

PPSP
INTERNET-DRAFT
Intended Status: Standards Track
Expires: April 24, 2013

Rui S. Cruz
Mario S. Nunes
IST/INESC-ID/INOV
Yingjie Gu
Jinwei Xia
Huawei
Joao P. Taveira
IST/INOV
Deng Lingli
China Mobile
October 21, 2012

PPSP Tracker Protocol--Base Protocol (PPSP-TP/1.0)
draft-cruz-ppsp-base-tracker-protocol-01

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 (audio, video, associated timed text and metadata) delivery, such as adaptive multi-rate, layered (scalable) and multi-view (3D), in live, time-shifted and on-demand modes.

Status of this Memo

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

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

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

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

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

Copyright and License Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	4
1.1. Use Scenarios and Streaming Modes	4
1.2. Assumptions	5
1.2.1. Enrollment and Bootstrap	6
1.2.2. NAT Traversal	7
1.2.3. Content Information Metadata	8
1.2.4. Authentication, Confidentiality, Integrity	8
2. Terminology	8
3. Architectural and Functional View	11
4. Messaging Model	13
5. Request/Response model	13
6. State Machines and Flows of the Protocol	14
6.1. Normal Operation	16
6.2. Error Conditions	17
7. Protocol Specification	17
7.1. Request/Response Syntax and Format	17
7.2. Semantics of PPSPTrackerProtocol elements	20
7.3. Request element in request Messages	22
7.4. Response element in response Messages	23
8. Request/Response Processing	24
8.1. CONNECT Request	24
8.2. STAT_REPORT Request	28
8.3. Error and Recovery conditions	29
9. Security Considerations	30
9.1. Authentication between Tracker and Peers	30
9.2. Content Integrity protection against polluting peers/trackers	31
9.3. Residual attacks and mitigation	31
9.4. Pro-incentive parameter trustfulness	31
10. Guidelines for Extending PPSP-TP	32
10.1. Forms of PPSP-TP Extension	33
10.2. Issues to Be Addressed in PPSP-TP Extensions	34
11. IANA Considerations	35
12. Acknowledgments	35
13. References	36
13.1. Normative References	36
13.2. Informative References	37
Appendix A. PPSP-TP Message Syntax for HTTP/1.1	39
A.1. Header Fields	40
A.2. Methods	40
A.3. Message Bodies	40
A.4. Message Response Codes	41
Authors' Addresses	43

1. Introduction

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

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

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

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

The Client Media Player is a logical entity providing direct interface to the end user at the client device, and includes the functions to select, request, decode and render contents. The Client Media Player may interface with the local peer application using request and response standard formats for HTTP Request and Response messages [RFC2616].

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

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

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

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

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

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

1.2. Assumptions

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

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

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

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

1.2.1. Enrollment and Bootstrap

In order to join an existing P2P streaming service and to participate in content sharing, any peer must first locate a tracker, using for example, the following methods (illustrated in Figures 1, 2):

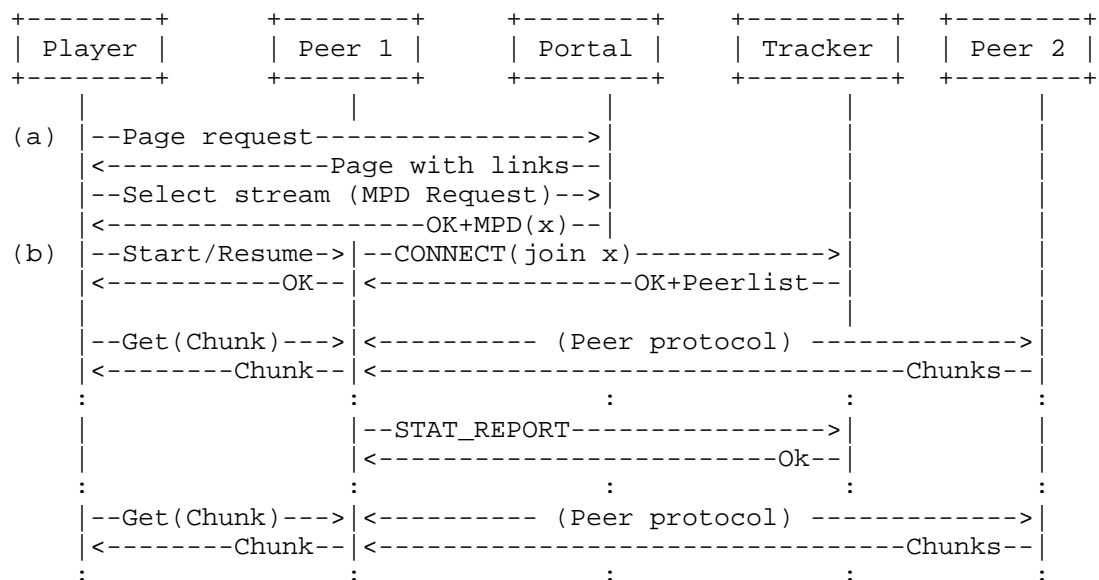


Figure 1: A typical PPSP session for watching a streaming content.

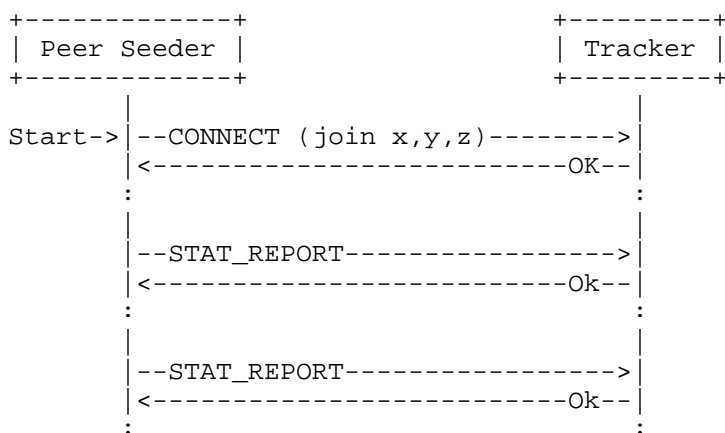


Figure 2: A typical PPSP session for a streaming Seeder

1. From a content service provider provisioning mechanism: this is the typical case used for the provider Seeders (edge caches and/or Media Servers), where some "Start" activation signal triggers the process for contents x, y, z (Figure 2).
2. From a web page: a Publishing and Searching Portal may provide tracker location information to end users.
3. From the Media Presentation Description (MPD) file of a content: this meta-info file must contain information about the address of one or more trackers (that can be grouped by tiers of priority) which are controlling the swarm for that media content, as illustrated in Figure 1 for a content x.

As illustrated in Figure 1, a peer may initiate a stream using the above method 3 starting at point (a), or resume a previously initiated stream, using also method 3 but starting at point (b).

In order to be able to bootstrap, 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.

The specification of the mechanisms used by the Client Media Player 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.

1.2.2. NAT Traversal

It is assumed that all trackers must be in the public Internet and

have been placed there deliberately. This document will not describe NAT Traversal mechanisms but the protocol facilitates flexible NAT Traversal techniques, such as those based on ICE [RFC5245], considering that the tracker node may provide NAT traversal services, as a STUN-like tracker. Being a STUN-like tracker, it can discover the reflexive candidate addresses of a peer and make them available in responses to other requesting peers.

1.2.3. Content Information Metadata

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

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

These characteristics may be described in a Media Presentation Description (MPD) file. It is envisioned that the content information metadata used in PPSP may align with MPD formats, such as ISO/IEC 23009-1 [ISO.IEC.23009-1] and [I-D.pantos-http-live-streaming].

1.2.4. Authentication, Confidentiality, Integrity

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

Due to the transactional nature of the communication between peers and tracker a method for adding authentication and data security services via replaceable mechanisms may be employed. One such method is the OAuth 2.0 Authorization [RFC6749] with bearer token, providing the peer with the information required to successfully utilize the access token to make protected requests to the tracker [RFC6750].

2. Terminology

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

This draft uses terms defined in [I-D.ietf-ppsp-problem-statement].

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

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

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

Chunk: A chunk is a basic unit of data block organized in P2P streaming for storage, scheduling, advertisement and exchange among peers. A chunk therefore refers to a segment of partitioned streaming media.

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

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

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

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

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

Leecher: A Peer that has not yet completed the transfer of all chunks of the media content.

Live streaming: The scenario where all clients receive streaming content for the same ongoing event. The lags between the play points

of the clients and that of the streaming source are small.

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

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

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

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

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

Peer-ID: Unique identifier for the peer. The Peer-ID 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 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-Peer Messages (i.e., Peer Protocol): The Peer Protocol messages enable each peer to exchange content availability with other peers and request other peers for content.

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

Representation: Structured collection of one or more media components.

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

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

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

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

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

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

Swarm-ID: Unique identifier for a swarm. 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.

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

Tracker-Peer Messages (i.e., Tracker Protocol): The Tracker Protocol messages provide communication between peers and trackers, by which peers can provide content availability, report streaming status and request peer lists from trackers.

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

3. Architectural and Functional View

The PPSP Tracker Protocol architecture 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 Messaging 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).

The messaging layer deals with the framing format, for encoding and transmitting the data through the underlying transport protocol, and 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, or TLS [RFC5246] over it.

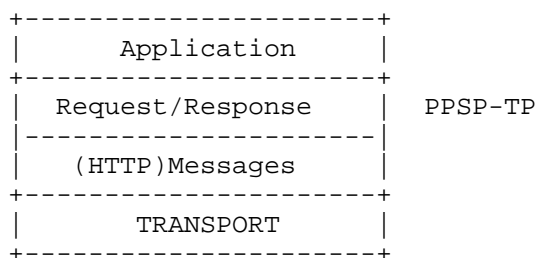


Figure 3: Abstract layering of PPSP-TP

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

A Peer corresponds to a logical entity (typically in a user device) that actually participate 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 segments 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 nodes, 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.

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

1. Peer connects to a local connection tracker and joins a swarm.
2. Peer acquires a list of peers from the connection tracker.

3. Peer exchanges its content availability with the peers on the obtained peer list (via peer protocol).
4. Peer identifies the peers with desired content.
5. Peer requests content from the identified peers (peer protocol).

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 (joins).

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

4. 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 and framing formats used for encoding and transmitting the data over the wire, e.g., the format proposed in [I-D.montenegro-httpbis-speed-mobility]. The exchange of messages of PPSP-TP is envisioned to be performed over a stream-oriented reliable transport protocol, like TCP.

Appendix A describes the message syntax when using HTTP/1.1 messages.

5. Request/Response model

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

PPSP-TP 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.

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

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

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

PPSP-TP/1.0
Req. Messages
CONNECT
STAT_REPORT

Table 1: Request Messages

CONNECT: This Request message is used when a peer registers (or if already registered) in the tracker notifies 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 link status. 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 8.1)

STAT_REPORT: This Request message allows an active peer to send status 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.

6. 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.

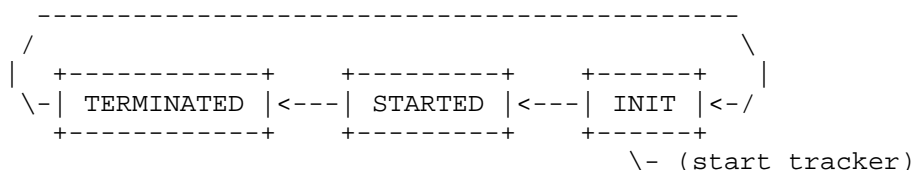


Figure 4: Tracker State Machine

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.

In addition to the tracker state machine, each Peer-ID is modeled with its own transaction state machine (Figure 5), instantiated per Peer-ID in the tracker, and deleted when it is removed.

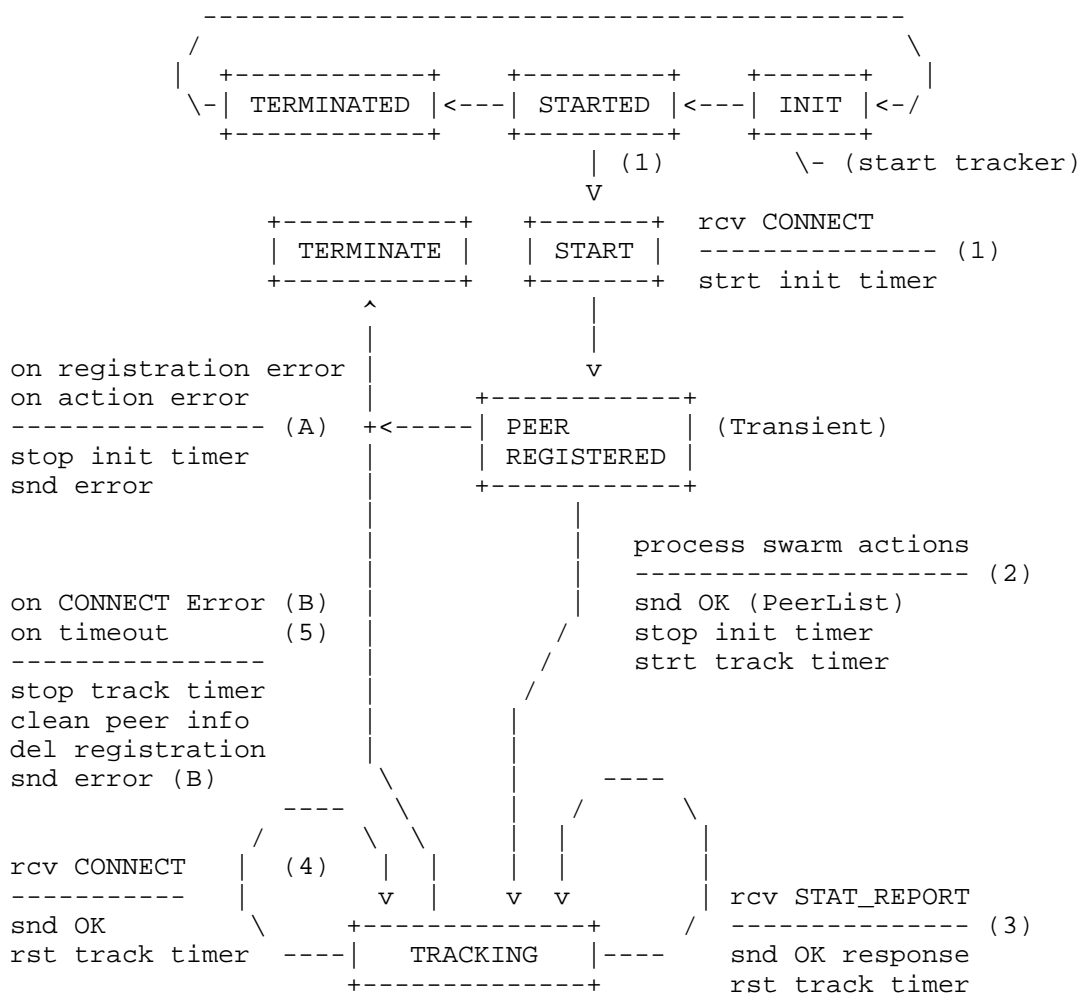


Figure 5: Per-Peer-ID Transaction State Machine and Flow Diagram

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 "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 and STAT_REPORT messages, or
- o Timeout events.

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

6.1. Normal Operation

On normal operation the process consists of the following steps:

- 1) When a 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 CONNECT request triggers a new "per-Peer-ID" State machine. In the START state the tracker initiates the registration of 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 processes the requested action(s) for the valid swarm information. 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), depending on peer mode (section 8.1). At the PEER REGISTERED state, if Peer-ID is considered invalid, the tracker responds with either error codes 401 Unauthorized or 403 Forbidden (described in section 7), and transitions to TERMINATE state for that Peer-ID.

- 3) In TRACKING state, a STAT_REPORT message received from the peer resets the "track timer" and is responded with a successful condition.
- 4) While TRACKING, a CONNECT message received with valid swarm actions information (section 8.1) from the peer resets the "track timer" and is responded with a successful condition.
- 5) 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, and transitions to TERMINATE state for that Peer-ID.

6.2. Error Conditions

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

- A) At PEER REGISTERED state, when a CONNECT Request only contains invalid swarm actions (section 8.1), the tracker responds with error code 403 Forbidden, deletes the registration and transition to TERMINATE state for that Peer-ID.
- B) At the TRACKING state (while the "track timer" has not expired) receiving a CONNECT message from the peer with invalid swarm actions (section 8.1) is considered an error condition. The tracker responds with error code 403 Forbidden (described in section 7), stops the "track timer", deletes the registration and transitions to TERMINATE state for that Peer-ID.

NOTE: These situations may correspond to a malfunction at the peer or to malicious conditions. Therefore, the adequate preventive measure is to proceed to TERMINATE state for the Peer-ID by de-registering the peer and cleaning all peer information.

7. Protocol Specification

7.1. Request/Response Syntax and Format

The message-body for Requests and Responses requiring it, correspond to a XML document [XML], optionally represented in a binary compact form using for example Efficient XML Interchange (EXI) Format [EXI Format].

The XML message-body MUST begin with an XML declaration line specifying the version of XML being used and indicating the character encoding, which SHOULD be "UTF-8". The root element MUST begin with an <PPSPTrackerProtocol> element. This element identifies the start of an PPSP-TP protocol element, the namespace used within the protocol, and the location of the protocol schema.

The generic format of a Request is the following:

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

The generic format of a Response is the following:

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

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

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

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

The element SwarmID MUST be present in requests to identify the actions to be taken in the specified swarms.

The version of PPSP-TP being used is indicated by the attribute @version of the root element, specified in an XML Schema ([XMLSchema.1] and [XMLSchema.2]) with XML namespaces [XMLNameSpace] to identify protocol grammars.

All Request messages MUST contain a PeerID element to uniquely

identify the peer (Peer-ID) in the network.

The PeerID information may be present on the following levels:

- On PPSPTrackerProtocol level in PPSPTrackerProtocol.PeerID element. For details refer to 7.2.1 Table 2.
- On PeerGroup level in PeerGroup.PeerInfo.PeerID element. For details refer to 7.2.1 Table 3.

The SwarmID element MUST be present in CONNECT Requests and SHOULD be present in STAT_REPORT Requests on the following levels:

- On PPSPTrackerProtocol level in PPSPTrackerProtocol.SwarmID element. For details refer to 7.2.1 Table 2.
- On StatisticsGroup level in StatisticsGroup.Stat.SwarmID element. For details refer to 7.2.1 Table 4.

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

The StatisticsGroup element MAY be present in STAT_REPORT requests. Details of usage in 8.2.

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

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

7.2. Semantics of PPSPTrackerProtocol elements

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

Element Name or Attribute Name	Use	Description
PPSPTrackerProtocol	1	The root element.
@version	M	Provides the version of PPSP-TP.
Request	0...1	Provides the request method and MUST be present in Request.
Response	0...1	Provides the response method and MUST be present in Response.
TransactionID	M	Root transaction Identification.
Result	0...N	Result of @action MUST be present in Responses.
@transactionID	CM	Identifier of the @action.
PeerID	0...1	Peer Identification. MUST be present in Request.
SwarmID	0...N	Swarm Identification. MUST be present in Requests.
@action	CM	Must be set to JOIN or LEAVE.
@peerMode	CM	Mode of Peer participation in the swarm, "LEECH" or "SEED".
@transactionID	CM	Identifier for the @action.
PeerNUM	0...1	Maximum peers to be received with capabilities indicated.
@abilityNAT	CM	Type of NAT traversal peers, as "No-NAT", "STUN", "TURN" or "PROXY"
@concurrentLinks	CM	Concurrent connectivity level of peers, "HIGH", "LOW" or "NORMAL"
@onlineTime	CM	Availability or online duration of peers, "HIGH" or "NORMAL"
@uploadBWlevel	CM	Upload bandwidth capability of peers, "HIGH" or "NORMAL"
PeerGroup	0...1	Information on peers (Table 3)
StatisticsGroup	0...1	Statistic data of peer (Table 4)
Legend: Use for attributes: M=Mandatory, OP=Optional, CM=Conditionally Mandatory Use for elements: minOccurs...maxOccurs (N=unbounded) Elements are represented by their name (case-sensitive) Attribute names (case-sensitive) are preceded with an @		

Table 2: Semantics of the base PPSPTrackerProtocol.

Element Name or Attribute Name	Use	Description
PeerGroup	0...1	Contains description of peers.
PeerInfo	1...N	Provides information on a peer.
@swarmID	0...1	Swarm Identification.
PeerID	0...1	Peer Identification.
		MAY be present in responses.
PeerAddress	0...N	IP Address information.
@addrType	M	Type of IP address, which can be "ipv4" or "ipv6"
@priority	CM	The priority of this interface. Used for NAT traversal.
@type	CM	Describes the address for NAT traversal, which can be "HOST" "REFLEXIVE" or "PROXY".
@connection	OP	Access type ("3G", "ADSL", etc.)
@asn	OP	Autonomous System number.
@ip	M	IP address value.
@port	M	IP service port value.
@peerProtocol	OP	PPSP Peer Protocol supported.
Legend: Use for attributes: M=Mandatory, OP=Optional, CM=Conditionally Mandatory Use for elements: minOccurs...maxOccurs (N=unbounded) Elements are represented by their name (case-sensitive) Attribute names (case-sensitive) are preceded with an @		

Table 3: Semantics of PeerGroup.

If STUN-like functions are enabled in the tracker and a PPSP-ICE method is used the attributes @type and @priority MUST be returned with the transport address candidates in responses to CONNECT requests.

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

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

Element Name or Attribute Name	Use	Description
StatisticsGroup	0...1	Provides statistic data on peer and content.
Stat	1...N	Groups statistics property data.
@property	M	The property to be reported. Property values in Table 6.
SwarmID	0...1	Swarm Identification.
UploadedBytes	0...1	Bytes sent to swarm.
DownloadedBytes	0...1	Bytes received from swarm.
AvailBandwidth	0...1	Upstream Bandwidth available.
Legend: Use for attributes: M=Mandatory, OP=Optional, CM=Conditionally Mandatory Use for elements: minOccurs...maxOccurs (N=unbounded) Elements are represented by their name (case-sensitive) Attribute names (case-sensitive) are preceded with an @		

Table 4: Semantics of StatisticsGroup.

The Stat element 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.

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

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

7.3. Request element in request Messages

Table 5 defines the valid string representations for the requests. These values MUST be treated as case-sensitive.

XML Request Methods String Values
CONNECT STAT_REPORT

Table 5: Valid Strings for Request element of requests.

7.4. Response element in response Messages

Table 6 defines the valid string representations for Response messages that require message-body. These values MUST be treated as case-sensitive.

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

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

Table 6: Valid Strings for Response element of responses.

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

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 join @action requested.

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

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

8. Request/Response Processing

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

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

8.1. CONNECT Request

This method is used when a peer registers to the system and/or requests swarm actions.

The peer **MUST** properly form the XML message-body, set the Request method to **CONNECT**, generate and set the TransactionIDs, set the PeerID with the identifier of the peer and include SwarmID action(s) to be performed. The peer **MAY** also include the IP addresses of its network interfaces in the **CONNECT** message.

An example of the message-body of a **CONNECT** Request for a peer Seeder is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD" schemaLocation="TBD" version="1.0">
  <Request>CONNECT</Request>
  <PeerID>656164657220</PeerID>
  <SwarmID action="JOIN" peerMode="SEED"
    transactionID="12345.1">1111</SwarmID>
  <SwarmID action="JOIN" peerMode="SEED"
    transactionID="12345.2">2222</SwarmID>
  <TransactionID>12345.0</TransactionID>
</PPSPTrackerProtocol>
```

An example of the message-body of a **CONNECT** Request for a peer Leecher requesting join to a swarm and providing optional information on the addresses of its interfaces is the following:


```

<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD" schemaLocation="TBD" version="1.0">
  <Request>CONNECT</Request>
  <PeerID>656164657221</PeerID>
  <PeerNum abilityNAT="STUN">5</PeerNum>
  <SwarmID action="JOIN" peerMode="LEECH"
    transactionID="12345.1">1111</SwarmID>
  <TransactionID>12345.0</TransactionID>
  <PeerGroup>
    <PeerInfo>
      <PeerAddress addrType="ipv4" ip="192.0.2.2" port="80"
        priority="1" peerProtocol="PPSP-PP" />
      <PeerAddress addrType="ipv6" ip="2001:db8::2" port="80"
        priority="2" peerProtocol="PPSP-PP" />
    </PeerInfo>
  </PeerGroup>
</PPSPTrackerProtocol>

```

An example of the message-body of a CONNECT Request for a peer Leecher "leaving" a previously joined swarm and requesting join to a new swarm is the following:

```

<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD" schemaLocation="TBD" version="1.0">
  <Request>CONNECT</Request>
  <PeerID>656164657221</PeerID>
  <PeerNum abilityNAT="STUN">5</PeerNum>
  <SwarmID action="LEAVE" peerMode="LEECH"
    transactionID="12345.1">1111</SwarmID>
  <SwarmID action="JOIN" peerMode="LEECH"
    transactionID="12345.2">2222</SwarmID>
  <TransactionID>12345.0</TransactionID>
</PPSPTrackerProtocol>

```

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 element SwarmID MUST be present with cardinality 1 to N, each containing the request for @action, the @peerMode of the peer and the child @transactionID for that swarm. The @peerMode element MUST be set to the type of participation of the peer in the swarm (SEED or LEECH).

The valid sets of SwarmID @action combinations for the CONNECT Request logic in PPSP-TP base protocol are summarized in Table 7 (referring to the tracker State Machine).

SwarmID Elements	@peerMode 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 7: Validity of SwarmID @action combinations in CONNECT Request.

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

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

If STUN-like function is enabled in the tracker, the response MAY include the peer reflexive address.

The response MUST have the same TransactionID values as the corresponding request and actions.

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

following:

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD" schemaLocation="TBD" version="1.0">
  <Response>SUCCESSFUL</Response>
  <TransactionID>
    <Result transactionID="12345.0">200 OK</Result>
    <Result transactionID="12345.1">200 OK</Result>
    <Result transactionID="12345.2">200 OK</Result>
  </TransactionID>
  <PeerGroup>
    <PeerInfo>
      <PeerID>656164657221</PeerID>
      <PeerAddress addrType="ipv4" ip="198.51.100.1" port="80"
        priority="1"
        type="REFLEXIVE"
        connection="3G"
        asn="64496" />
    </PeerInfo>
    <PeerInfo swarmID="2222">
      <PeerID>956264622298</PeerID>
      <PeerAddress addrType="ipv4" ip="198.51.100.22" port="80"
        asn="64496" peerProtocol="PPSP-PP" />
    </PeerInfo>
    <PeerInfo swarmID="2222">
      <PeerID>3332001256741</PeerID>
      <PeerAddress addrType="ipv4" ip="198.51.100.201" port="80"
        asn="64496" peerProtocol="PPSP-PP" />
    </PeerInfo>
  </PeerGroup>
</PPSPTrackerProtocol>
```

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

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

In case the @peerMode is LEECH the tracker will search and select an appropriate list of peers satisfying the conditions 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 [I.D.ietf-alto-protocol].

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

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

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

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD" schemaLocation="TBD" version="1.0">
  <Request>STAT_REPORT</Request>
  <PeerID>656164657221</PeerID>
  <TransactionID>12345</TransactionID>
  <StatisticsGroup>
    <Stat property="StreamStatistics">
      <SwarmID>1111</SwarmID>
      <UploadedBytes>512</UploadedBytes>
      <DownloadedBytes>768</DownloadedBytes>
      <AvailBandwidth>1024000</AvailBandwidth>
    </Stat>
  </StatisticsGroup>
</PPSPTrackerProtocol>
```

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 TransactionID value as the request.

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

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

8.3. Error and Recovery conditions

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

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

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

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

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

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

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

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

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

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

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

9. Security Considerations

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

No security system can 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].

9.1. Authentication between Tracker and Peers

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

9.2. Content Integrity protection against polluting peers/trackers

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

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

9.3. Residual attacks and mitigation

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

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

9.4. Pro-incentive parameter trustfulness

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

Trustworthiness of these pro-incentive parameters is critical to the effectiveness of the incentive mechanisms.

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

10. 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.

Since the beginning that extensibility has been a design goal of PPSP-TP, as the protocol is represented in the Extensible Markup Language [XML].

A powerful feature of XML is the extensibility, but this feature also provides multiple opportunities to make poor design decisions.

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.

10.1. Forms of PPSP-TP Extension

PPSP-TP extensions MUST be specified in XML. This ensures that parsers capable of processing PPSP-TP structures will also be capable of processing PPSP-TP extensions. Guidelines for the use of XML in IETF protocols can be found in RFC 3470 [RFC3470].

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 StatisticsGroup (Section 7.2.1, Table 4) 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:

```
<Stat property="StreamStatistics">
  <OnlineTime>3600</OnlineTime>
</Stat>
```

Another example also for the StatisticsGroup is for additional @property attributes and elements, such as those necessary to report content chunk availability information:

```
<Stat property="ContentMap">
  <SegmentInfo chunkmapSize="25">
    A/8D/wP/A/8D/wP/A/8D/wP/A/8D/wP/....
  </SegmentInfo>
</Stat>
```

- 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 Leecher, 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  |
+-----+
|
|  --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.

10.2. 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.

11. IANA Considerations

There are presently no IANA considerations with this document.

12. Acknowledgments

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

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

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the SARACEN project or the European Commission.

13. References

13.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.
- [RFC5245] Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", RFC 5245, April 2010.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC5322] Resnick, P., Ed., "Internet Message Format", RFC 5322, October 2008.
- [ISO.8601.2004] International Organization for Standardization, "Data elements and interchange formats - Information interchange - Representation of dates and times", ISO Standard 8601, December 2004.
- [XML] Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E. and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth Edition)", W3C xml, November 2008, <<http://www.w3.org/TR/xml/>>.
- [XMLSchema.1] Thompson, H., Beech, D., Maloney, M. and N. Mendelsohn, "XML Schema Part 1: Structures Second Edition", W3C xmlschema-1, October 2004, <<http://www.w3.org/TR/xmlschema-1/>>.
- [XMLSchema.2] Biron, P. and A. Malhotra, "XML Schema Part 2: Datatypes Second Edition", W3C xmlschema-2, October 2004, <<http://www.w3.org/TR/xmlschema-2/>>.

[XMLNameSpace] Bray, T., Hollander, D., Layman, A., Tobin, R. and H. Thompson, "Namespaces in XML", W3C REC-xml-names, December 2009, <<http://www.w3.org/TR/REC-xml-names/>>.

[EXI Format] Schneider, J. and T. Kamiya, "Efficient XML Interchange (EXI) Format 1.0", W3C exi, March 2011, <<http://www.w3.org/TR/exi/>>.

13.2. Informative References

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

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

[RFC3470] Hollenbeck, S., Rose, M., and L. Masinter, "Guidelines for the Use of Extensible Markup Language (XML) within IETF Protocols", BCP 70, RFC 3470, January 2003.

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

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, October 2012.

[I-D.ietf-ppsp-problem-statement] Zhang, Y., Zong, N., Camarillo, G., Seng, J. and Y. Yang, "Problem Statement of P2P Streaming Protocol (PPSP)", draft-ietf-ppsp-problem-statement-11 (work in progress), October 2012.

[I-D.pantos-http-live-streaming] Pantos. R. and W. May, "HTTP Live Streaming", draft-pantos-http-live-streaming-10 (work in progress), October 2012.

[I.D.ietf-alto-protocol] Alimi, R., Penno, R. and Y. Yang, "ALTO Protocol", draft-ietf-alto-protocol-13, (work in progress), September 2012.

[I-D.montenegro-httpbis-speed-mobility] Trace, R., Foresti, A., Singhal, S., Mazahir, O., Nielsen, H., Raymor, B., Rao, R. and G. Montenegro, "HTTP Speed+Mobility," draft-montenegro-httpbis-speed-mobility-02 (work in progress), June 2012.

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

[refs.saracenwebpage] "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. PPSP-TP Message Syntax for HTTP/1.1

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

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

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

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

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

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

A Request message example is the following:

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

[Request_Body]
```

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

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

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

A Response message example is the following:

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

```
[Response_Body]
```

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

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

A.1. Header Fields

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

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

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

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

The Content-Encoding entity-header field MAY be used in response messages with "gzip" compression scheme [RFC1952] for faster transmission times and less network bandwidth usage.

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

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

A.2. Methods

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

A.3. Message Bodies

PPSP-TP requests MUST contain message-bodies.

PPSP-TP responses MAY include a message-body.

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

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

A.4. Message Response Codes

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

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

2xx: Success -- the action was successfully received, understood, and accepted;

4xx: Peer Error -- the request contains bad syntax or cannot be fulfilled at this tracker;

5xx: Tracker Error -- the tracker failed to fulfill an apparently valid request;

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

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

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

401 Unauthorized -- The request requires authentication.

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

404 Not Found -- This status is returned if the tracker did not find

anything matching the Request-URI.

- 408 Request Timeout -- The peer did not produce a request within the time that the tracker was prepared to wait.
- 411 Length Required -- The tracker refuses to accept the request without a defined Content-Length. The peer MAY repeat the request if it adds a valid Content-Length header field containing the length of the message-body in the request message.
- 414 Request-URI Too Long -- The tracker is refusing to service the request because the Request-URI is longer than the tracker is willing to interpret. This rare condition is likely to occur when the tracker is under attack by a client attempting to exploit security holes.
- 500 Internal Server Error -- The tracker encountered an unexpected condition which prevented it from fulfilling the request.
- 503 Service Unavailable -- The tracker is currently unable to handle the request due to a temporary overloading or maintenance condition.

Authors' Addresses

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

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

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

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

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

Deng Lingli
China Mobile

PPSP
INTERNET-DRAFT
Intended Status: Standards Track
Expires: April 25, 2013

Rachel Huang
Huawei
Rui S. Cruz
Mario S. Nunes
IST/INESC-ID/INOV
Joao P. Taveira
IST/INOV
October 22, 2012

PPSP Tracker Protocol--Extended Protocol (PPSP-TP/1.1)
draft-huang-ppsp-extended-tracker-protocol-01

Abstract

This document specifies the extended Peer-to-Peer Streaming Protocol - Tracker Protocol (PPSP-TP/1.1), a new extension protocol complementing the basic core messages and usages specified in PPSP-TP/1.0 for the exchange of meta information between trackers and peers, such as initial offer/request of participation in multimedia content streaming, content information, peer lists and reports of activity and status. It extends PPSP-TP/1.0 to include new optional messages providing granular controls and usages in the communications between peer and tracker. The extension protocol is retro-compatible with PPSP-TP/1.0.

Status of this Memo

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

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

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

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

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

Copyright and License Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	4
2. Terminology	4
3. PPSP-TP/1.1 Overview	4
3.1. Request Messages	4
3.1.1. Enhanced Request Messages	4
3.1.1.1 CONNECT	4
3.1.1.2 STAT_REPORT	5
3.1.2. New Request Messages	5
3.1.2.1. JOIN	5
3.1.2.2. DISCONNECT	6
3.1.2.3. FIND	6
3.2. Extended Tracker Transaction State Machine	8
3.2.1. Normal Operation	9
3.2.2. Error Conditions	11
3.3. Extended Request/Response Syntax and Format	11
3.3.1. Extended Semantics of PPSPTrackerProtocol Elements	14
3.3.2. Extended Request/Response Element in Request Messages	18
3.4. Compatibility with PPSP-TP/1.0	18
4. Request/Response Processing	19
4.1. Enhanced CONNECT Request	19
4.1.1 Registration CONNECT Request	20
4.1.2 Fast CONNECT Request	21
4.2. DISCONNECT Request	24
4.3. JOIN Request	26
4.4. FIND Request	29
4.5. Enhanced STAT_REPORT Request	32
4.6. Error and Recovery Conditions	34
5. Security Considerations	34
6. IANA Considerations	34
7. Acknowledgments	34
8. References	35
8.1 Normative References	35
8.2 Informative References	35
Authors' Addresses	36

1. Introduction

The PPSP Tracker Protocol is one of the Peer-to-Peer Streaming Protocol which specifies standard format/encoding of information and messages between PPSP peers and PPSP trackers. Based on the requirements defined in [I-D.ietf-ppsp-problem-statement], PPSP-TP/1.0 specified in [I-D.cruz-ppsp-base-tracker-protocol] has provided the basic core messages to be exchanged between trackers and peers in order to carry out some fundamental operations. They are mandatory messages covering most basic use cases and MUST be implemented in all PPSP-based streaming systems.

This document specifies PPSP-TP/1.1 to complement the basic core messages and usages specified in [I-D.cruz-ppsp-base-tracker-protocol]. It extends PPSP-TP/1.0 to include some new optional messages for providing granular controls and usages in some dedicated scenarios. The extension protocol is retro-compatible with PPSP-TP/1.0. For a peer, it can implement either PPSP-TP/1.0 or PPSP-TP/1.1. For a tracker, it is recommended to implement PPSP-TP/1.1, which is also able to deal with the requests of PPSP-TP/1.0.

2. Terminology

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

This draft uses terms defined in [I-D.ietf-ppsp-problem-statement] and [I-D.cruz-ppsp-base-tracker-protocol].

3. PPSP-TP/1.1 Overview

3.1. Request Messages

3.1.1. Enhanced Request Messages

In this section, the request messages specified in PPSP-TP/1.0 are extended to enable granular control and optional information delivery.

3.1.1.1 CONNECT

Besides the semantics defined in PPSP-TP/1.0 specification, the enhanced CONNECT Request message is also used when a peer registers in the tracker without simultaneously requesting additional actions to be carried upon successful registration. In such a case, the tracker records the Peer-ID, connect-time (referenced to the absolute time), optional peer IP addresses and link status (if present in the

request), and waits for further requests from the peer.

This message is also extended to allow a peer participating in some swarms as LEECH and simultaneously joining other swarms as SEED when it registers in the tracker. For example, a peer is a seeder of several certain contents that it generated, but it is also a consumer of a streaming program. For some reason the connectivity failed and it wants to reconnect from the point it was before. Then the enhanced CONNECT request message could satisfy this kind of usage.

3.1.1.2 STAT_REPORT

The STAT_REPORT Request message is extended to allow the exchange of content data information, like chunkmaps, between an active peer and a tracker. The information can be used by a tracker as a qualification to select appropriate peer lists when peers request to the tracker for the peer lists of some specific contents. An example of a STAT_REPORT for multiple properties is illustrated in subsection 4.5.

3.1.2. New Request Messages

Three new messages, listed in Table 1, are introduced in this section to extend those specified in PPSP-TP/1.0 [I-D.cruz-ppsp-base-tracker-protocol].

+-----+	
	PPSP Tracker
	Extension
	Req. Messages
+-----+	
	DISCONNECT
	JOIN
	FIND
+-----+	

Table 1: Extended Request Messages

3.1.2.1. JOIN

The JOIN Request message is used for a peer to notify the tracker that it wishes to participate in a swarm. The tracker records the content availability, i.e., adds the peer to the peers list for the swarm. On receiving a JOIN message, the tracker first checks the PeerMode type and then decides the next step (more details are referred in section 4.3).

A use case of this request message is showed in Figure 1. In this

case, the seeder of a certain provider has already registered in a tracker through the CONNECT message. When it receives some "Start" activation signal for a program x, it could send the JOIN message to the connected tracker to join the swarm. When it receives another signal for joining another program y, it can send the JOIN message to the tracker to request for joining the other swarm, but remaining sharing the content for the first one.

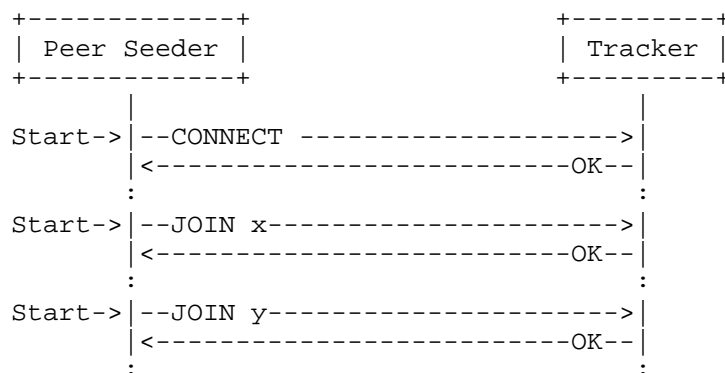


Figure 1: An Example of JOIN Request Message Usages

3.1.2.2. DISCONNECT

The DISCONNECT Request message is used when the peer intends to no longer participate in a specific swarm, or in all swarms. When receiving the message from a peer, the tracker deletes the corresponding activity records related to the peer (including its status and all content status for the corresponding swarms).

Continuing the example in 3.1.1, when the newly joined program is no longer available, a "DISCONNECT z" message is sent by the seeder to indicate that it leaves swarm z. Note that this activity does not affect other swarms (swarm x, y). The seeder still maintains active all the other swarms it participates in.

Another use case is that this request message can be used to stop peer participation in all swarms and de-register from the system. In such a case, DISCONNECT message will have the same effect of timer expiring (from the tracker perspective), but providing a graceful disconnect from the system.

3.1.2.3. FIND

The FIND Request message allows a peer to request the tracker for the peer list of a swarm. The request can include specific content

scopes, either media content representations or specific chunks of a media representation in a 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.

This message can be used in scenarios when peers want to obtain the updated peer lists from the tracker. For example, when a peer acting as leech has joined a swarm, and after a while part of the content is not available in the previously known peers, it can send a FIND request message to the tracker to update the peer lists which could satisfy the peer's requirement.

Another scenario could be found when a peer has changed its primary network attachment. One example is that a peer with a LAN and a WiFi interface which are going through different routers. The peer is using some PPSP-based P2P application which can keep working when the peer switches from the LAN to the WiFi (for example, unplugging the Ethernet cable, the P2P connection can recover automatically). In this case, the peer can send a FIND Request message to the tracker for the updated peer lists. And the response replied by the tracker to the FIND Request message should include the requesting peer transport address as well as the required peer list.

An example of usage of the enhanced request messages in PPSP-TP/1.1 is the illustrated in Figure 2.

In that figure a peers starts by connecting to the system (using the semantics of PPSP-TP/1.1 Registration CONNECT) after which JOINS a specific swarm (swarm_a) in SEED mode.

While active the peer periodically updates the tracker using STAT_REPORT messages. Later, the peer JOINS another swarm (swarm_b) but in LEECH mode, i.e., the end-user intended to watch that content while still sharing the first one. During the stream the peer requests an updated list of peers in that swarm to the tracker.

When finished streaming the second content the peer DISCONNECTs from the corresponding swarm (swarm_b) while still sharing the first content (swarm_a).

Later the peer DISCONNECTs from the system.

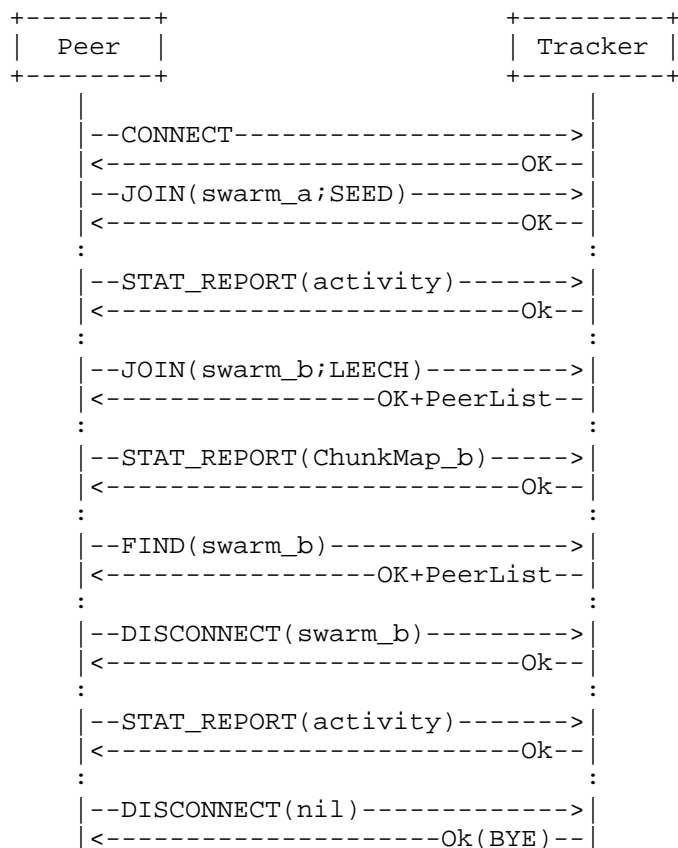


Figure 2: Example of a session for a PPSP-TP extended version.

3.2. Extended Tracker Transaction State Machine

The tracker state machine has been introduced in PPSP-TP/1.0 [I-D.cruz-ppsp-base-tracker-protocol]. Every tracker MUST keep a tracker state machine in which the state transitions are triggered by peer registrations. In addition to the tracker state machine, a transaction state machine for each registered Peer-ID is also specified. In this specification, as some additional granularity messages have been introduced, an extended "per-Peer-ID" transaction state machine (Figure 2) is specified to provide more functionality and detailed control to the tracker protocol. PPSP-TP/1.1 MUST include both "per-Peer-ID" transaction state machines to remain compatible with PPSP-TP/1.0. A Tracker implementing PPSP-TP/1.1 could instantiate corresponding "per-Peer-ID" transaction state machine based on the version of the CONNECT request message received from the peer.

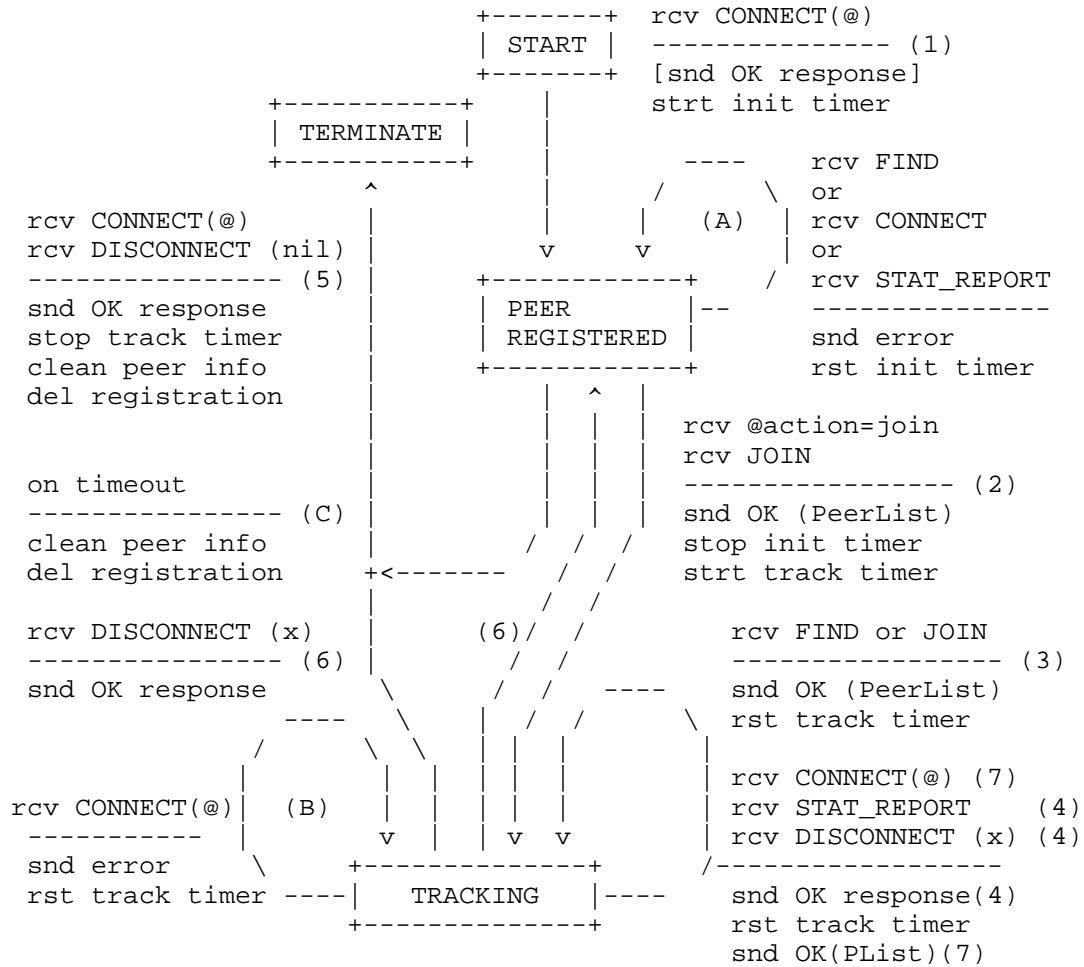


Figure 3: Extended Per-Peer-ID Transaction State Machine

The state diagram in Figure 3 illustrates the complete state changes together with the causing events and resulting actions when implementing basic tracker protocol with extended protocol. Note that specific error conditions are not shown in the state diagram.

3.2.1. Normal Operation

On normal operation the extended process consists of the following steps:

- 1) When an enhanced CONNECT message is received from a peer, if

successfully authenticated and validated, the tracker registers the Peer-ID and associated information (IP addresses). In case the CONNECT request also includes Swarm @action requests (using the same semantics of PPSP-TP/1.0), it moves to PEER REGISTERED state carrying the Swarm @action to be performed. In case the CONNECT request does not contain any Swarm @action requests, it sends the response of successful registration to peer, starts the "init timer" and moves to PEER REGISTERED state.

- 2) While PEER REGISTERED, when a JOIN message or swarm @action="JOIN" request is received with valid swarm information, the tracker stops the "init timer", starts the "track timer" and sends the response of successful join to the peer. The response MAY contain the appropriate list of peers in the swarm, depending on PeerMode. A successful request in this state starts the TRACKING state associated with the peer-ID for the requested swarm.
- 3) While TRACKING, a JOIN or FIND message received with valid swarm information from the peer resets the "track timer" and is responded with a successful condition, either for the JOIN to (an additional) swarm or for including the appropriate list of peers for the scope in the FIND request.
- 4) While TRACKING, a DISCONNECT(x) message received from the peer, containing a valid x=Swarm-ID resets the "track timer" and is responded with a successful condition. The tracker cleans the information associated with the participation of the Peer-ID in the specified swarm(s).

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

- 5) From either PEER REGISTERED or TRACKING states a DISCONNECT(x) message received from the peer, where x=nil, the tracker stops the "track timer", cleans the information associated with the participation of the Peer-ID in the the swarm(s) joined, responds with a successful condition, deletes the registration of the Peer-ID and transitions to TERMINATED state for that Peer-ID. The same operations are performed if a CONNECT message including valid Swarm @action="LEAVE" requests (using the same semantics of PPSP-TP/1.0) is received while in TRACKING state (valid actions in Table 9).
- 6) From TRACKING state a DISCONNECT(x) message received from the peer, where x=ALL or x=Swarm-ID is the last swarm, the tracker stops the "track timer", cleans the information associated with

the participation of the Peer-ID in the the swarm(s) joined, responds with a successful condition and transitions to PEER REGISTERED state.

- 7) While TRACKING, a CONNECT message received from the peer including a valid pair of Swarm @action requests (one being @action="LEAVE" and the other being @action="JOIN", using the same semantics of PPSP-TP/1.0), resets the "track timer" and is responded with a successful condition (valid actions in Table 9). For the @action="JOIN" the response to the request MAY include the appropriate list of peers.

3.2.2. Error Conditions

- A) At the PEER REGISTERED state (while the "init timer" has not expired) receiving FIND, CONNECT or STAT_REPORT messages from the peer is considered as an error condition. The tracker responds with error code 403 Forbidden, and resets the "init timer" one last time.
- B) At the TRACKING state (while the "track timer" has not expired) receiving an enhanced CONNECT message or swarm @action="JOIN" request from the peer is considered an error condition. The tracker responds with error code 403 Forbidden, and resets the "track timer".

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

- C) Without receiving messages from the peer, either from PEER REGISTERED state (controlled by init timer) or TRACKING state (controlled by track timer), on timeout the tracker cleans all the information associated with the Peer-ID in all swarms it was joined, deletes the registration, and transitions to TERMINATE state for that Peer-ID. The same action is taken if no valid message is received at the PEER REGISTERED state after the last "init timer" expires.

3.3. Extended Request/Response Syntax and Format

The architecture of PPSP-TP/1.1 is consistent with the architecture of the the basic tracker protocol specified in PPSP- TP/1.0 [I-D.cruz-ppsp-base-tracker-protocol]. It still uses the layered architecture of PPSP-TP/1.0. Besides that, the message syntax is identical with that used by PPSP-TP/1.0. Note that PPSP-TP/1.1 is

compatible with all the syntax and formats of PPSP-TP/1.0. The difference is that some request/response syntax has been extended in PPSP-TP/1.1 to contain the new optional messages. This section extends the generic format of Request to satisfy the extended request messages. The extended generic format of a Request is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
                      schemaLocation="TBD"
                      version="1.1">
  <Request></Request>
  <TransactionID></TransactionID>
  <PeerID></PeerID>
  <SwarmID></SwarmID>
  <PeerNum></PeerNum>
  <PeerGroup></PeerGroup>
  <ContentGroup></ContentGroup>
  <StatisticsGroup></StatisticsGroup>
</PPSPTrackerProtocol>
```

The generic format of a Response is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
                      schemaLocation="TBD"
                      version="1.1">
  <Response></Response>
  <TransactionID></TransactionID>
  <PeerGroup></PeerGroup>
</PPSPTrackerProtocol>
```

The version of PPSPTrackerProtocol has been changed to 1.1 which indicates that the peer is using extended tracker protocol (PPSP-TP/1.1).

Besides that, syntax of some elements has been extended for three extended request messages.

The SwarmID element MUST be present in FIND, JOIN and DISCONNECT requests.

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

The PeerGroup element MAY be present in CONNECT requests and

responses and MAY be present in responses to JOIN and FIND requests if the corresponding responses return information about peers.

One element "ContentGroup" is added to the format of Request. It MAY be present in requests referencing content, i.e., JOIN and FIND, if the request includes a content scope.

The extended semantics of the attributes and elements within a PPSPTrackerProtocol root element is described in subsection 3.3.1.

3.3.1. Extended Semantics of PPSPTrackerProtocol Elements

The extended semantics for PPSP-TP/1.1 are described bellow.

Element Name or Attribute Name	Use	Description
PPSPTrackerProtocol	1	The root element.
@version	M	Provides the version of PPSP-TP.
Request	0...1	Provides the request method and MUST be present in Request.
Response	0...1	Provides the response method and MUST be present in Response.
TransactionID	M	Root transaction Identification.
Result	0...N	Result of @action MUST be present in Responses.
@transactionID	CM	Identifier of the @action.
PeerID	0...1	Peer Identification. MUST be present in Request.
SwarmID	0...N	Swarm Identification. MUST be present in Requests.
@action	CM	Must be set to JOIN or LEAVE.
@peerMode	CM	Mode of Peer participation in the swarm, "LEECH" or "SEED".
@transactionID	CM	Identifier for the @action.
PeerNUM	0...1	Maximum peers to be received with capabilities indicated.
@abilityNAT	CM	Type of NAT traversal peers, as "No-NAT", "STUN", "TURN" or "PROXY"
@concurrentLinks	CM	Concurrent connectivity level of peers, "HIGH", "LOW" or "NORMAL"
@onlineTime	CM	Availability or online duration of peers, "HIGH" or "NORMAL"
@uploadBWlevel	CM	Upload bandwidth capability of peers, "HIGH" or "NORMAL"
PeerGroup	0...1	Information on peers (Table 3)
ContentGroup	0...1	Information on content (Table 4)
StatisticsGroup	0...1	Statistic data (Table 5)
Legend: Use for attributes: M=Mandatory, OP=Optional, CM=Conditionally Mandatory Use for elements: minOccurs...maxOccurs (N=unbounded) Elements are represented by their name (case-sensitive) Attribute names (case-sensitive) are preceded with an @		

Table 2: Semantics of the Extended PPSPTrackerProtocol.

The semantics of PeerGroup element is almost identical with that in PPSP-TP/1.0. It is listed below for convenience of reading. The implementation of this element has been extended in this specification, which is described in 4.3.

Element Name or Attribute Name	Use	Description
PeerGroup	0...1	Contains description of peers.
PeerInfo	1...N	Provides information on a peer.
@swarmID	0...1	Swarm Identification.
PeerID	0...1	Peer Identification.
PeerAddress	0...N	MAY be present in responses.
@addrType	M	IP Address information.
@priority	CM	Type of IP address, which can be "ipv4" or "ipv6"
@type	CM	The priority of this interface.
@connection	OP	Used for NAT traversal.
@asn	OP	Describes the address for NAT traversal, which can be "HOST" "REFLEXIVE" or "PROXY".
@ip	M	Access type ("3G", "ADSL", etc.)
@port	M	Autonomous System number.
@peerProtocol	OP	IP address value.
		IP service port value.
		PPSP Peer Protocol supported.
Legend: Use for attributes: M=Mandatory, OP=Optional, CM=Conditionally Mandatory Use for elements: minOccurs...maxOccurs (N=unbounded) Elements are represented by their name (case-sensitive) Attribute names (case-sensitive) are preceded with an @		

Table 3: Semantics of PeerGroup.

Table 4 describes the semantics of StatisticsGroup element.

Element Name or Attribute Name	Use	Description
StatisticsGroup	0...1	Provides statistic data on peer and content.
Stat	1...N	Groups statistics property data.
@property	M	The property to be reported. Property values in Table 5.
SwarmID	0...1	Swarm Identification.
UploadedBytes	0...1	Bytes sent to swarm.
DownloadedBytes	0...1	Bytes received from swarm.
AvailBandwidth	0...1	Upstream Bandwidth available.
Representation	0...N	Describes a component of content.
@id	CM	Unique identifier for this Representation.
SegmentInfo	1...N	Provides segment information by segment range. The chunkmap can be encoded in Base64 [RFC4648].
@startIndex	CM	The index of the first media segment in the chunkmap report for this Representation.
@endIndex	CM	The index of the last media segment in the chunkmap report for this Representation.
@chunkmapSize	CM	Size of chunkmap reported.
Legend: Use for attributes: M=Mandatory, OP=Optional, CM=Conditionally Mandatory Use for elements: minOccurs...maxOccurs (N=unbounded) Elements are represented by their name (case-sensitive) Attribute names (case-sensitive) are preceded with an @		

Table 4: Semantics of StatisticsGroup.

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

Table 5: StatisticsGroup Default Stat @property Values.

ContentGroup is a new element extended in this specification. The semantics of this element is described in Table 6.

Element Name or Attribute Name	Use	Description
ContentGroup	0...1	Provides information on content.
Representation	1...N	Describes a component of content.
@id	M	Unique identifier for this Representation.
SegmentInfo	1	Provides segment information.
@startIndex	M	The index of the first media segment in the request scope for this Representation.
@endIndex	OP	The index of the last media segment in the request scope for this Representation.
Legend: Use for attributes: M=Mandatory, OP=Optional, CM=Conditionally Mandatory Use for elements: minOccurs...maxOccurs (N=unbounded) Elements are represented by their name (case-sensitive) Attribute names (case-sensitive) are preceded with an @		

Table 6: Semantics of ContentGroup

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

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

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

component. An example is illustrated in subsection 4.3.

3.3.2. Extended Request/Response Element in Request Messages

Table 7 specifies the valid string representations for the requests in PPSP-TP/1.1. These values MUST be treated as case-sensitive.

Extended XML Request Methods String Values
CONNECT
DISCONNECT
JOIN
FIND
STAT_REPORT

Table 7: Extended Valid Strings for Request Element of Requests.

The response elements in response messages of PPSP-TP/1.1 are identical with those specified in PPSP-TP/1.0, which can be found in subsection 7.2.3 of [I-D.cruz-ppsp-base-tracker-protocol].

3.4. Compatibility with PPSP-TP/1.0

Peers and trackers may implement different versions of PPSP tracker protocol. Table 8 illustrate the different conditions when peer and tracker have different PPSP-TP versions.

Peer Version	Tracker Version	Request Version	Response Version
1.0	1.0	1.0	1.0
1.0	1.1	1.0	1.0
1.1	1.0	1.1	invalid
		1.0	1.0
1.1	1.1	1.1	1.1

Table 8: The Implementations of Different PPSP-TP Versions

Just as Table 8 shows, when both peer and tracker implement PPSP-

TP/1.0, all the requests and responses are processed based on semantics and schema of PPSP-TP/1.0.

When peer implements PPSP-TP/1.0 and tracker implements PPSP-TP/1.1, the peer sends CONNECT 1.0 and tracker responds with the semantics and version of PPSP-TP/1.0.

When peer implements PPSP-TP/1.1 and tracker implements PPSP-TP/1.0, the peer sends CONNECT 1.1 and tracker will respond with 400 (Bad request, with reason-phrase "PPSP version 1.0") to indicate that the peer has sent the invalid version of the request. After the peer receives the response with 400, it MUST send CONNECT 1.0 to be retro-compatible with the tracker implementing PPSP-TP/1.0.

When both peer and tracker implement PPSP-TP/1.1, peer may use an advanced CONNECT 1.1 with the semantics, rules and restrictions similar to the PPSP-TP/1.0 to switch channels, re-connect after a failure or use an advanced CONNECT 1.1 to register first and then wait for all other messages to perform other steps, such as JOIN, FIND, etc.

4. Request/Response Processing

4.1. Enhanced CONNECT Request

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

The peer MUST properly form the XML message-body, set the Request method to CONNECT, generate and set the TransactionID, and set the PeerID with the identifier of the peer. The peer MAY optionally include the IP addresses of its network interfaces in the CONNECT message, and so, the element PeerInfo MAY contain multiple PeerAddress child elements with attributes @addrType, @ip, @port and @peerProtocol, and optionally @priority and @type (if PPSP-ICE NAT traversal techniques are used) corresponding to each of the network interfaces of the peer.

The enhanced CONNECT Request has two variants: a registration CONNECT Request and a fast CONNECT request (retro-compatible with PPSP-TP/1.0).

4.1.1 Registration CONNECT Request

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

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
    schemaLocation="TBD"
    version="1.1">
  <Request>CONNECT</Request>
  <PeerID>656164657221</PeerID>
  <TransactionID>12345</TransactionID>
  <PeerGroup>
    <PeerInfo>
      <PeerAddress addrType="ipv4" ip="192.0.2.1" port="80"
        priority="1"
        type="HOST"
        peerProtocol="PPSP-PP" />
      <PeerAddress addrType="ipv6" ip="2001:db8::1" port="80"
        priority="2"
        type="HOST"
        peerProtocol="PPSP-PP" />
    </PeerInfo>
  </PeerGroup>
</PPSPTrackerProtocol>
```

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

The response to the registration CONNECT request MUST have the same TransactionID value as the request. An example of a Response message for the CONNECT Request is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
    schemaLocation="TBD"
    version="1.1">
  <Response>SUCCESSFUL</Response>
  <TransactionID>12345</TransactionID>
  <PeerGroup>
    <PeerInfo>
      <PeerAddress addrType="ipv4" ip="198.51.100.1" port="80"
        priority="1"
        type="REFLEXIVE"
        connection="ADSL"
        asn="64496" />
    </PeerInfo>
  </PeerGroup>
</PPSPTrackerProtocol>
```

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

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

4.1.2 Fast CONNECT Request

Fast CONNECT request is retro-compatible with the CONNECT request message defined in PPSP-TP/1.0 specification. An example of the message-body of the fast CONNECT Request is the following. This example is identical with PPSP-TP/1.0.

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
    schemaLocation="TBD"
    version="1.1">
  <Request>CONNECT</Request>
  <PeerID>656164657220</PeerID>
  <SwarmID action="JOIN" peerMode="SEED"
    transactionID="12345.1">1111</SwarmID>
  <SwarmID action="JOIN" peerMode="SEED"
    transactionID="12345.2">2222</SwarmID>
  <TransactionID>12345.0</TransactionID>
</PPSPTrackerProtocol>
```

Another example of fast CONNECT request usage is showed in below:


```

<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
    schemaLocation="TBD"
    version="1.1">
  <Request>CONNECT</Request>
  <PeerID>656164657220</PeerID>
  <SwarmID action="JOIN" peerMode="SEED"
    transactionID="12345.1">1111</SwarmID>
  <SwarmID action="JOIN" peerMode="SEED"
    transactionID="12345.2">2222</SwarmID>
  <SwarmID action="JOIN" peerMode="LEECH"
    transactionID="12345.3">3333</SwarmID>
  <TransactionID>12345.0</TransactionID>
</PPSPTrackerProtocol>

```

In this last example, the peer wants to re-join and participate in swarm 3333 to watch the program as a leecher, while sharing as seeder swarm 1111 and swarm 2222 with other peers.

SwarmID Elements	@peerMode 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	LEECH	JOIN	START	TERMINATE	Invalid
1	LEECH	LEAVE			
1	LEECH	JOIN	TRACKING	TRACKING	Valid
1	LEECH	LEAVE			
x	LEECH	JOIN	START	TRACKING	Valid
y	SEED	JOIN			
N	SEED	JOIN	START	TRACKING	Valid
N	SEED	JOIN	TRACKING	TERMINATE	Invalid
N	SEED	LEAVE	TRACKING	TERMINATE	Valid

Table 9: Validity of SwarmID @action combinations in CONNECT Request

It is also possible to simultaneously participate in multiple swarms

as LEECH, for example in situations when the peer has stopped watching several programs at a certain moment of the timeline (it may not have all the chunks of the contents) but may wish to continue sharing these programs with other peers. As such, a fast CONNECT request message would include some @action="JOIN" in LEECH mode and some in SEED mode.

In Table 9, the valid sets of SwarmID @action combinations is extended for the fast CONNECT Request logic in PPSP-TP/1.1 (referring to the "per-peer-ID" transaction state machine). The allowed number of SwarmID elements in a request is indicated, where x, y and N take values greater or equal to 1. When N is indicated, it means that if N requests are @action="JOIN" for the named swarms, the same N requests MUST be used for the corresponding @action="LEAVE" of the same swarms when using the fast CONNECT Request logic.

The response MUST have the same TransactionID values as the corresponding request and actions.

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

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

In case the @peerMode is LEECH the tracker will search and select an appropriate list of peers satisfying the conditions 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 [I.D.ietf-alto- protocol].

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

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
    schemaLocation="TBD"
    version="1.1">
  <Response>SUCCESSFUL</Response>
  <TransactionID>
    <Result transactionID="12345.0">200 OK</Result>
    <Result transactionID="12345.1">200 OK</Result>
    <Result transactionID="12345.3">200 OK</Result>
  </TransactionID>
  <PeerGroup>
    <PeerInfo>
      <PeerID>656164657221</PeerID>
      <PeerAddress addrType="ipv4" ip="198.51.100.1" port="80"
        priority="1"
        type="REFLEXIVE"
        connection="3G"
        asn="64496" />
    </PeerInfo>
    <PeerInfo swarmID="2222">
      <PeerID>956264622298</PeerID>
      <PeerAddress addrType="ipv4" ip="198.51.100.22" port="80"
        asn="64496" peerProtocol="PPSP-PP" />
    </PeerInfo>
    <PeerInfo swarmID="2222">
      <PeerID>3332001256741</PeerID>
      <PeerAddress addrType="ipv4" ip="198.51.100.201" port="80"
        asn="64496" peerProtocol="PPSP-PP" />
    </PeerInfo>
  </PeerGroup>
</PPSPTrackerProtocol>
```

4.2. DISCONNECT Request

This method is used when the peer intends to leave one or multiple specific swarms, or the system, and no longer participate.

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

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

The element SwarmID MUST be present with cardinality 1 to N, each

containing no child elements. The SwarmID value MUST be either specific Swarm-ID the peer had previously joined, the value "ALL" to designate all joined swarms, or the value "nil" to completely disconnect from the system.

An example of the message-body of a DISCONNECT Request for the peer leaving all joined swarms is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
                      schemaLocation="TBD"
                      version="1.1">
  <Request>DISCONNECT</Request>
  <PeerID>656164657221</PeerID>
  <SwarmID>ALL</SwarmID>

  <TransactionID>12345</TransactionID>
</PPSPTrackerProtocol>
```

An example of the message-body of a DISCONNECT Request for a peer seeder leaving several specific swarms is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
                      schemaLocation="TBD"
                      version="1.1">
  <Request>DISCONNECT</Request>
  <PeerID>656164657220</PeerID>
  <SwarmID transactionID="12345.1">1111</SwarmID>
  <SwarmID transactionID="12345.2">2222</SwarmID>
  <TransactionID>12345.0</TransactionID>
</PPSPTrackerProtocol>
```

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

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
    schemaLocation="TBD"
    version="1.1">
  <Response>SUCCESSFUL</Response>
  <TransactionID>12345</TransactionID>
</PPSPTrackerProtocol>
```

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

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
    schemaLocation="TBD"
    version="1.1">
  <Response>SUCCESSFUL</Response>
  <TransactionID>
    <TransactionID>12345</TransactionID>
    <Result transactionID="12345.0">200 OK</Result>
    <Result transactionID="12345.1">200 OK</Result>
    <Result transactionID="12345.2">200 OK</Result>
  </TransactionID>
</PPSPTrackerProtocol>
```

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

4.3. JOIN Request

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

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

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

A peer seeder may send the JOIN message to participate in several swarms. The PeerMode element SHOULD be set to SEED.

An example of the message-body of a JOIN Request for a peer seeder joining two swarms is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
    schemaLocation="TBD"
    version="1.1">
  <Request>JOIN</Request>
  <PeerID>656164657221</PeerID>
  <TransactionID>12345.0</TransactionID>
  <SwarmID peerMode="SEED" transactionID="12345.1">1111</SwarmID>
  <SwarmID peerMode="SEED" transactionID="12345.1">2222</SwarmID>
</PPSPTrackerProtocol>
```

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

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
    schemaLocation="TBD"
    version="1.1">
  <Response>SUCCESSFUL</Response>
  <TransactionID>
    <Result transactionID="12345.0">200 OK</Result>
    <Result transactionID="12345.1">200 OK</Result>
    <Result transactionID="12345.2">200 OK</Result>
  </TransactionID>
</PPSPTrackerProtocol>
```

As a leecher only, the peer can participate in one swarm at a time using JOIN requests. In this case, the JOIN request message may include the ContentGroup element to indicate a specific point in the stream. The PeerMode element SHOULD be set to LEECH.

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

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
                      schemaLocation="TBD"
                      version="1.1">
  <Request>JOIN</Request>
  <PeerID>656164657220</PeerID>
  <TransactionID>12345</TransactionID>
  <SwarmID peerMode="LEECH">1111</SwarmID>
  <PeerNum abilityNAT="STUN"
           concurrentLinks="HIGH"
           onlineTime="NORMAL"
           uploadBWlevel="NORMAL">5</PeerNum>
  <ContentGroup>
    <Representation id="tag0">
      <SegmentInfo startIndex="20" />
    </Representation>
    <Representation id="tag6">
      <SegmentInfo startIndex="20" />
    </Representation>
  </ContentGroup>
</PPSPTrackerProtocol>
```

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

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

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

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
    schemaLocation="TBD"
    version="1.1">
  <Response>SUCCESSFUL</Response>
  <TransactionID>12345</TransactionID>
  <PeerGroup>
    <PeerInfo>
      <PeerID>956264622298</PeerID>
      <PeerAddress addrType="ipv4" ip="198.51.100.22" port="80"
        asn="64496" peerProtocol="PPSP-PP" />
    </PeerInfo>
    <PeerInfo>
      <PeerID>3332001256741</PeerID>
      <PeerAddress addrType="ipv4" ip="198.51.100.201" port="80"
        asn="64496" peerProtocol="PPSP-PP" />
    </PeerInfo>
  </PeerGroup>
</PPSPTrackerProtocol>
```

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

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

4.4. FIND Request

This method allows peers to request to the tracker, whenever needed, a new peer list for the swarm or for specific scope of chunks of a media content representation of that swarm.

The peer MUST properly form the XML message-body, set the request method to FIND, set the PeerID with the identifier of the peer, set the SwarmID with the identifier of the swarm the peer is interested in. Optionally, in order to find peers having the specific chunks, the peer also may include information about the desired content in the JOIN request message.

This message is mainly used for leechers to update the peer list.

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
    schemaLocation="TBD"
    version="1.1">
  <Request>FIND</Request>
  <PeerID>656164657221</PeerID>
  <SwarmID>1111</SwarmID>
  <TransactionID>12345</TransactionID>
  <PeerNum abilityNAT="STUN"
    concurrentLinks="HIGH"
    onlineTime="NORMAL"
    uploadBWlevel="NORMAL">5</PeerNum>
  <ContentGroup>
    <Representation id="tag4">
      <SegmentInfo startIndex="110" endIndex="150" />
    </Representation>
  </ContentGroup>
</PPSPTrackerProtocol>
```

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

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

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

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
    schemaLocation="TBD"
    version="1.1">
  <Response>SUCCESSFUL</Response>
  <TransactionID>12345</TransactionID>
  <PeerGroup>
    <PeerInfo>
      <PeerID>956264622298</PeerID>
      <PeerAddress addrType="ipv4" ip="198.51.100.22" port="80"
        asn="64496" peerProtocol="PPSP-PP" />
    </PeerInfo>
    <PeerInfo>
      <PeerID>3332001256741</PeerID>
      <PeerAddress addrType="ipv4" ip="198.51.100.201" port="80"
        asn="64496" peerProtocol="PPSP-PP" />
    </PeerInfo>
  </PeerGroup>
</PPSPTrackerProtocol>
```

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

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

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

An example of a Response message for the FIND Request including the requesting peer public IP address is the following:

```

<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
    schemaLocation="TBD"
    version="1.1">
  <Response>SUCCESSFUL</Response>
  <TransactionID>12345</TransactionID>
  <PeerGroup>
    <PeerInfo>
      <PeerID>656164657221</PeerID>
      <PeerAddress addrType="ipv4" ip="198.51.100.1" port="80"
        priority="1"
        type="REFLEXIVE"
        connection="3G"
        asn="64496" />
    </PeerInfo>
    <PeerInfo>
      <PeerID>956264622298</PeerID>
      <PeerAddress addrType="ipv4" ip="198.51.100.22" port="80"
        asn="64496" peerProtocol="PPSP-PP" />
    </PeerInfo>
    <PeerInfo>
      <PeerID>3332001256741</PeerID>
      <PeerAddress addrType="ipv4" ip="198.51.100.201" port="80"
        asn="64496" peerProtocol="PPSP-PP" />
    </PeerInfo>
  </PeerGroup>
</PPSPTrackerProtocol>

```

4.5. Enhanced STAT_REPORT Request

This message still uses the specifications of PPSP-TP/1.0 defined in [I-D.cruz-ppsp-base-tracker-protocol]. The Stat element has been extended with one property, "ContentMap", to allow peers reporting map of chunks they have. The tracker would not have the ability to treat the FIND and JOIN requests for specific content chunks, unless peers report this kind of information. Examples are provided below.

An example of the message-body of an enhanced STAT_REPORT request is the following:

```

<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
    schemaLocation="TBD"
    version="1.1">
  <Request>STAT_REPORT</Request>
  <PeerID>656164657221</PeerID>
  <TransactionID>12345</TransactionID>
  <StatisticsGroup>

```

```
<Stat property="StreamStatistics">
  <SwarmID>1111</SwarmID>
  <UploadedBytes>512</UploadedBytes>
  <DownloadedBytes>768</DownloadedBytes>
  <AvailBandwidth>1024000</AvailBandwidth>
</Stat>
<Stat property="StreamStatistics">
  <SwarmID>2222</SwarmID>
  <UploadedBytes>1024</UploadedBytes>
  <DownloadedBytes>2048</DownloadedBytes>
  <AvailBandwidth>512000</AvailBandwidth>
</Stat>
<Stat property="ContentMap">
  <SwarmID>1111</SwarmID>
  <Representation id="tag0">
    <SegmentInfo startIndex="0" endIndex="24"
      chunkmapSize="25">
      A/8D/wP/A/8D/wP/A/8D/wP/A/8D/wP/....
    </SegmentInfo>
  </Representation>
  <Representation id="tag1">
    <SegmentInfo startIndex="0" endIndex="14"
      chunkmapSize="15">
      A/8D/wP/A/8D/wP/A/8D/wP/A/8D/wP/....
    </SegmentInfo>
    <SegmentInfo startIndex="20" endIndex="24"
      chunkmapSize="5">
      A/8D/wP/A/8D/wP/A/8D/wP/A/8D/wP/....
    </SegmentInfo>
  </Representation>
</Stat>
<Stat property="ContentMap">
  <SwarmID>2222</SwarmID>
  <Representation id="tag5">
    <SegmentInfo startIndex="0" endIndex="4"
      chunkmapSize="5">
      A/8D/wP/A/8D/wP/A/8D/wP/A/8D/wP/....
    </SegmentInfo>
  </Representation>
  <Representation id="tag6">
    <SegmentInfo startIndex="0" endIndex="4"
      chunkmapSize="5">
      A/8D/wP/A/8D/wP/A/8D/wP/A/8D/wP/....
    </SegmentInfo>
  </Representation>
</Stat>
</StatisticsGroup>
</PPSPTrackerProtocol>
```

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

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<PPSPTrackerProtocol xmlns="TBD"
                      schemaLocation="TBD"
                      version="1.1">
  <Response>SUCCESSFUL</Response>
  <TransactionID>12345</TransactionID>
</PPSPTrackerProtocol>
```

4.6. Error and Recovery Conditions

This document does not introduce any new error and recovery conditions. The implementation of error treatment MUST refer to PPSP-TP-1.0 specification [I-D.cruz-ppsp-base-tracker-protocol], subsection 8.6.

5. Security Considerations

The PPSP-TP/1.1 proposed in this document introduces no new security considerations beyond those described in PPSP-TP/1.0 specification [I-D.cruz-ppsp-base-tracker-protocol].

6. IANA Considerations

There are presently no IANA considerations with this document.

7. Acknowledgments

The authors would like to thank many people for their help and comments, particularly: Zhang Yunfei, Zong Ning, Martin Stiernerling, Johan Pouwelse and Arno Bakker. The authors would also like to thank the people participating in the EU FP7 project SARACEN (contract no. ICT-248474) [refs.saracenwebpage] for contributions and feedback to this document.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the SARACEN project or the European Commission.

8 References

8.1 Normative References

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

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

8.2 Informative References

[I-D.ietf-ppsp-problem-statement] Zhang, Y., Zong, N., Camarillo, G., Seng, J., and Y. Yang, "Problem Statement of P2P Streaming Protocol (PPSP)", draft-ietf-ppsp-problem-statement-11 (work in progress), October 2012.

[I-D.cruz-ppsp-base-tracker-protocol] Cruz, R., Nunes, M., Gu, Y., Xia, J., J. Taveira and D. Lingli, "PPSP Tracker Protocol-Base Protocol (PPSP-TP/1.0)", draft-cruz-ppsp-base-tracker-protocol-00 (work in progress), June 2012.

[I-D.ietf-alto-protocol] Alimi, R., Penno, R., Yang, Y., "ALTO Protocol", draft-ietf-alto-protocol-13, (work in progress), September 2012.

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

Authors' Addresses

Rachel Huang
Huawei
Phone: +86-25-56623633
EMail: rachel.huang@huawei.com

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

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

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

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

A. Bakker
R. Petrocco
V. Grishchenko
Technische Universiteit Delft
October 22, 2012

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

Abstract

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

Status of this Memo

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

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

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

This Internet-Draft will expire on April 25, 2013.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	5
1.1. Purpose	5
1.2. Requirements Language	6
1.3. Terminology	6
2. Overall Operation	8
2.1. Joining a Swarm	8
2.2. Exchanging Chunks	9
2.3. Leaving a Swarm	9
3. Messages	10
3.1. HANDSHAKE	10
3.2. HAVE	10
3.3. DATA	11
3.4. ACK	11
3.5. INTEGRITY	11
3.6. SIGNED_INTEGRITY	11
3.7. REQUEST	11
3.8. CANCEL	12
3.9. CHOKe and UNCHOKe	12
3.10. Peer Address Exchange and NAT Hole Punching	13
3.10.1. PEX_REQ and PEX_RES Messages	13
3.10.2. Hole Punching via PPSP Messages	13
3.11. Keep Alive Signalling	13
3.12. Storage Independence	14
4. Chunk Addressing Schemes	14
4.1. Start-End Ranges	14
4.1.1. Chunk Ranges	14
4.1.2. Byte Ranges	14
4.2. Bin Numbers	14
4.3. In Messages	16
4.3.1. In HAVE Messages	16
4.3.2. In ACK Messages	16
4.4. Compatibility	16

5.	Content Integrity Protection	17
5.1.	Merkle Hash Tree Scheme	18
5.2.	Content Integrity Verification	19
5.3.	The Atomic Datagram Principle	20
5.4.	INTEGRITY Messages	20
5.5.	Discussion and Overhead	21
6.	Merkle Hash Trees and The Automatic Detection of Content Size	22
6.1.	Peak Hashes	22
6.2.	Procedure	24
7.	Live Streaming	24
7.1.	Content Authentication	25
7.1.1.	Sign All	25
7.1.2.	Unified Merkle Tree	25
7.2.	Forgetting Chunks	28
8.	Protocol Options	29
8.1.	End Option	29
8.2.	Version	29
8.3.	Swarm Identifier	29
8.4.	Content Integrity Protection Method	30
8.5.	Merkle Tree Hash Function	30
8.6.	Live Signature Algorithm	30
8.7.	Chunk Addressing Method	31
8.8.	Live Discard Window	31
8.9.	Supported Messages	32
9.	UDP Encapsulation	32
9.1.	Chunk Size	33
9.2.	Datagrams and Messages	33
9.3.	Channels	34
9.4.	HANDSHAKE	34
9.5.	HAVE	35
9.6.	DATA	36
9.7.	ACK	36
9.8.	INTEGRITY	36
9.9.	SIGNED_INTEGRITY	37
9.10.	REQUEST	37
9.11.	CANCEL	37
9.12.	CHOKE and UNCHOKE	37
9.13.	PEX_REQ, PEX_RES and PEX_RESv6	37
9.14.	KEEPALIVE	37
9.15.	Flow and Congestion Control	37
10.	Extensibility	38
10.1.	Chunk Picking Algorithms	38
10.2.	Reciprocity Algorithms	38
11.	Acknowledgements	38
12.	IANA Considerations	39
13.	Security Considerations	39
13.1.	Security of the Handshake Procedure	39

13.1.1.	Protection against attack 1	40
13.1.2.	Protection against attack 2	40
13.1.3.	Protection against attack 3	41
13.2.	Secure Peer Address Exchange	41
13.2.1.	Protection against the Amplification Attack	42
13.2.2.	Example: Tracker as Certification Authority	42
13.2.3.	Protection Against Eclipse Attacks	43
13.3.	Support for Closed Swarms (PPSP.SEC.REQ-1)	43
13.4.	Confidentiality of Streamed Content (PPSP.SEC.REQ-2+3)	44
13.5.	Limit Potential Damage and Resource Exhaustion by Bad or Broken Peers (PPSP.SEC.REQ-4+6)	44
13.5.1.	HANDSHAKE	44
13.5.2.	HAVE	44
13.5.3.	DATA	45
13.5.4.	ACK	45
13.5.5.	INTEGRITY and SIGNED_INTEGRITY	45
13.5.6.	REQUEST	45
13.5.7.	CANCEL	46
13.5.8.	CHOKe	46
13.5.9.	UNCHOKe	46
13.5.10.	PEX_RES	46
13.5.11.	Unsolicited Messages in General	46
13.6.	Exclude Bad or Broken Peers (PPSP.SEC.REQ-5)	47
14.	References	47
14.1.	Normative References	47
14.2.	Informative References	47
Appendix A.	Revision History	50
Authors' Addresses	54

1. Introduction

1.1. Purpose

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

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

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

At present, PPSPP is set to run on top of UDP using LEDBAT for congestion control [I-D.ietf-ledbat-congestion]. Using LEDBAT enables PPSPP to serve the content after playback (seeding) without disrupting the user who may have moved to different tasks that use its network connection. LEDBAT may be replaced with a different algorithm when the work of the IETF working group on RTP Media Congestion Avoidance Techniques (RMCAT) [RMCATCHART] matures.

PPSPP is also flexible and extensible in the mechanisms it uses to promote client contribution and prevent freeriding, that is, how to deal with peers that only download content but never upload to others. It also allows different schemes chunk addressing and content integrity protection, if the defaults are not fit for a particular use case. In addition, it can work with different peer

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

1.2. Requirements Language

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

1.3. Terminology

message

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

datagram

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

content

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

chunk

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

chunk ID

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

chunk specification

An expression that denotes one or more chunk IDs.

chunk addressing scheme

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

chunk availability map

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

bin

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

content integrity protection scheme

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

hash

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

Merkle hash tree

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

root hash

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

swarm

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

swarm ID

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

tracker

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

choking

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

seeding/leeching

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

2. Overall Operation

The basic unit of communication in PPSP is the message. Multiple messages are multiplexed into a single datagram for transmission. A datagram (and hence the messages it contains) will have different representations on the wire depending on the transport protocol used (see Section 9).

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

2.1. Joining a Swarm

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

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

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

2.2. Exchanging Chunks

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

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

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

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

2.3. Leaving a Swarm

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

3. Messages

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

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

3.1. HANDSHAKE

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

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

3.2. HAVE

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

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

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

3.3. DATA

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

3.4. ACK

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

3.5. INTEGRITY

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

3.6. SIGNED_INTEGRITY

The SIGNED_INTEGRITY message carries digitally signed information required by the receiver to verify the integrity of a chunk in live streaming. It logically contains a chunk specification and a digital signature. Its exact payload depends on the live content integrity protection scheme used, see Section 7.1.

3.7. REQUEST

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

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

A peer MAY send a REQUEST message that MUST contain the specification of the chunks it wants to download. A peer receiving a REQUEST message MAY send out requested pieces. When peer Q receives multiple REQUESTs from the same peer P peer Q SHOULD process the REQUESTs sequentially. Multiple REQUEST messages MAY be sent in one datagram, for example, when a peer wants to request several rare chunks at once.

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

3.8. CANCEL

When downloading on demand or live streaming content, a peer MAY request urgent data from multiple peers to increase the probability of it being delivered on time. In particular, when the specific chunk picking algorithm (see Section 10.1), detects that a request for urgent data might not be served on time, a request for the same data MAY be sent to a different peer. When a peer P decides to request urgent data from a peer Q, peer P SHOULD send a CANCEL message to all the peers to which the data has been previously requested. The CANCEL message contains the specification of the chunks P no longer wants to request. In addition, when peer Q receives a HAVE message for the urgent data from peer P, peer Q MUST also cancel the previous REQUEST(s) from P. In other words, the HAVE message acts as an implicit CANCEL.

3.9. CHOKE and UNCHOKE

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

3.10. Peer Address Exchange and NAT Hole Punching

3.10.1. PEX_REQ and PEX_RES Messages

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

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

3.10.2. Hole Punching via PPSPP Messages

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

3.11. Keep Alive Signalling

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

3.12. Storage Independence

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

4. Chunk Addressing Schemes

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

4.1. Start-End Ranges

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

4.1.1. Chunk Ranges

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

4.1.2. Byte Ranges

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

4.2. Bin Numbers

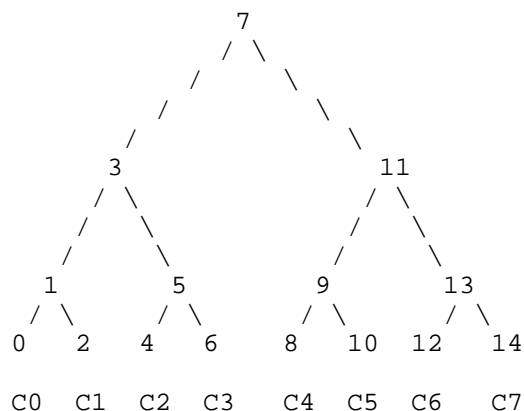
PPSPP introduces a novel method of addressing chunks of content called "bin numbers" (or "bins" for short). Bin numbers allow the addressing of a binary interval of data using a single integer. This reduces the amount of state that needs to be recorded per peer and the space needed to denote intervals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

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

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

$$\text{binP} = (\text{binL} + \text{binR}) / 2$$

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



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

Figure 1

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

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

4.3. In Messages

4.3.1. In HAVE Messages

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

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

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

4.3.2. In ACK Messages

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

4.4. Compatibility

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

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

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

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

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

$$\text{startrange} = (\text{bin AND } (\text{bin} + 1)) / 2$$
$$\text{endrange} = ((\text{bin OR } (\text{bin} + 1)) - 1) / 2$$

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

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

5. Content Integrity Protection

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

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

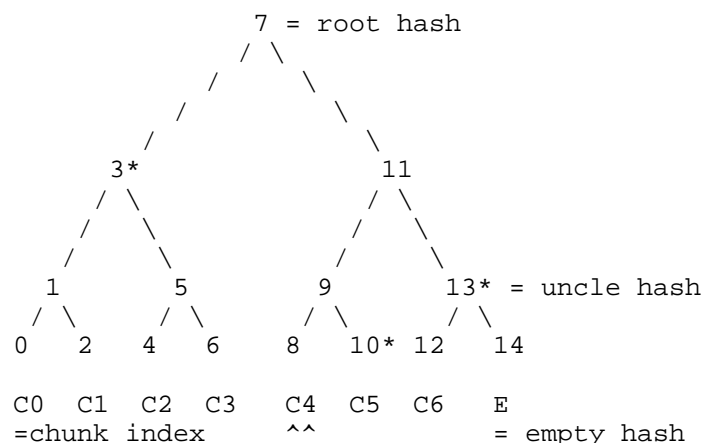
The Merkle hash tree scheme can use different chunk addressing schemes. All it requires is the ability to address a range of chunks. In the following description abstract node IDs are used to

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

5.1. Merkle Hash Tree Scheme

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

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



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

Figure 2

5.2. Content Integrity Verification

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

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

5.3. The Atomic Datagram Principle

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

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

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

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

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

5.4. INTEGRITY Messages

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

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

5.5. Discussion and Overhead

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

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

$$nnodes = \text{math.pow}(2, \text{math.log}(nchunks, 2) + 1)$$

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 $\text{hashsize} * nnodes$. However, the actual number of hashes transmitted can be optimized as described in Section 5.3. To see a peer can verify all chunks whilst receiving not all hashes, consider the example tree in Section 5.1.

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

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

Table 1: Overhead for the example tree

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

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

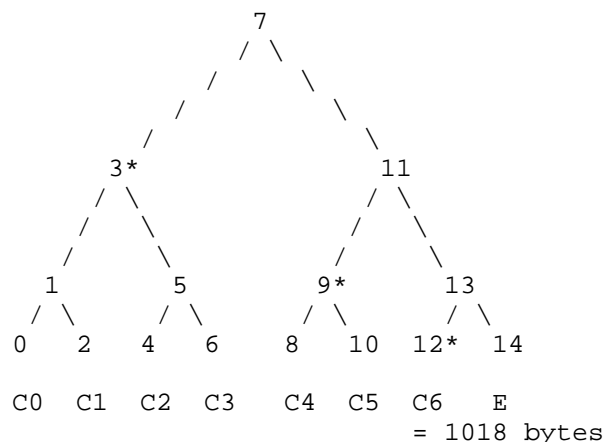
In PPSPP, the root hash of a static content asset, such as a video file, along with some peer addresses is sufficient to start a download. In addition, PPSPP can reliably and automatically derive the size of such content from information received from the network when fixed sized chunks are used. As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

6.1. Peak Hashes

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

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

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



Peak hashes in a Merkle hash tree.

Figure 3

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

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

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

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

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

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

$$(\text{number of chunks} - 1) * \text{fixed chunk size} + \text{size of last chunk}$$

6.2. Procedure

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

7. Live Streaming

The set of messages defined above can be used for live streaming as well. In a pull-based model, a live streaming injector can announce the chunks it generates via HAVE messages, and peers can retrieve them via REQUEST messages. Areas that need special attention are

content authentication and chunk addressing (to achieve an infinite stream of chunks).

7.1. Content Authentication

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

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

In the "Unified Merkle Tree" method, PPSP combines the Merkle hash tree scheme for static content with signatures to unify the video-on-demand and live streaming case. The use of Merkle hash trees reduces the number of signing and verification operations per second, that is, provide signature amortization similar to the approach described in [SIGMCAST].

7.1.1. Sign All

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

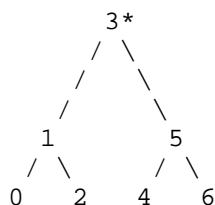
7.1.2. Unified Merkle Tree

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

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

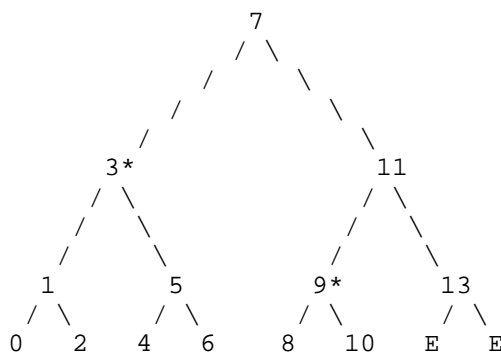
The injector now signs only the peak hashes in this set that are not in the old set of peak hashes. For N being a power of 2 there will just be one new peak hash (see below). This complementary signed peak is distributed to the peers. Receiving peers will verify the signature on the signed peak against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

To illustrate this procedure, consider the injector has generated the tree shown in Figure 4 and it is connected to several peers that currently have the same tree and all chunks. In this tree the root node 3 is also the peak node for this tree. Now the injector generates $N=2$ new chunks. As a result the tree expands as shown in Figure 5. The two new pieces 8 and 10 extend the tree on the right side, and to accommodate them a new root is created, node 7. As this tree is wider at the base than the actual number of chunks, there are currently two empty leaves. The peak nodes in this tree are 3 and 9.



Current live tree

Figure 4



Next current live tree

Figure 5

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

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

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

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

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

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

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

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

7.2. Forgetting Chunks

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

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

8. Protocol Options

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

8.1. End Option

A peer **MUST** conclude the list of protocol options with the end option. Subsequent octets should be considered protocol messages. The code for the end option is 255, and its length is 1 octet.

```

+-----+
| Code |
+-----+
|  255 |
+-----+

```

8.2. Version

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

```

+-----+-----+
| Code | Version |
+-----+-----+
|   0  |    v   |
+-----+-----+

```

8.3. Swarm Identifier

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

```

+-----+-----+-----+
| Code | Length | Swarm Identifier |
+-----+-----+-----+
|   1  | n (16 bits) | i1,i2,... |
+-----+-----+-----+

```

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

8.4. Content Integrity Protection Method

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

Code	Method
2	m

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

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

8.5. Merkle Tree Hash Function

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

Code	Hash Func
3	h

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

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

8.6. Live Signature Algorithm

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

Code	Sig Alg
4	s

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

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

8.7. Chunk Addressing Method

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

Code	Scheme
5	a

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

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

8.8. Live Discard Window

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

Code	Scheme
6	w (32 or 64-bits)

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

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

8.9. Supported Messages

Peers may support just a subset of the PPSPP messages. For example, peers running over TCP may not accept ACK messages, or peers used with a centralized tracking infrastructure may not accept PEX messages. For these reasons, peers who support only a proper subset of the PPSPP messages **MUST** signal which subset they support by means of this protocol option. The value of this option is a 256-bit bitmap where each bit represents a message type. The bitmap may be truncated to the last non-zero byte.

Code	Length	Message Bitmap
7	n	m1,m2,...

9. UDP Encapsulation

Currently, PPSPP-over-UDP is the preferred deployment option. Effectively, UDP allows the use of IP with minimal overhead and it also allows userspace implementations. LEDBAT is used for congestion control [I-D.ietf-ledbat-congestion]. Using LEDBAT enables PPSPP to serve the content after playback (seeding) without disrupting the user who may have moved to different tasks that use its network connection. LEDBAT may be replaced with a different algorithm when the work of the IETF working group on RTP Media Congestion Avoidance Techniques (RMCAT) [RMCATCHART] matures.

9.1. Chunk Size

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

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

9.2. Datagrams and Messages

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

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

- o PEX_RESv6 = 0x0c

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

All messages are idempotent or recognizable as duplicates. In particular, a peer MAY resend DATA, ACK, HAVE, INTEGRITY, PEX_*, SIGNED_INTEGRITY, REQUEST, CANCEL, CHOKE and UNCHOKe messages without problems when loss is suspected. When a peer resends a HANDSHAKE message it can be recognized as duplicate by the receiver and be dealt with.

9.3. Channels

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

9.4. HANDSHAKE

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

1. the IP address of a peer
2. peer's UDP port and
3. the swarm id of the content (see Section 5.1 and Section 7).
4. the chunk size used, unless the 1 KB default

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

On the wire the datagram will look something like this:

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

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

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

Peer's response datagram on the wire:

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

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

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

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

On the wire:

(CHANNEL) 00000022 HANDSHAKE 00000000

9.5. HAVE

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

A HAVE message for bin 3 on the wire:

HAVE 00000003

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

9.6. DATA

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

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

DATA 00000000 12345678 48656c6c6f20776f726c6421

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

9.7. ACK

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

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

ACK 00000002 12345678

9.8. INTEGRITY

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

An INTEGRITY message for bin 0 with a SHA1 hash on the wire:

HASH 00000000 123412341234123412341234123412341234123412341234

9.9. SIGNED_INTEGRITY

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

9.10. REQUEST

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

9.11. CANCEL

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

9.12. CHOKE and UNCHOKE

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

9.13. PEX_REQ, PEX_RES and PEX_RESv6

A PEX_REQ (0x06) message has no payload. A PEX_RES (0x05) message consists of a IPv4 address in big endian format followed by a UDP port number in big endian format. A PEX_RESv6 (0x0c) message contains a 128-bit IPv6 address instead of an IPv4 one.

9.14. KEEPALIVE

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

On the wire:

(CHANNEL) 00000022

9.15. Flow and Congestion Control

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

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

10. Extensibility

10.1. Chunk Picking Algorithms

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

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

10.2. Reciprocity Algorithms

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

11. Acknowledgements

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

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

12. IANA Considerations

To be determined.

13. Security Considerations

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

13.1. Security of the Handshake Procedure

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

1. DOS amplification attack: attackers try to use a PPSPP peer to generate more traffic to a victim.
2. DOS flood attack: attackers try to deny service to other peers by allocating lots of state at a PPSPP peer.
3. Disrupt service to an individual peer: attackers send bogus e.g. REQUEST and HAVE messages appearing to come from victim peer A to the peers B1..Bn serving that peer. This causes A to receive chunks it did not request or to not receive the chunks it requested.

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

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

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

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

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

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

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

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

13.1.1. Protection against attack 1

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

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

13.1.2. Protection against attack 2

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

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

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

13.1.3. Protection against attack 3

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

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

13.2. Secure Peer Address Exchange

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

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

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

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

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

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

13.2.1. Protection against the Amplification Attack

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

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

13.2.2. Example: Tracker as Certification Authority

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

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

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

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

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

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

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

13.2.3. Protection Against Eclipse Attacks

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

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

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

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

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

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

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

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

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

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

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

13.5.1. HANDSHAKE

- 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.

13.5.2. HAVE

- o 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.

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

Solution: Peer and chunk selection algorithms external to the

protocol will implement fairness and provide sharing incentives.

13.5.3. DATA

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

Solution: The content integrity protection schemes defend against this.

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

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

13.5.4. ACK

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

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

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

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

13.5.5. INTEGRITY and SIGNED_INTEGRITY

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

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

13.5.6. REQUEST

- o 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 quatum is the bitrate of the content, taking into account that this may be variable.

13.5.7. CANCEL

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

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

13.5.8. CHOKE

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

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

13.5.9. UNCHOKE

- o 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.

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

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

13.5.10. PEX_RES

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

13.5.11. Unsolicited Messages in General

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

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

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

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

14. References

14.1. Normative References

[FIPS180-3]

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

[I-D.ietf-ledbat-congestion]

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

[IANADNSSECCALGNUM]

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

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

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

14.2. Informative References

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

- [BINMAP] Grishchenko, V. and J. Pouwelse, "Binmaps: hybridizing bitmaps and binary trees", Technical Report PDS-2011-005, Parallel and Distributed Systems Group, Fac. of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, The Netherlands, Apr 2009.
- [BITOS] Vlavianos, A., Iliofotou, M., Mathieu, F., and M. Faloutsos, "BiToS: Enhancing BitTorrent for Supporting Streaming Applications", IEEE INFOCOM Global Internet Symposium Barcelona, Spain, Apr 2006.
- [BITTORRENT] Cohen, B., "The BitTorrent Protocol Specification", BitTorrent Enhancement Proposal 3, Feb 2008, <http://bittorrent.org/beps/bep_0003.html>.
- [CLOSED] Borch, N., Mitchell, K., Arntzen, I., and D. Gabrijelcic, "Access Control to BitTorrent Swarms Using Closed Swarms", ACM workshop on Advanced Video Streaming Techniques for Peer-to-Peer Networks and Social Networking (AVSTP2P '10), Florence, Italy, Oct 2010, <<http://doi.acm.org/10.1145/1877891.1877898>>.
- [DETMAL] Shetty, S., Galdames, P., Tavanapong, W., and Ying. Cai, "Detecting Malicious Peers in Overlay Multicast Streaming", IEEE Conference on Local Computer Networks (LCN'06). Tampa, FL, USA, Nov 2006.
- [ECLIPSE] Sit, E. and R. Morris, "Security Considerations for Peer-to-Peer Distributed Hash Tables", IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems pp. 261-269, Springer-Verlag, 2002.
- [ECS] Jovanovikj, V., Gabrijelcic, D., and T. Klobucar, "Access Control in BitTorrent P2P Networks Using the Enhanced Closed Swarms Protocol", International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2011), pp. 97-102, Nice, France, Aug 2011.
- [EPLIVEPERF] Bonald, T., Massoulie, L., Mathieu, F., Perino, D., and A. Twigg, "Epidemic Live Streaming: Optimal Performance Trade-offs", Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems Annapolis, MD, USA, Jun 2008.

- [GIVE2GET] Mol, J., Pouwelse, J., Meulpolder, M., Epema, D., and H. Sips, "Give-to-Get: Free-riding Resilient Video-on-demand in P2P Systems", Proceedings Multimedia Computing and Networking conference (Proceedings of SPIE Vol. 6818) San Jose, California, USA, Jan 2008.
- [HAC01] Menezes, A., van Oorschot, P., and S. Vanstone, "Handbook of Applied Cryptography", CRC Press, (Fifth Printing, August 2001), Oct 1996.
- [I-D.ietf-ppsp-reqs] Williams, C., Xiao, L., Zong, N., Pascual, V., and Y. Zhang, "P2P Streaming Protocol (PPSP) Requirements", draft-ietf-ppsp-reqs-05 (work in progress), October 2011.
- [I-D.narten-iana-considerations-rfc2434bis] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", draft-narten-iana-considerations-rfc2434bis-09 (work in progress), March 2008.
- [JIM11] Jimenez, R., Osmani, F., and B. Knutsson, "Sub-Second Lookups on a Large-Scale Kademlia-Based Overlay", IEEE International Conference on Peer-to-Peer Computing (P2P'11), Kyoto, Japan, Aug 2011.
- [MERKLE] Merkle, R., "Secrecy, Authentication, and Public Key Systems", Ph.D. thesis Dept. of Electrical Engineering, Stanford University, CA, USA, pp 40-45, 1979.
- [POLLIVE] Dhungel, P., Hei, Xiaojun., Ross, K., and N. Saxena, "Pollution in P2P Live Video Streaming", International Journal of Computer Networks & Communications (IJCNC) Vol.1, No.2, Jul 2009.
- [RFC2132] Alexander, S. and R. Droms, "DHCP Options and BOOTP Vendor Extensions", RFC 2132, March 1997.
- [RFC4034] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Resource Records for the DNS Security Extensions", RFC 4034, March 2005.
- [RFC4301] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.
- [RFC5389] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing, "Session Traversal Utilities for NAT (STUN)", RFC 5389,

October 2008.

- [RFC5971] Schulzrinne, H. and R. Hancock, "GIST: General Internet Signalling Transport", RFC 5971, October 2010.
- [RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, January 2012.
- [RMCATCHART]
Eggert, L. and others, "RTP Media Congestion Avoidance Techniques (rmcat) Description of Working Group", 2012, <<http://datatracker.ietf.org/wg/rmcat/charter/>>.
- [SECDHTS] Urdaneta, G., Pierre, G., and M. van Steen, "A Survey of DHT Security Techniques", ACM Computing Surveys vol. 43(2), Jun 2011.
- [SIGMCAST]
Wong, C. and S. Lam, "Digital Signatures for Flows and Multicasts", IEEE/ACM Transactions on Networking 7(4), pp. 502-513, 1999.
- [SNP] Ford, B., Srisuresh, P., and D. Kegel, "Peer-to-Peer Communication Across Network Address Translators", Feb 2005, <<http://www.brynosaurus.com/pub/net/p2pnat/>>.
- [SPS] Jesi, G., Montresor, A., and M. van Steen, "Secure Peer Sampling", Computer Networks vol. 54(12), pp. 2086-2098, Elsevier, Aug 2010.
- [SWIFTIMPL]
Grishchenko, V., Paananen, J., Pronchenkov, A., Bakker, A., and R. Petrocco, "Swift reference implementation", 2012, <<https://github.com/triblerteam/libswift/>>.
- [TIT4TAT] Cohen, B., "Incentives Build Robustness in BitTorrent", 1st Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, USA, Jun 2003.

Appendix A. Revision History

- 00 2011-12-19 Initial version.
- 01 2012-01-30 Minor text revision:
 - * Changed heading to "A. Bakker"

- * Changed title to **Peer** Protocol, and abbreviation PPSPP.
- * Replaced swift with PPSPP.
- * Removed Sec. 6.4. "HTTP (as PPSP)".
- * Renamed Sec. 8.4. to "Chunk Picking Algorithms".
- * Resolved Ticket #3: Removed sentence about random set of peers.
- * Resolved Ticket #6: Added clarification to "Chunk Picking Algorithms" section.
- * Resolved Ticket #11: Added Sec. 3.12 on Storage Independence
- * Resolved Ticket #14: Added clarification to "Automatic Size Detection" section.
- * Resolved Ticket #15: Operation section now states it shows example behaviour for a specific set of policies and schemes.
- * Resolved Ticket #30: Explained why multiple REQUESTs in one datagram.
- * Resolved Ticket #31: Renamed PEX_ADD message to PEX_RES.
- * Resolved Ticket #32: Renamed Sec 3.8. to "Keep Alive Signaling", and updated explanation.
- * Resolved Ticket #33: Explained NAT hole punching via only PPSPP messages.
- * Resolved Ticket #34: Added section about limited overhead of the Merkle hash tree scheme.

-02 2012-04-17 Major revision

- * Allow different chunk addressing and content integrity protection schemes (ticket #13):
- * Added chunk ID, chunk specification, chunk addressing scheme, etc. to terminology.
- * Created new Sections 4 and 5 discussing chunk addressing and content integrity protection schemes, respectively and moved relevant sections on bin numbering and Merkle hash trees there.

- * Renamed Section 4 to "Merkle Hash Trees and The Automatic Detection of Content Size".
- * Reformulated automatic size detection in terms of nodes, not bins.
- * Extended HANDSHAKE message to carry protocol options and created Section 8 on Protocol options. VERSION and MSGTYPE_RCVD messages replaced with protocol options.
- * Renamed HASH message to INTEGRITY.
- * Renamed HINT to REQUEST.
- * Added description of chunk addressing via (start,end) ranges.
- * Resolved Ticket #26: Extended "Security Considerations" with section on the handshake procedure.
- * Resolved Ticket #17: Defined recently as "in last 60 seconds" in PEX.
- * Resolved Ticket #20: Extended "Security Considerations" with design to make Peer Address Exchange more secure.
- * Resolved Ticket #38+39 / PPSP.SEC.REQ-2+3: Extended "Security Considerations" with a section on confidentiality of content.
- * Resolved Ticket #40+42 / PPSP.SEC.REQ-4+6: Extended "Security Considerations" with a per-message analysis of threats and how PPSP is protected from them.
- * Progressed Ticket #41 / PPSP.SEC.REQ-5: Extended "Security Considerations" with a section on possible ways of excluding bad or broken peers from the system.
- * Moved Rationale to Appendix.
- * Resolved Ticket #43: Updated Live Streaming section to include "Sign All" content authentication, and reference to [SIGMCAST] following discussion with Fabio Picconi.
- * Resolved Ticket #12: Added a CANCEL message to cancel REQUESTs for the same data that were sent to multiple peers at the same time in time-critical situations.

-03 2012-10-22 Major revision

- * Updated Abstract and Introduction, removing download case.
- * Resolved Ticket #4: Added explicit CHOKE/UNCHOKE messages.
- * Removed directory lists unused in streaming.
- * Resolved Ticket #22, #23, #28: Failure behaviour, error codes and dealing with peer crashes.
- * Resolved Ticket #13: Chunk ranges are the default chunk addressing scheme that all peers MUST support.
- * Added a section on compatibility between chunk addressing schemes.
- * Expanded the explanation of Unified Merkle Trees as a method for content integrity protection for live streams.
- * Added a section on forgetting chunks in live streaming.
- * Added "End" option to protocol options and corrected bugs in UDP encapsulation, following Karl Knutsson's comments.
- * Added SHA-2 support for Merkle Hash functions.
- * Added content integrity protection methods for live streaming to the relevant protocol option.
- * Added a Live Signature Algorithm protocol option.
- * Resolved Ticket #24+27: The choice for UDP + LEDBAT as transport has now been reflected in the draft. TCP and RTP encapsulations have been removed.
- * Superfluous parts of Section 10 on extensibility have been removed.
- * Removed appendix with Rationale.
- * Resolved Ticket #21+25: PPSPP currently uses LEDBAT and the DATA and ACK messages now contain the time fields it requires. Should other congestion control algorithms be supported in the future, a protocol option will be added.

Authors' Addresses

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

Phone:
Email: arno@cs.vu.nl

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

Phone:
Email: r.petrocco@gmail.com

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

Phone:
Email: victor.grishchenko@gmail.com

