Building a neighbourhood with Dispersy

Boudewijn Schoon

October 30, 2013

Contents

1 Concept

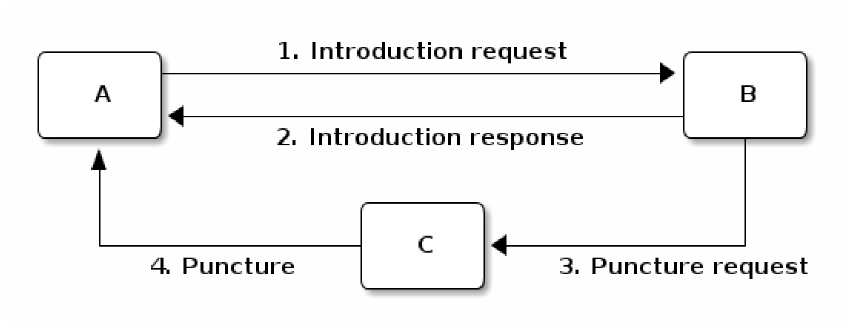
This technical report is part of a series that covers Dispersy, a library designed for Tribler with the purpose of maintaining a distributed overlay with peer discovery, message synchronisation, and rights management. This document focuses on peer discovery using what we call the *Dispersy walker*.

We designed the walker with the following problems in mind:

- approximately 64% of computers are behind a NAT[1];
- distributed systems allow malicious DDoS attacks[2].

Furthermore, we wanted the following features:

- resilient to churn;
- require little or no state;
- small bandwidth overhead.



This resulted in peer discovery as shown in the figure above, with the following phases take place:

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

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

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

[1] `http://pds.twi.tudelft.nl/reports/2010/PDS-2010-007.pdf`

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

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

We call these four phases *taking a step*, and when we take multiple steps we call this *walking*. Walking is the Dispersy strategy of peer discovery and gives each peer a set of known peers which we call its *neighbourhood*.

The remainder of this document will explain the walker design in detail and gives the reasoning behind design choices.

- Section 2 explains how we handle addresses and identities using candidates;
- Section `Who to walk towards` explains how peer A chooses from its neighbourhood;
- Section 4 explains how peer B chooses 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 trues to follow the 5 second rule;
- Section 8 explains public key exchange;
- And finally Section 9 explains what to debug output to expect.

2 IP addresses and member identities

We designed Dispersy with rights management in mind, therefore peers need to be able to verify their identity. To this end we use public/private key pairs to allow peers to cryptographically identify themselves. A key pair represents a single Member instance which we use to sign our messages and verify messages created by other peers.

Ideally we want assign one IP address to each member, and have this mapping be the same for every peer in the system. However, we can not assume that this IP address will remain the same (i.e. IP addresses may change between sessions) or even that every peer will assign the same IP address to a member (i.e. someone behind a symmetric NAT will use different ports for communication with other peers).

Because of this we distinct between Members and Candidates. Where a member (i.e. a cryptographic key) uniquely identifies each peer throughout the system and a candidate is temporary pointer to an IP address.

Once find a Member at a specific address we add this member to the associated Candidate, this gives every Candidate a list of Member instances that we have seen at this address. Providing a temporary mapping between Members and Candidates.

2.1 Candidate categories

A Candidate maintains time stamps for events that take place during walking, i.e. last time we sent an introduction-request. Using these time stamps we assign a candidate with a category, which determines how we can use it while walking.

Candidate is always assigned one of the following four categories, although this changes over time. The following is from peer A's point of view:

walk: peer B is a walk-Candidate when peer A receives an introduction-response from peer B no more than *walk lifetime* seconds ago (note that peer A will only accept an introduction-response from peer B shortly after sending an introduction-request to peer B);

stumble: peer B is a stumble-Candidate when it is *not* a walk-Candidate and peer A receives an introduction-request from peer B no more than *stumble lifetime* seconds ago;

intro: peer C is an intro-Candidate when it is neither a walk nor a stumble-Candidate, and peer A receives an introduction-response from peer B with an introduction to peer C no more than *intro lifetime* seconds ago;

none: in all other cases.

Walk lifetime and stumble lifetime are both set to 57.5 seconds. We have chosen this value with regard to most NAT boxes closing a punctured 'hole' 60 seconds after receiving the last packet.

We have chosen 27.5 seconds for the intro lifetime with regard to most NAT boxes closing a punctured 'hole' 30 seconds after puncturing the hole without receiving any packets through it[3].

2.2 (Un)verified candidates

The Dispersy code provides two main methods to obtain available Candidate instances:

`dispersy_yield_candidates` 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 not actually had any contact with them ourselves.

`dispersy_yield_verified_candidates` returns an iterator with all walk and stumble-Candidate instances, in a randomised order. In most cases verified candidates are better than unverified ones.

[3] `http://pds.twi.tudelft.nl/reports/2010/PDS-2010-007.pdf`

2.3 Candidates we can walk towards

We are only allowed to walk towards a Candidate when it meets the two criteria described below. A candidate that meets these criteria is *eligible* for a walk.

1. the category is either walk, stumble, or intro, and
2. the previous walk to this candidate was more than *eligible delay* seconds ago.

We have chosen 27.5 seconds for the eligible delay, with the exception of bootstrap candidates which require a 57.5 seconds delay. This delay ensures that (bootstrap) peers are not contacted to frequently. This feature was initially introduced to reduce the numbers of walks towards trackers in overlays with no or few other peers.

2.4 References to the source code

The file dispersy/candidate.py defines the delay and lifetime values discussed in this section, as well as the Candidate class which provides methods to determine and influence the category, see below:

```
CANDIDATE_ELIGIBLE_DELAY = 27.5
CANDIDATE_ELIGIBLE_BOOTSTRAP_DELAY = 57.5
CANDIDATE_WALK_LIFETIME = 57.5
CANDIDATE_STUMBLE_LIFETIME = 57.5
CANDIDATE_INTRO_LIFETIME = 27.5
CANDIDATE_LIFETIME = 180.0

class WalkCandidate(Candidate):
    def get_category(self, now): pass
    def walk(self, now, timeout_adjustment): pass
    def walk_response(self): pass
    def stumble(self, now): pass
    def intro(self, now): pass
    def is_eligible_for_walk(self, now): pass
```

The file dispersy/community.py defines the Community class which contains the methods used to obtain Candidate instances in the neighborhood:

```
class Community(object):
    def dispersy_yield_candidates(self): pass
    def dispersy_yield_verified_candidates(self): pass
```

3 Who to walk to

In **phase 1** of the walk schema (see Section 1) peer A chooses a known peer B from its neighbourhood and sends it an introduction-request. Method `dispersy_get_walk_candidate()` chooses peer B and returns a Candidate instance pointing to it, or it returns None when no eligible candidates are available.

Choosing a Candidate to walk to heavily influences how large your neighbourhood will be. Based on your walks alone you will know approximately 11 Candidates (assuming you take one step every 5 seconds, see section 7), since at most 11 steps can fit within the 57.5 seconds *walk lifetime* window. At the same time other peers may chose you to walk to, hence incoming walks from unmet peers also increase your neighbourhood by the number of incoming unmet walks within the 57.5 seconds *stumble lifetime* window.

When we assume that there is at least one eligible Candidate in every category we can give the following simplified representation of the selection strategy:

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

Table 1 contains all possible combinations, the first column *has-WSIB* specifies if there is at least one walk, stumble, intro, or bootstrap candidate available. For example, 1000 means that the only available candidates are walk candidates, hence there is a 100% chance to for `dispersy_get_walk_candidate()` to return a walk-candidate.

This design takes into account that malicious peers can easily pollute our neighbourhood by walking towards us from multiple distinct addresses, effectively adding an arbitrary number of stumble-Candidates to our neighbourhood. Therefore, 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

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%	

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 500 * 15 * 60 / 5 bootstrap peers 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, you can apply the diff in `./minimal_bootstrap.diff`. This will result in the combinations shown in Table 2.

3.2 References to the source code

The file dispersy/community.py defines the method discussed in this section, see below:

```
class Community(object):
    def dispersy_get_walk_candidate(self): pass
```

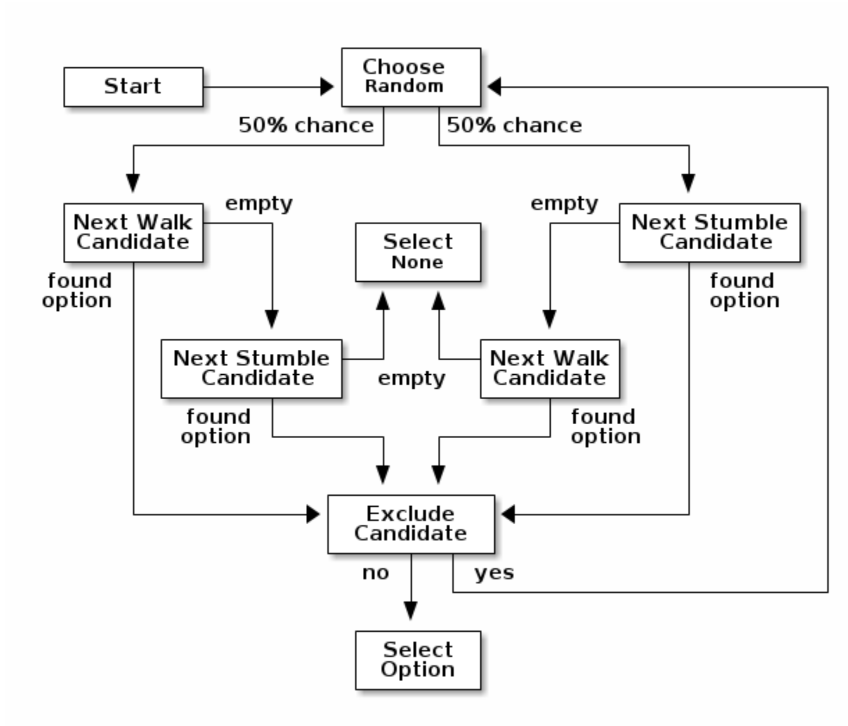
Table 2: Suggested chance to select a category based depending on which categories has eligible candidates.

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 schema (see Section 1) peer B chooses a known peer C from its neighbourhood and introduces it to peer A. Method `dispersy_get_introduce_candidate(exclude_candidate)` chooses peer C from the verified (and not excluded) available candidates and returns it, or, when no candidates are available, it returns None.

We designed `dispersy_get_introduce_candidate(exclude_candidate)` to return 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 above schema shows how we select a Candidate, however, in most cases we can simplify it as follows:

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

There are reasons why we can not introduce one candidate to another. Peer B can not introduce peer C to A when:

- when C and A are the same Candidate;
- when 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 and it can only receive messages with this prefix. We introduced tunnelling at the end of 2012 to allow all Dispersy traffic to be send through libswift. We introduced the ability for Dispersy to recognise the `FFFFFFFF` prefix without using libswift, while older Dispersy clients will believe the prefix is part of the message, making them unable to decode it. Because we can not distinguish between older and newer

code we are currently assuming all code is 'old'.
TODO add a picture to clarify

- when C and A are both behind a NAT that changes the outgoing port number and they are not within the same LAN.

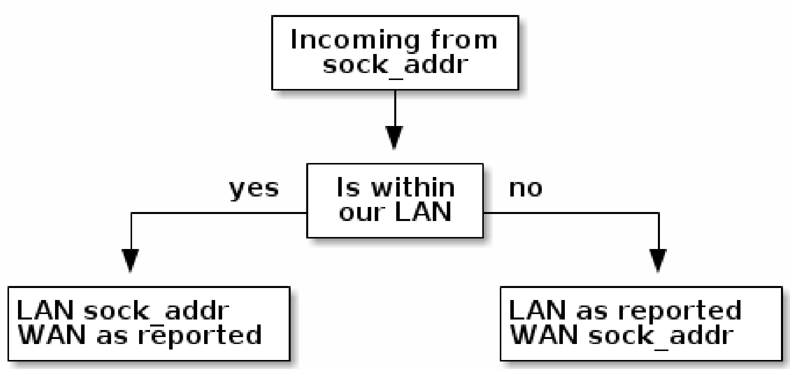
4.2 References to the source code

The file dispersy/community.py defines the method discussed in this section, see below:

```
class Community(object):
    def dispersy_get_introduce_candidate(self, exclude_candidate): pass
```

5 LAN and WAN address

In **phase 2** of the walk schema (see Section 1) peer B tries to determine the LAN and WAN address of peer A, it does this using the address reported in the UDP header (i.e. the `sock_addr`) of the incoming introduction-request combined with the WAN and LAN address that A reports that it has. We follow the schema shown below:



We implement this in method =estimate_lan_and_wan_addresses(sock_addr, lan_addr, wan_addr)=[4] which uses a simple assumption: when peer B sees that the message originated from within the same LAN it will assume that peer A's LAN address is the `sock_addr`. But when the message originated from outside its LAN then peer A's WAN address is the `sock_addr`.

[4]The word estimate is used as historically this code was not able to make this decision as cleanly as is described here.

Dispersy determines whether an address is within its own LAN by checking if it corresponds with one of its local interfaces, with regards to its netmask. We do this using the method `_get_interface_addresses()` 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, as discusses in Section 6.

5.1 References to the source code

The file dispersy/dispersy.py defines the methods discussed in this section, see below:

```
class Dispersy(object):
    @staticmethod
    def _get_interface_addresses(): pass
    def estimate_lan_and_wan_addresses(self, sock_addr, lan_address,
                                       wan_address): pass
```

6 WAN address voting

In **phase 2** of the walk schema (see Section 1) 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, see section 4.

Most of the magic happens in the method `wan_address_vote(address, B)` 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: when we change our WAN address we also 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.

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 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.

6.2 References to the source code

The file dispersy/dispersy.py defines the methods discussed in this section, see below:

```
class Dispersy(object):
    def wan_address_unvote(self, voter): pass
    def wan_address_vote(self, address, voter): pass
```

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[5];
2. taking a step involves performing the bloom filter synchronisation, see component_synchronization.org;

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 (see component_synchronization.org), 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 a step includes creating a bloom filter making it one of the most CPU intensive parts of Dispersy. Below we show a naive approach, where we simply schedule 5 seconds between each step. For the purpose of simplicity we will assume that it takes 1 second to create a bloom filter. This example holds choosing a more realistic value.

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 is increasing the gap between steps even further, resulting in only 7 steps instead of the expected 10.

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

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

7.2 Walking in multiple overlays

While ever widening gaps between steps is already a bad thing, it will only get worse when we need to maintain multiple 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. 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 lower letter 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 when the previous step was less than 4.5 seconds ago. This is a large performance boost 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 doesn't 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 diff in walk_multiplier.diff modifies all constants (as known at October 2013). Changing the `WALK_MULTIPLIER` constant 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.

7.4 References to the source code

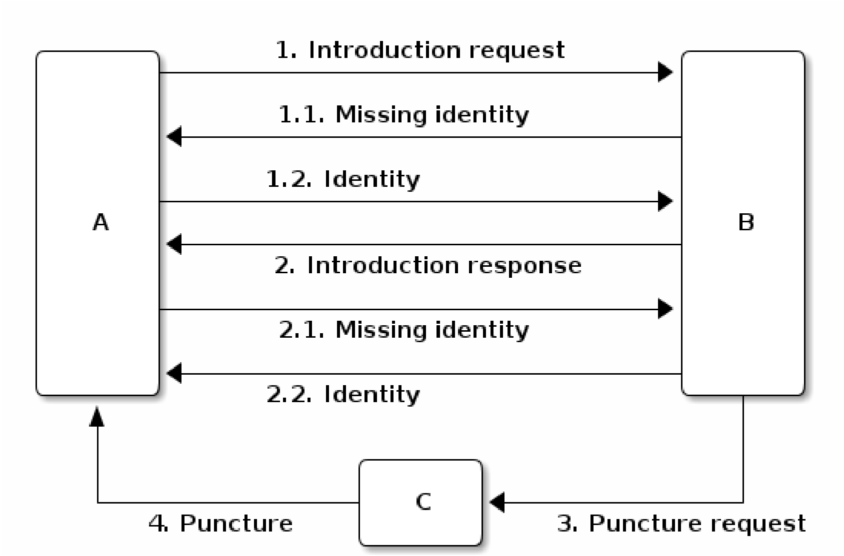
The file dispersy/dispersy.py defines the method discussed in this section, see below:

```
class Dispersy(object):
    def _candidate_walker(self): pass
```

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 the figure below.



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 is not behaving as expected.

9.1 Bootstrapping

To bootstrap an overlay we contact one of the bootstrap servers. When we have never encountered this bootstrap server before we 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 taking some steps we will have started building our neighbourhood. Below we see that we contact someone at 74.96.92.***:7759, we no longer need to exchange public keys, but 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 the a logger with the name `dispersy-stats-detailed-candidates`. When enabling INFO level messages this logger 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.