## TRIBLER PROTOCOL SPECIFICATION

## **V0.0.2, JANUARY 2009**

A. Bakker, J.J.D. Mol, J. Yang, L. d'Acunto, J.A. Pouwelse, J. Wang, P. Garbacki, A. Iosup, J. Doumen, J. Roozenburg, Y. Yuan, M. ten Brinke, L. Musat, F. Zindel, F. van der Werf, M. Meulpolder, J. Taal, B. Schoon Large-Scale Distributed Systems, Vrije Universiteit, Amsterdam Parallel and Distributed Systems, Delft University of Technology Information and Communication Theory, Delft University of Technology Distributed and Embedded Systems, University of Twente

## **ABSTRACT**

This document contains the specifications for the protocols used by the Tribler content-delivery library in January 2009.

## **Revision History**

2009-01-12 Created initial document from Freeband I-Share deliverables D3.104, D3.7, D3.10, D3.14.

# **Contents**

1	Intro	oduction	on .		1
2	Vide	o-On-D	Demand		3
	2.1	Introdu	luction		. 3
	2.2	Wire F	Format		. 3
	2.3	Torren	nt File Extension		. 4
		2.3.1	MetaDiscussion		. 4
3	Live	Stream	ning		5
	3.1	Suppor	orting Infinite Duration		. 5
	3.2	Source	ee Authentication		. 6
	3.3	Live P	Playback		. 7
	3.4	Auxilia	liary Seeders		. 7
4	Sma	ll Torre	ent Files		9
	4.1	Introdu	luction		. 9
	4.2	Simple	le Merkle Hashes		. 9
	4.3	Inclusi	sion in BitTorrent		. 10
		4.3.1	MetaDiscussion		. 11
5	Secu	ıre, Pern	rmanent Peer Identifiers		13
	5.1	Introdu	luction		. 13
	5.2	PermII	IDs		. 13
		5.2.1	Definitions and Terminology		. 13
		5.2.2	Authentication Protocol		. 14
		5.2.3	Integration into BitTorrent		. 15
		5.2.4	Torrent Signatures		. 15
	5.3	The O	Overlay Swarm		. 15
	5.4	Protoc	col Versioning		. 16
		5.4.1	Basic Protocol Versioning		. 16
		5.4.2	Overlay-swarm Protocol Versioning		. 16
		5.4.3	Protocol History		. 17
		5.4.4	MetaDiscussion		. 17

6	Dece	ntralized Recommendation 19
	6.1	Introduction
	6.2	BuddyCast Protocol
	6.3	Detailed Algorithm
		6.3.1 Pseudo Code
		6.3.2 Valid Peers and Bootstrapping
		6.3.3 Rate Control
	6.4	Wire Format
	0.4	6.4.1 History and Open Issues
		6.4.2 MetaDiscussion
	6.5	Obtaining Metadata
	0.5	6.5.1 MetaDiscussion
	6.6	
	0.0	,
		6.6.1 Wire Format
		6.6.2 MetaDiscussion
7	Rom	ote Search 27
′	7.1	Introduction
	7.1	Protocol
	7.3	Wire Format
	1.5	whe round
8	Coor	perative Downloading 29
	8.1	Introduction
	8.2	Protocol
	8.3	Wire Format
	0.0	8.3.1 MetaDiscussion
9	Socia	al Networking 31
	9.1	Introduction
	9.2	Nickname and Picture Exchange
		9.2.1 Wire Format
	9.3	Friendship Making
		9.3.1 Wire format
10		/Firewall Detection and Traversal 33
	10.1	Introduction
	10.2	External Address Discovery
		10.2.1 Wire Format
		10.2.2 MetaDiscussion
	10.3	Detection of Network Address Translator (NAT) Type
		10.3.1 Type Detection
		10.3.2 Timeout Detection
11		ote Monitoring 37
	11.1	Introduction
	11.2	Protocol
	11.3	Wire format

11.4	References																																													39	9
------	------------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----	---

## Introduction

This document contains the specifications for the protocols for our Tribler content delivery library. The reader is assumed to be familiar with the BitTorrent protocol [4] [5] on which Tribler is based. For convenience, we include a brief summary.

BitTorrent has an interesting design that enables each individual downloader to maximize his own download rate and locks out users who do not contribute to the system. A peer wishing to download a particular file through BitTorrent first needs to obtain a *torrent* metafile for the file from, for example, a Web site or RSS news feed. The metafile gives the peer the address of a *tracker* for the file and checksums to verify downloaded parts of the file. The peer then contacts the tracker to obtain a list of peers currently involved in downloading the file, implying they have pieces of the file to share.

Next, the peer contacts a random peer to obtain a first piece of the file itself. With this piece in hand, the peer starts to contact other peers in the list to see if they will trade its piece for another part of the file. If so, the contacted peer sends a few blocks of the negotiated piece, and continues to do so as long as the other does the same. This tit-for-tat mechanism automatically locks out peers who are unwilling to upload themselves. By monitoring the download rate obtained from its current set of peers and randomly trying other peers to see if faster peers are available, a user can maximize its download rate. By always selecting a rare part of the file from the pieces on offer, a peer ensures it always has a piece of the file that other peers are interested in. These policies for piece selection and bandwidth trading lead to a balanced economy with suppliers meeting demand and achieving their own goal (fast download) at the same time. Once the peer has obtained the complete file it will become a *seeder* and altruistically provide pieces to other peers without any return. The set of all peers currently actively exchanging pieces of the file is called the file's *swarm*.

We describe the protocols and the features they enable one by one in a separate chapter. [ *TODO* 

• (use-local-network-if-same-NAT (post Jan2009))

## Video-On-Demand

#### 2.1. INTRODUCTION

In current BitTorrent, a user must wait for the download of a media file to be completed before it can be played. To play a media file while it is being downloaded the download protocol must guarantee that the media player receives the pieces of the file in-order and in time. The time requirement means that the download protocol must supply the piece before the media player has to display it on the screen. The BitTorrent protocol provides no guarantees about the order and timeliness of file pieces.

Instead, its piece-picking policy is based on an economic model of supply and demand [4]. The model uses the basic principle of tit-for-tat, that is, a peer cannot download a piece of the file unless it uploads a piece in return. To support this principle, the BitTorrent piece-picking policy states that a peer should download the piece of the file with the least replicas in the network of peers. As a result, a peer will always have a rare piece and thus is able to find a lot of peers willing to trade with him (i.e., low supply, high demand). Unfortunately, this policy does not provide any ordering or timeliness guarantees.

We have extended BitTorrent such that it allows media files to be played while they are downloading, if the user's network connection is fast enough. We call our approach Give-to-Get and we refer to [12] for a detailed description of this approach.

#### 2.2. WIRE FORMAT

For completeness we will describe the G2G\_PIECE\_XFER protocol message that is used. The idea of Give-to-Get is to discourage free-riding by letting peers favour uploading to other peers who have proven to be good uploaders. As a consequence, free-riders are only tolerated as long as there is spare capacity in the system. To enable this, peers need to inform other peers about how much they upload, for which they used the G2G\_PIECE\_XFER message.

A peer that supports Give-to-Get should signal this fact to other peers by including the method Tr\_G2G in its BitTorrent extension protocol handshake [13]. This protocol is a simple mechanism that allows peers to communicate what BitTorrent extensions they support and to exchange the associated messages. We refer to it as the EXTEND protocol.

When it meets another peer that supports Give-to-Get in the same swarm, the peers send each other G2G\_PIECE\_XFER messages every time they upload a piece of content to any other peer. By default this message has message ID 235, unless negotiated otherwise via the EXTEND handshake. The message body consists of a piece number, an offset and a length (4-byte integers), detailing which part of which piece they uploaded.

### Version 2

During testing it was revealed that sending a G2G\_PIECE\_XFER message in the above format for every uploaded piece generated too much traffic, especially with the small piece sizes we use. Hence, we changed the message format such that it allows batched information transfer. Version 2 of the G2G\_PIECE\_XFER message therefore has the following payload. It consists of a bencoded dictionary, mapping the piece number uploaded (as a string) to string of length 1, encoding the percentage of the piece uploaded. The string contains the character obtained when the percentage of the piece uploaded is multiplied by 100 and converted to an integer, i.e., chr(int((100.0 \* perc)) to give a compact message. A peer supporting version 2 of the Give-to-Get message should use include the method 'Tr\_G2G\_v2' in the EXTEND protocol handshake, rather than 'Tr\_G2G'.

#### 2.3. TORRENT FILE EXTENSION

To estimate the amount of content to prebuffer we extend a torrent file to include the bitrate of the content. For security reasons it is encoded in the 'info' field of the torrent file, such that it cannot be modified without changing the identity of the torrent file (i.e., its infohash). For a single-file torrent the 'info' dictionary just contains an extra 'bitrate' field specifying the bitrate in bytes per second. For multi-file torrents, the per-file dictionary in the 'files' list is extended with a 'bitrate' field.

## 2.3.1. MetaDiscussion

In the current Tribler implementation (January 2009) we actual use the torrent-file extension made by Azureus which encodes the bitrate and other properties in a azureus\_properties field in the top-level dictionary. See http://svn.tribler.org/abc/branches/release-4.5/Tribler/Main/vwxGUI/zudeo\_torrent\_description. txt

# **Live Streaming**

Live streaming is the real-time broadcast of video or audio over a network of BitTorrent clients. Live streaming differs from video-on-demand in two important aspects: first, a live broadcast can have an infinite duration. Second, as a result, the content to be broadcast is not known beforehand. BitTorrent heavily depends on the content being known beforehand for its algorithms and integrity protection. We have extended the BitTorrent design to cover these two differences. For completeness, we also include our policy for playing back a live stream and how to replicate the source to get more seeding capacity.

#### 3.1. SUPPORTING INFINITE DURATION

The BitTorrent protocol assumes the number of pieces is known in advance. The number of pieces is used throughout a typical implementation to allocate arrays with an element for every piece. For that reason, it is not practical to simply increase the number of pieces such that the last piece will not be reached (for instance, 2<sup>32</sup>). Instead, we use a sliding window which rotates over a fixed number of pieces. Pieces which fall out of the window will be considered out-dated. Each peer deletes out-dated pieces, and will consider them to be deleted by its neighbours as well, thereby avoiding the need for additional messages. If a piece is out-dated by the time it is downloaded, it will not be offered for download to other peers. Within the sliding window, each peer barters for pieces as per the BitTorrent protocol.

The sliding window of the injector consists of W pieces up to and including the latest generated piece. Since the injector has to be able to serve any peer in case all other peers depart, no peer is allowed to have a playback position of more than W pieces before the latest generated piece. We use a rather large value of W (15 minutes), which allows the system to operate in a broad range of networks.

The sliding windows of the peers cannot be perfectly synchronized due to the differences in delays within the network. Since not all peers are connected to the injector, a peer P assumes newly generated pieces are at most W pieces beyond P's current playback position. On the other hand, the neighbours of P can request pieces up to W pieces behind P's current playback position. Every peer therefore maintains a sliding window of both W pieces before and W pieces after its current playback position. Note that not all of the pieces within the sliding window of a peer actually exist, as most of the sliding windows extend beyond the latest generated piece.

When a peer joins the network and connects to other peers, it learns about which pieces are available and has to decide which pieces are the latest. The other peers will report which pieces they own within their own sliding window. The pieces available in the neighbourhood of any peer span at

most 2W (barring network latencies and clock differences), as any piece is discarded after at most 2W after it has been generated. To be able to distinguish the beginning and end of the sliding windows, we need to rotate it over a total number of pieces corresponding to a ring of 4W pieces. Such a ring will always contain an empty range of at least 2W pieces in length. A peer can therefore synchronize on the latest piece available before an empty range of at least 2W pieces.

Malfunctioning and malicious clients may claim to have pieces outside the range which is considered valid by the other peers. To avoid basing the sliding window on such clients, a peer uses majority voting to determine the correct current position of the sliding window: only the pieces which are available at more than half the neighbours are considered.

### 3.2. SOURCE AUTHENTICATION

The BitTorrent protocol computes a hash for each piece and includes these hashes in the torrent file. In the case of live streaming, these hashes have to be filled with dummy values or not be included at all, because the data is not known in advance and the hashes can therefore not be computed when the torrent file is created. However, the lack of hashes makes it impossible for the peers to validate the data they download, leaving the system prone to injection attacks and data corruption.

We prevent data corruption by using asymmetric cryptography to sign the data, which has been found to be superior to several other methods for data validation in live P2P video streams. The injector publishes its public key by putting it in the torrent file. Each piece is assigned an absolute sequence number, starting from zero. Each piece, along with its absolute sequence number and time stamp, is signed by the injector. Such a scheme allows any peer to validate any packet as originating from the injector. By including the sequence number and time stamp, a peer is able to verify that the signed piece is actually the piece it requested. The sequence number allows a peer to confirm that the piece is recent, since the actual piece numbers are reused by the rotating sliding window. As a bonus, the included time stamps allow a peer to estimate the delay between the injector and its own playback position.

Our design allows for different methods of source authentication. We currently support ECDSA-based authentication as follows. A regular torrent file contains a pieces field in its info dictionary that contains the SHA1 hashes of the published content. We replace this pieces field with a live field that contains a dictionary. This live dictionary contains a single required field authmethod that specifies the authentication method to use (either ECDSA or None). If the method is ECDSA the dictionary contains an additional field pubkey that contains the EC-public key of the source in binary (DER) format. Putting this information in the info dictionary of the torrent makes it part of the identify of the torrent, and therefore it cannot be modified by malicious parties without changing the torrent's identity.

The payload of each piece is slightly reduced to make room for the metadata and the signature. In our case, the signature adds 65 bytes of overhead to each piece, and the piece number and timestamp add 16 bytes in total. The layout is as follows:

- an 8 byte sequence number
- an 8 byte real-time timestamp in UTC
- a 1 byte length field followed by
- a variable-length ECDSA signature in ASN.1 (max 64 bytes)
- optionally 0x00 padding bytes, if the ECDSA sig is less than 64 bytes

The data is stored in the last 81 bytes of the piece.

If a peer downloads a piece and detects an invalid signature or finds it outdated, it will disconnect from the neighbour that provided the piece. A piece is outdated if the timestamp indicates it was generated W pieces or longer ago.

#### 3.3. LIVE PLAYBACK

Before a peer starts downloading any video data, it has to decide at which piece number it will start watching the stream. In order to obtain the full download speed, a peer should not focus on the latest pieces if those pieces are only available at a small number of neighbours. As a trade-off, we let peers start downloading at *B* pieces before the latest piece which is available at at least half of the neighbours. In this *prebuffering phase*, a peer waits until it has downloaded at least 90% of these *B* pieces, to avoid waiting for pieces that are lost in the network, in transit, or take too long to download.the *prebuffering phase*.

During playback, the peer maintains a playback buffer of pieces to send to the video player inorder. The first amount of data sent to the video player is discarded as the video player seeks to the first complete picture to display. Since our algorithm deals with byte streams and does not know any frame boundaries, the amount of data discarded is hard to predict accurately. As a result, it is theoretically possible for a peer to have a buffer underrun immediately after playback has commenced.

A buffer underrun occurs when a piece i is to be played but has not been downloaded. Since pieces are downloaded out of order, the peer can nevertheless have pieces after i available for playback. If a peer has more than B/2 pieces available after i, the peer will drop missing piece i. Otherwise, it will stall playback in order to allow data to be accumulated in the buffer. Once more than B/2 pieces are available, playback is resumed, which could result in dropping piece i after all if it still has not been downloaded.

We employ this trade off since dropping nor stalling is a strategy that can be used in all situations. For instance, if a certain piece is lost because it never reached a peer's neighbours, it should be dropped and playback can just ignore the missing piece. On the other hand, a peer's playback position can suddenly become unsustainable if the neighbourhood of the peer changes. The new neighbours may only be able to provide the peer with older data than it needs. In that case, a peer should stall playback in order to synchronize its playback position with its new neighbours.

The value of *B* to use depends on the bitrate and the average network connection of a peer (latency and bandwidth). Previously we have used simulations to determine the optimal value of *B* for a given set of parameters.

#### 3.4. AUXILIARY SEEDERS

In BitTorrent swarms, a fraction of the peers has completed their download and seeds the content to others. The presence of seeders significantly improves the download performance of the other peers (the leechers). However, such peers do not exist in a live streaming environment as no peer has ever finished downloading the video stream.

We redefine a *seeder* in a live streaming environment to be a peer which is always unchoked by the injector and is guaranteed enough bandwidth in order to obtain the full video stream. The injector has a list of peer identifiers (for example, IP addresses and port numbers) representing trusted peers which are allowed to act as seeders if they connect to the injector. The seeders and leechers use the exact same protocol, but the seeders are guaranteed to be unchoked by the injector. The identity of the seeders is not known to the other peers to prevent malicious behaviour targeted at the seeders.

The injector thus controls the set of seeders, and has two incentives for maintaining that set. First, the seeders can provide their upload capacity to the network, taking load off the injector. The seeders behave like a small Content Delivery Network (CDN), which boosts the download performance of other peers as in regular BitTorrent swarms. Second, the seeders increase the availability of the individual pieces. All pieces are pushed to all seeders, which reduces the probability of an individual piece not being pushed into the rest of the swarm.

## **Small Torrent Files**

### 4.1. INTRODUCTION

BitTorrent requires a torrent file containing a cryptographic digest of every piece of the file to allow the verification of pieces during the download. Serving torrent files for large files puts a strain on the Web servers used. The idea is to replace the torrent file with a short, secure identifier. For example, a single cryptographic digest or BitTorrent URL.

A related problem is the use of large piece sizes. To keep the size of a torrent file small (as to not overload the Web servers) the number of hashes for a file is being kept small. For large files this implies that the piece size over which digests are calculated must go up (up to 2MB pieces are used). The large piece sizes affect the ability of peers to barter pieces. Only when a piece has been completely received and verified using the digest may it be traded with other peers. This means that it may be some time before a node starts bartering with others.

The idea is to replace the list of digests with a single Merkle hash [11]. A Merkle hash can be used to verify the integrity of the total file as well as the individual blocks via a hierarchical scheme. It works by constructing a hash tree of the content and using just the root hash as data integrity protection. The simple root hash value also allows for smaller piece sizes to be used. A common form of hash trees is the Merkle hash tree, hence the name.

## 4.2. SIMPLE MERKLE HASHES

We have chosen a minimalistic design that does not affect the existing BitTorrent protocol and clients very much. The design is backwards compatible in the sense that clients supporting the Simple Merkle Hash extension can still be made to process regular torrent files easily.

From the content we construct as hash tree as follows. Given a piece size, we calculate the hashes of all the pieces in the file set. Next, we create a binary tree of sufficient height. Sufficient height means that the lowest level in the tree has enough nodes to hold all piece hashes in the set. We place all piece hashes in the tree, starting at the left-most leaf. The remaining leaves in the tree are assigned a hash value of 0 (see Discussion). Finally, we calculate the hash values of the higher levels in the tree, 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 we call the *root hash*.

The root hash along with the total size of the file set and the piece size are now the only information in the system that needs to come from a trusted source. A client that has only the root hash of a file set can check any piece as follows. It first calculates the hash of the piece it received. Along with this

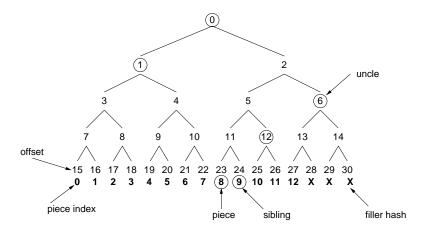


Figure 4.1: A Merkle tree for a file consisting of 12 pieces of content. Verifying the hash of piece index 8 by recomputing the root hash 0 requires all circled hashes (9, 12, 6, 1).

piece it should have received the hashes of the piece's sibling and of its "uncles", that is the sibling Y of its parent X, and the uncle of that Y until the root is reached. Using this information the client recalculates the root hash of the tree, and compares it to the root hash it received from the trusted source.

#### Discussion

We chose a binary tree for simplicity. Trees with larger degrees are also possible. However, the number of hashes that need to be sent with each piece is already small at about  $^2log$  of the file-set size.

The leaves of the tree that do not contain a piece hash currently get the value 0, e.g. 20 bytes with value 0 for a SHA-1 hash. Cryptographically, it is probably stronger to use other values there. The problem is, however, that the filler values would have to be known by any initial seeder, or seeder that lost its state (but not the set of files). Otherwise, it cannot construct the same hash tree. If we assume that a seeder should be able to start with just a torrent file and the set of files, the filler values must be public information. The cryptographic question is whether using public filler values is stronger than using 0s.

## 4.3. INCLUSION IN BITTORRENT

The original publisher of the file set creates a so-called *Merkle torrent* which is a torrent file that contains a 'root hash' key in its info part instead of a 'pieces' key. Merkle torrents have the extension .merkle.torrent.

When a seeder starts it uses the information in the Merkle torrent and the file set to reconstruct the hash tree and registers itself with the tracker using the hash value of the info part of the Merkle torrent, as usual (see Discussion).

A BitTorrent client that supports the Simple Merkle Hash extension must set bit 42 in the 8 reserved bytes in the BT header, where the left-most bit is bit 0 and the right-most is bit 63. Such a

client must not send PIECE messages but must use the new message type HASHPIECE to send pieces.

A HASHPIECE message consists of an index, begin, hashlist and piece. The hashlist consists of the piece's own hash, the piece's sibling hash, and the uncles of the piece up until and including the root hash (see above and Discussion). In particular, the hashlist is a list of 2-element lists. The first element denotes the node offset in the tree, the second element is the hash value. The node offset is the number of the node when numbered in a breadth-first fashion (i.e., going left to right starting at the top).

Only the HASHPIECE message with begin = 0 must contain a filled hashlist, for all other begin values the hashlist must be empty. In other words, the message containing the first subpiece should have a filled hashlist, subsequent subpieces should not.

A HASHPIECE message is a regular BT protocol message (as opposed to an overlay-swarm protocol message) with message ID 250 and has the following payload:

- 1. 4-byte index
- 2. 4-byte begin
- 3. 4-byte length of bencoded hashlist
- 4. the bencoded hashlist
- 5. the piece

Upon receipt of a HASHPIECE message, the receiver recomputes the root hash using the hashlist and compares it to the root hash in the Merkle torrent. If they match, all the hash values are saved in the receiver's own hash tree, such that they can be passed on to others when the piece is downloaded from this receiver. When all subpieces have come in, the piece is checked using the hash from the hash tree.

## Discussion

Using the hash of the 'info' part for registering at the tracker means that for a given file set, the swarm that use a conventional torrent file and the swarm that uses a Merkle torrent will be disjunct. The hash value is different, hence the swarms are known under different identifiers at the trackers.

In theory we can create one swarm. In that swarm, new clients could serve pieces to old clients. For the new clients to benefit from the old clients, however, we need to add some way for the new to obtain the hashes required to check a piece. Designing a fool proof solution for this problem is not trivial.

Because we let the initial seeders recalculate the hash tree, this extension is incompatible with the proposed HTTP Seeding extension by John Hoffman that allows seeding directly from Web sites [9]. Including the root hash in a HASHPIECE message allows a quick sanity check.

#### 4.3.1. MetaDiscussion

We should replace the mechanism for signaling we support the Merkle hash extension from the bit in the 8 reserved bytes in the BT header to using the official BitTorrent extension protocol [13].

# Secure, Permanent Peer Identifiers

### 5.1. INTRODUCTION

At present, BitTorrent does not require strong authentication of peers, as peer-to-peer interactions are transient and shortlived and security stems from the digests in the trusted torrent file. We want to establish longer term relationships between peers and introduce a number of privileged operations which should only be available to friends. We therefore extend the protocol with secure, permanent peer identifiers called *PermIDs*. We assume a PermID maps to a single IP address and port number and is initially also used to identify users. The mapping of PermID to IP address is controlled by the owner of the PermID (a user). Initially we used PermIDs for authentication of friends in cooperative downloads.

The idea is to use public-key cryptography and give each peer a public/private keypair, where the public key will act as the PermID. We intend to use Elliptic Curve-based public key cryptography [15] because it provides stronger protection using small keys than e.g. RSA-based algorithms [1]. Having small PermIDs is useful to allow caching of large numbers of (PermID,IP) pairs, as discussed next.

## 5.2. PERMIDS

In Tribler, each client creates a public/private key pair based on Elliptic Curve Cryptography. The (currently uncertified) public key acts as the PermID for the user. Users distribute this PermID to their friends out-of-band to establish trusted friend relationships. When two peers connect as part of a download, the Tribler client checks whether the peer supports our PermID extension. If so, it will also setup a overlay-swarm connection to the peer. To successfully set up an overlay-swarm connection both peers need to authenticate themselves using the standard ISO/IEC 9798-3 challenge/response identification protocol.

If the peer is successfully authenticated but not a friend of the user (i.e., does not appear in the list of friends' PermIDs), the Tribler client will allow it to request non-privileged operations, such as exchanging file preferences (see Section 6.2). If the peer is a friend, it may request privileged operations such as coordinating a friends-assisted download (see Section 8.2).

## **5.2.1.** Definitions and Terminology

**PermID** A PermID or permanent identifier is a public key in DER encoding. More specifically, it is a public key of an Elliptic Curve Cryptography (ECC) keypair that is generated on the sect233k1 curve, see [3].

It can be generated using OpenSSL 0.9.8 with point conversion set to POINT\_CONVERSION\_UNCOMPRESSED and the ASN.1 flag set.

#### **5.2.2.** Authentication Protocol

We implement a ISO/IEC 9798-3 challenge-response identification protocol, as described in \$10.3.3 (ii) (2) of [10].

Alice (A) and Bob (B) both have a ECC keypair as indicated above. Assume B initiates a connection to A. Assume B's BT handshake has been sent and A's received, so both sides know eachother's BT peer ID, and B has sent a BITFIELD message indicating what content it already has, if applicable (the BT spec says that this may only ever be sent as the first message, we stick to that). The exact message layout and content is described below.

- 1. To start the authentication protocol, B creates a 8092-bit random number  $r_B$  and sends this number in a CHALLENGE message to A.
- 2. On receiving the CHALLENGE message A also generates a 8092-bit random number  $r_A$ . It creates a signature  $S_A$  over  $r_A$ ,  $r_B$  and the BT peer ID of B, using its private key  $K_A$ . The signature is placed in a RESPONSE1 message along with  $pub_A$ ,  $r_A$ , and the BT peer ID of B.  $pub_A$  takes the role of  $cert_A$  in the ISO/IEC protocol specification.
- 3. A sends the RESPONSE1 message to B.
- 4. B verifies that the peer ID in the RESPONSE1 message is his peer ID. B verifies that the signature  $S_A$  is correct using the fields in the RESPONSE1 message.
- 5. If the signature  $S_A$  is correct, B believes it has successfully authenticated A.
- 6. Next, B creates a signature  $S_B$  over  $r_B$ ,  $r_A$ , and the peer ID of A, using its private key  $K_B$ . This signature is placed in a RESPONSE2 message along with  $pub_B$  and the BT peer ID of A.  $pub_B$  takes the role of  $cert_B$  in the ISO/IEC protocol specification.
- 7. A verifies that the peer ID in the RESPONSE2 message is his peer ID. A verifies that the signature  $S_B$  is correct using the fields in the RESPONSE2 message.
- 8. If the signature  $S_B$  is correct, A believes it has successfully authenticated B.

If any check fails in this protocol, the party discovering the problem simply does not continue with the protocol. At present, the peers are allowed to continue their BT interaction (i.e., we don't break the connection).

**CHALLENGE** Message ID 253. It consists of the message ID followed by a fixed length bencoded array of 1024 random bytes.

**RESPONSE1** Message ID 252. It consists of the message ID followed by a variable length bencoded dictionary. The dictionary has the following keys:

'certA' A variable-length public key in DER format.

'rA' 1024 random bytes.

**'B'** A 20-byte BitTorrent peer ID.

**'SA'** An ASN.1 encoded ECDSA signature (see "ANSI X9.62-1999: Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA).").

The signature is computed as follows. First we create a list L consisting of three items:  $r_A$ ,  $r_B$  and the BT peer ID of B, in that order. Next, we bencode that list, creating array M. We then compute the SHA-1 hash of M, creating hash H. This hash H is used as input to OpenSSL's ECDSA\_sign function, to create the ASN.1 encoded ECDSA signature.

**RESPONSE2** Message ID 251. It consists of the message ID followed by a variable length bencoded dictionary. The dictionary has the following keys:

'certB' A variable-length public key in DER format.

'A' A 20-byte BitTorrent peer ID.

'SB' An ASN.1 encoded ECDSA signature.

The signature is computed as described for RESPONSE1 except the list L now consists of three items:  $r_B$ ,  $r_A$  and the BT peer ID of A, in that order.

## 5.2.3. Integration into BitTorrent

See Sec. 5.3 on how to signal to other peers that the PermID extension is supported.

## **5.2.4.** Torrent Signatures

Torrents (either regular, or Merkle torrents (see Section 4.2) may be signed using a PermID, as follows. The metainfo, i.e. the dict in the torrent file, is extended with two entries (in addition to 'announce' and 'info')

'signer' The PermID of the signer (i.e., its public key in DER format).

**'signature'** The ECDSA signature in ASN.1 format.

The signature is computed over the SHA1 digest of the bencoded version of the metainfo without the two new entries. In other words, to verify the signature, make a deep copy of the metainfo, remove the 'signer' and 'signature' entries, bencode the resulting dictionary, compute the SHA-1 of the bencoded data, and verify it using OpenSSL's ECDSA\_verify with the PermID and signature as parameters.

#### **5.3. THE OVERLAY SWARM**

The recommendation and cooperative download feature both require new BitTorrent-protocol messages. We require a clean method for extending the protocol because our aim is to include more features in the future. Another requirement is being the least invasive in existing implementations. Furthermore, the current BitTorrent protocol does not allow communication outside the context of a swarm, that is, clients can only communicate with clients that are downloading the same file. For our extensions, we must be able to communicate outside the context of a single file swarm.

We therefore propose to create a new virtual swarm that encompasses all peers that are using the system, called the *overlay swarm* for high-level communication between peers. The swarm to which a

peer connection belongs is defined by the infohash field during the initial BitTorrent handshake. This infohash normally contains the SHA1 hash of the contents of the torrent file. In case of the overlay swarm, the infohash must contain a value of all zeros. The overlay swarm has no central BitTorrent tracker. A peer that wants, for example, to exchange preference lists with another peer must use the overlay swarm. The peer connects to the other peer's listen socket and uses the zero infohash value in the handshake. If the handshake is successful both parties know that new extension messages can be exchanged. After connecting to a peer on the overlay swarm the peers must run the challenge/response protocol from Section 5.2 to exchange and validate their PermIDs before any other communication.

By using this non-valid infohash value we remain fully backwards compatible and also are minimally invasive to the BitTorrent protocol. It also does not require extra TCP listen ports. The latter implies that no extra configuration of firewalls or Network Address Translators (NATs) is required by the user. This overlay can be extended in the future with new messages for secure gossip, sharing ratio enforcement, social networking, voting/moderation, reputation management, etc.

### 5.4. PROTOCOL VERSIONING

As we expect the overlay-swarm protocol to change frequently as new features are added or improved, we have to allow for many different versions of the protocol. Traditionally, the BitTorrent protocol has been versioned using the 64 reserved bits in the BitTorrent header:

'19'	'BitTorrent protocol'	reserved	info hash	peer ID
1	19	8	20	20

More accurately phrased, the client's features are expressed using the reserved bits. See [6] for the current allocation of the reserved bits. More recently, various vendors have started using the (now) official BT extension protocol [13]. We do the same as it has a larger identifier space and thus doesn't require coordination with other BitTorrent vendors to prevent clashes.

However, our Merkle-torrents extension (see (see Sec. 4.2) was developed before the BT extension protocol and currently still uses a reserved bit.

### **5.4.1.** Basic Protocol Versioning

We currently have one change to the basic protocol, namely, Merkle torrents. For this feature, we used a reserved bit. At the time, there was only one officially reserved bit, the right-most bit. This is used by BitTorrent 4 to indicated DHT-based tracker support. The "de facto" standards were as follows:

**Azureus** The left-most bit indicates support for the Azureus Messaging Protocol.

**BitComet** The first 2 bytes spell 'ex'.

Therefore, to prevent clashes we used bits in the middle of the 64-bit sequence. To indicate Merkle-torrent support, a client must set bit 42 (where the left-most bit is bit 0 and the right-most is 63).

## 5.4.2. Overlay-swarm Protocol Versioning

We expect a lot of protocol evolution in the swarm protocol, so it will be versioned differently. An important requirement is to allow backward compatibility. That is, if the protocol has been upgraded to V2, but a V2 client can still talk to a V1 client and vice versa, this should be possible.

To indicate support for any version of the overlay-swarm protocol, a client must use the BT extension protocol. In particular, it must define the extension Tr\_OVERLAYSWARM in the EXTEND handshake message 'm' field. Preferably the extension should have message ID 253.

Which versions of the protocol are spoken is currently communicated via the peer-ID field, as we do not use the peer ID in the overlay swarm. The versioning information will be encoded into the 20-byte peer ID field of the header as follows:

## Bytes 0–15 Used as before.

Bytes 16+17 16-bits big endian unsigned integer indicating the lowest protocol version this client supports.

Bytes 18+19 16-bits big endian unsigned integer indicating the current protocol version of this client.

In general, version negotiation works as follows. Peer A initiates an overlay-swarm connection to B encoding the lowest and current versions in A's peer ID. B checks if it supports a protocol in A's range. If not, it closes the connection. If so, it does not close the connection and sends its own handshake if it did not already do so (B may send it before the check). Upon receipt of B's info A does the same check. If it does not fail, A and B will choose the highest support protocol version.

Normally, the lowest protocol field should be set to the lowest version supported. Alternatively, a client may set it lower. Consider the following example: Assume there are 3 versions of the protocol: 3,4 and 5. Protocol 3 and 5 are good, but protocol 4 is broken. Client A wants to support all the good protocols, but not broken protocol 4. A then sets the oldest protocol it can support to 3 instead of 4 and the current one to 5.

To prevent using protocol 4, A then acts as follows. Assume B supports at most protocol 4 and at least protocol 2. A initiates an overlay swarm connection with B. Via the normal procedure the candidate protocol would be 4, which is unacceptable to A. To fix this situation A closes the current connection, and then reconnects to B with the oldest protocol and current protocol set to 3 (i.e., the only good protocol that both A and B can speak).

### **5.4.3. Protocol History**

- **v1** Used only internally.
- v2 First public release, Tribler>= 3.3.4
- v3 Second public release, Tribler>= 3.6.0, Dialback, BuddyCast 2.
- v4 Third public release, Tribler>= 3.7.0, BuddyCast 3.
- v5 Fourth public release, Tribler>= 4.0.0, Social Networking (SOCIAL\_OVERLAP message).
- **v6** Fifth public release, Tribler>= 4.1.0, Remote query, extra BC fields.
- v7 Sixth public release, Tribler>= 4.5.0, Remote monitoring and friendship making support.

## 5.4.4. MetaDiscussion

Should use just the EXTEND protocol to convey versioning information.

## **Decentralized Recommendation**

### 6.1. INTRODUCTION

The list of content a person downloads via BitTorrent can be considered the taste of the user. There are well-known centralized techniques for using your list of downloads and those of other users to discover new content that you will want to download. An example of such a recommendation technique is *user-based collaborative filtering* [2]. We have developed a decentralized version of this algorithm that will allow the Tribler client to make such personal recommendations.

### 6.2. BUDDYCAST PROTOCOL

Through its downloads the user builds up a *preference list* of content. The preference list contains by default all downloaded files from which the user can add or remove entries. These preference lists are exchanged freely amongst peers using the *Buddycast* algorithm. Using this algorithm the user builds a collection of a few hundred or more of such preference lists. This collection is called the *Preference Cache*.

The recommender component uses the Preference Cache to calculate both similarity between peers and to recommend certain content the user is predicted to like, using a special collaborative filtering algorithm. When a certain peer has a preference list with high similarity to the user's they have the same download taste. We call such similar peers *taste buddies*.

## **Epidemic Protocol**

The Buddycast algorithm is based on an epidemic protocol and roughly works as follows. Each peer maintains two lists of peers: (1) a list of its top-*N* taste buddies along with their current preference lists, and (2) a list of random peers. Periodically, a peer selects an entry from one of the lists and sends it its preference list, taste-buddy list and a selection of random peers. The receiving peer stores the preference list and uses the taste buddy and random peer info to update its own lists.

Furthermore, if the sending peer has downloaded some content which is of interested to the receiving peer (according to the collaborative filtering algorithm), the receiving peer may request the associated torrent file for the content from the sender, using a GET\_METADATA message. It will also download the torrent files from some randomly selected content, to improve the spread of information through the network. The whole process is referred to as *torrent collecting*. We alternatingly select a random peer and a taste buddy to exchange with. The exact protocol is described below.

BuddyCast takes into consideration the *connectivity* of the peers. A peer is connectible if it can be reached by another peer from the Internet. An unconnectible peer can only talk to other peers if it itself initiates a connection. We found that many peers are, unfortunately, unconnectible due to extensive firewalls and the dynamic property of peer-to-peer networks. To counter this problem, BuddyCast keeps open connections with a number of peers and only uses the addresses of the peers it is currently connect to fill the outgoing message, so all the peers broadcast by a peer are online.

Initially, the links to similar peers created by the BuddyCast algorithm were used just for recommending new content in a passive way. Later, we started using the links to answer active keyword searches from the user for particular content. The rationale is still that the similar peers are more likely to have the content the user is searching for and thus keeping links to them gives a higher hit rate. The details of the search protocol are described in Sec. 7.2. To improve the keyword search capability we extended the protocol. A peer now sends a list of recently collected torrents, that is, torrents he recently retrieved from other peer or obtains from an RSS feed. This list ensures that when two random peers meet, they both can discover and exchange a fresh torrent file.

### 6.3. DETAILED ALGORITHM

A peer running BuddyCast 2 keeps the following data structures in memory:

**Connection List** C - A list of peers to whom we keep open TCP connection.

C consists of the following 3 sublists.

- Connectible Connected Taste Buddy List  $C_T$ : A list of connectible peers to whom we keep a connection and which have similar tastes as us. The maximum number of peers in this list is 10.
- Connectible Connected Random Peer List  $C_R$ : A list of connectible peers who established connection with us most recently and are not in  $C_T$ . The maximum number of peers in this list is 10.
- Unconnectible Connected Peer List  $C_U$ : A list of unconnectible peers who connected to us. The maximum number of peers in this list is 10.

Connection Candidates List  $C_C$  - A list of peers which we can select as the target for a BUDDY-CAST message. The maximum number of peers in this list is 100.

**Block List** B - It contains a number of peers which you should not contact in a period (4 hours). It includes a Send Block List  $B_S$  (do not send message to any peer in this list) and a Receive Block List  $B_R$  (discard messages received from any peer in this list).

In addition to the in-core lists, every Tribler client has several database to store the information of peers, torrents and preferences it discovered in the network. We call these database the *megacaches*. Using the megacaches, a Tribler client can calculate similarity between peers and recommend torrents to download.

## 6.3.1. Pseudo Code

When Tribler starts it executes the algorithm shown in Figure 6.1, and sends out BUDDYCAST messages periodically. When it receives a BUDDYCAST message, the client executes the algorithm of Figures 6.2, updating its in-core lists and the megacaches. Both the send and receive algorithms use the

Table 6.1: Functions in BuddyCast

Function	Description
blockPeer(Q, block_list, time)	Add the peer Q into block_list for a period of time
fillPeers(message)	Put the addresses from the indicated list in the <i>message</i>
connectPeer(Q)	Connect to peer Q
getSimilarity(Q)	Get the similarity between peer $Q$ and myself

```
1: loop
       wait(\Delta T time units) {15 seconds in current implementation}
2:
       remove any peer from B_S and B_R if its block time was expired.
 3:
       keep connection with all peers in C_T, C_R and C_U
 4:
      if idle\_loops > 0 then
 5:
         idle\_loops \leftarrow idle\_loops - 1 {skip this loop for rate control}
6:
      else
 7:
 8:
         if C_C is empty then
            C_C \leftarrow select 5 peers recently seen from Mega Cache
9:
         end if
10:
         Q \leftarrow select a most similar taste buddy or a most likely online random peer from C_C
11:
         connectPeer(Q)
12:
         blockPeer(Q, B_S, 4hours)
13:
         remove Q from C_C
14:
         if Q is connected successfully then
15:
            buddycast\_msg\_send \leftarrow \textbf{createBuddycastMsg}()
16:
            send buddycast_msg_send to Q
17:
            receive buddycast_msg_recv from Q
18:
            C_C \leftarrow \text{fillPeers(buddycast\_msg\_recv)}
19:
            addConnectedPeer(Q) {add Q into C_T, C_R or C_U according to its similarity}
20:
21:
            blockPeer(Q, B_R, 4hours)
         end if
22:
      end if
23:
24: end loop
```

Figure 6.1: The protocol of an active peer.

createBuddycastMsg, addConnectedPeer and miscellaneous methods shown in Figure 6.3, Figure 6.4, and Table 6.1, respectively.

### **6.3.2.** Valid Peers and Bootstrapping

The BUDDYCAST message requires that each client knows other online peers. After the software is installed the client needs to obtain an initial online peer. We call this process bootstrapping and use well known superpeers to solve it. The addresses of the super peers are preloaded in the client's Peer Cache.

After installation the Tribler client will:

• Select a superpeer randomly from the Peer Cache.

```
1: loop
       receive buddycast_msg_recv from Q
 2:
      C_C \leftarrow \text{fillPeers(buddycast\_msg\_recv)}
3:
       addConnectedPeer(Q)
 4:
      blockPeer(Q, B_R, 4hours)
 5:
      buddycast\_msg\_send \leftarrow createBuddycastMsg()
 6:
       send buddycast_msg_send to Q
 7:
      blockPeer(Q, B_S, 4hours)
8:
       remove Q from C_C
9:
       idle\_loops \leftarrow idle\_loops + 1 {idle for a loop for rate control}
11: end loop
```

Figure 6.2: The protocol of a passive peer.

## function **createBuddycastMsg()**

```
My\_Preferences \leftarrow the most recently 50 preferences of the active peer Taste\_Buddies \leftarrow all peers from C_T Random\_Peers \leftarrow all peers from C_R buddycast\_msg\_send \leftarrow create an empty message buddycast\_msg\_send attaches the active peer's address and My\_Preferences buddycast\_msg\_send attaches addresses of Taste\_Buddies buddycast\_msg\_send attaches at most 10 preferences of each peer in Taste\_Buddies buddycast\_msg\_send attaches addresses of Random\_Peers
```

Figure 6.3: The function of creating a BUDDYCAST message

## function addConnectedPeer(Q)

```
if Q is connectable then Sim_Q \leftarrow \text{getSimilarity}(Q) {similarity between Q and the active peer} Min_{Sim} \leftarrow \text{similarity of the least similar peer in } C_T if Sim_Q \geq Min_{Sim} or (C_T \text{ is not full and } Sim_Q > 0) then C_T \leftarrow C_T + Q move the least similar peer to C_R if C_T overloads else C_R \leftarrow C_R + Q remove the oldest peer to C_R if C_R overloads end if else C_U \leftarrow C_U + Q end if
```

Figure 6.4: The function of adding a peer into  $C_T$  or  $C_R$ 

- Connect to this superpeer via the overlay swarm.
- Send a BUDDYCAST message with an empty preference list.
- Receive a BUDDYCAST message from the superpeer with filled in random-peers list...
- Initiate Buddycast using the superpeer's random peers.

When a superpeer is sent a BUDDYCAST message, this superpeer will respond with random-peers list. It may also have a filled in taste buddies list and preference lists. The latter can be used to promote certain important content and bootstrap a taste network around it.

### 6.3.3. Rate Control

A vital part of any epidemic protocol such as Buddycast is controlling the bandwidth it uses. Within a single minute it is possible to exchange preferences with many peers. When file downloads take days to complete it is important that no excessive amount of bandwidth is consumed by Buddycast. However, discovery of new files and new peers means that some amount of bandwidth needs to be spent.

Currently we use a simple policy to control rate: When starting for the very first time, we contact a peer every second for the first 2 minutes. For subsequent starts we contact a peer every 5 seconds for the first 30 minutes. From 30 minutes till 24 hours that we contact a peer every 15 seconds, after that once every minute. If we exchanged preference with a peer in the last 4 hours, we will not contact it again (but that peer can still connect you since it may have changed its preference).

### **6.4. WIRE FORMAT**

The preference and taste buddy lists of a client are exchanged via the overlay swarm (see Section 5.3), using a new BUDDYCAST message. The payload of this message contains 50 recent entries from your preference list, as well as the address information of your 10 most similar taste buddies and 10 random peers from your Peer Cache.

The exact format of the BUDDYCAST message is as follows. Its message ID is 249 and its payload consists of a bencoded dictionary with the following keys. All character string values are UTF-8 encoded.

'connectable' Whether I am directly reachable from the Internet (Boolean)

'ndls' My total number of downloads (integer)

'nfiles' Total number of torrents I collected (integer)

'npeers' Total number of peers discovered (integer)

'name' My name as a string.

'ip' My current IP address (string encoding, in dotted quad format).

'port' My listen port number (integer encoding).

**'preferences'** List of infohashes, one per preferred file (byte string). The minimum length of this list is 0, the maximum length is 50.

- 'taste\_buddies' A list of taste-buddy records. The minimum length of this list is 0, the maximum length is 10. A taste-buddy record is a dictionary containing the following keys:
  - **'preferences'** List of infohashes, one per preferred file (byte string). The maximum length is 0 (i.e., unused, see below)
  - 'PermID' Public key of the taste buddy (string encoding).
  - 'ip' The last known IP address of the taste buddy (string encoding, in dotted quad format).
  - **'port'** The port number that the taste buddy is listening on (integer encoding).
  - **'similarity'** Similarity between me and this taste buddy as an integer, higher meaning more similar.
  - 'connect\_time' When I established a connection with this taste buddy as an integer.
  - 'oversion' The overlay-protocol version (see Sec. 5.3 spoken by this taste buddy.
  - 'nfiles' Number of torrent files this taste buddy has collected as an integer (used to select peers to send remote keyword searches to).
- **'random\_peers'** List of peer addresses. The minimum length of this list is 0, the maximum length is 10. A peer address is a dictionary with the following keys:
  - 'PermID' Public key of the random peer (string encoding).
  - **'ip'** The last known IP address of the random peer (string encoding, in dotted quad format).
  - **'port'** The port number that the random peer is listening on (integer encoding).
  - **'similarity'** Similarity between me and this random peer as an integer, higher meaning more similar.
  - 'connect\_time' When I established a connection with this random peer as an integer.
  - **'oversion'** The overlay-protocol version (see Sec. 5.3 spoken by this random peer.
  - 'nfiles' Number of torrent files this random peer has collected as an integer (used to select peers to send remote keyword searches to).
- **'collected torrents'** List of infohashes, one per recently collected file (byte string). The minimum length of this list is 0, the maximum length is 50.

After receiving a BUDDYCAST message a peer must directly send its own message in reply, filled with the peer's own preferences and taste buddies. After sending the reply, the peer updates its *Preference Cache* and its *Peer Cache*. We do not yet take security into account and thus simply overwrite existing entries if the age the obtained preference list is superior then any possible previous entry. If a peer encounters unknown infohashes in the preference lists it may send a GET\_METADATA message to obtain the metadata (i.e., the torrent file) of this new content, as described below.

To keep the overlay-swarm connections to taste buddies and random peers open, (see Figure 6.1) an active peer may send KEEP\_ALIVE messages periodically. The message ID of a KEEP\_ALIVE message is 240 and it has no body.

## 6.4.1. History and Open Issues

The above specification describes the BUDDYCAST message as it is in version 6+7 of the overlay-swarm protocol. Previous versions have the following changes:

- v3 Added 'connectable' field.
  - Removed 'age' field from per taste-buddy/random-peer dict.
- Added 'collected torrents' field.
  - Added 'similarity' field to per taste-buddy/random-peer dict.
  - Deprecated 'preferences' field in per taste-buddy/random-peer dict, now always empty.
- Added 'npeers', 'ndls' and 'nfiles' field.
  - Added 'oversion' field to per taste-buddy/random-peer dict.
  - Added 'nfiles' field to per taste-buddy/random-peer dict.

In January 2009 we started working on a new version of BUDDYCAST that uses overlay-protocol version 8, to be documented here later.

#### 6.4.2. MetaDiscussion

- The 'ip' and 'port' fields are superfluous, a receiver should use the IP address and listen port obtained during overlay-connection establishment (see Sec. 5.3).
- The protocol needs a security review. E.g. the 'permid', 'ip' and 'port' fields of taste buddies and random peers should be secured by self-signing such that malicious peers cannot change the contact info (ip+port) for those peers.
- The empty 'preferences' field in a per-taste-buddy dict is superfluous.
- The 'name' field is superfluous when BUDDYCAST is used in combination with the social network SOCIAL\_OVERLAP message, see Sec. 9.2.1.

#### 6.5. OBTAINING METADATA

After a preference exchange the GET\_METADATA message is used to obtain information on a unknown infohash. The response is a METADATA message containing the torrent file for the given infohash. Since this mechanism was added to Tribler in 2005, an official BitTorrent extension has been proposed for obtaining the torrent file of a swarm. See [8].

The exact format of the GET\_METADATA message is as follows. Its message ID is 248 and its payload consists of a bencoded infohash.

The exact format of the METADATA message is as follows. Its message ID is 247 and its payload consists of a bencoded metadata record. A metadata record is a dictionary with the following keys:

'torrent\_hash' The infohash of the torrent (byte string)

**'metadata'** The torrent (byte string)

'last\_check\_time' Time of last check at the torrents tracker for how many peers are in the swarm, UTC in seconds as integer.

**'status'** String indicating the status of the checks:

- 'good' The tracker is responding.
- 'dead' The tracker has not responded on a number of tries.
- 'unknown' The tracker is flaky.
- 'leecher' Number of downloaders at last check.
- 'seeder' Number of seeders at last check.

The check fields were added in version 4 of the overlay-swarm protocol.

## 6.5.1. MetaDiscussion

There is no need to bencode infohash in the GET\_METADATA message payload.

## 6.6. FUTURE SECURITY

As part of our research into security for epidemic protocols we are looking into *Socially-Inspired Reputations* (SIR). The challenge of security is to identify malicious peers, to prevent DoS attacks, and to prevent information poisoning. SIR presumes that distributed reputation systems must be inspired by human mechanisms to judge believability of gossip. Our vehicle for demonstrating SIR is a new epidemic protocol called *BarterCast* that continously spreads information about the positive behavior of peers.

BarterCast uses the connections established by the peer-selection algorithm of BuddyCast to exchange information on positive actions of other peers. By storing all this "gossip" in the Tribler MegaCache, each peer can use the opinion of others to estimate trustworthiness levels. In BarterCast we gossip on the donation of upload bandwidth. Each peer keeps track of which peers have given him the most bandwidth and uses this to form an outgoing BARTERCAST message. This message contains a complete barter record, with both the number of uploaded and of downloaded MBytes, for 10 peers. Consequently, every four hour BuddyCast cycle a peer tries to receive fresh information on barter activities from 960 gossipers. A BARTERCAST message is a single message and is exchanged after a BUDDYCAST message.

## 6.6.1. Wire Format

A BARTERCAST message has message ID 236 and its payload consists of a bencoded dictionary with the following fields:

- 'totals' A tuple containing the total amount of data uploaded by me and the total amount of data downloaded by me, respectively. Both are integers indicating kibibytes.
- 'data' A dictionary mapping a peer's PermID to a barter record. A barter record is a dictionary with two keys:
  - **'u'** The amount of data I uploaded to this peer (integers indicating kibibytes).
  - 'd' The amount of data I downloaded from this peer (integers indicating kibibytes).

## 6.6.2. MetaDiscussion

As the BARTERCAST info is not yet used for building reputations the feature is as yet incomplete.

## **Remote Search**

## 7.1. INTRODUCTION

The BuddyCast protocol establishes connections between a user and its taste buddies, see Sec. 6.2. Initially, these connections were established to allow meaningful recommendation of new content to users following the principle of collaborative filtering. This principle says that if two people, *A* and *B*, have a similar taste in content, then any new content that *A* downloads is also likely to be appealing to *B* and vice versa. At the moment, taste buddies periodically exchange information about new content which then results in a list of recommended items for the user. As a next step we now use these connections to taste buddies to implement an efficient search mechanism for content. If a user wants to watch some new content that he heard about, but it is not on his list of recommended items yet, he can now do an explicit search of the databases of his taste buddies to see if they have already found this content. This mechanism, known as semantic-overlay search, has been shown to yield high hit rates [16].

## 7.2. PROTOCOL

The remote search mechanism queries the megacaches of the peers you are currently connected to (as a result of the BuddyCast protocol). To query a peer's database, the client sends an overlay-swarm QUERY message to the peer, containing a query ID and a query specification. The receiving peer checks if the sender has not exceeded the quota for QUERY messages. For senders who are marked as friends by the receiver's user, the quota is unlimited. For unknown senders there is a 100 query quota. If the quota has not been exceeded, the receiver parses the query and executes it on its megacache. The results are then sent back in an overlay-swarm QUERY\_REPLY message that carries the same query ID and a set of answers. At the moment, queries are limited to simple keyword searches in the receiver's torrent database, but it can be extended to full-fledged SQL-like queries in the future.

When the query results come in, they are displayed to the user. When the user decides to download one of the found torrents, the user clicks on the result, and its client then sends a GET\_METADATA message to the peer that returned the result. The peer, if still online, will then return the desired .torrent file in a METADATA message. This is the same mechanism for obtaining a torrent file from a peer as used in the Cooperative Download feature, see Sec. 8.1.

#### 7.3. WIRE FORMAT

A QUERY message has message ID 238 and its body consists of a bencoded dictionary with the following fields:

- **'id'** A 20-byte random query ID.
- 'q' A string which consists of the fixed word SIMPLE followed by a space and a number of space-separated keywords.

A QUERY\_REPLY message has message ID 237 and its body consists of a bencoded dictionary with the following fields:

- **'id'** A 20-byte random query ID.
- 'a' A dictionary containing the query results.

For the current simple queries the 'a' dictionary is a map from torrent IDs (infohashes) to torrent records. A torrent record is itself a dictionary with the following keys:

'content\_name' The 'name' field of the torrent as string.

**'length'** The total size of the content in the torrent as 4-byte integer

'leecher' The current number of downloaders as 4-byte integer.

**'seeder'** The current number of uploaders as 4-byte integer.

'category' A list of strings denoting the categories this torrent was classified into.

# **Cooperative Downloading**

### 8.1. INTRODUCTION

When there are few seeders in BitTorrent, your download speed is equal to your upload speed. As most people have an asymmetrical network connection with a maximum download speed that is larger than the upload speed, the downlink is often not fully exploited. If you have a group of friends whose connections are idle, they can be used to fill the downlink. For example, the friends can each download a piece of the file that you not yet have using standard BitTorrent. Once the piece is received, they then send it to you over your underutilized downlink without expecting any data in return. So by doing barter-free downloads with your friends you can utilize your asymmetric network link to its fullest.

## 8.2. PROTOCOL

Peers from a social group that decide to participate in a cooperative download take one of two roles: they are either *coordinators* or *helpers*. A coordinator is the peer that is interested in obtaining a complete copy of a particular file, and a helper is a peer that is recruited by a coordinator to assist in downloading that file. Both coordinator and helpers start downloading the file using the classical BitTorrent tit-for-tat and cooperative download extensions. Before downloading, a helper asks the coordinator what piece it should download. After downloading a file piece, the helper sends the piece to the coordinator without requesting anything in return. In addition to receiving file pieces from its helpers, the coordinator also optimizes its download performance by dynamically selecting the best available data source from the set of helpers and other peers in the BitTorrent network. Helpers give priority to coordinator requests and are therefore preferred as data sources.

## The Protocol in Detail

To invoke the help of a friend, the coordinator opens an overlay-swarm connection (see Section 5.3) to the helper and sends a DOWNLOAD\_HELP request. When the DOWNLOAD\_HELP message is received, and the helper is willing to help, it obtains the torrent file to use from the coordinator using a GET\_METADATA message. After receiving the corresponding METADATA message, the helper establishes two connections with the coordinator: a *control connection* and a *data-exchange connection*.

The control connection is used by the helper for claiming pieces at the coordinator. Control connections are blocking; the helper can block waiting for the response from the coordinator. Only the control messages RESERVE\_PIECES and PIECES\_RESERVED are sent over the control connections.

The pieces of the file the helper downloaded on behalf of the coordinator are transferred over the dataexchange connection. The control connection was introduced to make sure the helper immediately knows whether a particular piece was claimed elsewhere or not.

Whenever helper H wants to download a piece P, H first contacts the coordinator and tries to reserve (claim) P using a RESERVE\_PIECE message. If P was not claimed by anybody else, the coordinator sends a PIECE\_RESERVED message in return, and H, in turn, sends requests for piece P to the offering peer in the swarm. Otherwise, H checks its download bandwidth utilization. If the amount of unused download bandwidth is above a certain threshold, H requests P (although P was claimed by some other helper).

In order to decrease the number of messages exchanged between the coordinator and its helpers, the coordinator from time to time appends a list of all pieces that have been already claimed by others to the PIECE\_RESERVED reply to a helper. This optimization greatly improves the performance in the later stages of the download when most of the pieces have already been claimed, and only a few still have to be downloaded. With this list, the helper can then determine locally which pieces have been already obtained without asking coordinator for the status of each piece separately.

The coordinator decides to download a missing piece from either one of its helpers or any other peer in the swarm using the standard BT peer selection mechanism. A helper which is getting a REQUEST message for a piece from the coordinator puts this request in front of its sending queue, consequently giving them the highest priority. The connections between helper and coordinator are never choked.

#### 8.3. WIRE FORMAT

The control connection is an overlay-swarm connection, see Sec.5.3. The data-exchange connection is a regular BT connection for the torrent in question.

The DOWNLOAD\_HELP message has message ID 246 and its body consists of the infohash of the torrent to help with.

The STOP\_DOWNLOAD\_HELP message has message ID 245 and its body consists of the infohash of the torrent to stop helping with.

The RESERVE\_PIECES message has message ID 242 and is a concatenation of a 20-byte infohash, a 1-byte *all-or-nothing* field, and a bencoded list of piece numbers (integer encoding). The all-or-nothing field has value 0x1 if the helper wants a reservation for all given pieces, but if that is not possible wants no reservation of pieces at all. A value of 0x0 indicates that the helper is willing to accept a partial reservation.

The PIECES\_RESERVED message has message ID 241 and is a concatenation of a 20-byte infohash, and a bencoded list of piece numbers (integer encoding). Piece numbers may be positive or negative. In the latter case they represent pieces that have already been reserved by others.

#### 8.3.1. MetaDiscussion

We should overhaul to protocol to be more in-line with other messages, i.e., use a bencoded dictionary with keys as payload instead of concatenation.

# **Social Networking**

#### 9.1. INTRODUCTION

The social networking features of Tribler consist of a message for exchanging nickname and avatar pictures and a mechanism for friendship establishment that does not require the two parties involved to be online at the same time. It also does not require a central component to achieve this asynchronicity.

## 9.2. NICKNAME AND PICTURE EXCHANGE

In the text we will use the terms public name and picture. A user's public name is the name chosen by the user by which he will be publicly known. As it is chosen by the user, it is not necessarily system-wide unique. A user's public picture is a picture chosen by the user that will be shown to other users.

## 9.2.1. Wire Format

This feature is implemented by extending the overlay-swarm protocol with one message. After identification via the challenge/response protocol, if the peer A does not know peer B, the peer with the lexicographically smallest PermID sends a SOCIAL\_OVERLAP message. A may also send a SOCIAL\_OVERLAP message to peer B it has met before, but a long time ago, so they both get a up-to-date view of each other's info.

A SOCIAL\_OVERLAP message has message ID 239 and consists of a dictionary with the following keys:

'persinfo' A dictionary containing the personal information about the peer.

A persinfo dictionary has the following keys:

**'name'** Public name of the peer (string encoding)

**'icondata'** Public picture of the peer (string encoding)

**'icontype'** The MIME type of the public picture

#### 9.3. FRIENDSHIP MAKING

To facilitate the creation of social networks we designed a protocol for friendship establishment that does not require the two parties involved to be online at the same time. It also does not require a central component to achieve this asynchronicity.

The basic friendship making protocol is a simple request-reply protocol. A peer A wanting to become friends with peer B sends him a FRIENDSHIP message with message type 'REQ'. When peer B is online and receives this request this is somehow signaled to the user of B. If the user responds to the request, peer B returns a FRIENDSHIP message with a message type 'RESP' conveying the user's positive or negative response.

The special feature of this friendship protocol is its support for offline peers. If B is offline at the time of the friendship request by A, A will (after a retry) delegate the forwarding of the message to its fellow peers. This also holds for peer B sending a friendship response when A is down. In particular, a client will ask a mix of friends and online taste buddies in a ratio of 70% friends (if he has them) and 30% taste buddies, where taste buddies are a concept in BuddyCast that denote people that have similar taste in content. The client sends each helper a FRIENDSHIP message of type 'FWD' containing the contact info for the destination peer and the message body of the REQ or RESP message to forward. The original client and its helpers will attempt to contact the destination for a certain period of time, after which the message is discarded. Forwarded messages are not forwarded to others when the final destination peer remains offline.

### 9.3.1. Wire format

A FRIENDSHIP overlay-protocol message has message ID 234 and its body consists of a bencoded dictionary with a single common field:

'msg type' A variable-length string, currently 'REQ', 'RESP' or 'FWD'.

When the message type is 'REQ' there are no other fields. When the message type is 'RESP' it contains a single field:

**'response'** Whether the request from the peer was approved or denied encoded as a 1 or 0 integer value.

The 'FWD' message has three extra fields:

**'source'** Contact info of the original source of the message.

'dest' Contact info of the final destination of the message.

'msg' Dictionary with the body of the forwarded message.

Contact information itself, in turn, is a dictionary containing three fields:

'permid' The binary PermID of the peer.

**'ip'** The IP address of the peer as a string in dotted notation.

'port' The port of the peer as an integer.

## NAT/Firewall Detection and Traversal

#### 10.1. INTRODUCTION

With the increasing shortage of IP-address space and rising security concerns, more and more users will access the Internet from behind a Network-Address Translator (NAT) or firewall. We have designed a mechanism for a client to detect if it is behind a firewall. In addition, we added a platform-independent implementation of the Universal Play and Play (UPnP) protocol (Internet Gateway Device) for remotely configuring firewalls (http://www.upnp.org/). As another step towards full support for dealing with NATs we have implemented detection for which type of NAT is being used (as opposed to detecting just the presence of a NAT).

## 10.2. EXTERNAL ADDRESS DISCOVERY

Many of the clients run on a machine with or behind a firewall or Network Address Translator (NAT). This poses several problems:

- 1. Unless the Tribler listening port (7762 by default) is opened on the firewall, other peers cannot connect to it.
- 2. The Tribler client can no longer obtain the IP address via which it is reachable on the Internet from the operating system. As a result, it is not able to provide this address to others.

To solve these problems we have extended Tribler with a facility for detecting a firewall, and discovering a client's external IP address. In particular, we added two messages to the overlay-swarm protocol called the *dialback messages*. These messages are used as follows. At client startup, 7 peers are selected from the database of encountered peers. This database is initially filled with the addresses of the 8 Tribler superpeers. The client attempts to send a DIALBACK\_REQUEST to each of the 7 peers using the overlay swarm.

When a peer B receives a DIALBACK\_REQUEST it closes the existing overlay-swarm connection. It then tries to connect back to the initiating peer A. In particular, it will try to connect back to the IP address X that initiated the previous connection and the listen port that peer A specified in the Bit-Torrent handshake message (see Sec. 5.3). If the connection succeeds, B sends an DIALBACK\_REPLY message containing the IP address X it used to connect. Peer B thus informs the initiating client A that (1) the client is reachable from the Internet and (2) what its external IP address is (which is X).

To protect against malicious peers, the client will record the external IP addresses returned by the 7 peers and select the address the majority agrees on as being its real address. If there was no majority

either because not enough peers replied or they disagreed, the client start the process over again with 7 other peers after 30 seconds. The client will retry 5 times, so contact at most 35 peers.

In order to improve reachability of peers the client warns the user. The user interface clearly indicates when Tribler is not reachable and that port forwarding should be turned on its firewall for full performance.

#### **10.2.1.** Wire Format

A DIALBACK\_REQUEST message has message ID 244 and has no body.

A DIALBACK\_REPLY message has message ID 243 and its body consists of a bencoded IP address in dotted string notation.

### 10.2.2. MetaDiscussion

No need to bencode the DIALBACK\_REPLY message payload. There is now an official BT proposal to let the tracker return the external IP address of a client, see [7]. Also the EXTEND protocol handshake defines a 'yourip' field, see [13].

### 10.3. DETECTION OF NETWORK ADDRESS TRANSLATOR (NAT) TYPE

The NAT-detection functionality performs a simple check of the NAT/firewall type. In addition, it tests the duration of a given mapping from (internal address, internal port) to (external address, external port) in the NAT/firewall when not used for a while. In other words, when does a mapping in a NAT/firewall expires if there is no communication going through. This value is referred to as the *NAT timeout*. The NAT type check is done using a simplified version of the STUN protocol [14] and requires the assistance of several STUN servers, while the timeout check requires the assistance of a timeout server. The NAT-type detection can be activated remotely via the Remote Monitoring facility described below, and is used to measure the network connectivity of the average peer.

## 10.3.1. Type Detection

The NAT/firewall type detection process is depicted in Figure 10.1. The process is performed using the STUN algorithm, and therefore it requires the peer to know the addresses of at least 2 STUN servers. When performing the Test I, the peer sends a ping1 message to a STUN server. Upon receiving such a request, the server replies sending back a message containing the IP address and port it received the request at. Test I is used to understand whether a peer has a public IP address. If Test I is repeated using two different STUN servers (different addresses), the peer can also understand whether the NAT is allocating for it the same port in both communications. Test II and Test III are needed to check the filtering characteristics of a NAT/firewall. More specifically, when a peer is running Test II, it sends a ping2 message to a STUN server. The server then delegates another STUN server (residing on a machine with a different IP address) to send a reply back to the peer from which the request was originated. During Test III, instead, the peer sends a ping3 message to a STUN server, that replies from a different socket (bound on a different port). All the replies the STUN servers send to the requesting peers are in the format *ipaddress:port*.

## 10.3.2. Timeout Detection

The NAT timeout detection is based on a very simple algorithm. The peer sends messages in the format ping:  $\langle delay \rangle$  to the timeout server, meaning that it expects the server to reply after delay seconds. More specifically, the peer opens a new socket for each delay value in  $\{25, 35, 55, 85, 115, 145\}$  and assumes as NAT timeout the largest delay value for which it receives a reply from the server. The server's response is in the format pong:  $\langle delay \rangle$ .

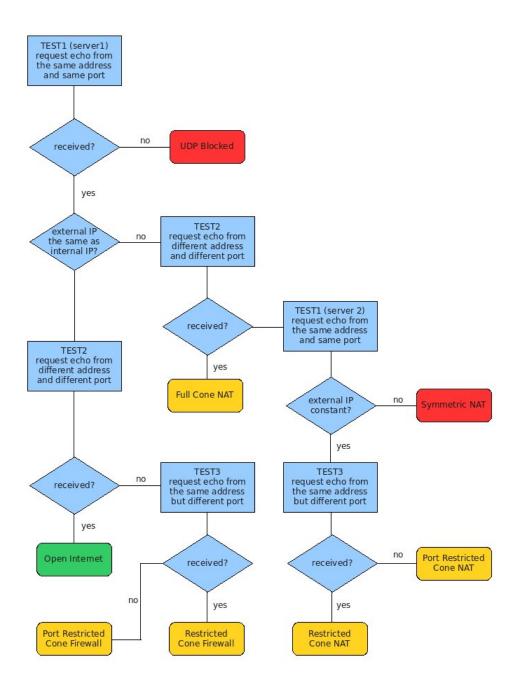


Figure 10.1: The process for detecting the NAT type used.  $^{36}$ 

# **Remote Monitoring**

#### 11.1. INTRODUCTION

To measure the behaviour of the system the client a generic crawler architecture was added. It allows monitoring of the system from different hosts to gain a broader picture. Its primary purpose is to let researchers to measure different aspects of the system, but can also be used for general health monitoring and statistics gathering.

Remote monitoring is performed by *Crawler processes*. Crawler processes send Crawler-request messages to peers they find through BuddyCast (i.e, a pull-based mechanism). A Crawler can request specific statistics or, for instance, request a NAT-type detection (see Sec 10.3.1). A Crawler request is only accepted when it is sent by an authorized Crawler. An authorized Crawler is a crawler that has authenticated itself via the overlay protocol and whose public key appears in the a specific text file present on all clients..

## 11.2. PROTOCOL

The basic Crawler protocol is a simple request-reply protocol. Administrators or researchers wishing to extend it for a particular monitoring tasks by defining an additional message subtype. Currently we have four subtypes:

**CRAWLER\_DATABASE\_QUERY** For free-form queries on the Tribler Megacache.

**CRAWLER\_NATCHECK** For requesting NAT-type detection.

**CRAWLER\_SEEDINGSTATS\_QUERY** For queries about seeding behaviour.

**CRAWLER\_FRIENDSHIP\_STATS** For queries about friendship making behaviour.

The protocol in addition has support for returning large amounts of statistics via a channel mechanism that correlates parts of the same reply, and some protection against too many crawling requests.

#### 11.3. WIRE FORMAT

Crawler messages are sent via the overlay protocol and have message ID 232 (CRAWLER\_REQUEST) and 231 (CRAWLER\_REPLY). They have a common prefix consisting of a fixed number of bytes:

1. 1 byte message subtype

2. 1 byte channel identifier (for returning large replies)

A CRAWLER\_REQUEST in addition has two extra elements following the prefix:

- 3. 2 byte Crawl frequency in network-byte order
- 4. n byte request payload

A CRAWLER\_REQUEST in addition has two extra elements following the prefix:

- 3. 1 byte giving number of parts left
- 4. 1 byte indicating success (0) or failure (non 0)
- 5. n bytes reply payload

#### 11.4. REFERENCES

- [1] BARKER, E., W., B., BURR, W., POLK, W., AND SMID, M. Recommendation for Key Management Part 1: General. Special Publication 800-57, National Institute of Standards and Technology, Gaithersburg, MD, USA, Aug. 2005.
- [2] Breese, J., Heckerman, D., and Kadie, C. Empirical Analysis of Predictive Algorithms for Collaborative Filtering. Tech. Rep. MSR-TR-98-12, Microsoft Research, Redmond, WA, USA, May 1998.
- [3] CERTICOM RESEARCH. Standards for Efficient Cryptography 2: Recommended Elliptic Curve Domain Parameters. Standard SEC2, Certicom Corp., Mississauga, ON, USA, Sept. 2000.
- [4] COHEN, B. Incentives to Build Robustness in BitTorrent. In *Proceedings 1st Workshop on Economics of Peer-to-Peer Systems* (Berkeley, CA, June 2003). http://bittorrent.com/bittorrentecon.pdf.
- [5] COHEN, B. BitTorrent Protocol. http://www.bittorrent.org/protocol.html, Jan. 2006.
- [6] HARRISON, D. Assigned Numbers. BEP 4, BitTorrent Community Forum, www.bittorrent.org, Jan. 2008.
- [7] HARRISON, D. Tracker Returns External IP. BEP 24, BitTorrent Community Forum, www. bittorrent.org, May 2008.
- [8] HAZEL, G., AND NORDBERG, A. Extension for Peers to Send Metadata Files. BEP 9, BitTorrent Community Forum, www.bittorrent.org, Jan. 2008.
- [9] HOFFMAN, J., AND DEHACKED. HTTP Seeding. BEP 17, BitTorrent Community Forum, www.bittorrent.org, Feb. 2008.
- [10] MENEZES, A., VAN OORSCHOT, P., AND VANSTONE, S. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL, USA, 2001.
- [11] MERKLE, R. A Digital Signature Based on a Conventional Encryption Function. In *Proceedings CRYPTO'87* (Santa Barbara, CA, USA, Aug. 1987), C. Pomerance, Ed., no. 293 in Lecture Notes in Computer Science, Springer-Verlag, pp. 369–378.
- [12] MOL, J., POUWELSE, J., MEULPOLDER, M., EPEMA, D., AND SIPS, H. Give-to-Get: Free-riding Resilient Video-on-demand in P2P Systems. In *Proceedings Multimedia Computing and Networking conference (Proceedings of SPIE Vol. 6818)* (San Jose, California, USA, Jan. 2008).
- [13] NORDBERG, A., STRIGEUS, L., AND HAZEL, G. Extension Protocol. BEP 10, BitTorrent Community Forum, www.bittorrent.org, Jan. 2008.
- [14] ROSENBERG, J., MAHY, R., MATTHEWS, P., AND WING, D. RFC 5389: Session Traversal Utilities for NAT (STUN), Oct. 2008.
- [15] SCHNEIER, B. Applied Cryptography, 2nd ed. John Wiley & Sons, New York, NY, USA, 1996.
- [16] VOULGARIS, S., AND VAN STEEN, M. Epidemic-style Management of Semantic Overlays for Content-Based Searching. In *Proceedings 11th International Euro-Par Conference* (Lisbon, Portugal, Aug. 2005), pp. 1143–1152.