Dispersy Documentation

Release 0.0.1

Tribler

Oct 15, 2018

Contents

1 Dispersy **3**
- 1.1 Integrated NAT Puncturing . . . 3
- 1.2 Decentralized . . . 3
- 1.3 Goal . . . 4
- 1.4 Key Features . . . 4
- 1.5 Documentation . . . 4

2 Installation **5**
- 2.1 Installation Methods . . . 5
- 2.2 Installing Dependencies . . . 5
- 2.3 Documentation . . . 7

3 System Overview **9**
- 3.1 Overlay . . . 9
- 3.2 Peer Selection . . . 9
- 3.3 Peer Discovery . . . 10
- 3.4 Bloom filter . . . 15

4 Wire Protocol **17**
- 4.1 Global time . . . 17
- 4.2 Dispersy message types . . . 17

5 Usage **33**
- 5.1 Payload . . . 33
- 5.2 Conversion . . . 34
- 5.3 Community . . . 36
- 5.4 Running Dispersy . . . 42

6 Contributing **45**

7 dispersy **47**
- 7.1 dispersy package . . . 48

Contents:

CHAPTER 1

Dispersy

The **Dis**tributed **Per**mission **Sy**stem, or Dispersy, is a platform to simplify the design of distributed communities. At the heart of Dispersy lies a simple identity and message handling system where each community and each user is uniquely and securely identified using elliptic curve cryptography.

1.1 Integrated NAT Puncturing

Nowadays almost all devices have a network connection, with a lot of them running in challenged network environments. Challenging conditions can be found in a wide range of networks, i.e. Peer-to-Peer networks (P2P), and delay tolerant networks (DTNs). These networks have several limitations, like having long communication delays, very low data rates, and unstable links.

P2P networks are particularly challenging due to nodes not always being online, NAT-firewall constrained Internet connections, and frequent interaction with potentially malicious nodes. Smartphones pose another challenge due to their limited processing capability and battery lifetime.

Dispersy was designed to be capable of running in challenged network environments. It does this by minimizing the needed resources by using optimized algorithms and protocols.

1.2 Decentralized

Dispersy is fully decentralized. It does not require any server infrastructure and can run on systems consisting of a large number of nodes. Each node runs the same algorithm and performs the same tasks. All nodes are equally important, resulting in increased robustness. Dispersy offers distributed system developers both one-to-many and many-to-many data dissemination capabilities. Data is forwarded between nodes. All injected data will eventually reach all nodes, overcoming challenging network conditions.

Dispersy uses elliptic curve cryptography to identify the different nodes in a secure and anonymous way.

1.3 Goal

Dispersy is designed as a building block for implementing fully decentralized versions of, for instance Facebook, Wikipedia, Twitter, or Youtube. These Web 2.0 applications often require on a direct Internet connection to their central servers, but can now be implemented in a distributed way

1.4 Key Features

Key features of Dispersy are:

- stateless synchronization using Bloomfilters
- decentralized NAT traversal
- performance that can scale to over 100,000 bundles

Dispersy is intergated in the BitTorrent client Tribler and show that it is performing very well in various real-time challenged network scenario's (3G and WIFI).

1.5 Documentation

The documentation for this project can be found at ReadTheDocs

CHAPTER 2

Installation

2.1 Installation Methods

To add Dispersy to your program you can use one of the following methods:

2.1.1 Git Submodule

You can add Dispersy to your program by adding it as a submodule of your git repository:

```
git submodule add git@github.com:Tribler/dispersy.git dispersy
```

Then you need to initialize the submodules by running:

```
git submodule update --init --recursive
```

then you can start building your communities.

2.2 Installing Dependencies

- python 2.7 (Dispersy runs only on python 2.7 because of twisted)
- twisted
- netifaces
- M2Crypto
- Libsodium
- PyCrypto

2.2.1 Linux

Apt-get

```
sudo apt-get install python-twisted
sudo apt-get install python-netifaces
sudo apt-get install python-m2crypto
sudo apt-get install libsodium-dev
pip install pycrypto
```

if your system is not Debian >= 8 or Ubuntu >= 15.04 use:

```
sudo add-apt-repository ppa:chris-lea/libsodium
sudo apt-get update && sudo apt-get install libsodium-dev
```

Yum

```
sudo yum install epel-release
pip install twisted
sudo yum install python-netifaces
pip install M2Crypto
sudo yum install libsodium-devel
pip install pycrypto
```

2.2.2 Windows

Check if you have the 32 bits version of python or the 64 bits version. You can use:

```
python -c "import struct;print( 8 * struct.calcsize('P'))"
```

32 bits

```
pip install twisted
pip install netifaces
pip install --egg M2CryptoWin32

Microsoft Visual C++ Compiler for Python 2.7
```

http://aka.ms/vcpython27

> Download the latest msvc version of libsodium from https://download.libsodium.org/libsodium/releases/ Extract libsodium.dll from LIBSODIUM_ROOTx32Releasev140dynamicon your hard-drive and add that directory to your path Test if it works with: python -c “import ctypes; ctypes.cdll.LoadLibrary(‘libsodium’)”
>
> pip install pycrypto

64 bits

```
pip install twisted
pip install netifaces
pip install --egg M2CryptoWin64

Download the latest msvc version of libsodium from https://download.libsodium.org/
↪libsodium/releases/
Extract libsodium.dll from LIBSODIUM_ROOT\x64\Release\v140\dynamic\ on your harddrive␣
↪and add that directory to your path
Test if it works with: python -c "import ctypes; ctypes.cdll.LoadLibrary('libsodium')"

pip install pycrypto
```

2.2.3 Mac

```
pip install twisted
pip install netifaces
pip install M2Crypto
brew install libsodium
pip install pycrypto
```

2.3 Documentation

To compile the documentation on your own you need:

```
pip install sphinx
pip install sphinx-rtd-theme
```

You can read a precompiled version on ReadTheDocs

CHAPTER 3

System Overview

3.1 Overlay

The design of Dispersy is modular and extensible and lets applications define their own overlay types for specific goals. In Dispersy all nodes communicate in one or more overlays which are called communities. An overlay is an additional network built on top of an existing one, most commonly the Internet. By defining a community, an application can determine which messages are going to be used, and which need to be synchronized. The body of a message is application specific, but its headers are added by Dispersy and can contain if and to whom this message needs to be synchronized, the id and or signature of the creator, etc.

After defining a community type, a peer needs to instantiate it. Instantiating a community requires a peer to generate an cryptographic identifier for it. This identifier has to be known to all peers who attempt to join this instance. Dispersy communities are completely separated from each other, through including the identifier of the overlay in the messages. After joining a community, peers synchronize by advertising their locally available messages using a Bloom filter.

3.2 Peer Selection

Dispersy maintains a list of peers for each overlay, this list contains peers which are deemed to be connectable. Initially, the peer list is empty, requiring a peer to initiate the peer discovery algorithm, i.e. bootstrap. For this it uses trackers, which must not be behind a NAT-firewall. Trackers maintain lists of peers for several overlays, and return these upon request. After a peer has populated its peer list with initial peers, it can use those to discover others and selects a random peer to perform a synchronization step with.

In order to create a less predictable, more robust node selection algorithm a Dispersy node will divide his candidate list into three categories:

- Trusted nodes;
- Nodes we have successfully contacted in the past;
- **Nodes who have contacted us in the past; either through**
    - Nodes that have sent an introduction-request; or

  - Nodes that have been introduced.

The node that is chosen for the next synchronization step is based on the probability of the different groups. After the group is chosen, Dispersy chooses the oldest node from that group to make sure the NAT-firewall timeout does not expire.

3.2.1 Trusted Nodes

Trusted nodes only consists of the tracker nodes. The tracker nodes have a probability of 1% to be selected. This is to not overload the tracker nodes, but still keep connecting to them to clear unwanted nodes. Contacting a trusted node will completely reset the candidate list, thereby removing any attacker nodes which could be present.

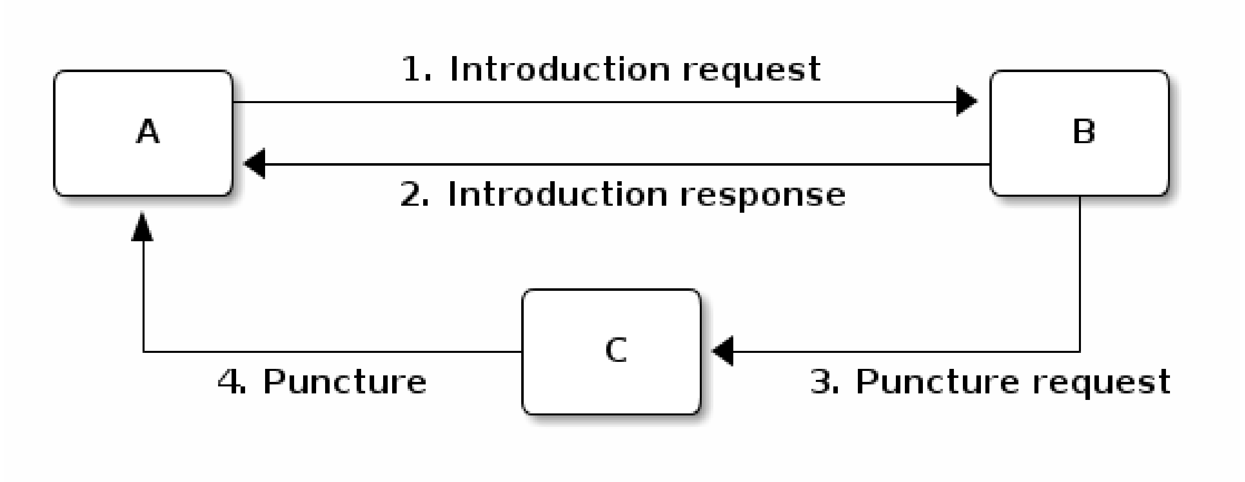
3.2.2 Nodes we have successfully contacted in the past

This group contains nodes that have responded to our introduction-request message. This group has a probability of 49.5% to be selected.

3.2.3 Nodes who have contacted us in the past

This group has two subgroups. One group contains nodes that have sent us introduction-request messages and the other group contains nodes that have been introduced to us by nodes we have successfully connected to in the past. Both of these groups have a probability of 24.75% to be selected. Both these groups contains nodes that might possible be malicious. The probability approach to picking the new node to communicate with is to make sure that even is this group becomes bigger the number of nodes chosen from this group stays the same.

3.3 Peer Discovery



There are four phases in discovering a new peer:

1. peer A chooses a peer B from its neighbourhood and it sends to peer B an introduction-request;
2. peer B chooses a peer C from its neighbourhood and sends peer A an introduction-response containing the address of peer C;
3. peer B sends to peer C a puncture-request containing the address of peer A;

4. peer C sends peer A a puncture message to puncture a hole in its own NAT.

These four phases constitute a step, and multiple steps constitute a walk. By walking, each peer discovers a set of known peers which we define as its neighbourhood.

3.3.1 NAT traversal

We integrate in Dispersy a NAT traversal technique able to puncture 77% of NAT firewalls. Not puncturing these firewalls would prevent up to 64% of peers from receiving any synchronization requests.

3.3.2 IP addresses and member identities

In Dispersy, each peer needs to be able to verify the identities of other peers because we use a right management mechanism. According to our right management mechanism, each peer has different access rights based on the history of rights granted and revoked. To this end, we use public/private key pairs to allow peers to cryptographically identify themselves. The public/private key pair of a peer represents a single Member instance which signs or verifies the messages created by the peer itself or other peers, respectively. Ideally, we want to assign one IP address to each member, and have this mapping be the same for every peer in the system. However, an IP address may change between successive sessions or some peers may assign the same IP address to a Member (i.e. someone behind a symmetric NAT uses different ports for communication with other peers). For those reasons, besides a Member (i.e. the cryptographic key) we use an additional instance, named a Candidate which is a temporary pointer to the current IP address of the corresponding peer. In order to provide a mapping between Members and Candidates, we create for every Candidate a list of Member instances seen at this address. In other words, once having found a Member at a specific IP address, we associate this member with the corresponding Candidate.

3.3.3 Candidate categories

Each candidate holds a list of timestamps to compare to other candidates and place them into different categories.

We define *walk difference*, *stumble difference*, and *intro difference* as follows according to the peer discovery diagram:

- Peer A computes the walk difference of peer B by taking the difference between the current time and receiving an introduction-response to an introduction-request from peer B.
- Peer B computes the stumble difference of peer A by taking the difference between the current time and receiving an introduction-request from peer A.
- Peer A computes the intro difference of peer C by taking the difference between the current time and receiving an introduction-response introducing peer C.

Using the computed time differences a peer assigns all other peers in one of the four categories:

1. **walk** when its walk difference is less or equal to the walk lifetime,
2. **stumble** when it is not a walk-Candidate and its stumble difference is less or equal to the stumble lifetime,
3. **intro** when it is neither a walk or a stumble-Candidate, and its intro difference is less or equal to the intro lifetime, and
4. **none** when it does not fulfil the criteria for its assignment to one of the previously mentioned categories.

We set both the walk lifetime and the stumble lifetime equal to 57.5 seconds because most NAT boxes close a punctured ‘hole’ 60 seconds after receiving the last packet. Moreover, we set the intro lifetime equal to 27.5 seconds because most NAT boxes close a punctured ‘hole’ after 30 seconds when no packets are received through it.

3.3.4 (Un)verified candidates

The Dispersy code provides two main methods to obtain available Candidate instances: the dispersy_yield_candidates method returns an iterator with all walk, stumble, and intro-Candidate instances, in a randomised order. Note that intro-Candidates are unverified, i.e. we have only heard about their existence, but did not actually have any contact with them ourselves. The dispersy_yield_verified_candidates method returns an iterator with all walk and stumble-Candidate instances, in a randomised order. We call these Candidates verified because we have received a message from them at most 57.5 seconds ago (i.e. the walk and stumble lifetime). This means that, unless the peer went offine in the mean time, the peer is still there and the NAT has, most likely, not closed yet. Note that there are NATs that close within 57.5 seconds, those will not be reachable. Because of this, communicating with verified candidates is often better than using unverified candidates.

3.3.5 Candidates we can walk towards

A peer is only allowed to walk towards a Candidate when the Candidate is eligible for a walk namely, it meets the two criteria described below:

1. the category is either walk, stumble, or intro
2. the last time that this peer walked to this specific candidate, occurred at least eligible delay second ago.

We have chosen 27.5 seconds for the eligible delay, with the exception of bootstrap candidates which require a 57.5 seconds of eligible delay. As a result, the bootstrap peers are not contacted to frequently. This feature was initially introduced to reduce the numbers of walks towards trackers in overlays with few peers.

3.3.6 Who to walk to

In phase 1 of the walk, peer A chooses a known peer B from its neighbourhood and sends it an introduction-request. The dispersy_get_walk_candidate method chooses peer B and returns a Candidate instance pointing to it. If there are no available eligible candidates, this method returns None. The choice of a Candidate to walk determines the size of the neighbourhood of peer A. Based on its walks, peer A is able to know at most 11 Candidates because according to our design, a peer takes one step every 5 seconds. As a result in a walk lifetime window of 57.5 seconds, it can take at most 11 steps. Nevertheless, other peers may chose to walk to peer A. Hence, the incoming walks to peer A, that occurred within the stumble lifetime window, increase the size of its neighbourhood accordingly. Assuming that there is at least one eligible Candidate in every category, the selection strategy can be simplified in the following rules. Peer A chooses with probability:

- 49.75% to revisit the oldest eligible walk-Candidate
- 24.825% to visit the oldest eligible stumble-Candidate
- 24.825% to visit the oldest eligible intro-Candidate
- 0.5% to visit a random eligible Candidate from the predefined list of bootstrap candidates

If one category is empty, the probabilities of choosing a peer from this category becomes 0.

Malicious peers can easily pollute our neighbourhood by walking towards a peer from multiple distinct addresses and adding an arbitrary number of stumble-Candidates to its neighbourhood. To avoid such a neighbourhood pollution, we assume that a successfully visited peer is safe. Hence, half of the time we revisit such a peer (i.e. from the walk category) while the remaining 50% is evenly spread between the intro category and the risky stumble category. Method dispersy_get_walk_candidate implements this design.

3.3.7 Who to introduce

In phase 2 of the walk, peer B chooses a known peer C from its neighbourhood and introduces it to peer A. The dispersy_get_introduce_candidate method chooses peer C from the verified available candidates and returns it, or, when no candidates are available, it returns None. Using dispersy_get_introduce_candidate returns a verified candidate in semi round robin fashion. To this end each Community maintains two dynamic iterators _walked_candidates and _stumbled_candidates which iterate over all walk-Candidates and stumble-Candidates in round-robin, respectively.

The selection process of a Candidate then becomes: 1. choose either the walk-Candidate or stumble-Candidate iterator 2. select the next Candidate in the iterator if it is not excluded, otherwise go back to step 1.

3.3.8 Candidate exclusion

Peer B can not introduce peer C to A when:

- C and A are the same Candidate
- C and A are both behind a NAT and they are not within the same LAN

3.3.9 Duplicate candidates

It is possible that peer B introduces an already known peer to peer A. We could have excluded the known peers by having peer A sending a list of known peers that peer B can exclude. However, we decided not to do this because:

1. it would increase the size of the introduction-request
2. it would give peer B information about peer A
3. the larger the overlay, the smaller the chance that peer B will introduce a peer that peer A already knows

3.3.10 LAN and WAN address

In phase 2 of the walk, peer B determines the LAN and WAN address of peer A by using the UDP header (i.e. the sock_addr) of the incoming introduction-request combined with the WAN and LAN address as reported by A. We implement this in method estimate_lan_and_wan_addresses using a simple rule: when peer B sees that the corresponding message originates from its LAN, it decides that peer A's LAN address is the sock_addr. If the message originates outside its LAN, then peer A's WAN address is the sock_addr.

Dispersy determines whether an address originates within its own LAN or not by checking if it corresponds with one of its local interfaces, with regards to its netmask. We do this using the _get_interface_addresses method and the Interface instances that it returns. Peer B uses the result of this estimation to update the lan_address and wan_address properties of the Candidate instance pointing to peer A. These values are also added to the introduction-response, allowing peer A to assess its own WAN address.

3.3.11 WAN address voting

In phase 2 of the walk, peer A receives an introduction-response containing the LAN and WAN address that peer B believes it has. This dial back allows peer A to determine how other peers perceive it, and thereby whether a NAT is affecting its address. When peer A is not affected by a NAT the voting will provide it with its own address. This is useful when peer A and B are both within the same LAN while peer C is not. In this case peer A will send an introduction-request (which includes the WAN address determined by voting) to peer B, peer B will inform peer C of both A's LAN (as determined by the UDP header) and WAN address (as reported by A), allowing peer C to determine that peer A is not within its LAN address, hence it will use peer A's reported WAN address to puncture its own NAT. When a NAT affects peer A the voting will provide information about the type of NAT, i.e. the connection type, that it is behind, as described below. This connection type effects who a peer introduces when receiving an introduction-request. Most of the magic happens in the wan_address_vote method and goes roughly as follows:

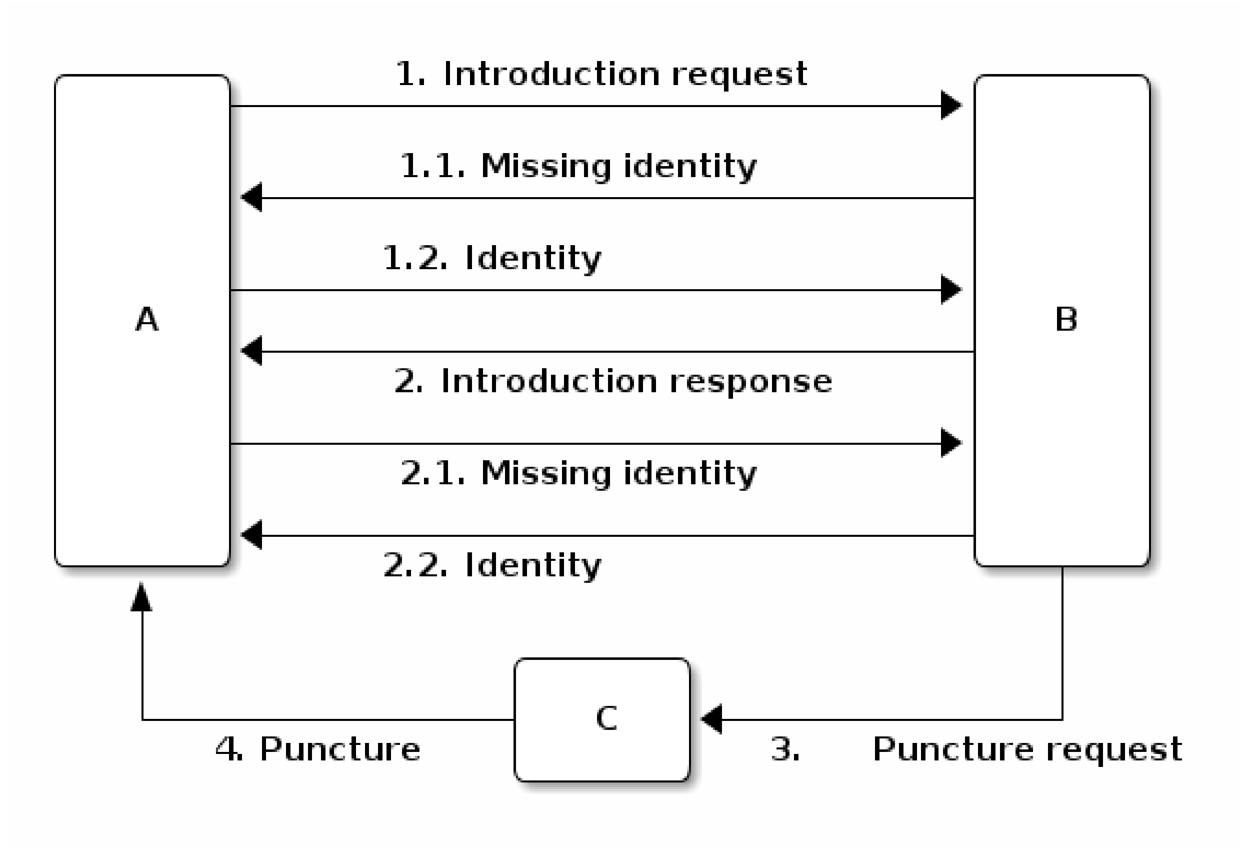
1. remove whatever B voted for before,
2. if the address is valid and B is outside our LAN then add the vote
3. select the new address as our WAN address if it has equal or more votes than our current WAN address. Note that changing our WAN address also makes us re-evaluate our LAN address;
4. **determine our connection type based on the following rules:**
    - **public**, when all votes have been for the same address and our LAN and WAN addresses are the same
    - **symmetric-NAT**, when we have votes for more than one different addresses
    - **unknown**, in all other cases

3.3.12 Cleanup old voting data

To allow for changes in the connectivity, i.e. when running on a roaming machine that changes IP addresses periodically, we must remove older votes, by calling the wan_address_unvote method5, that may no longer apply. Dispersy does this by periodically (every five minutes) checking for obsolete Candidate instances. Where we consider a Candidate to be obsolete when the last walk, stumble, or intro was more than lifetime seconds ago, where lifetime is three minutes. This means that it can take anywhere between five and eight minutes before removing old votes.

3.3.13 Transferring the public key

The messages introduction-request and introduction-response used to discover new nodes do not contain the public key of the sender, we transfer this key using a missing-identity request and a identity message response. Luckily this is only needed for public keys that we do not yet have. So the first time a node connects the interaction between the nodes goes like:



3.4 Bloom filter

In Dispersy, nodes synchronize bundles by advertising their locally available bundles using Bloomfilters. The Bloomfilters contain the bundles which a sending node has received previously, allowing the receiving node to check for missing bundles. Included in the introduction-request messages is a Bloom filter which advertises the locally available messages of a peer. We only include a subset of all messages locally available in the Bloom filter in order to keep the false positive rate low. Moreover, as Dispersy is based on UDP we limit the size of the message which includes the Bloom filter to the maximum transmission unit (MTU) of the link in order to avoid fragmentation. IP fragmentation is caused by a router in the path having a lower MTU than the packet size. A typical MTU size on the Internet is 1500 bytes, for 802.11 it is 2304 bytes.

We define message synchronization as creating consistency in a network of peers by resolving differences in messages received between two peers at a time. Peers regularly request missing messages by advertising their locally available messages using a Bloom filter. A Bloom filter is a compact hash-representation which allows for membership testing. Upon receiving such a request, a peer uses the Bloom filter to test if it has messages available the other has not.

Our synchronization technique consists of five steps:

1. Select a node from our candidate list
2. Select a range of bundles for synchronization
3. Create a Bloomfilter by hashing the selected bundles
4. Send the Bloomfilter to the selected node
5. Pause for a fixed interval and goto step 1

CHAPTER 4

Wire Protocol

This document describes the Dispersy wire protocol version 2 and its intended behaviors. Version 2 is **not** backwards compatible. The most notable changes when compared with the previous version are the use of google protocol buffers for the wire format, protection against IP spoofing, and session usage. Messages can be divided in two categories:

- Temporary message: A control message that is not stored on disk. Messages of this type are immediately discarded after they are processed.
- Persistent message: A message that contains information that must be retained across sessions. Effectively this includes every message that must be disseminated through the network.

4.1 Global time

Global time is a lamport clock used to provide message ordering withing a community. Using global time, every message can be uniquely identified using community, member, and global time.

Dispersy stores global time values using, at most, 64 bits. Therefore there is a finite number of global time values available. To avoid malicious peers from quickly pushing the global time value to the point where none are left, peers will only accept messages with a global time that is within a locally evaluated limit. This limit is set to the median of the neighbors' global time values plus a predefined margin.

Persistent messages that are not within the acceptable global time range are ignored.

4.2 Dispersy message types

4.2.1 Dispersy-message

Protocol Buffers allows messages to be defined, encoded, and finally decoded again. However, the way that we intend to use protocol buffers caused two issues to arise:

- Multiple different messages over the same communication channel requires a method to distinguish message type. The recommended method, as described by Google in self describing messages, is to encapsulate the message by a message that contains all possible messages as optional fields;
- Adding one or more signatures to a message requires the entire message (including the message type) to be serialized and passed to the cryptography layer, resulting signatures can only be placed in a wrapping message.

This wrapping message must store the message in binary. Otherwise changes to protocol buffers' internal implementation may cause one client to produce a different, yet compatible, binary representation. This would make it impossible to verify the signature.

Therefore, the Dispersy protocol will use two wrapping messages:

- **Descriptor** will allow message types to be assigned
- **Message** will contain the raw message bytes and optional signatures.

```
message Message {
   extensions 1024 to max;
   required bytes descriptor;
   repeated bytes signatures;
}
```

Descriptor limitations:

- Every temporary or persistent message must have an optional field in the Descriptor message. Community messages must use the field values assigned to extensions.
- A dispersy-message may only contain one message, i.e. only one optional field may be set.

```
message Descriptor {
   enum Type {
      // frequent temporary messages (uses <15 values)
      INTRODUCTIONREQUEST = 1;
      INTRODUCTIONRESPONSE = 2;
      SESSIONREQUEST = 3;
      SESSIONRESPONSE = 4;
      PUNCTUREREQUEST = 5;
      PUNCTURERESPONSE = 6;
      COLLECTION = 7;
      IDENTITY = 8;

      // infrequent temporary messages (uses >15 values)
      MISSINGIDENTITY = 16;
      MISSINGSEQUENCE = 17;
      MISSINGMESSAGE = 18;
      MISSINGLASTMESSAGE = 19;
      MISSINGPROOF = 20;
      SIGNATUREREQUEST = 21;
      SIGNATURERESPONSE = 22;

      // persistent messages (uses >63 values)
      AUTHORIZE = 64;
      REVOKE = 65;
      UNDOOWN = 66;
      UNDOOTHER = 67;
      DYNAMICSETTINGS = 68;
      DESTROYCOMMUNITY = 69;
   }
   extensions 1024 to max;
```

(continues on next page)

(continued from previous page)

```
   optional IntroductionRequest introduction_request = 1;
   optional IntroductionResponse introduction_response = 2;
   optional SessionRequest session_request = 3;
   optional SessionResponse session_response = 4;
   optional PunctureRequest puncture_request = 5;
   optional PunctureResponse puncture_response = 6;
   optional Collection collection = 7;
   optional Identity identity = 8;

   optional MissingIdentity missing_identity = 16;
   optional MissingSequence missing_sequence = 17;
   optional MissingMessage missing_message = 18;
   optional MissingLastMessage missing_last_message = 19;
   optional MissingProof missing_proof = 20;
   optional SignatureRequest signature_request = 21;
   optional SignatureResponse signature_response = 22;

   optional Authorize authorize = 64;
   optional Revoke revoke = 65;
   optional UndoOwn undo_own = 66;
   optional UndoOther undo_other = 67;
   optional DynamicSettings dynamic_settings = 68;
   optional DestroyCommunity destroy_community = 69;
}
```

Note that field numbers that are higher than 15 are encoded using two bytes, whereas lower field numbers require one byte, see defining a message type . Hence the fields that are most common should use low field numbers.

4.2.2 Dispersy-collection

A temporary message that contains one or more persistent Dispersy messages. It is required because persistent Dispersy messages do not have a session identifier.

Collection limitations:

- Collection.session is associated with the source address.
- Collection.messages contains one or more messages.

```
message Collection {
   extensions 1024 to max;
   required uint32 session = 1;
   repeated Message messages = 2;
}
```

4.2.3 Dispersy-identity

A temporary message that contains the public key for a single member. This message is the response to a *dispersy-missing-identity* request.

Identity limitations:

- Identity.session is associated with the source address.
- Identity.member must be no larger than 1024 bytes.
- Identity.member must be a valid ECC public key.

```
message Identity {
   extensions 1024 to max;
   required uint32 session = 1;
   required bytes member = 2;
}
```

4.2.4 Dispersy-authorize

A persistent message that grants permissions (permit, authorize, revoke, or undo) for one or more messages to one or more public keys. This message must be wrapped in a *dispersy-collection* and is a response to a *dispersy-introduction-request* or a *dispersy-missing-proof*.

Authorize limitations:

- Authorize.version is 1.
- Authorize.community must be 20 bytes.
- Authorize.member must be no larger than 1024 bytes.
- Authorize.member must be a valid EEC public key.
- Authorize.global_time must be one or higher and up to the local acceptable global time range.
- Authorize.sequence_number must follow already processed Authorize messages from Authorize.member. Sequence numbers start at one. No sequence number may be skipped.
- Authorize.targets must contain one or more entries.
- Authorize.targets[].member must be no larger than 1024 bytes.
- Authorize.targets[].member must be a valid EEC public key.
- Authorize.targets[].permissions must contain one or more entries.
- Authorize.targets[].permissions[].message must represent a known message in the community.
- Can not be undone using *dispersy-undo-own* or *dispersy-undo-other*.
- Requires a signature matching the Authorize.member.

```
message Authorize {
   enum Type {
      PERMIT = 1;
      AUTHORIZE = 2;
      REVOKE = 3;
      UNDO = 4;
   }
   message Permission {
      required Message.Type message = 1;
      required Type permission = 2;
   }
   message Target {
      required uint64 global_time = 1;
      required bytes member = 2;
      repeated Permission permissions = 3;
   }
   extensions 1024 to max;
   required uint32 version = 1;
   required bytes community = 2;
```

(continues on next page)

(continued from previous page)

```
   required bytes member = 3;
   required uint64 global_time = 4;
   required uint32 sequence_number = 5;
   repeated Target targets = 6;
}
```

4.2.5 Dispersy-revoke

A persistent message that revokes permissions (permit, authorize, revoke, or undo) for one or more messages from one or more public keys. This message must be wrapped in a *dispersy-collection* and is a response to a *dispersy-introduction-request* or a *dispersy-missing-proof*.

Revoke limitations:

- Revoke.version is 1.
- Revoke.community must be 20 bytes.
- Revoke.member must be no larger than 1024 bytes.
- Revoke.member must be a valid EEC public key.
- Revoke.global_time must be one or higher and up to the local acceptable global time range.
- Revoke.sequence_number must follow already processed Revoke messages from Revoke.member. Sequence numbers start at one. No sequence number may be skipped.
- Revoke.targets must contain one or more entries.
- Revoke.targets[].member must be no larger than 1024 bytes.
- Revoke.targets[].member must be a valid EEC public key.
- Revoke.targets[].permissions must contain one or more entries.
- Revoke.targets[].permissions[].message must represent a known message in the community.
- Can not be undone using *dispersy-undo-own* or *dispersy-undo-other*.
- Requires a signature matching the Revoke.member.

```
message Revoke {
   enum Type {
      PERMIT = 1;
      AUTHORIZE = 2;
      REVOKE = 3;
      UNDO = 4;
   }
   message Permission {
      required Message.Type message = 1;
      required Type permission = 2;
   }
   message Target {
      required uint64 global_time = 1;
      required bytes member = 2;
      repeated Permission permissions = 3;
   }
   extensions 1024 to max;
   required uint32 version = 1;
   required bytes community = 2;
```

(continues on next page)

(continued from previous page)

```
    required bytes member = 3;
    required uint64 global_time = 4;
    required uint32 sequence_number = 5;
    repeated Target targets = 6;
}
```

4.2.6 Dispersy-undo-own

A persistent message that marks an older message with an undone flag. This allows a member to undo her own previously created messages. This message must be wrapped in a *dispersy-collection* and is a response to *dispersy-introduction-request* or a *dispersy-missing-proof*. Undo messages can only be created for messages that allow being undone.

The dispersy-undo-own message contains a target global time which, together with the community identifier and the member identifier, uniquely identifies the message that is being undone. This message target must allow being undone.

To impose a limit on the number of dispersy-undo-own messages that can be created, a dispersy-undo-own message may only be accepted when the message that it points to is available and no dispersy-undo-own has yet been created for it.

UndoOwn limitations:

- UndoOwn.version is 1.
- UndoOwn.community must be 20 bytes.
- UndoOwn.member must be no larger than 1024 bytes.
- UndoOwn.member must be a valid EEC public key.
- UndoOwn.global_time must be one or higher and up to the local acceptable global time range.
- UndoOwn.sequence_number must follow already processed UndoOwn messages from UndoOwn.member. Sequence numbers start at one. No sequence number may be skipped.
- UndoOwn.target_global_time must be one or higher and smaller than UndoOwn.global_time.
- Can not be undone using dispersy-undo-own or *dispersy-undo-other*.
- Requires a signature matching the UndoOwn.member.

```
message UndoOwn {
    extensions 1024 to max;
    required uint32 version = 1;
    required bytes community = 2;
    required bytes member = 3;
    required uint64 global_time = 4;
    required uint32 sequence_number = 5;
    required uint64 target_global_time = 5;
}
```

4.2.7 Dispersy-undo-other

A persistent message that marks an older message with an undone flag. This allows a member to undo a previously created messages created by someone else. This message must be wrapped in a *dispersy-collection* and is a response to *dispersy-introduction-request* or a *dispersy-missing-proof*. Undo messages can only be created for messages that allow being undone.

The dispersy-undo-other message contains a target public key and target global time which, together with the community identifier, uniquely identifies the message that is being undone. This target message must allow being undone.

A dispersy-undo-other message may only be accepted when the message that it points to is available. In contrast to a *dispersy-undo-own* message, it is allowed to have multiple dispersy-undo-other messages targeting the same message. To impose a limit on the number of dispersy-undo-other messages that can be created, a member must have the undo permission for the target message.

UndoOther limitations:

- UndoOther.version is 1.
- UndoOther.community must be 20 bytes.
- UndoOther.member must be no larger than 1024 bytes.
- UndoOther.member must be a valid EEC public key.
- UndoOther.global_time must be one or higher and up to the local acceptable global time range.
- UndoOther.sequence_number must follow already processed UndoOther messages from UndoOther.member. Sequence numbers start at one. No sequence number may be skipped.
- UndoOther.target_global_time must be one or higher and smaller than UndoOther.global_time.
- UndoOther.target_member must be no larger than 1024 bytes.
- UndoOther.target_member must be a valid EEC public key.
- Can not be undone using *dispersy-undo-own* or dispersy-undo-other.
- Requires a signature matching the UndoOther.member.

```
message UndoOther {
   extensions 1024 to max;
   required uint32 version = 1;
   required bytes community = 2;
   required bytes member = 3;
   required uint64 global_time = 4;
   required uint32 sequence_number = 5;
   required uint64 target_global_time = 6;
   required bytes target_member = 7;
}
```

4.2.8 Dispersy-dynamic-settings

A persistent message that changes one or more message policies. When a message has two or more policies of a specific type defined, i.e. both PublicResolution and LinearResolution, the dispersy-dynamic-settings message allows switching between them. This message must be wrapped in a *dispersy-collection* and is a response to a *dispersy-introduction-request* or a *dispersy-missing-proof*.

The policy change is applied from the next global time increment after the global time given by the dispersy-dynamic-settings message.

DynamicSettings limitations:

- DynamicSettings.version is 1.
- DynamicSettings.community must be 20 bytes.
- DynamicSettings.member must be no larger than 1024 bytes.
- DynamicSettings.member must be a valid EEC public key.

- DynamicSettings.global_time must be one or higher and up to the local acceptable global time range.
- DynamicSettings.sequence_number must follow already processed DynamicSettings messages from DynamicSettings.member. Sequence numbers start at one. No sequence number may be skipped.
- DynamicSettings.target_message must represent a known message in the community.
- DynamicSettings.target_policy must be a policy that has dynamic settings enabled.
- DynamicSettings.target_index must be an existing index in the available dynamic settings.
- Can not be undone using *dispersy-undo-own* or *dispersy-undo-other*.
- Requires a signature matching the DynamicSettings.member.

```
message DynamicSettings {
   enum Policy {
      AUTHENTICATION = 1;
      RESOLUTION = 2;
      DISTRIBUTION = 3;
      DESTINATION = 4;
      PAYLOAD = 5;
   }
   extensions 1024 to max;
   required uint32 version = 1;
   required bytes community = 2;
   required bytes member = 3;
   required uint64 global_time = 4;
   required uint32 sequence_number = 5;
   required Message.Type target_message = 6;
   required Policy target_policy = 7;
   required uint32 target_index = 8;
}
```

4.2.9 Dispersy-destroy-community

A persistent message that forces an overlay to go offline. An overlay can be either soft killed or hard killed. This message must be wrapped in a *dispersy-collection* and is a response to *dispersy-introduction-request* (for soft kill) or a response to any temporary message (for hard kill).

A soft killed overlay is frozen. All existing persistent messages with global time lower or equal to DestroyCommunity.target_global_time will be retained but all other persistent messages are undone (where possible) and removed. New persistent messages with global time lower or equal to DestroyCommunity.target_global_time are accepted and processed but all other persistent messages are ignored. Temporary messages are not effected.

A hard killed overlay is destroyed. All persistent messages will be removed without undo, except the dispersy-destroy-community message and the authorize chain that is required to verify its validity. New persistent messages are ignored and temporary messages result in the dispersy-destroy-community and the authorize chain that is required to verify its validity.

A dispersy-destroy-community message can not be undone. Hence it is very important to ensure that only trusted peers have the permission to create this message.

DestroyCommunity limitations:

- DestroyCommunity.version is 1.
- DestroyCommunity.community must be 20 bytes.
- DestroyCommunity.member must be no larger than 1024 bytes.

- DestroyCommunity.member must be a valid EEC public key.
- DestroyCommunity.global_time must be one or higher and up to the local acceptable global time range.
- Can not be undone using *dispersy-undo-own* or *dispersy-undo-other*.
- Requires a signature matching the DestroyCommunity.member.

```
message DestroyCommunity {
   enum Degree {
      SOFT = 1;
      HARD = 2;
   }
   extensions 1024 to max;
   required uint32 version = 1;
   required bytes community = 2;
   required bytes member = 3;
   required uint64 global_time = 4;
   required Degree degree = 5;
}
```

4.2.10 Dispersy-signature-request

A temporary message to request a signature for an included message from another member. The included message may be modified before adding the signature. May respond with a *dispersy-signature-response* message.

SignatureRequest limitations:

- SignatureRequest.session is associated with the source address.
- SignatureRequest.request is a random number.
- SignatureRequest.message.signatures may not be set.

```
message SignatureRequest {
   extensions 1024 to max;
   required uint32 session = 1;
   required uint32 request = 2;
   required Message message = 3;
}
```

4.2.11 Dispersy-signature-response

A temporary message to respond to a *dispersy-signature-request* from another member. The included message may be different from the message given in the associated request.

SignatureResponse limitations:

- SignatureResponse.session is associated with the source address.
- SignatureResponse.request is SignatureRequest.request
- SignatureResponse.message.signatures must contain one signature.

```
message SignatureResponse {
   extensions 1024 to max;
   required uint32 session = 1;
   required uint32 request = 2;
```

(continues on next page)

(continued from previous page)

```
    required Message message = 3;
}
```

4.2.12 Dispersy-introduction-request

A temporary message to contact a peer that we may or may not have visited already. This message has two tasks:

- To maintain a semi-random overlay by obtaining one possibly locally unknown peer.
- To obtain eventual consistency by obtaining zero or more unknown persistent messages.

The dispersy-introduction-request, *dispersy-introduction-response*, *dispersy-session-request*, *dispersy-session-response*, *dispersy-puncture-request*, and *dispersy-puncture* messages are used together. The following schema describes the interaction between peers A, B, and C for a typical walk. Where we call A the initiator, B the invitor and C the invitee.

1. **A $\rightarrow$ B: dispersy-introduction-request**
   - $\{\text{shared}_{\text{AB}}$, $\text{identifier}_{\text{walk}}$, $\text{address}_{\text{B}}$, LAN_{A}, WAN_{A}, $\text{bloom}_{\text{A}}\}$
2. **B $\rightarrow$ A: *dispersy-session-request* (new session only)**
   - $\{\text{random}_{\text{B}}$, $\text{identifier}_{\text{walk}}$, $\text{address}_{\text{A}}$, LAN_{B}, $\text{WAN}_{\text{B}}\}$
3. **A $\rightarrow$ B: *dispersy-session-response* (new session only)**
   - random_{A}, $\text{identifier}_{\text{walk}}\}$
4. **B $\rightarrow$ C: *dispersy-puncture-request***
   - $\{\text{shared}_{\text{BC}}$, $\text{identifier}_{\text{walk}}$, LAN_{A}, $\text{WAN}_{\text{A}}\}$
5. **B $\rightarrow$ A: *dispersy-introduction-response***
   - $\{\text{shared}_{\text{AB}}$, $\text{identifier}_{\text{walk}}$, LAN_{C}, $\text{WAN}_{\text{C}}\}$
6. **B $\rightarrow$ A: *dispersy-collection***
   - $\{\text{shared}_{\text{AB}}$, missing messages$\}$
7. **C $\rightarrow$ A: *dispersy-puncture***
   - $\{\text{shared}_{\text{AC}}$, $\text{identifier}_{\text{walk}}$, LAN_{C}, $\text{WAN}_{\text{C}}\}$

IntroductionRequest limitations:

- IntroductionRequest.session is associated with the source address or zero to initiate a new session.
- IntroductionRequest.community must be 20 bytes.
- IntroductionRequest.global_time must be one or higher and up to the local acceptable global time range.
- IntroductionRequest.random must be a non-zero random value used for PunctureRequest.random and Puncture.random.
- IntroductionRequest.destination is the IPv4 address where the IntroductionRequest is sent.
- IntroductionRequest.source_lan is the senders IPv4 LAN address.
- IntroductionRequest.source_wan is the senders IPv4 WAN address.
- IntroductionRequest.connection_type is the senders connection type. The connection_type is only given when it is known.

- IntroductionRequest.synchronization contains a bloomfilter representation of a subset of the senders known persistent messages. It is only given when the sender wants to obtain new persistent messages.

```
message IntroductionRequest {
   enum ConnectionType {
      public = 1;
      unknown_NAT = 2;
   }
   message Address {
      optional fixed32 ipv4_host = 1;
      optional uint32 ipv4_port = 2;
      optional ConnectionType type = 3;
   }
   message Synchronization {
      required uint64 low = 1 [default = 1];
      required uint64 hight = 2 [default = 1];
      required uint32 modulo = 3 [default = 1];
      required uint64 offset = 4;
      required bytes bloomfilter = 5;
   }
   extensions 1024 to max;
   required uint32 session = 1;
   required uint32 walk = 2;
   required bytes community = 3;
   required uint64 global_time = 4;
   required Address destination = 5;
   repeated Address sources = 6;
   optional Synchronization synchronization = 9;
}
```

4.2.13 Dispersy-introduction-response

A temporary message to introduce a, possibly new, peer to the receiving peer. This message is a response to a *dispersy-introduction-request* (when a session exists) or a *dispersy-session-response* (when a session was negotiated).

SessionResponse limitations:

- SessionResponse.walk is IntroductionRequest.walk.

```
message IntroductionResponse {
   enum ConnectionType {
      public = 1;
      unknown_NAT = 2;
   }
   message Address {
      optional fixed32 ipv4_host = 1;
      optional uint32 ipv4_port = 2;
      optional ConnectionType type = 3;
   }
   extensions 1024 to max;
   required uint32 session = 1;
   required uint32 walk = 4;
   required uint64 global_time = 4;
   repeated Address invitee = 5;
}
```

4.2.14 Dispersy-session-request

A temporary message to negotiate a session identifier. This message is a response to a *dispersy-introduction-request* when the session is zero or unknown.

Negotiating a session identifier will prevent a malicious peer M from spoofing the address of peer A to deliver a *dispersy-introduction-request* to peer B because A will only accept packets from LAN_B or WAN_B containing $random_A$. Where $random_A$ is a random number generated by A. This will prevent DOS attacks through IP spoofing.

SessionRequest limitations:

```
message SessionRequest {
   enum ConnectionType {
      public = 1;
      unknown_NAT = 2;
   }
   message Address {
      optional fixed32 ipv4_host = 1;
      optional uint32 ipv4_port = 2;
      optional ConnectionType type = 3;
   }
   extensions 1024 to max;
   required uint32 version = 1;
   repeated uint32 version_blacklist = 3;
   required uint32 walk = 4;
   required uint32 random_b = 5;
   required Address destination = 5;
   repeated Address source = 6;
}
```

4.2.15 Dispersy-session-response

A temporary message to negotiate a session identifier. This message is a response to a *dispersy-session-request*.

Once this message has been received both sides can compute the session identifier \$session = ($random_A$ + $random_B$) mod 2^{32}. This session identifier is present in all temporary messages, except for *dispersy-session-request* and dispersy-session-response.

SessionResponse limitations:

- SessionResponse.walk is IntroductionRequest.walk.

```
message SessionResponse {
   extensions 1024 to max;
   required uint32 version = 1;
   required uint32 walk = 4;
   required uint32 random_a = 5;
}
```

4.2.16 Dispersy-puncture-request

A temporary message to request the destination peer to puncture a hole in it's NAT. This message is a consequence introducing a two peers after receiving a *dispersy-introduction-request*.

PunctureRequest limitations:

- PunctureRequest.walk is IntroductionRequest.walk.

- PunctureRequest.initiator is one or more addresses corresponding to a single peer. These addresses may be modified to the best of the senders knowledge.

```
message PunctureRequest {
   enum ConnectionType {
      public = 1;
      unknown_NAT = 2;
   }
   message Address {
      optional fixed32 ipv4_host = 1;
      optional uint32 ipv4_port = 2;
      optional ConnectionType type = 3;
   }
   extensions 1024 to max;
   required uint32 session = 1;
   required uint32 walk = 4;
   required uint64 global_time = 4;
   repeated Address initiator = 5;
}
```

4.2.17 Dispersy-puncture

A temporary message to puncture a hole in the senders NAT. This message is the consequence of being introduced to a peer after receiving a *dispersy-puncture-request*.

Puncture limitations:

- Puncture.walk is IntroductionRequest.walk.

```
message PunctureRequest {
   enum ConnectionType {
      public = 1;
      unknown_NAT = 2;
   }
   message Address {
      optional fixed32 ipv4_host = 1;
      optional uint32 ipv4_port = 2;
      optional ConnectionType type = 3;
   }
   extensions 1024 to max;
   required uint32 session = 1;
   required uint32 walk = 4;
   repeated Address source = 5;
}
```

4.2.18 Dispersy-missing-identity

A temporary message to requests the public keys associated to a member identifier. Receiving this request should result in a *dispersy-collection* message containing one or more *dispersy-identity* messages.

DispersyMissingIdentity limitations:

- DispersyMissingIdentity.session must be associated with the source address.
- DispersyMissingIdentity.random must be a non-zero random value used to identify the response *dispersy-collection*.

- DispersyMissingIdentity.member must be no larger than 1024 bytes.
- DispersyMissingIdentity.member must be a valid EEC public key.

```
message DispersyMissingIdentity {
   extensions 1024 to max;
   required uint32 session = 1;
   required uint32 random = 2;
   required bytes member = 3;
}
```

4.2.19 Dispersy-missing-sequence

A temporary message to requests messages in a sequence number range. Receiving this request should result in a *dispersy-collection* message containing one or more messages matching the request.

DispersyMissingSequence limitations:

- DispersyMissingSequence.session must be associated with the source address.
- DispersyMissingSequence.random must be a non-zero random value used to identify the response *dispersy-collection*.
- DispersyMissingSequence.member must be no larger than 1024 bytes.
- DispersyMissingSequence.member must be a valid EEC public key.
- DispersyMissingSequence.descriptor must be the persistent message identifier.
- DispersyMissingSequence.sequence_low must be the first sequence number that is being requested.
- DispersyMissingSequence.sequence_high must be the last sequence number that is being requested.

```
message DispersyMissingSequence {
   extensions 1024 to max;
   required uint32 session = 1;
   required uint32 random = 2;
   required bytes member = 3;
   required Descriptor.Type descriptor = 4;
   required uint32 sequence_low = 5;
   required uint32 sequence_high = 6;
}
```

4.2.20 Dispersy-missing-message

A temporary message to requests one or more messages identified by a community identifier, member identifier, and one or more global times. This request should result in a *dispersy-collection* message containing one or more message messages matching the request.

DispersyMissingMessage limitations:

- DispersyMissingMessage.session must be associated with the source address.
- DispersyMissingMessage.random must be a non-zero random value used to identify the response *dispersy-collection*.
- DispersyMissingMessage.member must be no larger than 1024 bytes.
- DispersyMissingMessage.member must be a valid EEC public key.

- DispersyMissingMessage.global_times must be one or more global_time values.

```
message DispersyMissingMessage {
   extensions 1024 to max;
   required uint32 session = 1;
   required uint32 random = 2;
   required bytes member = 3;
   repeated uint64 global_times = 4;
}
```

4.2.21 Dispersy-missing-last-message

A temporary message to requests one or more most recent messages identified by a community identifier and member. This request should result in a *dispersy-collection* message containing one or more messages matching the request.

DispersyMissingLastMessage limitations:

- DispersyMissingLastMessage.session must be associated with the source address.
- DispersyMissingLastMessage.random must be a non-zero random value used to identify the response *dispersy-collection*.
- DispersyMissingLastMessage.member must be no larger than 1024 bytes.
- DispersyMissingLastMessage.member must be a valid EEC public key.
- DispersyMissingLastMessage.descriptor must be the persistent message identifier.

```
message DispersyMissingLastMessage {
   extensions 1024 to max;
   required uint32 session = 1;
   required uint32 random = 2;
   required bytes member = 3;
   required Descriptor.Type descriptor = 4;
}
```

4.2.22 Dispersy-missing-proof

A temporary message to requests one or more persistent messages from the permission tree that prove that that a given message is allowed. This request should result in a *dispersy-collection* message containing one or more *dispersy-authorize* and/or *dispersy-revoke* messages.

DispersyMissingProof limitations:

- DispersyMissingProof.session must be associated with the source address.
- DispersyMissingProof.random must be a non-zero random value used to identify the response *dispersy-collection*.
- DispersyMissingProof.member must be no larger than 1024 bytes.
- DispersyMissingProof.member must be a valid EEC public key.
- DispersyMissingProof.global_times must be one or more global_time values.

```
message DispersyMissingProof {
   extensions 1024 to max;
   required uint32 session = 1;
```

(continues on next page)

(continued from previous page)

```
    required uint32 random = 2;
    required bytes member = 3;
    repeated uint64 global_times = 4;
}
```

CHAPTER 5

Usage

To start using Dispersy you need to first have the Dispersy library in your project. You can find instructions for that in the installation section. This guide explains how to work with Dispersy and highlights some of the concepts in Dispersy like payloads, conversions and communities. At the end of this guide, you will be able to run your own Dispersy instances and send messages between them.

5.1 Payload

The payload in Dispersy defines the individual messages that get send across the network. This is an example of what a payload can look like:

A payload with a property 'text'

```
from dispersy.payload import Payload

class ExamplePayload(Payload):
    class Implementation(Payload.Implementation):
        def __init__(self, meta, text):
            assert isinstance(text, str)
            super(ExamplePayload.Implementation, self).__init__(meta)
            self._text = text

        @property
        def text(self):
            return self._text
```

In this example we only use a single attribute to store in the payload. You can add more attributes by adding extra arguments to the *__init__* and making a function to access it. Like showed in the following example:

A payload with an extra property 'amount'

```
def __init__(self, meta, text, amount):
    assert isinstance(text, str)
```

(continues on next page)

(continued from previous page)

```
    assert isinstance(amount, int)
    super(ExamplePayload.Implementation, self).__init__(meta)
    self._text = text
    self._amount = amount

...

@property
def amount(self):
    return self._amount
```

In the payload you can do validation and type checking. Type checking was already showed in the previous examples by checking if the attributes are instances of one of the builtin python types. You could also check for the max length for the 'text' attribute or check if the amount is in between two numbers.

An unicode text payload with a maximum length of 255 characters

```
from dispersy.payload import Payload

class TextPayload(Payload):
    class Implementation(Payload.Implementation):
        def __init__(self, meta, text):
            assert isinstance(text, unicode)
            assert len(text.encode("UTF-8")) <= 255
            super(TextPayload.Implementation, self).__init__(meta)
            self._text = text

        @property
        def text(self):
            return self._text
```

When a message is received this text property is available at *message.payload.text*.

5.2 Conversion

The conversion is used to handle the conversion between the Message.Implementation instances used in the code and the binary string representation on the wire. It also allows you to convert between different versions of the community.

Example of a conversion

```
from Tribler.Core.Utilities.encoding import encode, decode
from dispersy.conversion import BinaryConversion
from dispersy.message import DropPacket


class ExampleConversion(BinaryConversion):

    def __init__(self, community):
        super(ExampleConversion, self).__init__(community, "\x01")
        self.define_meta_message(chr(1), community.get_meta_message(u"example"), self.
↪_encode_example, self._decode_example)

    def _encode_example(self, message):
        packet = encode((message.payload.text, message.payload.amount))
        return packet,
```

(continues on next page)

(continued from previous page)

```
    def _decode_example(self, placeholder, offset, data):
        try:
            offset, payload = decode(data, offset)
        except ValueError:
            raise DropPacket("Unable to decode the example-payload")

        if not isinstance(payload, tuple):
            raise DropPacket("Invalid payload type")

        text, amount = payload
        if not isinstance(text, str):
            raise DropPacket("Invalid 'text' type")
        if not isinstance(amount, int):
            raise DropPacket("Invalid 'amount' type")

        return offset, placeholder.meta.payload.implement(text, amount)
```

```
super(MarketConversion, self).__init__(community, "\x01")
```

This line marks the version of the community. The values 'x00' and 'xff' cannot be used, because they are used to indicate the default conversion and for when more than one byte is needed to indicate the version respectively. So you start your conversion with 'x01' and when you need to change something when it is already in use, you need to increase your version number to 'x02'.

```
self.define_meta_message(chr(1), community.get_meta_message(u"example"), self._encode_
↪example, self._decode_example)
```

This line is used to indicate how different payload classes should be converted. For each payload you have you need to add a *define_meta_message* statement. The 'chr(1)' is used to have a small indicator for this payload across the wire. So each different *define_meta_message* has a different indicator (e.g. chr(2)). The *community.get_meta_message(u"example")* gets the metadata for the specific payload implementations. It should use the same name as defined in the community. So in this case the message defined as *example* is retrieved from the community. The third and the fourth parameter are for specifying the encode and decode functions respectively. In this case the functions are called *_encode_example* and *_decode_example*. The two functions have the following arguments:

```
def _encode_example(self, message):

def _decode_example(self, placeholder, offset, data):
```

To make it easier to implement the functions, the following class can be used: Encoding utility class. It provides functions to convert the data to binary. The encode functions accepts a single object or a tuple of objects depending on the number of properties in the payload. So a payload with one property would have a encode function like:

Example of an encode function for one property named 'text'

```
def _encode_example(self, message):
    packet = encode(message.payload.text)
    return packet,
```

A payload with two properties would have an encode function like this:

Example of an encode function for two properties named 'text' and 'amount'

```
def _encode_example(self, message):
    packet = encode((message.payload.text, message.payload.amount))
    return packet,
```

If the payload has more properties then add these to the tuple. The decode functions for the two examples would be:

Example of a decode function for one property named 'text'

```
def _decode_example(self, placeholder, offset, data):
    try:
        offset, payload = decode(data, offset)
    except ValueError:
        raise DropPacket("Unable to decode the example-payload")

    text = payload

    if not isinstance(text, str):
        raise DropPacket("Invalid 'text' type")

    return offset, placeholder.meta.payload.implement(text)
```

Example of a decode function for two properties named 'text' and 'amount'

```
def _decode_example(self, placeholder, offset, data):
    try:
        offset, payload = decode(data, offset)
    except ValueError:
        raise DropPacket("Unable to decode the example-payload")

    if not isinstance(payload, tuple):
        raise DropPacket("Invalid payload type")

    text, amount = payload
    if not isinstance(text, str):
        raise DropPacket("Invalid 'text' type")
    if not isinstance(amount, int):
        raise DropPacket("Invalid 'amount' type")

    return offset, placeholder.meta.payload.implement(text, amount)
```

The same validation is used as in the payload to check for malformed messages and drop the packet if found.

5.3 Community

A community in Dispersy defines the overlay used for the communication within the network.

An example of a community

```
import logging

from .conversion import ExampleConversion
from .payload import ExamplePayload

from dispersy.authentication import MemberAuthentication
from dispersy.community import Community
from dispersy.conversion import DefaultConversion
```

(continues on next page)

(continued from previous page)

```
from dispersy.destination import CommunityDestination
from dispersy.distribution import DirectDistribution
from dispersy.message import Message, DelayMessageByProof
from dispersy.resolution import PublicResolution

logger = logging.getLogger(__name__)


class ExampleCommunity(Community):

    @classmethod
    def get_master_members(cls, dispersy):
        master_key = "<public-key>".decode("HEX")
        master = dispersy.get_member(public_key=master_key)
        return [master]

    def initialize(self):
        super(ExampleCommunity, self).initialize()
        logger.info("Example community initialized")

    def initiate_meta_messages(self):
        return super(ExampleCommunity, self).initiate_meta_messages() + [
            Message(self, u"example",
                    MemberAuthentication(encoding="sha1"),
                    PublicResolution(),
                    DirectDistribution(),
                    CommunityDestination(node_count=10),
                    ExamplePayload(),
                    self.check_message,
                    self.on_example),
        ]

    def initiate_conversions(self):
        return [DefaultConversion(self), ExampleConversion(self)]

    def check_message(self, messages):
        for message in messages:
            allowed, _ = self._timeline.check(message)
            if allowed:
                yield message
            else:
                yield DelayMessageByProof(message)

    def send_example(self, text, amount, store=True, update=True, forward=True):
        logger.debug("sending example")
        meta = self.get_meta_message(u"example")
        message = meta.impl(authentication=(self.my_member,),
                            distribution=(self.claim_global_time(),),
                            payload=(text, amount,))
        self.dispersy.store_update_forward([message], store, update, forward)

    def on_example(self, messages):
        for message in messages:
            logger.debug("received example message")
```

The community consists out of a couple different elements:

5.3.1 Master member

Each community must define a master member. This member is just a normal Dispersy member that is only used to identify the community uniquely across the overlay. To create a master member, a public/private cryptography keypair has to be generated first, which has to be known to all nodes attempting to join.. This can be done with the *createkey.py* tool located under the *tool* package. To use the tool you must first copy it to the base directory of your porject. Using this tool a *curves* argument must be given to create a key to the strength of your liking. The recommended curve to use is *high*:

```
python createkey.py high
```

You can also create multiple keys at once by passing the curve argument multiple times:

```
python createkey.py high low high
```

When the key is generated, the pub 170 bits identifier should be copied and put in place of the <*public-key*> in the following template:

```
master_key = "<public-key>".decode("HEX")
master = dispersy.get_member(public_key=master_key)
```

There are two ways to add the master member to the community. The first one showed here is the preferred way:

First approach: Added as part of the definition of the community

```
@classmethod
def get_master_members(cls, dispersy):
    master_key = "<public-key>".decode("HEX")
    master = dispersy.get_member(public_key=master_key)
    return [master]
```

With this approach the community has to be created in this way:

```
# arguments(<community>, <dispersy_member>, <load: if the community should be loaded>)
dispersy.define_auto_load(ExampleCommunity, my_member, load=True)
```

Second approach: Added when the community is created

```
master_key = "<public-key>".decode("HEX")
master = dispersy.get_member(public_key=master_key)

# arguments(<dispersy>, <master_member>, <dispersy_member>)
community = ExampleCommunity.init_community(dispersy, master, my_member)
dispersy.attach_community(community)
```

The first approach is preferred because is stores the identifier as part of the definition of the community and allows it to be a separate module.

5.3.2 Initialize

The initialize method can be used to perform some tasks right after the community is created. This method is automatically called.

5.3.3 Initiate meta messages

The *initiate_meta_messages* is used to define the different messages that can be send over the overlay.

```
def initiate_meta_messages(self):
    return super(ExampleCommunity, self).initiate_meta_messages() + [
        <messages>
    ]
```

The messages need to be defined between the list brackets and be comma separated. An example of a message is shown below:

```
Message(self, u"example",
        MemberAuthentication(encoding="sha1"),
        PublicResolution(),
        DirectDistribution(),
        CommunityDestination(node_count=10),
        ExamplePayload(),
        self.check_message,
        self.on_example)
```

5.3.4 Messages

Messages are application dependent, however Dispersy adds optional headers describing if and to whom this message needs to be synchronized, the id and or signature of the creator, etc.

A message has the following four different policies (headers), and each policy defines how a specific part of the message should be handled.

- Authentication defines if the message is signed, and if so, by how many members.
- Resolution defines how the permission system should resolve conflicts between messages.
- Distribution defines if the message is send once or if it should be gossiped around. In the latter case, it can also define how many messages should be kept in the network.
- Destination defines to whom the message should be send or gossiped.

To ensure that every node handles a messages in the same way, i.e. has the same policies associated to each message, a message exists in two stages. The meta-message and the implemented-message stage. Each message has one meta-message associated to it and tells us how the message is supposed to be handled. When a message is sent or received an implementation is made from the meta-message that contains information specifically for that message. For example: a meta-message could have the member-authentication-policy that tells us that the message must be signed by a member but only the an implemented-message will have data and this signature.

Authentication

Each Dispersy message that is send has an Authentication policy associated to it. This policy dictates how the message is authenticated, i.e. how the message is associated to the sender or creator of this message.

NoAuthentication

The NoAuthentication policy can be used when a message is not owned, i.e. signed, by anyone.

A message that uses the no-authentication policy does not contain any identity information nor a signature. This makes the message smaller –from a storage and bandwidth point of view– and cheaper –from a CPU point of view– to generate. However, the message becomes less secure as everyone can generate and modify it as they please. This makes this policy ill suited for gossiping purposes.

MemberAuthentication

The MemberAuthentication policy can be used when a message is owned, i.e. signed, by one member.

A message that uses the member-authentication policy will add an identifier to the message that indicates the creator of the message. This identifier can be either the public key or the sha1 digest of the public key. The former is relatively large but uniquely identifies the member, while the latter is relatively small but might not uniquely identify the member, although, this will uniquely identify the member when combined with the signature.

Furthermore, a signature over the entire message is appended to ensure that no one else can modify the message or impersonate the creator. Using the default curve, NID-sect233k1, each signature will be 58 bytes long.

The member-authentication policy is used to sign a message, associating it to a specific member. This lies at the foundation of Dispersy where specific members are permitted specific actions. Furthermore, permissions can only be obtained by having another member, who is allowed to do so, give you this permission in the form of a signed message.

DoubleMemberAuthentication

A message that uses the double-member-authentication policy is signed by two member. Similar to the member-authentication policy the message contains two identifiers where the first indicates the creator and the second indicates the members that added her signature.

Dispersy is responsible for obtaining the signatures of the different members and handles this using the messages dispersy-signature-request and dispersy-signature-response, defined below. Creating a double signed message is performed using the following steps: first Alice creates a message (M) where M uses the double-member-authentication policy. At this point M consists of the community identifier, the conversion identifier, the message identifier, the member identifier for both Alice and Bob, optional resolution information, optional distribution information, optional destination information, the message payload, and 0 bytes for the two signatures.

Message M is then wrapped inside a dispersy-signature-request message (R) and send to Bob. When Bob receives this request he can optionally apply changes to M2 and add his signature. Assuming that he does the new message M2, which now includes Bob's signature while Alice's is still 0, is wrapped in a dispersy-signature-response message (E) and sent back to Alice. If Alice agrees with the (possible) changes in M2 she can add her own signature and M2 is stored, updated, and forwarded to other nodes in the community.

Resolution

Resolution is used for determining who can create the message. This is part of the permission system in Dispersy. There are three types of resolutions:

PublicResolution

Public resolution allows any member to create a message. This is the most common type used.

LinearResolution

Linear resolution allows only members that have a specific permission to create a message. This resolution type checks the public identifier against the permission list to see if that user is allowed to create that message.

DynamicResolution

Dynamic resolution allows the resolution policy to change. A special dispersy-dynamic-settings message needs to be created and distributed to change the resolution policy. Currently the policy can dynamically switch between either PublicResolution and LinearResolution.

Distribution

Distibution determines how a message gets distributed across the network. There are five types of distibutions packaged in Dispersy:

SyncDistribution

Sync distribution allows gossiping and synchronization of messages throughout the community.

The PRIORITY value ranges [0:255] where the 0 is the lowest priority and 255 the highest. Any messages that have a priority below 32 will not be synced. These messages require a mechanism to request missing messages whenever they are needed.

The PRIORITY was introduced when we found that the dispersy-identity messages are the majority of gossiped messages while very few are actually required. The dispersy-missing-identity message is used to retrieve an identity whenever it is needed.

FullSyncDistibution

Full-sync distribution allows gossiping and synchronization of messages throughout the community.

Sequence numbers can be enabled or disabled per meta-message. When disabled the sequence number is always zero. When enabled the claim_sequence_number method can be called to obtain the next sequence number in sequence.

Currently there is one situation where disabling sequence numbers is required. This is when the message will be signed by multiple members. In this case the sequence number is claimed but may not be used (if the other members refuse to add their signature). This causes a missing sequence message. This in turn could be solved by creating a placeholder message, however, this is not currently, and may never be, implemented.

LastSyncDistribution

Last-sync distribution does the same as SyncDistribution but only for the last n messages. This number is determined by a input parameter.

DirectDistribution

Direct distibution is used to send a message to a node directly, without syncing the information. The information is processed and then thrown away.

RelayDistribution

Relay distribution does the same as DirectDistribution

Destination

The destination determines where or who the message is going to. There are two types of destination policies:

CandidateDestination

A destination policy where the message is sent to one or more specified candidates.

CommunityDestination

A destination policy where the message is sent to one or more community members selected from the current candidate list.

At the time of sending at most NODE_COUNT addresses are obtained using community.yield_random_candidates(. . .) to receive the message.

5.4 Running Dispersy

Dispersy uses Twisted for all low level network communications. It is not recommended to run twisted on a separate thread. A Dispersy based program should be async and use twisted, even better if it's a twisted plugin. That saves having to take care of the reactor lifetime, log rotation, pid file and suchlike.

5.4.1 Run Twisted in the main thread

Dispersy uses the Twisted reactor, which is an event driven networking framework. In the main function the function that starts Dispersy is passed unto the reactor before start is called.

A LoopingCall has been included to send a message every 1 second to members of the community with a timestamp. If you run this code on two seperate instances (if you use the same computer make sure to change the port and database name!) you will be able to see the messages if you add a print statement in the *ExampleCommunity.on_example* method. Don't forget to change the port and the public key of the master member in the example below. The variables between <> have to be replaced with values/objects belonging to your own project.

```
from twisted.internet import reactor
from twisted.internet.task import LoopingCall
import time

def start_dispersy():
    dispersy = Dispersy(StandaloneEndpoint(<port>, '0.0.0.0'), unicode('.'), u
↪'dispersy.db')
    dispersy.statistics.enable_debug_statistics(True)
    dispersy.start(autoload_discovery=True)

    my_member = dispersy.get_new_member()
    master_member = dispersy.get_member(public_key=<master_key>)

    community = ExampleCommunity.init_community(dispersy, master_member, my_member)

    LoopingCall(lambda:community.send_example("Time sent", int(time.time()))).start(1.
↪0)
```

(continues on next page)

(continued from previous page)

```

def main():
    reactor.callWhenRunning(start_dispersy)
    reactor.run()

if __name__ == "__main__":
    main()
```

CHAPTER 6

Contributing

CHAPTER 7

dispersy

7.1 dispersy package

7.1.1 Subpackages

dispersy.discovery package

Submodules

dispersy.discovery.bootstrap module

dispersy.discovery.community module

dispersy.discovery.conversion module

dispersy.discovery.payload module

Module contents

dispersy.tool package

Submodules

dispersy.tool.createkey module

dispersy.tool.lencoder module

dispersy.tool.main module

Module contents

dispersy.tracker package

Submodules