Delft University of Technology
Parallel and Distributed Systems Report Series

Dispersy Peer Discovery and NAT traversal

Boudewijn Schoon and Johan Pouwelse
dispersy@frayja.com

Completed 2013.

Report number PDS-2013-009

ISSN 1387-2109

Published and produced by:
Parallel and Distributed Systems Group
Department of Software and Computer Technology
Faculty Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

Information about Parallel and Distributed Systems Report Series:
`reports@pds.ewi.tudelft.nl`

Information about Parallel and Distributed Systems Section:
`http://www.pds.ewi.tudelft.nl/`

Abstract

In this technical report we present the peer discovery and NAT traversal strategies of Dispersy. Dispersy is a project that allows peer-to-peer engineers to design and deploy next generation self-organising socially intelligent information systems. The project aims to combine three recent trends within information systems: social networks, peer production, and peer-to-peer systems. Dispersy is expected to perform in a challenged environment and hence must deal with firewalls, Network Address Translation (NAT), churn.

We designed Dispersy in such a way that its core mechanisms, being peer discovery, message dissemination, and rights management, can be changed to fit what is needed. This technical report covers the first of these core mechanisms, explaining how the default Dispersy implementation handles peer discovery and where it needs to be modified to achieve different behaviour.

Contents

List of Figures

List of Tables

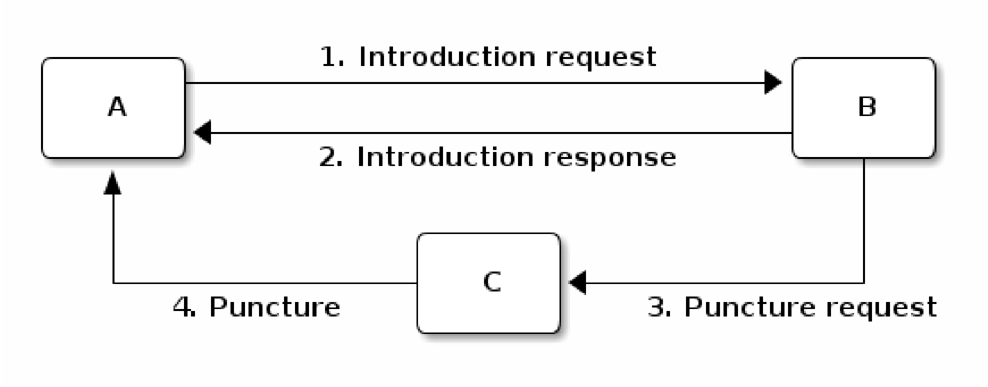

Figure 1: Peer discovery mechanism

1 Concept

This technical report is part of a series describing Dispersy, a library designed to maintain a distributed overlay with peer discovery, message synchronisation, and rights management. Currently, Dispersy is the peer discovery mechanism of our BiTorrent-based client, Tribler[1]. This document describes the peer discovery process using the *Dispersy walker*.

We designed Dispersy walker taking into consideration the challenges for peer discovery in P2P systems which are:

- approximately 64% of computers are behind a NAT[2],
- distributed systems allow malicious DDoS attacks[1],
- high churn rate,
- requirement of little or no state, and
- limited bandwidth resources.

Our peer discovery mechanism illustrated in Figure 1 consists of four phases:

phase 1: peer A chooses a peer B from its neighbourhood and it sends to peer B an introduction-request;

phase 2: peer B chooses a peer C from its neighbourhood and sends peer A an introduction-response containing the address of peer C;

phase 3: peer B sends to peer C a puncture-request containing the address of peer A;

phase 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*.

The remainder of this document explains the walker mechanism in detail and gives the reasoning behind our design choices. Section 2 explains how we handle addresses and identities using candidates, Section 3 explains how peer A chooses another peer from its neighbourhood,

[1] http://events.ccc.de/congress/2010/Fahrplan/events/4210.en.html

Section 4 explains how peer B chooses another peer from its neighbourhood, Section 5 explains how peer B determines peer A's LAN and WAN address, Section 6 explains how peer A determines its own WAN address, Section 7 explains how the walker tries to follow the 5 second rule, Section 8 explains public key exchange, and finally, Section 9 explains what debug output to expect.

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. We note that this mapping is temporary,

2.1 Candidate categories

Each Candidate maintains time stamps for important events taking place during a walk, i.e. the last time of an introduction-request. Using these time stamps, we assign Candidates into four categories: *walk*, *stumble*, *intro*, and *none*. These categories determine how we can use Candidates during a walk. Each Candidate is assigned only to one category, and this category may change over time.

Below, we describe the four categories and the assignment process using the example of peers A, B, and C from our illustration in Figure 1. When communicating peers update several time stamps that are used to compute how long ago an event occurred.

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:

walk when its walk difference is less or equal to the *walk lifetime*,

stumble when it is *not* a walk-Candidate and its stuble difference is less or equal to the *stumble lifetime*,

intro when it is neither a walk or a stumble-Candidate, and its intro difference is less or equal to the *intro lifetime*, and

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[1].

2.2 (Un)verified candidates

The Dispersy code provides two main methods to obtain available Candidate instances: the dispersy_yield_candidates method[2] 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[3] 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 offline 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[1], those will not be reachable.

Because of this, communicating with verified candidates is often better than using unverified candidates.

2.3 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 Who to walk to

In **phase 1** of the walk (as illustrated in Figure 1), peer A chooses a known peer B from its neighbourhood and sends it an introduction-request. The dispersy_get_walk_candidate method[3] 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

[2]Implemented in the Community class, see dispersy/community.py

peer takes one step every 5 seconds (see Section 7). 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, and
- 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. In Table 1, we present the probability of choosing categories when some of these categories are empty. The first column *has-WSIB* shows in binary form if there is at least one walk, stumble, intro, or bootstrap candidate available by setting the corresponding bit equal to 1. For example, 1000 means that the only available candidates are walk candidates.

Table 1: Chance to select a category based depending on which categories has eligible candidates.

has-WSIB	walk	stumble	intro	boot	none
0000					100%
0001				100%	
0010			100%		
0011			99.5%	0.5%	
0100		100%			
0101		99.5%		0.5%	
0110		50%	50%		
0111		49.75%	49.75%	0.5%	
1000	100%				
1001	99.5%			0.5%	
1010	50%		50%		
1011	49.75%		49.75%	0.5%	
1100	50%	50%			
1101	49.75%	49.75%		0.5%	
1111	49.75%	24.825%	24.825%	0.5%	

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.

3.1 Dissemination experiments

During experiments that want to focus on dissemination speed, it is possible to only visit bootstrap-Candidates during the bootstrap process. Otherwise there is a 0.5% chance each step to visit a bootstrap peer and not get any new data (since the bootstrap peers do not participate in data dissemination).

Approximately 450 bootstrap peers[3] will be unnecessarily visited in a 15 minute experiment where 500 peers disseminate data. When this is undesirable, perhaps because you do not want to explain why certain steps do not yield any new data, the file minimal_bootstrap.diff will remove the 0.5% chance to visit a bootstrap peer. This will result in the combinations shown in Table 2.

Table 2: Suggested chance to select a category based depending on which categories has eligible candidates.

has-WSIB	walk	stumble	intro	boot	none
0000					100%
0001				100%	
0010			100%		
0011			100%		
0100		100%			
0101		100%			
0110		50%	50%		
0111		50%	50%		
1000	100%				
1001	100%				
1010	50%		50%		
1011	50%		50%		
1100	50%	50%			
1101	50%	50%			
1111	50%	25%	25%		

4 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[3] 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[3] _walked_candidates and _stumbled_candidates which iterate over all walk-Candidates and stumble-Candidates in round-robin, respectively.

In Figure 2, we present the selection process of a Candidate. In most cases, this process is simplified in the following steps:

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.

4.1 Candidate exclusion

Peer B can not introduce peer C to A when:

[3]one peer takes 12 steps per minute, 500 peers take 90,000 steps in 15 minutes, 0.5% will be towards bootstrap peers, i.e. 450 steps.

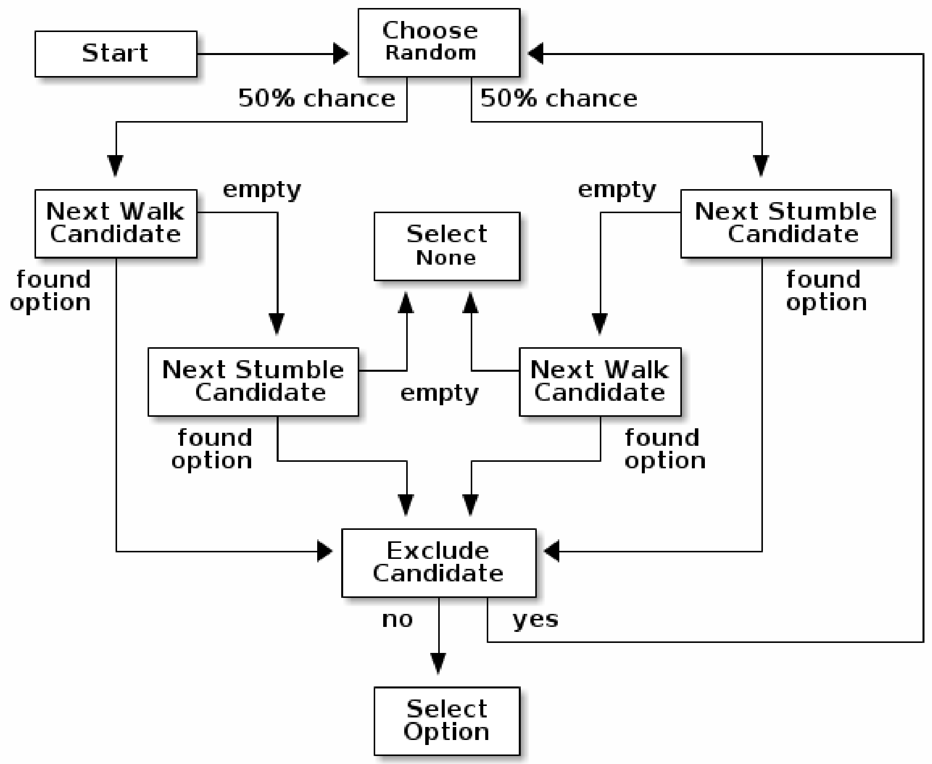

Figure 2: Introduction peer selection mechanism

- C and A are the same Candidate,
- C and A are both behind a NAT and they are not within the same LAN,
- C is behind a tunnel while A is not behind a tunnel. Peer C is behind a tunnel when all messages it sends have a `FFFFFFFF` prefix. In this case, it can only receive messages with this prefix. Tunnelling has been introduced to Dispersy at the end of 2012 so that traffic is send through libswift. Dispersy recognises the `FFFFFFFF` prefix without using libswift. However, older Dispersy clients cannot recognise this prefix and they see it as a part of the message. Since older and newer versions of Dispersy are not distinguishable, we are currently considering all peers to use an older version of Dispersy.

4.2 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.

5 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, as it is illustrated by Figure 3.

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 As LAN address is the sock_addr. If the message originates outside its LAN, then peer As WAN address is the sock_addr[fn::The word 'estimate' is for historical reasons since this code was not able to make this decision as cleanly as is described here.].

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[4] and the Interface instances that it returns.

Peer B uses the result of this estimation to update the lan_address and wan_address properties[5] 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, as discusses in Section 6.

6 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.

[4] Implemented in the Dispersy class, see dispersy/dispersy.py
[5] Implemented in the WalkCandidate class, see dispersy/candidate.py

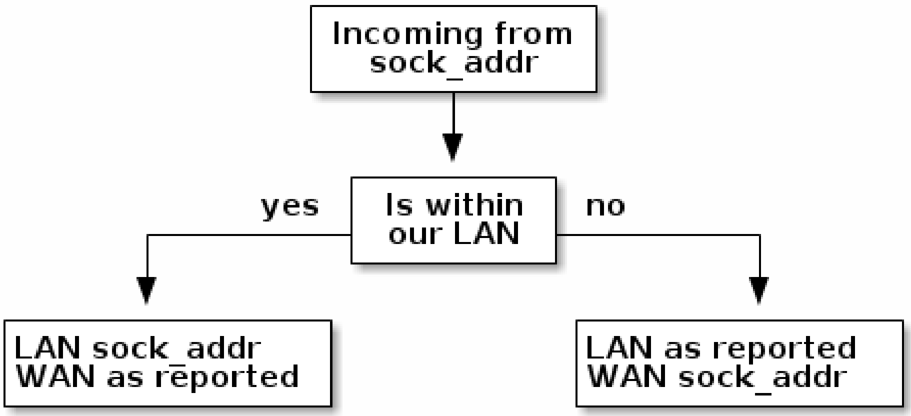

Figure 3: Determine the LAN or WAN address of a peer

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 As 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 As 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, see Section 4.

Most of the magic happens in the wan_address_vote method[5] 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, and

   unknown in all other cases.

6.1 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 method[5], 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.

7 The 5 second rule

When we decided on the design of the walker we took into account the following factors:

1. a significant number of NAT devices close a port 60 seconds after receiving the last packet though it[6], and
2. taking a step involves performing the bloom filter synchronisation. Synchronisation is not described in this report.

Obviously when we take more steps the neighbourhood will contain more walk and intro-Candidates (and since other peers also take more steps the neighbourhood will also, on average, contain more stumble-Candidates). This would advocate taking as many steps as possible.

However, every step also has a cost associated to it, the majority being in the bloom filter synchronisation. At the time we wanted every step to perform a synchronisation, and given that some peers might receive multiple incoming steps around the same time, we decided on a reserved value of 5 seconds. We expect this to be sufficient to perform one synchronisation for ourselves and, in the worst case, multiple incoming synchronisations.

Nowadays we have introduced mechanisms to reduce the workload by not always performing a bloom filter synchronisation, hence the 5 second rule is not strictly necessary anymore, however, the code contains constants derived from 5 seconds, making it difficult to change (see 7.3).

7.1 Walking in a single overlay

In the worst case, creating a bloom filter is one of the most CPU intensive parts of Dispersy. Below, we present an example of a naive approach where we simply schedule 5 seconds between each step. For simplicity, we will assume that it takes 1 second to create a bloom filter.

The schematic below shows a time line with + every 5 seconds when a step should take place. It shows that creating the bloom filter is causing walker X to take a step once every 6 seconds instead of every 5 seconds. Furthermore, a large delay caused by task T increases the gap between steps even further, resulting in only 7 steps instead of 10 which is the expected number of steps.

```
                delayed steps
+----+----+----+----+----+----+----+----+----+----+  (time line)
            TT            TT            TT        (task T)
X     X       X     X       X     X       X       (steps overlay X)
```

7.2 Walking in multiple overlays

The previous naive approach causes the gap between walks to be larger than the intended 5 seconds, this in turn results in fewer walks, hence slower data dissemination and fewer available candidates. The gap between walks will only get larger when we need to maintain multiple

[6] Do not confuse with NAT devices closing a port 30 seconds after puncturing it without receiving any packets through it

overlays at the same time. In this case the naive approach would result in both overlays X and Y walking immediately after one another, causing a spike in CPU traffic, as seen in the schematic below.

```
     delayed steps with multiple overlays
+----+----+----+----+----+----+----+----+----+  (time line)
            TT            TT            TT     (task T)
X     X       X     X       X     X       X    (steps overlay X)
   Y     Y     Y     Y       Y     Y       Y   (steps overlay Y)
```

We address both of these problems by what we call a *self healing walker*, implemented in the _candidate_walker method[5]. This walker takes into account both the number of overlays and the time between walks in individual overlays. The self healing walker has two major features:

- predicting the time when the next walk should occur to remove the delays the naive approach would introduce
- allowing more than one step in a single overlay within 5 seconds, as seen in the schematic below where the lowercase x and y are within 5 seconds of the previous step taken in its overlay.

```
              self healing walker
+----+----+----+----+----+----+----+----+----+  (time line)
          TT             TT             TT     (task T)
X    X      X  x    X      X  x    X      X  x  (steps overlay X)
   Y     Y    Y   y    Y     Y   y    Y     Y   (steps overlay Y)
```

To preserve resources, Dispersy will tell a community not to perform a bloom filter synchronisation (while still performing the walk to maintain the neighbourhood) when the previous step was less than 4.5 seconds ago. Since this will usually occur under heavy CPU load, the benefit is that it will reduce the load since synchronisation is the most expensive part of taking a step.

When we detect that the previous walk in an overlay was more than 5 seconds ago, a *walk reset* will occur to ensure we do not walk to often. This is especially useful when a computer running Dispersy goes into sleep mode, when it wakes up the walk may be hours behind, the walk reset will ensure that Dispersy doesnt try to catch up with the sleeping time by taking thousands of steps.

7.3 Walk multiplier

Sometimes it can be useful to change the 5 seconds delay between steps into something else. The problem is that all derived values must be appropriately changed. The best way to do this is to multiply all these values with the same constant.

The file walk_multiplier.diff will modify all these constants (as known at October 2013). Changing the `WALK_MULTIPLIER` constant[7] to 2 will result in a step every 10.0 seconds, i.e. slowing down the walker. Conversely, changing the constant to 0.5 will result in a step every 2.5 seconds, i.e. speeding up the walker.

[7] Implemented in the WalkCandidate class, see dispersy/candidate.py

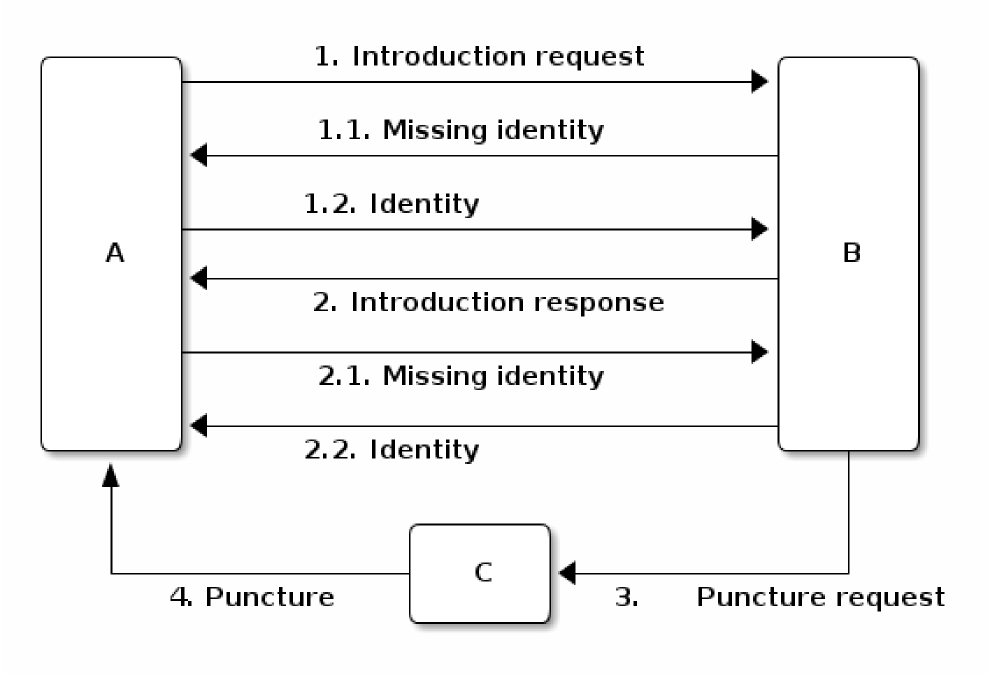

Figure 4: Peer dispersy mechanism between unknown peers

8 Transferring the public key

The signed walker messages introduction-request and introduction-response used in Section 1 do not contain the public key of the signer, 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, hence the first time that we encounter a peer the walk actually follows Figure 4.

9 Debug output

Dispersy uses the standard Python logger to output different message levels, i.e. DEBUG, INFO, WARNING, and ERROR. When enabling DEBUG messages the logger in dispersy/endpoint.py will log all incoming and outgoing packets, including their name when possible. This can give valuable information when something does not follow the expected behavior.

9.1 Bootstrapping

To bootstrap an overlay, each node contacts one of the bootstrap servers. If nodes have never encountered this bootstrap server before, they need to exchange public keys. This results in the following DEBUG output:

```
dispersy-introduction-request -> 130.161.211.245:6422   132 bytes
    dispersy-missing-identity <- 130.161.211.245:6422    51 bytes
            dispersy-identity -> 130.161.211.245:6422   177 bytes
```

```
dispersy-introduction-response <- 130.161.211.245:6422   126 bytes
     dispersy-missing-identity -> 130.161.211.245:6422    51 bytes
             dispersy-identity <- 130.161.211.245:6422   141 bytes
```

9.2 Building a neighbourhood

After have walked for some steps, each node builds its neighbourhood. Below we see that we contact someone at 74.96.92.***:7759. Nodes no longer need to exchange public keys, but only the incoming puncture message from 84.209.251.***:7759 is from someone not yet encountered, hence we exchange identities immediately.

```
 dispersy-introduction-request ->    74.96.92.***:7759   132 bytes
dispersy-introduction-response <-    74.96.92.***:7759   144 bytes
             dispersy-puncture <-  84.209.251.***:7759   125 bytes
     dispersy-missing-identity ->  84.209.251.***:7759    51 bytes
             dispersy-identity <-  84.209.251.***:7759   177 bytes
```

9.3 Candidate statistics

Dispersy provides a logger with the name dispersy-stats-detailed-candidates. When enabling DEBUG level messages for this logger, it will output a summary of its neighbourhood every five seconds. The example below is the summary as seen shortly after contacting 74.96.92.***:7759, see below:

```
--- 8164f55c2f828738fa779570e4605a81fec95c9d Community ---
  4.7s  E  intro   unknown       {192.168.1.35:7759 84.209.251.***:7759}
  9.7s  E  intro   unknown       {192.168.25.100:7759 177.157.54.***:7759}
 14.8s  E  intro   unknown       {192.168.0.3:34728 188.242.194.***:34728}
 19.9s  E  intro   unknown       {192.168.3.101:7759 67.33.160.***:7759}
 24.4s  E  intro   unknown       {192.168.178.21:7759 188.154.8.***:7759}
  5.0s     walk    unknown       {192.168.1.18:7759 74.96.92.***:7759}
 10.0s     walk    unknown       {192.168.0.100:7761 84.251.49.***:7761}
 15.0s     walk    symmetric-NAT {178.164.145.6:7759 94.21.97.***:7759}
 20.0s     walk    unknown       {192.168.1.27:7759 87.18.61.***:16409}
 25.0s     walk    symmetric-NAT {90.165.123.***:7759}
 30.0s  E  walk    unknown       {192.168.1.172:7759 76.115.137.***:7759}
 35.0s  E  walk    unknown       {192.168.2.3:7759 97.91.131.***:7759}
 45.0s  E  walk    unknown       {192.168.1.51:7749 109.208.189.***:7749}
 50.0s  E  walk    unknown       {192.168.0.3:7759 180.145.124.***:7759}
 55.0s  E  walk    unknown       {192.168.0.2:7759 83.153.18.***:7759}
```

The summary shows that the Candidate at 74.96.92.***:7759 is currently a walk-Candidate with age 5.0 seconds, i.e. we sent the introduction-request 5.0 seconds ago.

Furthermore, there is an intro-Candidate at 84.209.251.***:7759, which is the introduced Candidate from when we received a response to this walk 4.7 seconds ago. Note that this Candidate has the character E which signifies that this Candidate is eligible for a walk.

References

[1] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. Epema, M. Reinders, M. R. Van Steen, and H. J. Sips, "Tribler: a social-based peer-to-peer system," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 2, pp. 127–138, 2008. 4

[2] G. Halkes and J. Pouwelse, "Udp nat and firewall puncturing in the wild," in *NETWORKING 2011*, pp. 1–12, Springer, 2011. 4