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

V. Grishchenko
A. Bakker
TU Delft

October 26, 2011

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

Abstract

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

Status of this memo

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

Table of Contents

1. Introduction

1.1. Purpose

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

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

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

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

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

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

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

1.2. Conventions Used in This Document

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

1.3. Terminology

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

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

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

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

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

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

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

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

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

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

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

2. Overall Operation

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

2.1. Joining a Swarm

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

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

2.2. Exchanging Chunks

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

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

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

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

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

2.3. Leaving a Swarm

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

3. Messages

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

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

3.1. HANDSHAKE

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

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

3.3. HAVE

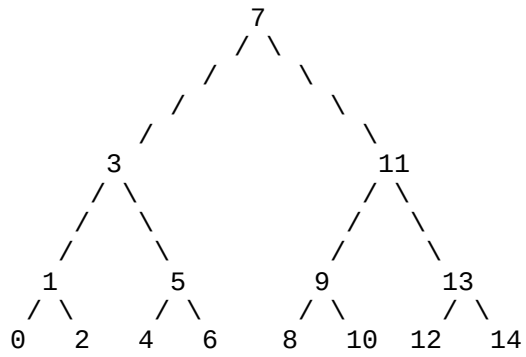
The HAVE message is used to convey which chunks a peers has available, expressed in a new content addressing scheme called "bin numbers".

3.3.1. Bin Numbers

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

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

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



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

3.3.2. HAVE Message

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

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

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

3.4. ACK

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

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

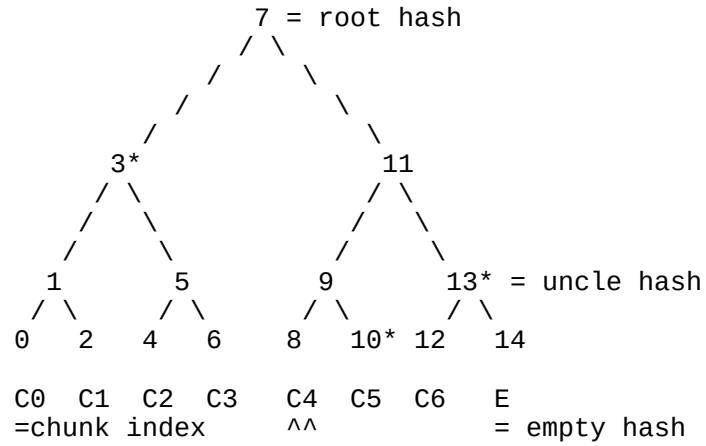
3.5. DATA and HASH

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

3.5.1. Merkle Hash Tree

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

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



3.5.2. Content Integrity Verification

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

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

3.5.3. The Atomic Datagram Principle

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

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

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

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

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

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

3.5.4. DATA and HASH Messages

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

3.6. HINT

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

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

3.7. Peer Address Exchange and NAT Hole Punching

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

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

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

3.8. KEEPALIVE

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

3.9. VERSION

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

3.10. Conveying Peer Capabilities

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

3.11. Directory Lists

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

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

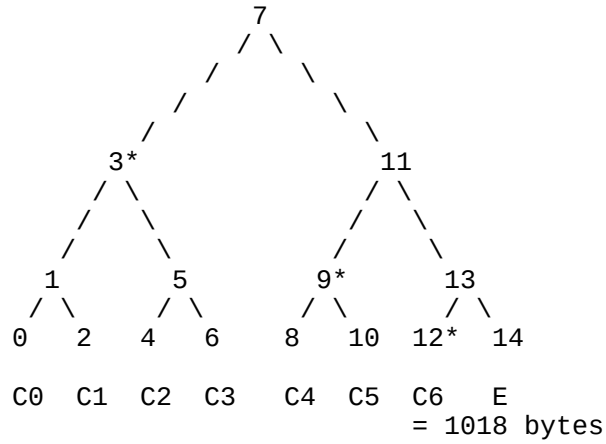
4. Automatic Detection of Content Size

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

4.1. Peak Hashes

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

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



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

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

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

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

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

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

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

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

4.2. Procedure

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

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

5. Live streaming

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

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

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

6. Transport Protocols and Encapsulation

6.1. UDP

6.1.1. Chunk Size

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

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

6.1.2. Datagrams and Messages

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

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

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

6.1.3. Channels

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

6.1.4. HANDSHAKE and VERSION

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

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

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

On the wire the datagram will look something like this:

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

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

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

Peer's response datagram on the wire:

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

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

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

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

On the wire:
00 00000000

6.1.5. HAVE

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

6.1.6. ACK

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

6.1.7. HASH

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

6.1.8. DATA

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

6.1.9. KEEPALIVE

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

On the wire:
00000022

6.1.10. Flow and Congestion Control

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

6.2. TCP

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

6.3. RTP Profile for PPSP

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

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

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

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

6.3.1. Design

6.3.1.1. Joining a Swarm

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

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

6.3.1.2. Exchanging Chunks

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

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

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

6.3.1.3. Leaving a Swarm

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

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

6.3.1.4. Discussion

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

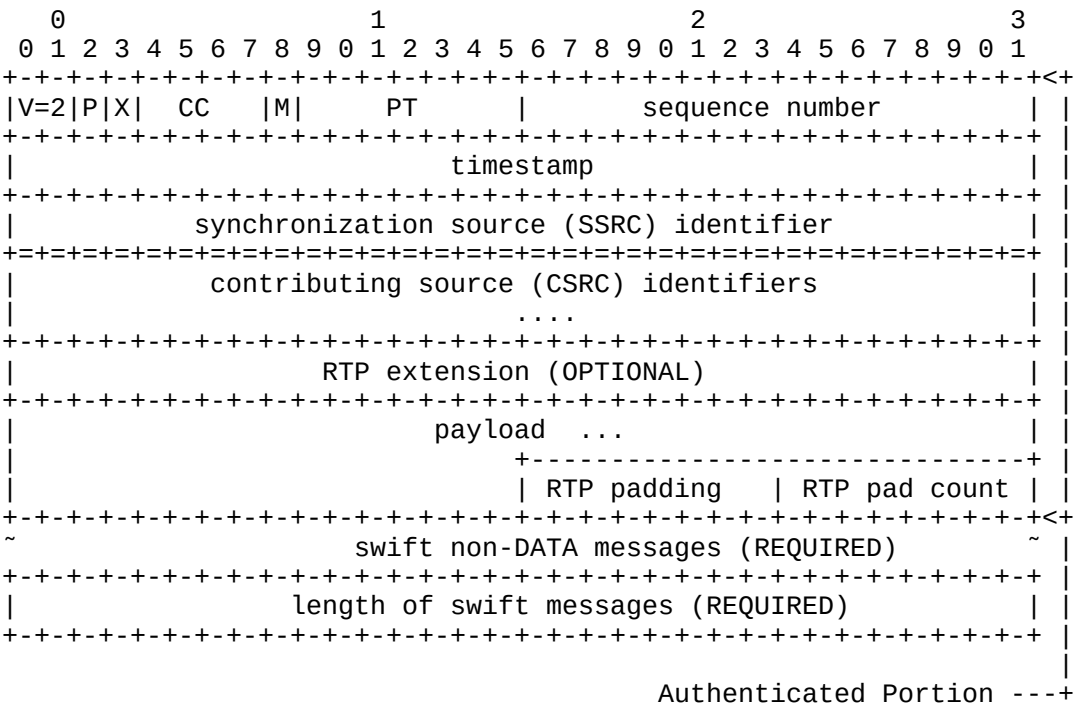


Figure: The format of an RTP-Swift packet.

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

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

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

6.3.2. PPSP Requirements

6.3.2.1. Basic Requirements

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

a tracker protocol, to be discussed elsewhere.

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

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

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

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

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

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

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

6.3.2.2. Peer Protocol Requirements

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

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

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

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

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

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

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

6.3.2.3. Security Requirements

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

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

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

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

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

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

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

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

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

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

6.4. HTTP (as PPSP)

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

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

6.4.1. Design

6.4.1.1. Joining a Swarm

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

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

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

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

6.4.1.2. Exchanging Chunks

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

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

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

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

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

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

6.4.1.3. Leaving a Swarm

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

6.4.1.4. Discussion

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

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

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

6.4.2. PPSP Requirements

6.4.2.1. Basic Requirements

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

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

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

IP+port is sufficient.

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

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

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

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

6.4.2.2. Peer Protocol Requirements

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

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

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

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

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

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

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

6.4.2.3. Security Requirements

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

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

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

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

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

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

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

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

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

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

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

7. Security Considerations

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

8. Extensibility

8.1. 32 bit vs 64 bit

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

```
44 000000000000000E 01234567890ABCDEF1234567890ABCDEF1234567
```

8.2. IPv6

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

8.3. Congestion Control Algorithms

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

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

8.4. Piece Picking Algorithms

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

8.5. Reciprocity Algorithms

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

8.6. Different crypto/hashing schemes

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

9. Rationale

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

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

TABLE 1. Use cases.

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

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

9.1. Design Goals

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

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

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

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

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

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

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

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

9.2. Not TCP

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

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

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

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

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

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

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

9.3. Generic Acknowledgments

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

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

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

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

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

Acknowledgements

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

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

References

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

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

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

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

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

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

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

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

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

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

Authors’ addresses

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

Email: arno@cs.vu.nl