extensions MAY be documented as Internet-Draft and RFC documents if there are requirements for coordination, interoperability, and broad distribution.

Extensions need not be published as Internet-Draft or RFC documents if they are intended for operation in a closed environment or are otherwise intended for a limited audience.

7.1 Forms of PPSP-TP Extension

In PPSP-TP two extension mechanisms can be used: a Request-Response Extension or a Protocol-level Extension.

o Request-Response Extension: Adding elements or attributes to an existing element mapping in the schema is the simplest form of extension. This form should be explored before any other. This task can be accomplished by extending an existing element mapping.

  For example, an element mapping for the Statistics Group can be extended to include additional elements needed to express status information about the activity of the peer, such as OnlineTime for the Stat element.

o Protocol-level Extension: If there is no existing element mapping that can be extended to meet the requirements and the existing PPSP-TP Request and Response message structures are insufficient, then extending the protocol should be considered in order to define new operational Requests and Responses.

  For example, to enhance the level of control and the granularity of the operations, a new version of the protocol with new messages (JOIN, DISCONNECT), a retro-compatible change in semantics of an existing CONNECT Request/Response and an extension in STAT_REPORT could be considered.

  As illustrated in Figure 6, the peer would use an enhanced CONNECT Request to perform the initial registration in the system. Then it would JOIN a first swarm as SEEDER, later JOIN a second swarm as LEECH, and then DISCONNECT from the latter swarm but keeping as SEEDER for the first one. When deciding to leave the system, the peer DISCONNECTs gracefully from it:

```
                 +--------+                      +---------+
                 |  Peer  |                      | Tracker |
                 +--------+                      +---------+
                     |                                |
                     |--CONNECT--------------------->|
                     |<--------------------------OK--|
                     |--JOIN(swarm_a;SEED)---------->|
                     |<--------------------------OK--|
                     :                                :
                     |--STAT_REPORT(activity)------->|
                     |<--------------------------Ok--|
                     :                                :
                     |--JOIN(swarm_b;LEECH)--------->|
                     |<-----------------OK+PeerList--|
                     :                                :
                     |--STAT_REPORT(ChunkMap_b)----->|
                     |<--------------------------Ok--|
                     :                                :
                     |--DISCONNECT(swarm_b)--------->|
                     |<--------------------------Ok--|
                     :                                :
                     |--STAT_REPORT(activity)------->|
                     |<--------------------------Ok--|
                     :                                :
                     |--DISCONNECT------------------>|
                     |<---------------------Ok(BYE)--|
```

Figure 6: Example of a session for a PPSP-TP extended version.

7.2 Issues to Be Addressed in PPSP-TP Extensions

There are several issues that all extensions should take into consideration.

- Overview of the Extension: It is RECOMMENDED that extensions to PPSP-TP have a protocol overview section that discusses the basic operation of the extension. The most important processing rules for the elements in the message flows SHOULD also be mentioned.

- Backward Compatibility: One of the most important issues to consider is whether the new extension is backward compatible with the base PPST-TP.

- Syntactic Issues: Extensions that define new Request/Response methods SHOULD use all capitals for the method name, keeping with a long-standing convention in many protocols, such as HTTP. Method names are case sensitive in PPSP-TP. Method names SHOULD be shorter than 16 characters and SHOULD attempt to convey the general meaning of the Request or Response.

- Semantic Issues: PPSP-TP extensions MUST clearly define the semantics of the extensions. Specifically, the extension MUST specify the behaviors expected from both the Peer and the Tracker in processing the extension, with the processing rules in temporal order of the common messaging scenario.

  Processing rules generally specify actions to be taken on receipt of messages and expiration of timers.

  The extension SHOULD specify procedures to be taken in exceptional conditions that are recoverable. Handling of unrecoverable errors does not require specification.

- Security Issues: Being security an important component of any protocol, designers of PPSP-TP extensions need to carefully consider security requirements, namely authorization requirements and requirements for end-to-end integrity.

- Examples of Usage: The specification of the extension SHOULD give examples of message flows and message formatting and include examples of messages containing new syntax. Examples of message flows should be given to cover common cases and at least one failure or unusual case.

8 IANA Considerations

8.1 MIME Type Registry

This document defines registry for application/ppsp-tracker+json media types.

Type name: application

Subtype name: ppsp-tracker+json

Required parameters: n/a

Optional parameters: n/a

Encoding considerations: Encoding considerations are identical to those specified for the "application/json" media type. See [RFC7159].

Security considerations: See Section 6.

Interoperability considerations: This document specifies format of conforming messages and the interpretation thereof.

Published specification: This document.

Applications that use this media type: PPSP trackers and peers either stand alone or embedded within other applications.

Additional information:

Magic number(s): n/a

File extension(s): This document uses the MIME type to refer to protocol messages, therefore it does not requires a file extension.

Macintosh file type code(s): n/a

Person & email address to contact for further information: See Authors’ Addresses section.

Intended usage: COMMON

Restrictions on usage: none

Author: See Authors’ Addresses section.

Change controller: IESG (iesg@ietf.org)

8.2 PPSP Tracker Protocol Version Number Registry

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

9 Acknowledgments

The authors would like to thank many people for for their help and comments, particularly: Zhang Yunfei, Liao Hongluan, Roni Even, Dave Cottlehuber, 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, Riccardo Petrocco and Arno Bakker.

Rui Cruz, Mario Nunes and Joao Taveira were partially supported by the SARACEN project [SARACEN], a research project of the European Union 7th Framework Programme (contract no. ICT-248474).

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, the European Commission, Huawei or China Mobile.

10 References

10.1 Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

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

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

10.2 Informative References

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

[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.

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

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

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

[RFC3410] Case, J., Mundy, R., Partain, D., and B. Stewart, "Introduction and Applicability Statements for Internet-Standard Management Framework", RFC 3410, December 2002.

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

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

[RFC4022] Raghunarayan, R., Ed., "Management Information Base for the Transmission Control Protocol (TCP)", RFC 4022, March 2005.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, July 2005.

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

[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.

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

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

[RFC5952] Kawamura, S. and M. Kawashima, "A Recommendation for IPv6 Address Text Representation", RFC 5952, August 2010.

[Fielding] Fielding, R., "Architectural Styles and the Design of Network-based Software Architectures", University of California, Irvine, Dissertation 2000, 2000.

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

[Contracts] Piatek, M., Venkataramani, A., Yang, R., Zhang, D. and A. Jaffe, "Contracts: Practical Contribution Incentives for P2P Live Streaming", in NSDI ’10: USENIX Symposium on Networked Systems Design and Implementation, April 2010.

Appendix A. Revision History

```
   -00      2013-02-14 Initial version.
   -01      2013-02-14 Minor revision.
   -02      2013-10-21 Minor revision.
   -03      2013-12-31 Major revision
       +    Introduced a generalization of the protocol specification
            using a C-style notation.
       -    removed all examples of protocol message encoding in XML
   -04      2014-07-01 Minor Revision
       -    removed Appendix referencing the use of HTTP
       +    refined the presentation language specification to include
            protocol elements definitions.
   -05      2014-07-04 Minor Revision
   -06      2014-10-27 Minor Revision
   -07      2014-12-12 Major Revision
       +    introduced a text-based (JSON) protocol encoding with
            examples for all the messages
       +    corrections in the specifications of protocol elements
       +    section 5 specification of protocol elements semantics
       +    introduced a IANA MIME Type registry
   -08      (Current)  Major Revision
       *    merge sections 5 and 4 with section 3; renumbered all other
       +    refined the protocol elements definitions for consistency
            with the JSON data structures
       +    revised protocol messages encoding examples
       +    additional IANA registry for protocol version
       *    editorial corrections
```

Authors’ Addresses

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

Rachel Huang
Huawei
Email: rachel.huang@huawei.com

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
Email: denglingli@chinamobile.com

Gu Yingjie
Email: guyingjie@gmail.com

PPSP A. Bakker
Internet-Draft Vrije Universiteit Amsterdam
Intended status: Standards Track R. Petrocco
Expires: June 1, 2015 V. Grishchenko
Technische Universiteit Delft
November 28, 2014

Peer-to-Peer Streaming Peer Protocol (PPSPP)
draft-ietf-ppsp-peer-protocol-12

Abstract

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

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

1.1. Purpose

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

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

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

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

PPSPP is also flexible and extensible in the mechanisms it uses to promote client contribution and prevent freeriding, that is, how to deal with peers that only download content but never upload to others. It also allows different schemes for chunk addressing and content integrity protection, if the defaults are not fit for a particular use case. In addition, it can work with different peer discovery schemes, such as centralized trackers or fast Distributed Hash Tables [JIM11]. Finally, in this default setup, PPSPP maintains only a small amount of state per peer. A reference implementation of PPSPP over UDP is available [SWIFTIMPL].

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

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

1.2. Requirements Language

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

1.3. Terminology

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

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

content

Either a live transmission or a pre-recorded multimedia file.

chunk
The basic unit in which the content is divided. E.g. a block of N kilobyte. A chunk may be of variable size.

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

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

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

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

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

content integrity protection scheme
Scheme for protecting the integrity of the content while it is being distributed via the peer-to-peer network. That is, methods for receiving peers to detect whether a requested chunk has been modified, either maliciously by the sending peer or accidentally in transit.

hash
The result of applying a cryptographic hash function, more specifically a modification detection code (MDC) [HAC01], such as SHA-256 [FIPS180-4], to a piece of data.

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

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

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

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

swarm ID
: Unique identifier for a swarm of peers, in PPSPP a sequence of bytes. For video-on-demand with content integrity protection enabled, the identifier is the so-called root hash of a Merkle hash tree over the content. For live streaming, the swarm ID is a public key.

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

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

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

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

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

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

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

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

2. Overall Operation

The basic unit of communication in PPSPP is the message. Multiple messages are multiplexed into a single datagram for transmission. A datagram (and hence the messages it contains) will have different representations on the wire depending on the transport protocol used (see Section 8).

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

2.1. Example: Joining a Swarm

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

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

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

2.2. Example: Exchanging Chunks

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

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

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

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

2.3. Example: Leaving a Swarm

To leave a swarm in a graceful way, peer A sends a specific HANDSHAKE message to all its peers (see Section 8.4) and deregisters from the tracker following the (PPSP) tracker protocol. Peers receiving the datagram should remove A from their current peer list. If A crashes ungracefully, peers should remove A from their peer list when they detect it no longer sends messages (see Section 3.12).

3. Messages

No error codes or responses are used in the protocol; absence of any response indicates an error. Invalid messages are discarded, and further communication with the peer SHOULD be stopped. The rationale is that it is sufficient to classify peers as either good or bad and only use the good ones. A good peer is a peer that responds with chunks; a peer that does not respond, or does not respond in time is classified as bad. The idea is that in PPSPP the content is available from multiple sources (unlike HTTP), so a peer should not invest too much effort in trying to obtain it from a particular source. This classification in good or bad allows a peer to deal with slow, crashed and (silent) malicious peers.

Multiple messages MUST be multiplexed into a single datagram for transmission. Messages in a single datagram MUST be processed in the strict order in which they appear in the datagram. If an invalid message is found in a datagram, the remaining messages MUST be discarded.

For the sake of simplicity, one swarm of peers deals with one content file or stream only. There is a single division of the content into chunks that all peers in the swarm adhere to, determined by the content publisher. Distribution of a collection of files can be done either by using multiple swarms or by using an external storage mapping from the linear byte space of a single swarm to different files, transparent to the protocol. In other words, the audio/video container format used is outside the scope of this document.

3.1. HANDSHAKE

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

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

(b) the chunk size used,

(c) the chunk addressing method used,

(d) the content integrity protection method used, and

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

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

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

The payload of the HANDSHAKE message contains a sequence of protocol options. The protocol options encode the swarm metadata just described to enable an end-to-end check whether the peers are in the right swarm, and a number of per-peer configuration parameters. The complete set of protocol options are specified in Section 7. The HANDSHAKE message also contains a channel ID, for multiplexing communication and security, see Section 3.11 and Section 13.1. A HANDSHAKE message MUST always be the first message in a datagram.

3.1.1. Handshake Procedure

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

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

   * A channel ID, chanP, randomly chosen as specified in Section 13.1.

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

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

   This first datagram MUST be prefixed with the (destination) channel ID 0, see Section 3.11. Hence, the datagram contains two channel IDs: the destination channel ID prefixed to the datagram, and the channel ID chanP included in the HANDSHAKE message inside the datagram. This datagram MAY also contain some minor additional payload, e.g. HAVE messages to indicate P's current progress, but MUST NOT include any heavy payload (defined in Section 1.3), such as a DATA message Allowing minor payload minimizes the number of initialization round-trips, thus improving time-till-playback. Forbidding heavy payload prevents an amplification attack (see Section 13.1.)

2. The receiving peer Q checks the HANDSHAKE message from peer P. If any check by Q fails, Q MUST NOT send a HANDSHAKE (or any other) message back, as the message from P may have been spoofed (see Section 13.1). Only if P and Q are in the same swarm, and Q is interested in communicating with P, Q MUST a datagram to P that starts with a HANDSHAKE message. This reply HANDSHAKE MUST contain:

   * A channel ID, chanQ, randomly chosen as specified in Section 13.1.

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

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

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

3. The initiating peer P checks the reply datagram from Q. If the reply datagram is not prefixed with (destination) channel ID chanP, peer P MUST discard the datagram. P SHOULD continue to process datagrams from Q that do meet this requirement. This check prevents interference by spoofing, see Section 13.1. If P’s channel ID is echoed correctly, the initiator P knows that the addressed peer Q really responds.

4. Next, peer P checks the HANDSHAKE message in the datagram from Q. If any check by P fails, or P is no longer interested in communicating with Q, P MAY send a HANDSHAKE message to inform Q it will cease communication. This closing HANDSHAKE message MUST contain an all 0-zeros channel ID and a list of protocol options. The list MUST be either empty or contain the maximum version number peer P supports, following the Min/max versioning scheme defined in [RFC6709], Section 4.1. The datagram containing this closing HANDSHAKE message MUST be prefixed with (destination) channel ID chanQ. Peer P MAY also simply cease communication.

5. If addressed peer Q does not respond to initiating peer P’s first datagram, peer P MAY resend that datagram, until peer Q is

considered dead, according to the rules specified in Section 3.12.

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

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

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

3.2. HAVE

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

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

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

3.3. DATA

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

3.4. ACK

ACK messages MUST be sent to acknowledge received chunks if PPSPP is run over an unreliable transport protocol. ACK messages MAY be sent if a reliable transport protocol is used. In the former case, a receiving peer that has successfully checked the integrity of a chunk, or interval of chunks C MUST send an ACK message containing a chunk specification for C. As LEDBAT is used, an ACK message MUST contain the one-way delay, computed from the peer’s current system time received in the DATA message. A peer MAY delay sending ACK messages as defined in the LEDBAT specification.

3.5. INTEGRITY

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

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

3.6. SIGNED_INTEGRITY

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

3.7. REQUEST

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

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

When live streaming via a push model, a peer receiving REQUESTs also MAY send some other chunks in case it runs out of requests or for some other reason. In that case the only purpose of REQUEST messages is to provide hints and coordinate peers to avoid unnecessary data retransmission.

3.8. CANCEL

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

3.9. CHOKE and UNCHOKE

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

3.10. Peer Address Exchange

3.10.1. PEX_REQ and PEX_RES Messages

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

PPSPP supports peer-address exchange on the Internet and in benign private networks, as an OPTIONAL feature (not mandatory to implement) under certain conditions. The general mechanism works as follows. To obtain some peer addresses a peer A MAY send a PEX_REQ message to peer B. Peer B MAY respond with one or more PEX_REScert messages. Logically, a PEX_REScert reply message contains the address of a single peer Ci. The address in the PEX_REScert message MUST be of a peer B has exchanged messages with in the last 60 seconds to guarantee liveliness. Upon receipt, peer A may contact any or none of the returned peers Ci. Alternatively, peers MAY ignore PEX_REQ and PEX_REScert messages if uninterested in obtaining new peers or because of security considerations (rate limiting) or any other reason. The PEX messages can be used to construct a dedicated tracker peer.

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

1. Peer transport addresses must be relatively stable.

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

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

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

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

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

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

3.11. Channels

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

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

```
_________    _________          _________
|       |    |       |          |       |
| Swarm |    | Swarm |          | Swarm |
|  Mgr  |    |   A   |          |   B   |
|_______|    |_______|          |_______|
    |            |                /   \
    |            |               /     \
____|____    ____|____    ______/__    _\_______
|       |    |       |    |       |    |       |
| Chan  |    | Chan  |    | Chan  |    | Chan  |
|   0   |    |  481  |    |  836  |    |  372  |
|_______|    |_______|    |_______|    |_______|
    |            |            |            |
    |            |            |            |
____|____________|____________|____________|____
|                                              |
|                     UDP                      |
|                  port 6778                   |
|______________________________________________|
```

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

Figure 1

3.12. Keep Alive Signalling

A peer SHOULD send a "keep alive" message periodically to each peer it is interested in, but has no other messages to send to them at present. The goal of the keep alives is to keep a signaling channel open to peers that are of interest. Which peers those are is determined by a policy that decides which peers are of interest now and in the near future. This document does not prescribe a policy, but examples of interesting peers are: (a) peers that have chunks on offer that this client needs, or (b) peers that currently do not have interesting chunks on offer (because they are still downloading themselves, or in live streaming), but gave good performance in the past. When these peers have new chunks to offer, the peer that kept a signaling channel open can use them again. Periodically sending "keep alive" messages prevents other peers declaring the peer dead. A guideline for declaring a peer dead when using UDP consists of a 3 minute delay since that last packet has been received from that peer, and at least 3 datagrams were sent to that peer during the same period. When a peer is declared dead, the channel to it is closed, no more messages will be sent to that peer and the local administration about the peer is discarded. Busy servers can force idle clients to disconnect by not sending keep alives. PPSPP does not define an explicit message type for "keep alive" messages. In the PPSP-over-UDP encapsulation they are implemented as simple datagrams consisting of a 4-byte channel ID only, see Section 8.3 and Section 8.4.

4. Chunk Addressing Schemes

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

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

4.1. Start-End Ranges

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

4.1.1. Chunk Ranges

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
~                     Start chunk (32 or 64)                    ~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
~                     End chunk (32 or 64)                      ~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

4.1.2. Byte Ranges

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Start byte offset (64)                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     End byte offset (64)                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

4.2. Bin Numbers

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

In bin addressing, the smallest binary interval is a single chunk (e.g. a block of bytes which may be of variable size), the largest interval is a complete range of 2**63 chunks. In a novel addition to the classical scheme, these intervals are numbered in a way which lays them out into a vector nicely, which is called bin numbering, as follows. Consider an chunk interval of width W. To derive the bin numbers of the complete interval and the subintervals, a minimal balanced binary tree is built that is at least W chunks wide at the base. The leaves from left-to-right correspond to the chunks 0..W-1 in the interval, and have bin number I*2 where I is the index of the chunk (counting beyond W-1 to balance the tree). The bin number of higher level nodes P in the tree is calculated as follows:

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

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

```
                   7
                  / \
                /     \
              /         \
            /             \
          3                 11
         / \               / \
        /   \             /   \
       /     \           /     \
      1       5         9       13
     / \     / \       / \     / \
    0   2   4   6     8   10  12   14

    C0  C1  C2  C3    C4  C5  C6   C7
```

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

Figure 2

So bin 7 represents the complete interval, bin 3 represents the interval of chunk C0..C3, bin 1 represents the interval of chunks C0 and C1, and bin 2 represents chunk C1. The special numbers 0xFFFFFFFF (32-bit) or 0xFFFFFFFFFFFFFFFF (64-bit) stands for an empty interval, and 0x7FFF...FFF stands for "everything".

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
~                     Bin number (32 or 64)                     ~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

4.3. In Messages

4.3.1. In HAVE Messages

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

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

Implementation note:

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

4.3.2. In ACK Messages

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

5. Content Integrity Protection

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

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

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

5.1. Merkle Hash Tree Scheme

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

The Merkle hash tree of a content file 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 SHA-256, the hashes of all the chunks of the content are calculated. Next, a binary tree of sufficient height is created. Sufficient height means that the lowest level in the tree has enough nodes to hold all chunk hashes in the set, as with bin numbering. The figure below shows the tree for a content file consisting of 7 chunks. As before with the content addressing scheme, the leaves of the tree correspond to a chunk and in this case are assigned the hash of that chunk, starting at the left-most leaf. As the base of the tree may be wider than the number of chunks, any remaining leaves in the tree are assigned an empty hash value of all zeros. Finally, the hash values of the higher levels in the tree are calculated, by concatenating the hash values of the two children (again left to right) and computing the hash of that aggregate. If the two children are empty hashes, the parent is an empty all zeros hash as well (to save computation). This process ends in a hash value for the root node, which is called the "root hash". Note the root hash only depends on the content and any modification of the content will result in a different root hash.

```
                     7 = root hash
                    / \
                  /     \
                /         \
              /             \
            3*                11
           / \               / \
          /   \             /   \
         /     \           /     \
        1       5         9       13* = uncle hash
       / \     / \       / \      / \
      0   2   4   6     8   10* 12   14

      C0  C1  C2  C3    C4  C5  C6   E
      =chunk index      ^^           = empty hash
```

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

Figure 3

5.2. Content Integrity Verification

Assuming a peer receives the root hash of the content it wants to download from a trusted source, it can check the integrity of any chunk of that content it receives as follows. It first calculates the hash of the chunk it received, for example chunk C4 in the previous figure. Along with this chunk it MUST receive the hashes required to check the integrity of that chunk. In principle, these are the hash of the chunk's sibling (C5) and that of its "uncles". A chunk's uncles are the sibling Y of its parent X, and the uncle of that Y, recursively until the root is reached. For chunk C4 its uncles are nodes 13 and 3 and its sibling is 10; all marked with a * in the figure. Using this information the peer recalculates the root hash of the tree, and compares it to the root hash it received from the trusted source. If they match the chunk of content has been positively verified to be the requested part of the content. Otherwise, the sending peer either sent the wrong content or the wrong sibling or uncle hashes. For simplicity, the set of sibling and uncles hashes is collectively referred to as the "uncle hashes".

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

5.3. The Atomic Datagram Principle

As explained above, a datagram consists of a sequence of messages. Ideally, every datagram sent must be independent of other datagrams, so each datagram SHOULD be processed separately and a loss of one datagram must not disrupt the flow of datagrams between two peers. Thus, as a datagram carries zero or more messages, both messages and message interdependencies SHOULD NOT span over multiple datagrams.

This principle implies that as any chunk is verified using its uncle hashes the necessary hashes SHOULD be put into the same datagram as the chunk's data. If this is not possible because of a limitation on datagram size, the necessary hashes MUST be sent first in one or more datagrams. As a general rule, if some additional data is still missing to process a message within a datagram, the message SHOULD be dropped.

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

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

In short, the sender MUST put into the datagram the hashes he believes are necessary for the receiver to verify the chunk. The receiver MUST remember all the hashes it needs to verify missing chunks that it still wants to download. Note that the latter implies that a hardware-limited receiver MAY forget some hashes if it does not plan to announce possession of these chunks to others (i.e., does not plan to send HAVE messages.)

5.4. INTEGRITY Messages

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

5.5. Discussion and Overhead

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

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

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

In principle, the hash values of all these nodes will have to be sent to a peer once for it to verify all chunks. Hence the maximum on-the-wire overhead is hashsize * nnodes. However, the actual number of hashes transmitted can be optimized as described in Section 5.3.

To see a peer can verify all chunks whilst receiving not all hashes, consider the example tree in Section 5.1. In case of a simple progressive download, of chunks 0,2,4,6, etc. the sending peer will send the following hashes:

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

Table 1: Overhead for the example tree

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

5.6. Automatic Detection of Content Size

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

5.6.1. Peak Hashes

The ability for a newcomer peer to detect the size of the content depends heavily on the concept of peak hashes. The concept of peak hashes depends on the concepts of filled and incomplete nodes. Recall that when constructing the binary trees for content verification and addressing the base of the tree may have more leaves than the number of chunks in the content. In the Merkle hash tree these leaves were assigned empty all-zero hashes to be able to calculate the higher level hashes. A filled node is now defined as a node that corresponds to an interval of leaves that consists only of hashes of content chunks, not empty hashes. Reversely, an incomplete (not filled) node corresponds to an interval that contains also empty hashes, typically an interval that extends past the end of the file. In the following figure nodes 7, 11, 13 and 14 are incomplete the rest is filled.

Formally, a peak hash is the hash of a filled node in the Merkle tree, whose sibling is an incomplete node. Practically, suppose a file is 7162 bytes long and a chunk is 1 kilobyte. That file fits into 7 chunks, the tail chunk being 1018 bytes long. The Merkle tree for that file is shown in Figure 4. Following the definition the peak hashes of this file are in nodes 3, 9 and 12, denoted with a *. E denotes an empty hash.

```
                    7
                   / \
                 /     \
               /         \
             /             \
           3*               11
          / \              / \
         /   \            /   \
        /     \          /     \
       1       5        9*      13
      / \     / \      / \     / \
     0   2   4   6    8   10  12*  14

     C0  C1  C2  C3   C4  C5  C6   E
                              = 1018 bytes
```

Peak hashes in a Merkle hash tree.

Figure 4

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

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

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

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

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

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

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

5.6.2. Procedure

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

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

6. Live Streaming

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

6.1. Content Authentication

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

In the "Sign All" method, the live injector signs each chunk of content using a private key and peers, upon receiving the chunk, check the signature using the corresponding public key obtained from a trusted source. Support for this method is OPTIONAL.

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

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

Algorithm Numbers [IANADNSSECALGNUM]. The RSASHA1 [RFC4034], RSASHA256 [RFC5702], and ECDSAP256SHA256 and ECDSAP384SHA384 [RFC6605] algorithms are MANDATORY to implement.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Algo Number(8)|                                               ~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
~                 DNSSEC Public Key (variable)                  ~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

6.1.1. Sign All

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

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

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

6.1.2. Unified Merkle Tree

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

6.1.2.1. Signed Munro Hashes

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

```
          3
         / \
        /   \
       /     \
      1       5$
     / \     / \
    0   2   4   6
```

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

Figure 5

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

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

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

```
                                     7
                                    / \
                                  /     \
                                /         \
                              /             \
                            3               11
                           / \              / \
                          /   \            /   \
                         /     \          /     \
                        1       5        9$      13
                       / \     / \      / \      / \
                      0   2   4   6    8   10   E   E
```

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

Figure 6

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

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

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

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

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

6.1.2.2. Munro Signature Calculation

The digital signature algorithm used is determined by the Live Signature Algorithm protocol option, see Section 7.7. The signature is computed over a concatenation of the on-the-wire representation of the chunk specification of the munro node (see Section 6.1.2.1), a timestamp in 64-bit NTP Timestamp format [RFC5905], and the hash associated with the munro node, in that order. The timestamp is the time signature that was made at the injector in UTC.

6.1.2.3. Procedure

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

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

6.1.2.4. Secure Tune In

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

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

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

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

6.2. Forgetting Chunks

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

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

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

7. Protocol Options

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

Code	Description
0	Version
1	Minimum Version
2	Swarm Identifier
3	Content Integrity Protection Method
4	Merkle Hash Tree Function
5	Live Signature Algorithm
6	Chunk Addressing Method
7	Live Discard Window
8	Supported Messages
9	Chunk Size
10-254	Unassigned
255	End Option

Table 2: PPSP Peer Protocol Options

7.1. End Option

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

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|1 1 1 1 1 1 1 1|
+-+-+-+-+-+-+-+-+
```

7.2. Version

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

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

Table 3: PPSP Peer Protocol Version Numbers

```
 0                   1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0|  Version (8)  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

7.3. Minimum Version

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

```
 0                   1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 1| Min. Ver. (8) |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

7.4. Swarm Identifier

When a peer initiates the handshake it MUST include a single swarm identifier option. If the peer is not the initiator, it MAY include a swarm identifier option, as an end-to-end check. This option has the following structure:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 1 0|     Swarm ID Length (16)      |               ~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
~                  Swarm Identifier (variable)                  ~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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

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

7.5. Content Integrity Protection Method

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

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

Table 4: PPSP Peer Content Integrity Protection Methods

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

```
 0                   1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 1 1|   CIPM (8)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

7.6. Merkle Tree Hash Function

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

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

Table 5: PPSP Peer Protocol Merkle Hash Functions

Implementations MUST support SHA-1 (see Section 13.5) and SHA-256. SHA-256 is the default.

```
 0                   1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 1 0 0|    MHF (8)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

7.7. Live Signature Algorithm

When the content integrity protection method is "Sign All" or "Unified Merkle Tree" this option MUST be defined. The code for this option is 5. The 8-bit value of this option is one of the Domain Name System Security (DNSSEC) Algorithm Numbers [IANADNSSECALGNUM]. The RSASHA1 [RFC4034], RSASHA256 [RFC5702], ECDSAP256SHA256 and ECDSAP384SHA384 [RFC6605] algorithms are MANDATORY to implement. Default is ECDSAP256SHA256.

```
 0                   1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 1 0 1|    LSA (8)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

7.8. Chunk Addressing Method

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

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

Table 6: PPSP Peer Chunk Addressing Methods

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

```
 0                   1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 1 1 0|    CAM (8)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

7.9. Live Discard Window

A peer in a live swarm MUST include the discard window it uses. The code for this option is 7. The unit of the discard window depends on the chunk addressing method used, see Table 6. For bins and chunk ranges it is a number of chunks, for byte ranges it is a number of bytes. Its data type is the same as for a bin, or one value in a range specification. In other words, its value is a 32-bit or 64-bit integer in big endian format. If this option is used, the Chunk Addressing Method MUST appear before it in the list. This option has the following structure:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 1 1 1|      Live Discard Window (32 or 64)           ~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
~                                                               ~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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

7.10. Supported Messages

Peers may support just a subset of the PPSPP messages. For example, peers running over TCP may not accept ACK messages, or peers used with a centralized tracking infrastructure may not accept PEX messages. For these reasons, peers who support only a proper subset of the PPSPP messages MUST signal which subset they support by means of this protocol option. The code for this option is 8. The value of this option is a length octet (SupMsgLen) indicating the length in bytes of the compressed bitmap that follows.

The set of messages supported can be derived from the compressed bitmap by padding it with bytes of value 0 until it is 256 bits in length. Then a 1 bit in the resulting bitmap at position X (numbering left to right) corresponds to support for message type X, see Table 7. In other words, to construct the compressed bitmap, create a bitmap with a 1 for each message type supported and a 0 for a message type that is not, store it as an array of bytes and truncate it to the last non-zero byte. An example of the first 16 bits of the compressed bitmap for a peer supporting every message except ACKs and PEXs is: 11011001 11110000.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 1 0 0 0| SupMsgLen (8) |                               ~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
~             Supported Messages Bitmap (variable, max 256)     ~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

7.11. Chunk Size

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 1 0 0 1|       Chunk Size (32)                         ~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
~               |
+-+-+-+-+-+-+-+-+
```

8. UDP Encapsulation

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

8.1. Chunk Size

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

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

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

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

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

8.2. Datagrams and Messages

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

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

Table 7: PPSP Peer Protocol Message Types

Furthermore, integers are serialized in the network (big-endian) byte order. So consider the example of a HAVE message (Section 3.2) using bin chunk addressing. It has message type of 0x03 and a payload of a bin number, a four-byte integer (say, 1); hence, its on the wire representation for UDP can be written in hex as: "0300000001".

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

8.3. Channels

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

8.4. HANDSHAKE

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Destination Channel ID (32)                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0|            Source Channel ID (32)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               |                                               ~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                       Protocol Options                        ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

where:

Destination Channel ID:

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

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

The octet 0x00: The HANDSHAKE message: 0x00

The Source Channel ID: A locally unused channel ID

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

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

8.5. HAVE

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

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 1 1|               Start chunk (32)                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               |                End chunk (32)                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               |
+-+-+-+-+-+-+-+-+
```

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

8.6. DATA

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

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 1|             Start chunk (32)                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               |              End chunk (32)                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               |                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Timestamp (64)                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               |                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                            Data                               ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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

8.7. ACK

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

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

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |0 0 0 0 0 0 1 0|               Start chunk (32)                |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |               |                End chunk (32)                 |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |               |                                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                  One-way delay sample (64)                    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |               |
   +-+-+-+-+-+-+-+-+
```

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

8.8. INTEGRITY

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

An INTEGRITY message using 32-bit chunk ranges as Chunk Addressing method and a SHA-256 hash:

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |0 0 0 0 0 1 0 0|               Start chunk (32)                |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |               |                End chunk (32)                 |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |               |                                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                                                               |
   ~                           Hash (256)                          ~
   |                                                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |               |
   +-+-+-+-+-+-+-+-+
```

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

8.9. SIGNED_INTEGRITY

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

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 1 1 1|                Start chunk (32)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               |                 End chunk (32)                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               |                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Timestamp (64)                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               |                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                          Signature                            ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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

The length of the digital signature can be derived from the Live Signature Algorithm protocol option and the swarm ID as follows. The first MANDATORY algorithms are RSASHA1 and RSASHA256. For those algorithms, the swarm ID consists of a 1-byte Algorithm field followed by a RSA public key stored as a tuple (exponent length,exponent,modulus) [RFC3110]. Given the exponent length and the length of the public key tuple in the swarm ID, the length of the modulus in bytes can be calculated. This yields the length of the signature as in RSA this is the length of the modulus [HAC01]. The other MANDATORY algorithms are ECDSAP256SHA256 and ECDSAP384SHA384 [RFC6605]. For these algorithms the length of the digital signature is 64 and 96 bytes, respectively.

8.10. REQUEST

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

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 1 0 0 0|                Start chunk (32)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               |                 End chunk (32)                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               |
+-+-+-+-+-+-+-+-+
```

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

8.11. CANCEL

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

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 1 0 0 1|                Start chunk (32)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               |                 End chunk (32)                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               |
+-+-+-+-+-+-+-+-+
```

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

8.12. CHOKE and UNCHOKE

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

A CHOKE message:

```
 0
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|0 0 0 0 1 0 1 0|
+-+-+-+-+-+-+-+-+
```

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

An UNCHOKE message:

```
 0
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|0 0 0 0 1 0 1 1|
+-+-+-+-+-+-+-+-+
```

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

8.13. PEX_REQ, PEX_RESv4, PEX_RESv6 and PEX_REScert

A PEX_REQ (0x06) message has no payload. A PEX_RESv4 (0x05) message consists of an IPv4 address in big endian format followed by a UDP port number in big endian format. A PEX_RESv6 (0x0c) message contains a 128-bit IPv6 address instead of an IPv4 one. If a PEX_REQ message does not originate from a private, unique-local, link-local or multicast address [RFC1918][RFC4193][RFC4291], then the PEX_RES* messages sent in reply MUST NOT contain such addresses. This is to prevent leaking of internal addresses to external peers.

A PEX_REQ message:

```
 0
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|0 0 0 0 0 1 1 0|
+-+-+-+-+-+-+-+-+
```

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

A PEX_RESv4 message:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 1 0 1|              IPv4 Address (32)                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               |             Port (16)         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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

A PEX_RESv6 message:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 1 1 0 0|                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   IPv6 Address (128)                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               |             Port (16)         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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

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

A PEX_REScert message:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 1 1 0 1|   Size of Memb. Cert. (16)    |               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                     Membership Certificate                    ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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

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

    file://192.0.2.0:6778/e5a12c7ad2d8fab33c699d1e198d66f79fa610c3

8.14. KEEPALIVE

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

A keepalive datagram:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Channel ID (32)                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

8.15. Flow and Congestion Control

Explicit flow control is not required for PPSPP-over-UDP. In the case of video-on-demand, the receiver explicitly requests the content from peers, and is therefore in control of how much data is coming towards it. In the case of live streaming, where a push-model may be used, the amount of data incoming is limited to the stream bitrate, which the receiver must be able to process for a continuous playback. Should, for any reason, the receiver get saturated with data, the congestion control at the sender side will detect the situation and adjust the sending rate accordingly.

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

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

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

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

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

8.16. Example of Operation

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

To do the handshake the initiating peer sends a datagram that MUST start with an all 0-zeros channel ID (0x00000000), followed by a HANDSHAKE message, whose payload is a locally unused, random channel ID (in this case 0x00000001) and a list of protocol options. Channel IDs MUST be randomly chosen, as described in Section 13.1.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   HANDSHAKE   |0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 1|    Version    |0 0 0 0 0 0 0 1|  Min Version  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 1|   Swarm ID    |0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 1 0 0 0 1 1 1 1 0 1 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 0 1 1 0|
~                              .....                            ~
|1 0 0 0 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 1 0 0 0 0 0 0 1 1 1 0 1 1|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Cont. Int.  |0 0 0 0 0 0 0 1| Mer.H.Tree F. |0 0 0 0 0 0 1 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Chunk Add.  |0 0 0 0 0 0 1 0|   Chunk Size  |0 0 0 0 0 0 0 0~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
~0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0|      End      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The protocol options are:

Version: 1

Minimum supported Version: 1

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

Content Integrity Protection Method: Merkle Hash Tree.

Merkle Tree Hash Function: SHA-256.

Chunk Addressing Method: 32-bit chunk ranges.

Chunk Size: 1024.

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   HANDSHAKE   |0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 1 0 0 0|    Version    |0 0 0 0 0 0 0 1|   Cont. Int.  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 1| Mer.H.Tree F. |0 0 0 0 0 0 1 0|   Chunk Add.  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 1 0|  Chunk Size   |0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
~0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0|      End      |      HAVE     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

With the protocol options the receiving peer agrees on speaking protocol version 1, on using the Merkle Hash Tree as Content Integrity Protection Method, SHA-256 hash as Merkle Tree Hash Function, 32-bit chunk ranges as Chunk Addressing Method, and Chunk Size 1024. Furthermore, it sends a HAVE message within the same datagram, announcing that it has locally available the first chunk of content.

At this point, the initiator knows that the peer really responds; for that purpose channel IDs MUST be random enough to prevent easy guessing. So, the third datagram of a handshake MAY already contain some heavy payload. To minimize the number of initialization round trips, the first two datagrams MAY also contain some minor payload, e.g. the HAVE message.

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    REQUEST    |0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0|    PEX_REQ    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     DATA      |0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 0 1 0 0 1|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 0 1 1 0 1 1 1 1 1 0 1 1 0 1 1|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 1 0 0 0 1 0 0|0 1 0 0 1 0 0 0 0 1 1 0 0 1 0 1 0 1 1 0 1 1 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
~                             .....                             ~
|0 1 1 0 1 1 0 0 0 1 1 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 1 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The DATA message consists of:

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

The timestamp value: 0004e94180b7db44

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

Note that the above datagram does not include the INTEGRITY message, as the entire content can fit into a single message, hence the initiating peer is able to verify it against the root hash. Also, in this example the peer does not respond to the PEX_REQ as it does not know any third peer participating in the swarm.

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      ACK      |0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 1 1 0 0 1 0 0|      HAVE     |0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   HANDSHAKE   |0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0|      End      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

9. Extensibility

9.1. Chunk Picking Algorithms

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

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

9.2. Reciprocity Algorithms

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

10. Acknowledgements

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

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

11. IANA Considerations

IANA is to create a new top-level registry called "Peer-to-Peer Streaming Peer Protocol (PPSPP)", which will host the six new sub-registries defined below for the extensibility of the protocol. For all registries, assignments consist of a name and its associated value. Also for all registries, the "Unassigned" ranges designated are governed by the policy 'IETF Review' as described in [RFC5226].

11.1. PPSP Peer Protocol Message Type Registry

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

11.2. PPSP Peer Protocol Option Registry

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

11.3. PPSP Peer Protocol Version Number Registry

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

11.4. PPSP Peer Protocol Content Integrity Protection Method Registry

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

11.5. PPSP Peer Protocol Merkle Hash Tree Function Registry

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

11.6. PPSP Peer Protocol Chunk Addressing Method Registry

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

12. Manageability Considerations

This section presents operations and management considerations following the checklist in [RFC5706], Appendix A.

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

12.1. Operations

12.1.1. Installation and Initial Setup

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

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

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

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

12.1.2. Requirements on Other Protocols and Functional Components

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

12.1.3. Migration Path

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

12.1.4. Impact on Network Operation

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

12.1.5. Verifying Correct Operation

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

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

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

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

12.1.6. Configuration

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

Name	Default	Definition
Chunk Size	var, 1024 bytes recommended	Section 8.1
Static Content Integrity Protection Method	1 (Merkle Hash Tree)	Section 7.5
Live Content Integrity Protection Method	3 (Unified Merkle Tree)	Section 7.5
Merkle Hash Tree Function	2 (SHA-256)	Section 7.6
Live Signature Algorithm	13 (ECDSAP256SHA256)	Section 7.7
Chunk Addressing Method	2 (32-bit chunk ranges)	Section 7.8
Live Discard Window	var	Section 6.2, Section 7.9
NCHUNKS_PER_SIG	var	Section 6.1.2.1
Dead peer detection	No reply in 3 minutes + 3 datagrams	Section 3.12

Table 8: PPSPP Defaults

12.2. Management Considerations

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

12.2.1. Management Interoperability and Information

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

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

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

12.2.2. Fault Management

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

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

12.2.3. Configuration Management

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

12.2.4. Accounting Management

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

12.2.5. Performance Management

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

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

12.2.6. Security Management

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

13. Security Considerations

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

13.1. Security of the Handshake Procedure

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

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

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

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

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

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

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

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

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

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

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

13.1.1. Protection Against Attack 1

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

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

13.1.2. Protection Against Attack 2

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

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

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

13.1.3. Protection Against Attack 3

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

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

13.2. Secure Peer Address Exchange

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

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

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

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

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

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

13.2.1. Protection against the Amplification Attack

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

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

13.2.2. Example: Tracker as Certification Authority

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

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

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

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

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

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

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

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

13.2.3. Protection Against Eclipse Attacks

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

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

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

13.3. Support for Closed Swarms ([RFC6972] PPSP.SEC.REQ-1)

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

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

13.4. Confidentiality of Streamed Content ([RFC6972] PPSP.SEC.REQ-1)

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

When transmitting over DTLS, PPSPP can obtain the PMTU estimate maintained by the IP layer to determine how much payload can be put in a single datagram without fragmentation ([RFC6347], Sec. 4.1.1.1). If PMTU changes and the chunk size becomes too large to fit into a single datagram, PPSPP can choose to allow fragmentation by clearing the DF-bit. Alternatively, the content publisher can decide to use smaller chunks and transmit multiple in the same datagram when the MTU allows.

13.5. Strength of the Hash Function for Merkle Hash Trees

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

In general, note that the hash function is used in a hash tree, which makes it more complex to create collisions. In particular, if attackers manage to find a collision for a hash it can replace just one chunk, so the impact is limited. If fixed sized chunks are used, the collision even has to be of the same size as the original chunk. For hashes higher up in the hash tree, a collision must be a concatenation of two hashes. In sum, finding collisions that fit with the hash tree are generally harder to find than regular collisions.

13.6. Limit Potential Damage and Resource Exhaustion by Bad or Broken Peers ([RFC6972] PPSP.SEC.REQ-2)

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

13.6.1. HANDSHAKE

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

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

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

13.6.2. HAVE

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

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

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

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

13.6.3. DATA

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

  Solution: The content integrity protection schemes defend against this.

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

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

13.6.4. ACK

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

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

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

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

13.6.5. INTEGRITY and SIGNED_INTEGRITY

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

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

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

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

13.6.6. REQUEST

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

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

13.6.7. CANCEL

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

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

13.6.8. CHOKE

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

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

13.6.9. UNCHOKE

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

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

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

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

13.6.10. PEX_RES

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

13.6.11. Unsolicited Messages in General

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

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

13.7. Exclude Bad or Broken Peers ([RFC6972] PPSP.SEC.REQ-2)

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

14. References

14.1. Normative References

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

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

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

[RFC1918] Rekhter, Y., Moskowitz, R., Karrenberg, D., Groot, G., and E. Lear, "Address Allocation for Private Internets", BCP 5, RFC 1918, February 1996.

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

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

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

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

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

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

[RFC5702] Jansen, J., "Use of SHA-2 Algorithms with RSA in DNSKEY and RRSIG Resource Records for DNSSEC", RFC 5702, October 2009.

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

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

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

14.2. Informative References

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

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

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

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

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

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

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

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

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

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

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

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

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

[I-D.ietf-ppsp-base-tracker-protocol]
Cruz, R., Nunes, M., Yingjie, G., Xia, J., Taveira, J., and D. Lingli, "PPSP Tracker Protocol-Base Protocol (PPSP-TP/1.0)", draft-ietf-ppsp-base-tracker-protocol-06 (work in progress), October 2014.

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

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

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

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

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

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

[PPSPPERF] Petrocco, R., Pouwelse, J., and D. Epema, "Performance analysis of the Libswift P2P streaming protocol", IEEE International Conference on Peer-to-Peer Computing (P2P'12), Tarragona, Spain, Sep 2012.

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

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

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

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

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

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

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

[RFC4193] Hinden, R. and B. Haberman, "Unique Local IPv6 Unicast Addresses", RFC 4193, October 2005.

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

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

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

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

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

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

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

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

[RFC6194] Polk, T., Chen, L., Turner, S., and P. Hoffman, "Security Considerations for the SHA-0 and SHA-1 Message-Digest Algorithms", RFC 6194, March 2011.

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

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

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

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

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

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

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

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

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

Appendix A. Revision History

-00 2011-12-19 Initial version.

-01 2012-01-30 Minor text revision:

* Changed heading to "A. Bakker"
* Changed title to *Peer* Protocol, and abbreviation PPSPP.
* Replaced swift with PPSPP.
* Removed Sec. 6.4. "HTTP (as PPSP)".
* Renamed Sec. 8.4. to "Chunk Picking Algorithms".
* Resolved Ticket #3: Removed sentence about random set of peers.

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

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

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

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

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

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

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

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

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

-02 2012-04-17 Major revision

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

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

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

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

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

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

* Renamed HASH message to INTEGRITY.

* Renamed HINT to REQUEST.

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

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

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

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

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

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

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

* Moved Rationale to Appendix.

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

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

-03 2012-10-22 Major revision

* Updated Abstract and Introduction, removing download case.

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

* Removed directory lists unused in streaming.

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

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

* Added a section on compatibility between chunk addressing schemes.

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

* Added a section on forgetting chunks in live streaming.

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

* Added SHA-2 support for Merkle Hash functions.

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

* Added a Live Signature Algorithm protocol option.

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

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

* Removed appendix with Rationale.

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

-04 2012-11-07 Minor revision

* Corrected typos.

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

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

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

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

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

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

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

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

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

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

* Defined value for version protocol option.

* Added small clarifications and corrected typos.

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

* Processed comments on unclear phrasing from Riccardo Bernardini.

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

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

-05 2013-01-23 Minor revision

* Corrected category to Standards Track.

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

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

-06 2013-02-11 Minor revision

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

* Clarified wording on PEX_REQ.

* Clarified wording on SIGNED_INTEGRITY.

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

* Added Manageability Consideration section following RFC5706.

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

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

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

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

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

* Updated reference to PPSP problem statement and requirements document.

* Used kibibyte unit in formal sections.

-07 2013-06-19 Revision following AD Review

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

1) Merkle Hash Trees I have found the document very confusing on whether Merkle Hash Trees (MHTs) and the for the MHT required bin numbering scheme are now optional or mandatory. Parts of the draft make the impression that either of them or both or optional (mainly in the beginning of the document), while Section 5 and later Sections are relying heavily on MHTs. My naive reading of the current draft is that you could rely on start-end ranges for chunk addressing and MHTs for content protection. However, I do know that this combination

is not working. If MHTs are really optional, including the bin numbering, the document should really state this and make clear what the operations of the protocol are with the mandatory to implement (MTI) mechanisms. The MHT, bins, and all the protocol handling should go in an appendix. There is a call to make for the WG: I do know that MHTs were considered by some as burden and they have called for a leaner way, i.e., the start-end ranges. The call for the leaner way has been implemented in the document but not fully.

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

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

+ A new paragraph has been added to Section 8.15 describing the widespread use of LEDBAT in current P2P systems. Hence, aim is a DOWNREF procedure.

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

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

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

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

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

+ Concept now introduced in Section 3 with a figure.

***Technicals:

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

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

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

+ Rephrased.

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

+ Phrase added.

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

+ Phrase removed.

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

+ Added.

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

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

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

+ DONE.

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

+ DONE.

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

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

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

+ DONE.

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

+ DONE.

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

+ DONE.

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

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

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

- Section 3.11 I don't see the 'MUST send keep-alive' as a mandatory requirement, as peers might have good reasons not to send any keep alive. Why not saying 'A peer can send a keep-alive' and it 'MUST use the simple datagram...' as already described. Though there is also no really need to say MUST.

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

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

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

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

+ DONE.

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

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

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

+ Rephrased, see High-level issues.

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

+ Rephrased.

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

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

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

+ Expanded. Added a formal definition.

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

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

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

+ Clarified.

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

+ Clarified.

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

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

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

+ Corrected, see High-level issues 4.)

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

+ Clarified.

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

+ Adjusted.

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

+ 1 KiB chunk size has been made a recommendation.

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

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

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

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

+ Concept now introduced in Section 3.11.

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

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

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

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

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

+ Clarified.

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

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

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

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

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

We added the usual bit-based ASCII art representations.

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

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

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

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

***Editorials:

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

+ DONE.

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

+ DONE.

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

+ DONE.

- Section 2.1 and following: These are examples, isn't it? If so, this should be mentioned or clarified.

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

- Section 2.1: What is the PPSP Url?

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

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

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

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

+ DONE.

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

+ Rephrased.

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

+ Rephrased.

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

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

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

+ Yes. Removed.

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

+ DONE, for byte ranges as well.

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

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

+ Removed, just similarity in format.

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

- Section 8.1 1 kibibyte is 1 kbyte?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

-09 2014-04-4 Nits fixed

Nits about e.g. newer references fixed.

-10 2014-06-17 DOWNREF restored

Reference to LEDBAT was not in Normative references as it should have been.

-11 2014-06-18 IANA not OK

In the 2nd Last Call IANA posed two questions:

" QUESTION: Are the authors intended to have one single top-level registry to host these six new registries defined in this draft? Please see http://www.iana.org/assignments/ancp as an example, that the ANCP registry hosts multiple sub-registries."

+ Yes. We updated the IANA Considerations section with IANA’s Pearl Liang proposed text to request this.

"QUESTION: Section 11.3 specifies that the values are integers in the range 0-255. However value 0 is not included in the above table. Section 7.2 (Version) does not clearly explain value 0."

+ 0 defined as reserved.

Text cleanup

+ Terminology: states chunks may be variable size explicitly also here.

+ Figure 3 label improved.

+ 5.4 Explicitly state that leaves have tree height 0.

+ 5.5 Repaired split paragraph.

+ 5.6 Rephrased start to be consistent with minimally required metadata.

+ 5.6.1 Removed remark about peak hashes role in static/live download unification as that is no longer the case.

+ 5.6.2 Explicitly stated that peak hashes are transported in INTEGRITY messages.

+ 6.1.2.1 Changed "computing the computed" to "comparing the computed"

+ 6.2 Changed sentence to "*is* related to bitrate"

+ 8.4 Explicitly stated that the *Source* Channel ID is all 0-zeros in closing HANDSHAKE.

+ 8.5 Removed spurious line.

+ 8.6 Explicitly stated that the chunk specification in a DATA message denotes a single chunk.

+ 8.8 Changed copy+paste from ACK to INTEGRITY message.

+ 8.12 Renamed PEX_RES to PEX_RESv4 where needed.

+ 8.17 Explicitly stated that the *Source* Channel ID is all 0-zeros in closing HANDSHAKE.

+ 12.1.5 Changed to read that a swarm’s operation can easily verified when swarm metadata and tracker info is available.

+ 13.2.1 "cert" -> "certificate"

+ Added refs to RFC6972 where required.

==
=========== IESG telechat DISCUSSES ==========================
===

Alissa Cooper:

"I’m a little surprised about the choice of LEDBAT for congestion control of live streams. It seems like LEDBAT is not what the receiver would want the sender to use for live-streamed content, because if a bottleneck is encountered on the path, the live stream will yield early, and the recipient’s perception of quality will degrade. If the bottleneck is near the recipient, then every sender sending chunks will yield early, and there may be no senders available to stream at an acceptable level of quality. I’m assuming the WG discussed this -- it would be helpful to understand why a more aggressive congestion control was not selected for live streaming."

+ Updated 8.16 to discuss why LEDBAT is good in a peer-to-peer context (can be friendly to network as a whole and has configurable aggressiveness)

Stephen Farrell:

"(1) 3.10: What is a "benign" environment? I actually do understand what is meant, but how could a program evaluate that in order to decicde whether or not to send a PES_RESv4? You then refer to a "potentially hostile environment" which could presumably be anywhere, so are you really saying that PES_REScert is the "right thing" to do, but you know it won’t be done so these are weasel words around that awkward fact? (Apologies if I’m wrong on that, but that’s the impression I got when reading this, but maybe that’s just my paranoia:-)"

+ Rephrased to show PEX_REScert is the only option on the Internet.

"(2) 6.1.2.2: What exactly are the "munro" bytes that are the first input to the signature? Where are those defined? (Sorry if I missed/skipped over that;-)"

+ Added to Terminology and added an explicit reference in this section.

"(3) 7.6 and 13.5: SHA1 as the MTI is wrong. Why is that ok, given the collision resistance is less that designed for? 7.7 also calls for SHA256 being implemented in any case. The run-time argument in 13.5 does not convince me. Attacks only ever get worse, so the collision resistance property which this protocol needs ought lead to selection of an as-far-as-known good hash function. Today that means SHA256 and not SHA1."

+ SHA-256 is now the default. SHA-1 is still MTI to give content providers a trade-off between performance and security, as the on-the-wire overhead is 37.5% smaller.

"(4) 7.7: Why RSASHA1 and not RSA with SHA256?"

+ RSA256 is now also MTI, but RSASHA1 is also required, as argued in the previous point.

"(5) 7.10: The message number is wrong in the figure."

+ Fixed.

"(6) 8.4: I don’t see the swarm’s metadata record in the ascii art diagram and you just say "look at section 7" so two questions: a) where is the "chunk size used" option in section 7? and b) do all the swarm metadata options have to be sent each time with no limit on ordering except as given in section 7 (which had one such order sensitive limit I think)?"

+ (a) We once envisioned that a peer could start with just a swarm ID+chunk size as metadata and obtain all protocol options (chunk addressing, integrity protection, etc.) from a peer. As this turned out to be too complex to secure (peers may lie about the options), we decided to make the options all part of the swarm metadata after the AD review. This renders the protocol options in the HANDSHAKE to an end-to-end test really. Chunk size was never part of that negotiation because writing code that would handle bad input on that parameter was definitely too complicated. Chunk size has now been added as a protocol option.

+ (b) The HANDSHAKE message and hence protocol options are sent only in the first datagram. After that this information is part of the context of the channel that has been established. We added a limit on ordering (sort on code value, ascending) as a simplification.

"(7) 8.13: Don't you need to register the ppsp URI scheme? In case its useful, which I doubt, if you have code: RFC6920 URIs could be used for this if you wanted and would save you adding ppsp to the IANA URI scheme registry (and having to deal with the URI police:-)"

+ Not sure. The problem is we need to denote a peer in a swarm here. This means we need to encode the swarm ID and the peer address. IMHO, this means we cannot use the ni: RFC6920 scheme here, because only the hash determines identity. If we would encode the swarm ID there (SHA-256 hash of the swarm ID), we need a place for the peer address and the authority part does not make the URL unique. The ni: URL will still only identify the swarm. Encoding the peer address in OtherNames instead of uniformResourceIdentifier is troublesome too. We could find no single object type to denote a transport address (IP+port) that supports both IPv4 and IPv6 (udpDomain is IPv4 only). Using SAN ipAddress for the address and a separate OtherName for the port number (e.g. udpEndpointLocalPort from [RFC4113]) is not ideal, as the port number by itself is not a name for the subject. Hence, we replaced the ppsp: scheme with the file: scheme, which has an authority part where we can naturally encode the peer address in.

"(8) 13.4: Wouldn't DTLS change the chunk size considerations and also influence how messages map to datagrams? Isn't more specification needed to say how to really use DTLS here? Just saying "use DTLS or IPsec or higher layer crypto" doesn't really seem sufficient. And doing the DTLS bits right shouldn't be very hard either."

+ According to RFC6347, for "DTLS over UDP, the upper layer protocol SHOULD be allowed to obtain the PMTU estimate maintained in the IP layer" (Sec. 4.1.1.1). So we know beforehand how much payload we can send in a datagram without fragmentation. If PMTU changes and the chunk size becomes too large, we can choose to allow fragmentation ("the upper layer protocol SHOULD be allowed to set the state of the DF bit (in IPv4) or prohibit local fragmentation (in IPv6)."). Alternatively, the content publisher can decide to use smaller chunks and transmit multiple in the same datagram when the MTU allows. We added an explanatory paragraph to Section 13.4.

Other DISCUSSes: TODO.

-12 2014-11-16 Other DISCUSSes and reviews

==
=========== IESG telechat DISCUSSES (continued) ==============
===

Richard Barnes:

"My DISCUSS here is based mainly on the readability of the document, which seems bad enough to be an impediment to interoperability. As far as I can tell, this document does not define a protocol, in the sense of a set of actions required to achieve a given objective. Instead, it presents a pile of piece parts with a couple of combinations, and notes that these combinations could be used to achieve, e.g., live streaming. (In the language of patents, it has not been "reduced to practice".) What are the steps an implementation follows to join a swarm? To connect to a new peer and request chunks? The pieces seem to be here, but the big picture is completely absent."

+ Clarified the DISCUSS via email. In response, we made the distinction between tracker protocol and peer protocol more clear in the Introduction and Section 2.

+ We also rewrote the section on the HANDSHAKE message to include an explicit handshake procedure in the format suggested.

Kathleen Moriarty:

"I am still reading this draft, but don’t see any response to the SecDir review that raised some very important points for discussion: http://www.ietf.org/mail-archive/web/secdir/current/msg04879.html I’ll amend this when I get further into my review and would appreciate a response to the SecDir review."

+ Please see our responses to the SecDir review via email: http://www.ietf.org/mail-archive/web/secdir/current/msg04914.html and below.

OpsDir review by Tina TSOU (replied to by email, July 9th, 2014):

"You probably want to mention SHA-256 rater than SHA-1 [...]"

+ SHA-256 made default. See reply to Farrell.

"Section 8.13: You should also include ULAs"

+ Added.

"Section 8.16: This doesn't seem like a good justification for not having flow control. Could you please elaborate on why flow control is not needed for this case?"

+ Explained in email reply.

"Section 8.17, page 53: The channel ID values employed might give the reader the impression that they are non-random."

+ Stated explicitly they must be random with a reference to why.

Nits

+ Processed

GenArt review by Christer Holmberg (replied to by email, July 9th, 2014)

"Q1: The sending of keep alives is a SHOULD, and there are no procedures on how to act if keep alives are not received. There isn't even a mechanism to negotiate the sending of keep alives.

+ Rewrote Section 3.12 to explain what happens when no keep alives are received. In particular, given certain conditions the peer is declared dead and no more messages are sent to it, and the local administration about that peer is discarded. The exact conditions were specified in

Sec. 8.15 but are now defined in 3.12 at Christer's suggestion in his follow-up to our reply. Section 8.15 removed.

"Q2: As the sending of keep alives is a SHOULD, are there example cases when keep alives would NOT be sent?"

+ Added example of busy server garbage collecting idle clients by not sending keep alives.

"Q3: The text saying "to each peer it wants to interact with in the future" sounds a little strange to me. How does a peer know with whom it wants to interact in the future? Perhaps the text instead should talk about peers with whom one wants to maintain a signaling channel, or something like that?"

+ Rephrased in Sec. 3.12 to "interesting peers" and explain there is a policy that determines which peers are of interest. E.g. peers that have chunks that the downloader is still missing.

SecDir review by David Harrington (replied to by email, July 10th, 2014)

Editorials:

+ Incorporated.

"6) tech: I feel uncomfortable with section 2 containing examples that describe the overall flow. Examples are non-normative text, usually contained in a non-normative appendix. These examples describe the order of messages, and it is "

+ Please see our actions taken on Richard Barnes' DISCUSS.

"7) in example 2.2, the integrity hash is provided by the peer that is providing the (potentially maliciously modified) content. Isn't that like asking the fox to verify that the henhouse is safe?"

+ Added that the hashes can be verified against the trusted swarm ID using the Merkle tree content integrity protection scheme, defined later in the document.

"9) in 3, paragraph 1, it says "this behavior", but I'm not sure which behavior it is referencing. It is unclear whether not sending error messages, or discarding messages, or stopping communication, or classifying peers is the behavior that allows a peer to deal with slow, crashed, or silent peers. I don't understand HOW any of the behaviors mentioned would allow a peer to deal with slow, crashed, or silent peers. I do not understand on what basis peers are judged "good" or "bad"."

+ Added explanation how in a peer-to-peer system with multiple sources to obtain the content from, the classification in good and bad can be used to deal with malfunctioning peers.

"11) in 3, paragraph 3, the second sentence seems to contradict the first sentence, and since neither is written using RFC2119 keywords, it seems to really leave the whole question open to implementer interpretation."

+ Added that the video container format used is outside the scope of this document.

''"A SIGNED_INTEGRITY message (type 0x07) consists of a chunk specification, a 64-bit NTP timestamp [RFC5905] and a digital signature encoded as a Signature field in a RRSIG record in DNSSEC without the BASE-64 encoding [RFC4034]." Can this work in an implementation with no NTP support?''

+ Yes. It is sufficient for the injector's and receivers' clocks to be roughly synchronized. Rephrased to "64-bit timestamp in NTP Timestamp format".

"8.14 describes a keep alive message format, but no processing instructions."

+ Please see our actions following the GenArt review.

"Multiple messages are multiplexed in a datagram. How are the messages delimited? If there is any corruption in one message, how does the receiver find the end of the message and the start of the next message? If I understand correctly, invalid messages are discarded and no error code is sent. If one of the messages are found to be invalid, are all messages in that datagram discarded? or are all subsequent messages in that datagram discarded? or is it valid to process the remaining messages in the datagram after an invalid message is detected? If so, would that conflict with the rule that all messages must be processed in order?"

+ Messages are fixed size, or contain size fields. Made it more clear in Section 3 that when an invalid message is encountered in a datagram, the remaining messages MUST be discarded.

** COMMENTS by Spencer Dawkins, July 8th 2014**

"3. Messages In general, no error codes or responses are used in the protocol; absence of any response indicates an error. Is there accurate qualifier more narrow than "in general" that you could substitute?"

+ Qualifier removed.

"3.1. HANDSHAKE [heavy/minor confusing]"

+ Heavy payload has been explicitly defined in Terminology. Please see our response to Richard Barnes' DISCUSS for a rephrased definition of the handshake procedure.

"3.2. HAVE In particular, whenever a receiving peer P has successfully checked the integrity of a chunk, or interval of chunks, it SHOULD send a ^^^^^^ HAVE message to all peers Q1..Qn it wants to interact with in the near future. A policy in peer P determines when the HAVE is sent. P may sent it directly, or peer P may wait until either it has other data to sent to Qi, or until it has received and checked multiple chunks. This wasn't clear to me. I'm not understanding why a SHOULD is appropriate, but I suspect I shouldn't be askig a 2119 question, because this is tangled between "send a HAVE to the peers you want to interact with in the near future" and "if you don't want to interact with a specific peer in the near future, you can wait to send a HAVE". Is that even close? "

+ Yes. Changed to a MUST and rephrased that the HAVE is sent only to the peers the sender wants to allow download of those chunks from.

"3.4. ACK [unreliable/reliable discussion in WG]"

+ The swift protocol on which this draft is based had been designed from the start to be transport-agnostic. We tried to preserve that as much as possible.

"5.3. The Atomic Datagram Principle [...] With that many SHOULDs, I'd be worried that implementations using PPSPP can't count on much. If I receive a message that spans multiple datagrams (even though it shouldn't), that don't include the necessary hashes (even though it should), and I don't drop a message with missing data (even though I should), is that all fine?"

+ Yes. Unfortunately, there are some exceptional cases that have to be dealt with. E.g. because of reordering you may want to hang on to a datagram with a DATA message that cannot yet be verified because the datagram with the required hashes was delayed. In general, there will be multiple sources to obtain the content from, so a peer/implementor may choose to get the chunks from a different source with a better path.

"5.4. INTEGRITY Messages Concretely, a peer that wants to send a chunk of content creates a datagram that MUST consist of a list of INTEGRITY messages followed by a DATA message. If the INTEGRITY messages and DATA message cannot be put into a single datagram because of a limitation on datagram size, the INTEGRITY messages MUST be sent first in one or more datagrams. Is this assuming that the path between peers will never reorder packets?"

+ No. Hence, the many SHOULDs in the previous Section. This facilitates processing in the normal case.

COMMENT by Jari Arkko, July 8th 2014

+ Please see responses to GenArt review.

COMMENTs by Barry Leiba, July 9th 2014

"General question on the chunking: Is it the case that a given piece of content is chunked in a specific way, with known chunk IDs, such that every peer that's serving that content up (at least in the same swarm) uses the same chunks with the same chunk IDs? One can guess that from the way things work, but shouldn't the document say that? Or does it, and I missed it?"

+ Explicitly stated this in Section 3.

"-- Section 3.7 -- When peer Q receives multiple REQUESTs from the same peer P, peer Q SHOULD process the REQUESTs in the order received. What happens if it doesn't? Is there an interoperability issue here? A performance issue? Or what? (That is, why is this a 2119 SHOULD?)"

+ Consider the case where peer P is operating near the playback deadline, i.e., the last chunk it has needs to be given to the video player very soon or it will stall (i.e., P has nearly run out of buffer). In that case it is important that the chunks P requests arrive in order (and it will be requesting a range of chunks at a time to get a pipeline going). So the SHOULD is there to help prevent a performance problem.

"-- Section 5.3 -- Thus, as a datagram carries zero or more messages, neither messages nor message interdependencies SHOULD span over multiple datagrams. The negatives in this sentence really make the SHOULD a hidden SHOULD NOT, and its meaning is unclear. I think it would be clearer if it were worded that way:"

+ Replaced with suggested wording.

COMMENTs by Alia Atlas, July 9th 2014

+ Requested modifications made.

"Sec 8.1: The paragraph on PLPMTUD is a bit confusing. Presumably this is between two peers - but the chunk sizes used by the swarm would be specified by the initial seeder. Thus I can see the PLPMTUD variant being useful to decide upon the PPSPP datagram size, but not the chunk size. Could you please clarify either what I'm missing?"

+ This section discussed considerations for choosing a chunk size. Deployments can use PLPMTUD, but in that case they should partition the content using a small chunk size, such that the datagram can be scaled up or down depending on the actual network properties.

COMMENTs by Stephen Farrell, July 10th 2014.

For DISCUSSes, please see above at revision -11.

"- The elephant is in the room, but not the intro:-) Surely some comparison with BT is needed in the intro?"

+ Paragraph added.

"- 1.1: I really dislike the term self-certification as its quite misleading."

+ AFAIK the term was quite common for this type of mechanism, see e.g. http://en.wikipedia.org/wiki/Self-certifying_File_System

"- 1.3, 'content': s/asset/file/ would be better I think and less capitalist;-)"

+ Done. We had commercial partners in our research project ;-)

"- 3: I don't get what is meant by this "an external storage mapping from the linear byte space of a single swarm to different files" I can sorta see what's meant, but am not sure. Maybe try clarify?"

+ In other words, we do not prescribe the video container format. Added to the paragraph.

"- 5.3, last para: Is the 1st MUST there really implementable in general? I think the MUST might be to include those hashes that the sender thinks the receiver needs."

+ Clarified

"- 6.1 - this defines two methods yet says "If the protocol operates in a benign environment the method MAY be used." Which is meant here?"

+ Clarified.

"- 6.1.2.1: what if different folks think NCHUNKS_PER_SIG has different values? How do we all agree on a value? (BTW, the last sentence of this section is a cool thing.)"

+ This value is set by the content publisher, and is then derived from the chunk specification of the signed munro hash by all peers.

"- 7.4: "In other cases a peer MAY include a swarm identifier option, as an end-to-end check." That's not clear to me, what other cases?"

+ Rephrased.

"- 7.8: The width of the figure seems wrong."

+ Corrected.

"- 7.10: An example compressed encoding would be useful."

+ Added a small example.

"- 8.16: "perfectly detected" - huh? what does that mean?"

+ Rephrased in revision -11.

Authors' Addresses

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

Email: arno@cs.vu.nl

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

Email: r.petrocco@gmail.com

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

Email: victor.grishchenko@gmail.com