

Dispersy: Distributed Permission System

Boudewijn Schoon

Abstract

Treating members of a distributed system equally is counter intuitive to what we do during human interactions. Having a system that allows certain activity with members based on who they are, or what social group they belong to, is something that people understand.

We feel that a permission system provides a strong incentive for members to participate in a constructive manner or risk losing the permissions that they have obtained.

This report will describe the benefits and a practical implementation of such a distributed permission system, which we call Dispersy.

1 Introduction

When designing any piece of software that requires communication between multiple parties, one of the first design choices is to either use a central server or a distributed system. Both have advantages and disadvantages, and they roughly come down to: central servers being more secure and easier to design, while distributed systems are more robust and scalable.

While we can not completely change this, we can strive to design a platform for distributed systems that provides security and common features to reduce software complexity. We call this platform Dispersy, which is an acronym for Distributed Permission System.

As Dispersy is design to be a platform, it will not provide features for the end user. Such features must be build on top of the platform in the form of a community, shown in Figure 1. Many different communities are possible, we are currently working on a barter community that gives us information on how generous other nodes in the system are. Also, in the near future we will use

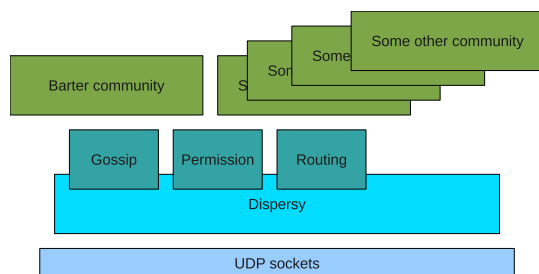


Figure 1: Hierarchy between Dispersy any other components

Dispersy to build a user community around BitTorrent, creating a distributed files sharing community.

There are a few design goals that we want to keep in mind when expanding Dispersy with new features:

- **Scalability** is important. Dispersy provides several different message policies that are designed to scale to several million simultaneous nodes. While large communities will result in reduced performance, dissemination of data must always converge over time.
- **Security** is moderate by design. The security features that Dispersy provides must be guaranteed to work. However, these features are fairly limited as security is inherently difficult to guarantee in any distributed system.
- **Security** is strong through social structure. As there is no central server that dictates policy, Dispersy creates a social structure where people higher up the chain override choices made by people lower in the chain.
- **Complexity** should be low. Most complex actions, such as user authentication and efficient dissemination of data, should be handled by Dispersy leaving the designer of the community free to focus on end user features.

In the following sections we will describe Dispersy and what it provides as from December 2010. This description will include the various message policies. We will conclude this report by describing the communities that are being build, and will be, build on top of Dispersy.

2 Dispersy

The Distributed Permission System, 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.

Since we can not guarantee each member to be online all the time, messages that they created at one point in time should be able to retain their meaning even when the member is off-line. This can be achieved by signing such messages and having them propagated though other nodes in the network. Unfortunately, this increases the strain on these other nodes, which we try to alleviate using specific message policies, which will be described below.

Following from this, we can easily package each message into one UDP packet to simplify connectability problems since UDP packets are much easier to pass though NAT's and firewalls.

Earlier we hinted that messages can have different policies. A message has the following four different policies, 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 gossipped 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 gossipped.

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 send 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.

A community can tweak the policies and how they behave by changing the parameters that the policies supply. Aside from the four policies, each meta-message also defines the community that it is part of, the name it uses as an internal identifier, and the class that will contain the payload. Below is an example of a meta-message definition.

```

1: Message(community = example,
2:   name = u"dispersy-example-message",
3:   authentication = MemberAuthentication(),
4:   resolution = LinearResolution(),
5:   distribution = FullSyncDistribution(),
6:   destination = CommunityDestination(),
7:   payload = ExamplePayload())

```

Since these message policies lie at the heart of Dispersy, we will describe them in depth in the following sections. As a reference Table 1 provides a list with all available policies and which policies are compatible with each other.

3 Permissions

One of the two most important features of Dispersy is its ability to handle permissions. Permissions tell us what a member is allowed to do, or in other words, which messages a member is allowed to send. The information required to make these choices is based on authorize and revoke messages. Each message can have three different types of payload:

- **Authorize** is used to grant a member the permission to use authorize, revoke, or permit payloads.
- **Revoke** is used to revoke a members permission to use authorize, revoke, or permit payloads.

<<25>	<25>	<25>	<25>	<25>
Message	*Authentication*	*Resolution*	*Distribution*	*Destination*
Authentication				
no-authentication		public	relay, direct	address, member, community
member-authentication		public, linear	relay, direct, full-sync, last-sync	address, member, community, similarity
multi-member-authentication		public, linear	relay, direct, full-sync\footnote(without sequence number), last-sync\footnote(without sequence number)	address, member, community, similarity
Resolution				
public-resolution	no, member, multi-member		relay, direct, full-sync, last-sync	address, member, community, similarity
linear-resolution	member, multi-member		relay, direct, full-sync	address, member, community, similarity
Distribution				
relay-distribution	no, member, multi-member	public, linear	address, member	
direct-distribution	no, member, multi-member	public, linear	address, member, community	community, similarity
full-sync-distribution	no, member, multi-member	member	public, linear	community, similarity
footnote(without sequence number)	member, multi-member	member	public, linear	community, similarity
last-sync-distribution	member, multi-member	member-member	public, linear	community, similarity
footnote(without sequence number)	member, multi-member	member	public, linear	community, similarity
Destination				
address-destination	no, member, multi-member	public, linear	relay, direct	
member-destination	no, member, multi-member	public, linear	relay, direct	
community-destination	no, member, multi-member	public, linear	direct, full-sync, last-sync	
similarity-destination	member, multi-member	public, linear	full-sync, last-sync	

Table 1: Available message policies.

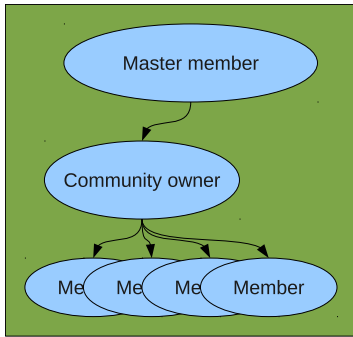


Figure 2: Authentication hierarchy starting with the mater member

- **Permit** is used to transfer the community defined payload for this message.

For example, before Bob is allowed to send a ‘hello-world’ message, Alice must first authorize Bob with the permission to send a ‘hello-world’ message. And this assumes that Alice has the permission to do this.

3.1 Global time

The above example introduces the aspect of time, because Bob obtained the permission at some point in time, and should not have been able to send a ‘hello-world’ message before that time. Furthermore, Alice can also revoke the permission, disallowing Bob to send ‘hello-world’ messages from some point in time. Global time is further discussed in Section 6.

3.2 Sequence number

Both authorize and revoke messages use the full-sync-distribution policy. This ensures that a member is unable to create these messages out of order, or that any such messages end up missing. Sequence numbers are further discussed in Section 6.4.

3.3 Master member

In the above example Alice gives permission to Bob. However, for this schema to work, another member must have given Alice the permission to do so. Dispersy solves this pitfall though a master member.

Each community has one master member, which is created whenever a new community is created. In fact, the master members’ public key is the unique identifier for the community. When a community is created, the master member authorizes one member, typically whomever created the community, with all available permissions. From this point other members can be authorized as well, if needed, as shown in Figure 2.

From a point of security, the master member serves as a final fall-back mechanism, if the member identity of someone in the

community has been compromised, the master member can always revoke all her permissions. Once the master member identity is compromised, the community is lost. To safeguard the master member identity as much as possible, a large elliptic curve should be used. By default this is the NID-sect571r1, where each signature is 142 bytes long.

4 Authentication policy

User management is very important to Dispersy, to ensure that nodes can not impersonate each other, each node has its own identity. This identity is, currently, given using a public/private elliptic curve key pair. Each member is free to choose which curve to use when creating its own identity, though we use NID-sect233k1 as a default.

4.1 No-authentication

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.

4.2 Member-authentication

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.

4.3 Multi-member-authentication

A message that uses the multi-member-authentication policy is signed by two or more member. Similar to the member-authentication policy the message contains two or more identifiers where the first indicates the creator and the following indicate the members that added their signature.

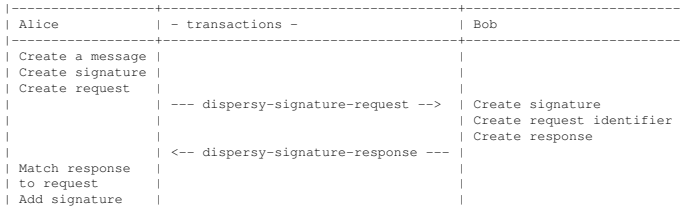


Table 2: Double signing a message using signature request and response messages.

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. Table 2 shows the steps required to double sign a message. First Alice creates a message (M), note that this message must use the multi-member-authentication policy, and signs it herself. At this point the message 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 finally the signature for Alice and enough bytes –all set to zero– to fit the signature for Bob.

```
1: Message(community = example,
2:   name = u"dispersy-signature-request",
3:   authentication = NoAuthentication(),
4:   resolution = PublicResolution(),
5:   distribution = DirectDistribution(),
6:   destination = MemberDestination(),
7:   payload = SignatureRequestPayload())
```

```
1: Message(community = example,
2:   name = u"dispersy-signature-response",
3:   authentication = NoAuthentication(),
4:   resolution = PublicResolution(),
5:   distribution = DirectDistribution(),
6:   destination = AddressDestination(),
7:   payload = SignatureResponsePayload())
```

This message is consequently wrapped inside a `dispersy-signature-request` message (R) and send to Bob. When Bob receives this request he is given the choice to add his signature, assuming that he does, both a signature and a request identifier will be generated. The signature signs the entire message (M) excluding the two signatures, while the request identifier is a sha1 digest over the request message (R).

Finally Bob sends a `dispersy-signature-response` message (E), containing the request identifier and his signature, back to Alice. Alice is able to match this specific response to the original request and adds Bob’s signature to message (M). This message, which is now double signed, can now be disseminated according to its own distribution policy.

The multi-member-authentication policy can be used to not only double sign, but also sign messages with even more members. The double sign mechanism is, for instance, used by the barter community to ensure that two members agree on the amount of bandwidth uploaded by both parties before disseminating this information to the rest of the community.

5 Resolution policy

Permissions tell us which messages a member is allowed to send. The resolution policy defines how the permission system resolves conflicts between messages, i.e. how it is decided whether or not a member was allowed to send a message. For any resolution policy it holds that all members must always converge toward exactly the same conclusion.

Each message that is received is checked. If the message is invalid, i.e. the creator of the message did not have the right permission at the global time defined in the message, a proof of validity is requested from the sender using a `dispersy-proof-request`¹. This proof should be in the form of a valid authorize message. The sender can now either provide a proof that has been made obsolete –in which case we can correct the sender– or the sender can provide a proof that is new to us –in which case we can correct our information–. Either way, the message is not processed until proof is supplied.

```
1: Message(community = example,
2:   name = u"dispersy-proof-request",
3:   authentication = NoAuthentication(),
4:   resolution = PublicResolution(),
5:   distribution = DirectDistribution(),
6:   destination = AddressDestination(),
7:   payload = ProofRequestPayload())
```

```
1: Message(community = example,
2:   name = u"dispersy-proof-response",
3:   authentication = NoAuthentication(),
4:   resolution = PublicResolution(),
5:   distribution = DirectDistribution(),
6:   destination = AddressDestination(),
7:   payload = ProofResponsePayload())
```

Do note that each member has three different permissions for each message in the community. Each member either has, or does not have, the permission to authorize, revoke, or permit one particular message. And each of these permissions needs to be explicitly granted with an authorize message and removed using a revoke message.

There is one exception hard-coded into Dispersy: the master member is allowed to send every message for its own community. Furthermore, revoking a permission for a master member will always fail.

5.1 Public-resolution

By far the easiest resolution policy is public-resolution. Every member is allowed to send messages that have this policy. Effectively this policy removes all permission capabilities that Dispersy has to offer. Even so, some messages might benefit from this policy, for instance for a search request message that everyone is always allowed to send.

5.2 Linear-resolution

A message that uses the linear-resolution policy an authorize or revoke-message will change one members’ permission after that point in time. In technical terms, authorizing or revoking a

¹Currently the `dispersy-proof-request` and `response` have not been implemented.

Time	Action	Bob (B)	Carol (C)
...		none	none
11	auth(A, B, permit)		
12	auth(A, B, authorize)	permit	
13	auth(A, B, revoke)	permit, authorize	
14		permit, authorize, revoke	
...			
42	auth(B, C, permit)		
43	auth(B, C, authorize)		permit
44	auth(B, C, revoke)		permit, authorize
45			permit, authorize, revoke
...			
56	auth(A, C, permit)		
57			
...			
166	revoke(A, B, revoke)		
167	revoke(A, B, authorize)	permit, authorize	
168	revoke(A, B, permit)	permit	
169		none	
...			

Table 3: Using the linear-resolution policy.

Time	Action	Bob (B)	Carol (C)
...		none	none
11	auth(A, B, 14, permit)		
12	auth(A, B, 14, authorize)		
13	auth(A, B, 14, revoke)		
14			permit, authorize, revoke
...			
27			everything revoked
...			
42	auth(B, C, 45, permit)		
43	auth(B, C, 45, authorize)		
44	auth(B, C, 45, revoke)		
45			everything revoked
...			
56	auth(A, C, 57, permit)		
57			permit
...			
166	revoke(A, B, 27, revoke)		
167	revoke(A, B, 27, authorize)		
168	revoke(A, B, 27, revoke)		
...			

Table 4: Using the cyclic-resolution policy.

permission at global time T will take effect at global time $T+1$ and onward until a new contradictory authorize or revoke message takes effect.

We will continue with a running example where we authorize and revoke the permit-, authorize-, and revoke-permissions for the ‘write’ message in a forum community. This example is illustrated in Table 3.

We start the example when Alice authorizes Bob with all permissions during T_{11} through T_{13} . This allows Bob to use all three permissions for the ‘write’ message starting at T_{14} .

Next we continue when Carol is authorized by both Alice and Bob with the permit permission at T_{42} and T_{56} , respectively. Furthermore, Bob authorizes Carol with the authorize and revoke permissions as well, at T_{43} and T_{44} , respectively. Note that Carol is authorized with the permit permission by both Alice and Bob from time T_{57} and onward, although this has no advantage.

Finally we conclude when Alice revokes all Bob’s permissions during T_{166} through T_{168} . Starting T_{169} and onward Bob no longer has any permissions for the ‘write’ message. Note that this does not affect the permissions that Carol obtained from Bob earlier.

Linear-resolution is the simplest resolution policy that we have available. Extending it in any way will quickly cause the complexity to increase significantly, as is shown with the cyclic-resolution policy below.

5.3 Cyclic-resolution

One thing that is clearly missing in the linear-resolution policy, is the possibility to undo damage caused by a malicious or careless member. However, going back in the timeline and invalidating a message that was previously valid causes many new challenges.

Do note however, that we have to face some of these challenges even when using the linear-resolution policy, since we are not able to disseminate all authorize- and revoke messages instantaneously.

The one difference between linear- and cyclic-resolution is that with the latter we specify -when- the authorize or revoke should commence, while with the former we always commenced at $T+1$ after creating the authorize- or revoke message.

This allows us to specify that a revoke should commence in the past, making all associated messages invalid. This does not only allow us to revoke permit messages, for instance writing to a forum, this may also revoke earlier authorize messages. Effectively, it is even possible to revoke your own permission to revoke. Clearly this requires a well defined schema, as it is most important that each member draws exactly the same conclusion, regardless of the order in which messages were arrived.

To elaborate we will again use a running example, shown in Table 4. Note that each authorize and revoke message now contains the global time at which it should commence.

We start at T_{11} through T_{13} where Alice authorizes Bob with all permission. This allows Bob to use permit-, authorize-, and revoke permissions starting at T_{14} as is defined in the authorize messages.

Next Bob authorizes Carol with all permissions at T_{42} through T_{44} . This allows Carol to use permit, authorize, and revoke starting at T_{45} as is defines in these authorize messages.

Next Alice also authorizes Carol with the permit permission at T_{56} . Even though Carol already had been given this permission by Bob at T_{42} .

Finally Alice revokes all of Bob’s permissions at T_{166} through T_{168} . However, this change is set to commence at T_{27} in the past. This retroactively changes everything that Bob has done starting at T_{27} . This includes the authorizations that Bob did for Carol at T_{42} through T_{44} . Therefore, Carol automatically loses all permissions except for the permit permission that she received from Alice at T_{56} .

This example shows both the power and the complexity of the cyclic-resolution policy. Especially considering that Alice, or any other member with the correct permissions, is also able to reactivate the permissions for either Bob or Carol.

We believe that the cyclic-resolution policy can be a very powerful asset to a community, however, because we do not yet have a practical need for this particular policy, we have chosen to focus on other aspects of Dispersy for now.

6 Distribution policy

The distribution policy defines how a message is send or distributed to one or more members. So far there are two distinct categories for distribution policies. The first category send a message from one member to another, this includes the direct-distribution and relay-distribution policies. The second category allows members to disseminate messages for each other, this includes the full-sync-distribution and last-sync-distribution policies.

All distribution policies have one property in common, they all contain the global time. As discussed before in section 3, global time is very important to decide message ordering. Whenever a message with a global time is received, this updates the local global time. Furthermore, whenever a message with a global-time is send, the current global time is incremented by one, and this new value is send with the message².

Using this schema ensures that all messages send by a member will have unique global time values associated to them, allowing us to order these messages. However, multiple members may still have messages with the same global time. To order these messages other ordering rules must be followed. The rule itself is not important, but, ensuring that each member follows the same rule is. Currently we order these messages using their sha1 digest.

The current implementation uses 64 bits to keep track of the global time, allowing an almost limitless supply of time. Although this can be reduces significantly when malicious messages are send. We believe that we can design an algorithm to detect these malicious messages, for instance by creating a global consensus on the current time. However, currently these algorithms are future work.

6.1 Direct-distribution

The direct-distribution policy is the simplest available distribution policy, it simply sends the message directly to the given destination. This policy increase the size of the message by only eight bytes, for the global time, making it a very simply and cheap policy.

However, it is unable to circumvent firewalls or NAT boxes, nor will other people store and forward this message once the message creator is off-line. Direct-distribution is often used for simple requests that need an immediate response or no reply at all.

6.2 Relay-distribution

The relay-distribution policy is a slight extension to direct-distribution, allowing firewall and NAT traversal by using a proxy member that can communicate with both the message sender and receiver.

Currently we have not yet implemented the relay-distribution policy, nor have we explicitly defined the interface yet. It might become a privacy mechanism where several proxies, or hops, may

²Currently not all messages that are send will increment the global time. This should be fixed.

be taken before the message reaches its destination. At this point in time we have not had the need for this policy.

6.3 Sync-distribution

The sync-distribution policy is not a usable policy in itself, however, it provides the basic functionality that both the full-sync-distribution and last-sync-distribution policies use, namely: message gossiping.

Whenever a valid message is received that uses either the full-sync-distribution or last-sync-distribution policy, this message is stored in the local Dispersy database. How long this message remains in the database depends on the specific distribution policy and its parameters and is discussed in the following sections.

Another member can obtain stored messages from another members' database by sending a dispersy-sync message. The properties for this message are shown below.

```
1: Message(community = example,
2:         name = u"dispersy-sync",
3:         authentication = MemberAuthentication(),
4:         resolution = PublicResolution(),
5:         distribution = DirectDistribution(),
6:         destination = CommunityDestination(),
7:         payload = SyncPayload())
```

A dispersy-sync message contains a global time (T) and a bloom filter. This bloom filter is filled with all the messages that the sender has in the global time range T though T+1000. The receiving member reads all messages in this time range from its database and checks if this message is contained in the bloom filter. Missing messages are send back. Currently we send back no more than five kilobytes worth of messages for each sync message that we receive, although this is likely to be increased in the future.

We choose the time range of 1000 because this results in a dispersy-sync message that is approximately one kilobyte large. This has the benefit that the message will not be split in multiple IP packets while traversing the internet. However, we have future plans to ensure that the dispersy-sync message sends bloom filters of variable size to ensure that large communities, where many more messages will be in the same global time range, will still be effective.

Furthermore, deciding how often, how many, and to whom the dispersy-sync messages should be send is a very delicate matter. For instance, sending ten sync messages every minute makes it more likely that duplicate messages are received, because multiple members could send back the same messages. However, sending to very few members very often increases the CPU usage and, depending on how we choose to whom we send, might reduce dissemination speed.

We are currently experimenting with these parameters to see which values and combinations gives us optimal gossiping performance.

6.4 Full-sync-distribution

The full-sync-distribution policy ensures full gossip. Each message that ever existed is replicated on all appropriate members. To

ensure that no messages are missing, each full-sync-distribution message contains a sequence number that is unique per member. The first message must have sequence number 1, and following messages must increment this value by one for each message that is created.

Using this sequence number we can ensure that no messages are missing. When a message is received with sequence number 3, while we only have sequence number 1 for this message, we will request the missing message using a dispersy-missing-sequence message, defined below.

```
1: Message(community = example,
2:     name = u"dispersy-missing-sequence",
3:     authentication = NoAuthentication(),
4:     resolution = PublicResolution(),
5:     distribution = DirectDistribution(),
6:     destination = AddressDestination(),
7:     payload = MissingSequencePayload())
```

Unless the sender of a dispersy-missing-sequence message is malicious, the receiver should have the missing messages in her database and she should respond by sending them back.

Unfortunately the full-sync-distribution policy will, over time, result in a large collection of stored messages, because no messages are ever discarded from the database. This presents us with scalability problems that we can only address by making concessions in recall-ability. The last-sync-distribution describes a policy that makes such a concession.

6.5 Last-sync-distribution

The last-sync-distribution policy ensures full gossip for the last N messages for each member. This results in older messages going extinct as new ones are generated. The largest reason for this policies existence is to reduce the amount of storage and bandwidth required for large communities, i.e. to make Dispersy scalable.

Messages with this policy do not have a sequence number, and only use the global time to determine which messages are to be discarded. However, the global time does not allow us to detect missing messages, therefore, no dispersy-missing-sequence message can be send. Messages that use the last-sync-distribution policy can therefore expect a slower convergence rate.

7 Destination policy

With many of the Dispersy features a member should not be concerned by where a message is send. For instance, with full gossip the message should end up at every member, therefore the end user does not actually care to whom it is send first. The destination policy is all about to whom a message is send.

7.1 Address-destination

The address-destination policy sends the message to one or more specified IP addresses and port combinations. After the message is send, no verification is given that the message is actually received.

7.2 Member-destination

The member-destination policy sends the message to one or more specified members. However, a translation is required to obtain the IP addresses and port numbers for each member. This translation uses dispersy-identity messages which are defined below. The payload of this message contains the IP address and port number that the member believes it is available at. Note that this message uses the last-sync-distribution policy with a history_size of 1. Therefore, only the most recently known address is disseminated for each member in the community.

When no dispersy-identity message, and hence no last known address, is available, a dispersy-identity-request message is used to obtain it from other members of the community. When this fails, the message can not be send.

```
1: Message(community = example,
2:     name = u"dispersy-identity",
3:     authentication = MemberAuthentication(encoding="pem"),
4:     resolution = PublicResolution(),
5:     distribution = LastSyncDistribution(cluster=254, history_size=1),
6:     destination = CommunityDestination(),
7:     payload = IdentityPayload())
```

```
1: Message(community = example,
2:     name = u"dispersy-identity-request",
3:     authentication = NoAuthentication(),
4:     resolution = PublicResolution(),
5:     distribution = DirectDistribution(),
6:     destination = AddressDestination(),
7:     payload = IdentityRequestPayload())
```

After the message is send, no verification is given that the message is actually received.

7.3 Community-destination

When a member is not interested in defining a destination herself, the community-destination policy can be used. This policy uses a routing table, which contains addresses and the most recent times when a message was either send too or received from that address.

Which addresses are picked from the routing table will greatly influence how large the group of people will be that a node regularly updates with. Therefore, it directly influences how effective the dissemination of data will be. Currently a slight emphasis is put on keeping contact with members that appear to be online.

The community-destination policy currently specifies how many addresses should be selected from the routing table, although, we will add properties to ensure that each message is able more control over their destination. These properties will include how recently we must have received something from that address, how recently we send something to that address, what to do when non such addresses exist, etc.

7.4 Similarity-destination

Especially when using one of the full gossip distribution policies, we are likely to encounter scalability problems. One way to alleviate this is to limit the number members to which a message needs to be spread. The similarity-destination policy does this though a form of semantic clustering. Clustering can take place on a variety of things, such as members that have:

- similar taste
- similar IP addresses
- similar sharing ratio
- similar online behavior
- interaction with each other
- externally defined as friends

We believe that a bloom filter –that uses only one bit per entry– can be used for a wide variety of these similarity checks. Each node is free to fill its bloom filter with items it likes or characteristics that it has. When two bloom filters are compared, the number of bits that are logical bi-conditional indicate how similar they are, or in other words. When the number of matching bits exceeds a specified threshold we define this as the nodes being in each others sphere of influence.

For example, below are given the bits that are set for eight words that a node might use to specify its taste. Following this are the members A, B, and C with their similarity bits.

```
# bitvector for eight words
00000000 00000000 10000000 00000000 = cake
00000000 00000000 00000000 00100000 = lemonade
00000000 00000010 00000000 00000000 = kittens
00000000 00000000 00000000 00000010 = puppies
00000010 00000000 00000000 00000000 = beer
00000010 00000000 00000000 00000000 = booze
00001000 00000000 00000000 00000000 = women
00000000 00000000 00000000 00000100 = pubs

# similarity bits for Alice, Bob, and Carol
00000000 00000010 10000000 00100010 = Alice: cake, lemonade, kittens, puppies
00000010 00000000 10000000 00100100 = Bob:  cake, lemonade, beer, pubs
00001010 00000000 00000000 00000100 = Carol: beer, booze, women, pubs

# logical bi-conditional between members
bi_conditional(Alice, Carol) = 25
bi_conditional(Alice, Bob) = 28
bi_conditional(Bob, Carol) = 29
```

From the resulting bi-conditionals we see that Alice and Carol –with value 25– are the least similar. This is certainly true as Alice and Carol are the only members who did not add the same words in their similarity bitvector. However, considering that in this example, we only used 32 bits for the bitvectors, 25 is still a high value.

Our experience with the Tribler overlay and its taste buddies tells us that members only very rarely have any overlapping taste. Hence we need to add more bits. Currently we choose to add random bits to ensure that semantic clustering becomes more distinct.

This allows Dispersy to adjust the semantic cluster that it is part of, by changing these random bits to match more or less, or even specific members in the community. We call this regulating the similarity bits³.

³Currently we have not yet implemented this regulation mechanism.