QLectives – Socially Intelligent Systems for Quality
Project no. 231200

Instrument: Large-scale integrating project (IP)
Programme: FP7-ICT

Deliverable D.4.1.2

QLectives Platform v2

Submission date: 2011-02-17

Start date of project: 2009-03-01 Duration: 48 months

Organisation name of lead contractor for this deliverable: TUDelft

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidental, only for members of the consortium (including the Commission Services)	

Document information

1.1 Author(s)

Author	Organisation	E-mail
Tamás Vinkó	TU Delft	T.Vinko@tudelft.nl
Nazareno Andrade	TU Delft	N.FerreiradeAndrade@tudelft.nl
Boudewijn Schoon	TU Delft	P.B.Schoon@tudelft.nl
Johan Pouwelse	TU Delft	J.A.Pouwelse@tudelft.nl

1.2 Other contributors

Name	Organisation	E-mail

1.3 Document history

Version#	Date	Change
V0.1	6 December, 2010	Starting version, template
V0.2	17 December, 2010	Complete first draft
V0.3	X, 2011	Complete revised version
V1.0	X, 2011	Approved version to be submitted to EU

1.4 Document data

Keywords	Peer-to-Peer, BitTorrent, Techno-Social Networking, Distributed Permission System, NAT puncturing
Editors address data	T.Vinko@tudelft.nl, N.FerreiradeAndrade@tudelft.nl
Delivery date	XX, 2011

1.5 Distribution list

Date	Issue	E-mail
	Consortium members	QLECTIVES@list.surrey.ac.uk
	Project officer	Jose.FERNANDEZ-VILLACANAS@ec.europa.eu
	EC archive	INFSO-ICT-231200@ec.europa.eu

QLectives Consortium

This document is part of a research project funded by the ICT Programme of the Commission of the European Communities as grant number ICT-2009-231200.

University of Surrey (Coordinator)
Department of Sociology/Centre
for Research in Social Simulation
Guildford GU2 7XH
Surrey
United Kingdom
Contact person: Prof. Nigel Gilbert
E-mail: n.gilbert@surrey.ac.uk

University of Fribourg
Department of Physics
Fribourg 1700
Switzerland
Contact person: Prof. Yi-Cheng Zhang
E-mail: yi-cheng.zhang@unifr.ch

Technical University of Delft
Department of Software Technology
Delft, 2628 CN
Netherlands
Contact Person: Dr Johan Pouwelse
E-mail: j.a.pouwelse@tudelft.nl

University of Warsaw
Faculty of Psychology
Warsaw 00927
Poland
Contact Person: Prof. Andrzej Nowak
E-mail: nowak@fau.edu

ETH Zurich
Chair of Sociology, in particular
Modelling and Simulation
Zurich, CH-8092
Switzerland
Contact person: Prof. Dirk Helbing
E-mail: dhelbing@ethz.ch

Centre National de la Recherche Scientifique, CNRS
Paris 75006,
France
Contact person: Dr. Camille ROTH
E-mail: camille.roth@polytechnique.edu

University of Szeged
MTA-SZTE Research Group on
Artificial Intelligence
Szeged 6720, Hungary
Contact person: Dr Mark Jelasity
E-mail: jelasity@inf.u-szeged.hu

Institut für Rundfunktechnik GmbH
Munich 80939
Germany
Contact person: Dr. Christoph Dosch
E-mail: dosch@irt.de

QLectives introduction

QLectives is a project bringing together top social modelers, peer-to-peer engineers and physicists 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** - in which people link to others over the Internet to gain value and facilitate collaboration
- **Peer production** - in which people collectively produce informational products and experiences without traditional hierarchies or market incentives
- **Peer-to-Peer systems** - in which software clients running on user machines distribute media and other information without a central server or administrative control

QLectives aims to bring these together to form Quality Collectives, i.e. functional decentralised communities that self-organise and self-maintain for the benefit of the people who comprise them. We aim to generate theory at the social level, design algorithms and deploy prototypes targeted towards two application domains:

- **QMedia** - an interactive peer-to-peer media distribution system (including live streaming), providing fully distributed social filtering and recommendation for quality
- **QScience** - a distributed platform for scientists allowing them to locate or form new communities and quality reviewing mechanisms, which are transparent and promote quality

The approach of the QLectives project is unique in that it brings together a highly inter-disciplinary team applied to specific real world problems. The project applies a scientific approach to research by formulating theories, applying them to real systems and then performing detailed measurements of system and user behaviour to validate or modify our theories if necessary. The two applications will be based on two existing user communities comprising several thousand people - so-called ”Living labs”, media sharing community tribler.org; and the scientific collaboration forum EconoPhysics.

Executive summary

This report accompanies and documents the version 2.0 of the QLectives Platform software. The aim of the QLectives Platform is to combine social networking, facilitation of quality and scalable peer-to-peer (P2P) technology into a next-generation peer-production platform. This platform can serve as the basis for the implementation of multiple peer-to-peer systems, and consists of several components which are continuously refined using input from other work packages. All the components of the QLectives Platform are (and will be) generic and re-usable as they can handle various content types (e.g. software, video, photo, text) and are not tied to a specific application domain.

The QLectives Platform version 1.0 was built on top of the already deployed and mature P2P tribler.org code-base, which provides most of the low-level P2P functionalities for the social networking and quality facilitation required. All the technical details about that version were supplied in the QLectives Deliverable D4.1.1.

This report describes the implementation progress in the QLectives Platform since QLectives Deliverable D4.1.1. This progress is centered on the development of one major component that implements alone most of the core features needed for the decentralized collectives envisioned in QLectives. This component is the Distributed Permission System, which we name **Dispersy**. Dispersy is the result of factoring out and advancing the common functionality implemented for all collectives-related aspects of QLectives Platform and QMedia version 1.0. This module provides the necessary primitives to implement functionalities for decentralized groups that share a robust, secure, and scalable set of messages and permissions. We envision that porting current modules to rely on Dispersy as well as implementing future functionality using this module will greatly improve the quality of QMedia and QLectives Platform.

Contents

Chapter 1

Introduction

This report accompanies and documents the version 2.0 of the QLectives Platform software. It is implemented in the context of the QLectives project and aims to serve as a generic middleware to develop peer-to-peer (P2P) applications. The central novelty of QLectives Platform's version 2.0 is the Distributed Permission System, which we name Dispersy. This module provides a platform to build up communities (see Figure 1.1 for a schematic view), for example a barter community, that gives information on how generous the nodes in the system are (this example is detailed in QLectives Deliverable D.4.3.2). In the future, the QLectives project will consider porting the current QScience living lab software to the QLectives Platform; that would be done using the version 2.0 code-base.

Dispersy's core functionality is to enable the creation of communities. A Dispersy community is a set of nodes that share a same set of messages and permission settings. The design of Dispersy and its functionality is guided by the

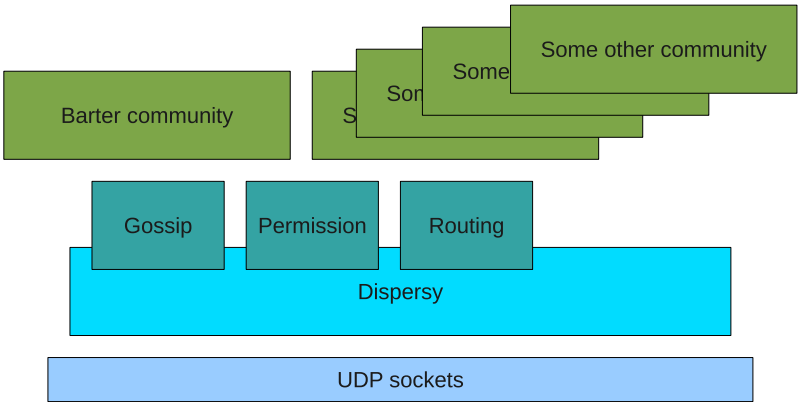

Figure 1.1: Hierarchy between Dispersy and other components

following design considerations:

- The first paramount factor to observe is scalability. Dispersy provides several different message policies that are designed to scale to million of simultaneous nodes. While large communities will result in reduced performance, dissemination of data must always converge over time.
- Security is mild 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 large-scale decentralized systems.
- Security is made strong through social structures. As there is no central server that dictates policy, Dispersy creates a social structure where people higher up the chain override choices by people lower in the chain.
- Complexity, regarding the actual software development for extension of the system, 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 Chapter, we give a detailed description of Dispersy and what it provides in QLectives Platform version 2.0 release.

Chapter 2

Dispersy

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

While we cannot fundamentally change this postulate, 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.

The central metaphor in Dispersy is a *community*. A community shares a set of messages known to all of its members, and a set of permissions implemented by the members.

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 [10]. Since we cannot 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 offline. 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. We alleviate this strain using specific message policies.

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 NATs and firewalls.

```
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())
```

Figure 2.1: Meta-message definition.

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 sent once or if it should be disseminated among nodes. 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 sent or synchronized.

To ensure that every node handles a messages in the same way, i.e. has the same policies associated with each message, a message exists in two parts: meta-message and the implemented-message. The meta-message part tells how the message is supposed to be handled. When a message is sent or received, an implementation is made from the meta-message that contains information specifically for that message. For example: a meta-message could have the member-authentication-policy that tells us that the message must be signed by a member but only the 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. Figure 2.1 shows an example of a meta-message definition.

Since these message policies lie at the heart of Dispersy, we describe them in depth in the following. As a reference, Table 2.1 provides a list with all available policies and which policies are compatible with each other. Before discussing policies in detail, however, we describe the permissions system in the next section.

2.1 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 member's permission to use authorize, revoke, or permit payloads.

- **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.

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 2.4.

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 also further discussed in Section 2.4.

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[a], last-sync[b]	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, last-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
full-sync-distribution[c]	member	public, linear		community, similarity
full-sync-distribution[d]	member, multi-member	public, linear		community, similarity
last-sync-distribution[e]	member-member	public, linear		community, similarity
last-sync-distribution[f]	member, multi-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	

[a] without sequence number
[b] without sequence number
[c] with sequence number
[d] without sequence number
[e] with sequence number
[f] without sequence number

Table 2.1: Available message policies.

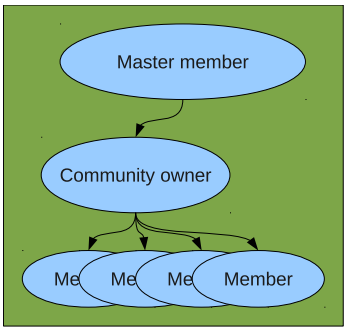

Figure 2.2: Authentication hierarchy starting with the mater member

2.1.1 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 problem 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 2.2.

From the point of view of security, the master member serves as a final fallback 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.

2.2 Authentication policy

User management is very important to Dispersy. To ensure that nodes cannot impersonate each other, each node has its own identity. Because our message oriented strategy often requires single messages to be signed, we choose elliptic curves over RSA cryptography. Using elliptic curves, the same level of security as RSA can be obtained with a smaller signature length [10]. A curve of 233 bits (sect233k1) was chosen because this is the same as the one that Tribler (and hence

QMedia) has been using over the past years to ensure that existing members can retain their current identity.

2.2.1 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 to be relayed, as everyone can generate and modify it as they please. This makes this policy ill suited for information dissemination, for example, through gossip [6]. In gossiping, all node in the network periodically propagate messages they have received in the past to a random subset of nodes.

2.2.2 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.

2.2.3 Multi-member-authentication

A message that uses the multi-member-authentication policy is signed by two or more members. Similar to the member-authentication policy, the message con-

```
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())
```

Figure 2.3: Meta-message definition for the dispersy-signature-request message.

tains two or more identifiers where the first indicates the creator and the following indicate the members that added their signature.

Alice	- transactions -	Bob
Create a message		
Create signature		
Create request		
	— dispersy-signature-request –>	Create signature
		Create request identifier
		Create response
	<– dispersy-signature-response —	
Match response		
to request		
Add signature		

Table 2.2: 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, shown in figures 2.3 and 2.4, respectively. Table 2.2 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.

This message is consequently wrapped inside a dispersy-signature-request message (R) and sent to Bob. When Bob receives this request he is given the

```
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())
```

Figure 2.4: Meta-message definition for the dispersy-signature-response message.

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.

Section 2.4.3 describes sequence numbers that can be included in a message. A message that uses the multi-member-authentication policy should not use sequence numbers as well. If the message owner claims a new sequence number for a message (M1) that uses the multi-member-authentication policy, he must wait until all signature responses are received until he can create a new message (M2) with the next sequence number. If he does not wait, his sequence chain will be broken and other members will want to obtain –when they receive M2– the missing M1, even though M1 does not yet exist. This is even excluding the problem of members -not- giving their signature to M1 at all.

This can be solved by adding a dummy message when the members do not want to add their signature, although this is not yet implemented. Furthemore,

```
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())
```

Figure 2.5: Meta-message definition for the dispersy-proof-request message.

performance can be improved by allowing messages to be processed out of sequence order, although this partially defeats the purpose of the sequence numbers in the first place.

2.3 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 and provided with a dispersy-proof-response message[1] shown in figures 2.5 and 2.6, respectively. 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]Currently the dispersy-proof-request and response have not been implemented.

```
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())
```

Figure 2.6: Meta-message definition for the dispersy-proof-response message.

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.

2.3.1 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.

2.3.2 Linear-resolution

A message that uses the linear-resolution policy and authorize- or revoke-message will change one members' permission after that point in time. In technical terms, authorizing or revoking a 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 2.3.

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 2.3: Using the linear-resolution policy.

We start the example when Alice authorizes Bob with all permissions during T_{11} trhough 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} trhough 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.

2.3.3 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 arrived.

To elaborate we will again use a running example, shown in Table 2.4. 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 permissions. 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 defined 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}.

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 2.4: Using the cyclic-resolution policy.

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 re-activate 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.

2.4 Distribution policy

The distribution policy defines how a message is sent or distributed to one or more members. So far there are two distinct categories for distribution policies. The first category concerns sending 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 in Section 2.1, global time is central to decide message ordering. Whenever a message with a global time is received, this updates the local register of the global time. Furthermore, whenever a message with a global-time is sent, the current global time is incremented by one, and this new value is sent with the message.

Using this schema ensures that all messages sent 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. Nevertheless, this can be reduced significantly when malicious messages are sent. 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.

2.4.1 Direct-distribution

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

2.4.2 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.

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

Figure 2.7: Meta-message definition for the dispersy-sync message.

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

2.4.3 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. With gossiping not only the member who created the message disseminates the message, but other members also help relaying the message to other members.

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 a member's database by sending a dispersy-sync message. The properties for this message are shown in Figure 2.7.

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 sent 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 mes-

```
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())
```

Figure 2.8: Meta-message definition for the dispersy-missing-sequence message.

sage 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 sent 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.

The sync-distribution policy provides the option to either enable or disable sequence numbers. When such numbers are enabled, Dispersy ensures that no messages go missing, and that each message contains a sequence number that is unique per member and per meta-message. 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 will be able to detect missing sequence numbers and hence missing messages. For instance: 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 in Figure 2.8.

Unless the sender of a dispersy-missing-sequence message is malicious, the receiver should have the missing messages in her database and she should re-

spond by sending them back.

2.4.4 Full-sync-distribution

The full-sync-distribution policy ensures full gossip, in other words, the policy ensures that each message that ever existed is replicated on all appropriate members.

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.

2.4.5 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 sent. Messages that use the last-sync-distribution policy can therefore expect a slower convergence rate.

2.5 Destination policy

With many of the Dispersy features a member should not be concerned by where a message is sent. 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 sent first. The destination policy is all about to whom a message is sent.

```
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())
```

Figure 2.9: Meta-message definition for the dispersy-identity message.

2.5.1 Address-destination

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

2.5.2 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 in Figure 2.9. 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, see Figure 2.10 is used to obtain it from other members of the community. When this fails, the message cannot be sent.

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

2.5.3 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

```
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())
```

Figure 2.10: Meta-message definition for the dispersy-identity-request message.

addresses and the most recent times when a message was either sent to 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 to have more control over their destination. These properties will include how recently we must have received something from that address, how recently we sent something to that address, what to do when no such addresses exist, etc.

2.5.4 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. 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 Alice: cake, lemonade, kittens, puppies
00000000 00000010 10000000 00100010

# similarity bits Bob: cake, lemonade, beer, pubs
00000010 00000000 10000000 00100100

# similarity bits Bob: beer, booze, women, pubs
00001010 00000000 00000000 00000100

# 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 bits.

2.6 Related literature

We are not aware of any platform that provides the same set of functionalities of Dispersy and could be integrated in the QLectives Platform. GO [11] is a platform that allows multiple gossip-based applications to share the underlying network resources. Although similar to Dispersy in this sense, GO does not focus on distributed permissions. Ricochet [1] considers the optimization of information dissemination among overlapping groups of nodes, but for time-critical applications most suitable for clusters. JXTA [8] is a general-purpose P2P middleware that supports multiple groups and applications, but does not allow the fine tuning of the information dissemination utilized by the applications built on top of it. This fine tuning, in turn, is a core requirement for the QLectives Platform, informed by our research on the impact of information dissemination on the protocols and mechanisms implemented in QMedia [5] (see also Deliverable 4.1.2).

The seminal work involving the information dissemination approach most used in Dispersy, gossiping, was done in the context of database replication [6]. This work has inspired a thread of developments in the application of gossip — also referred to as epidemic protocols — to several domains, and notably to P2P systems (for overviews, see [9, 2, 3]). At its core, Dispersy employs a form of

semantic overlay [12, 13]. Our approach, however, differs from most gossip applications in its use of more aggressive information dissemination policies based on bloom filters (for an overview of bloom filters, see [4]). Finally, the certification in Dispersy is similar to that defined by SPKI [7].

2.7 Implementation status

The majority of the features described in this report have been already implemented in the QLectives Platform version 2.0. However, for clarity of presentation when discussing Dispersy's design, we have also referred to ongoing and near-future implementation work. This section summarizes the implementation status of all features referred to in this report.

The source code of Dispersy is available at http://svn.tribler.org/abc/tags/tribler-dispersy-1.0.0rc1/. A formal documentation of the current implementation status of Dispersy, together with a developer-oriented documentation of the present API is available at http://svn.tribler.org/abc/tags/tribler-dispersy-1.0.0rc1/dispersy-api-december-2010.pdf

The Dispersy core has been implemented with the following set of features:

- Encoding messages from the internal message format to on-the-wire bytes (see Conversion.py).
- Decoding messages from the on-the-wire bytes to the internal message format (see Conversion.py).
- Delaying an incoming message when a previous message is missing. Once the required missing messages are retrieved using a dispersy-sequence-request the delayed message is processed (see Dispersy.py, Message.py, and Trigger.py).
- Delaying incoming messages when a public key is missing to properly verify the message content. Once the missing key is retrieved using a dispersy-identity-request the delayed message is processed (see Dispersy.py, Message.py, and Trigger.py).
- Triggers can be used to obtain a callback whenever the footprint of an incoming message matches a given regular expression. This is usually used

to wait, and provide a timeout, for a response message. (see Dispersy.py and Trigger.py).

- Routing information is gathered though dispersy-routing-request and dispersy-routing-response messages. Consequently, this information is used to disseminate the public address of each member using a dispersy-identity message (see Dispersy.py).
- Periodically one or more addresses will be choosen and an attempt will be made to obtain new messages using a dispersy-sync message (see Dispersy.py).

The following message policies, described in this report, have been implemented:

- No-authentication (see Authentication.py).
- Member-authentication (see Authentication.py).
- Multi-member-authentication (see Authentication.py and Dispersy.py).
- Public-resolution (see Resolution.py).
- Direct-distribution (see Distribution.py).
- Full-sync-distribution (see Distribution.py).
- Last-sync-distribution (see Distribution.py).
- Address-destination (see Destination.py).
- Member-destination (see Destination.py).
- Community-destination (see Destination.py, with the actual choice of to whom the message is sent implemented in Dispersy.py)

The following message policies, described in this report, have been partially implemented:

- Linear-resolution. The mechanics of the authorize and revoke messages are still under consideration.

- Cyclic-resolution.
- Relay-distribution. This policy, as described in its related section, has not yet been implemented. We do believe this message can have great value to provide a measure of anonymity and hence hope to see it implemented in the near future.
- Similarity-destination. This policy is only partially implemented and will require considerable work before it is able to do what it is intended for.

Several test cases are availble to ensure that, after a modification, the current functionality is not accidentally broken. These test cases cover the basic usage of the policies described in this report, although we feel they can be extended. The testcases can be executed using the steps below:

```
$ svn co http://svn.tribler.org/abc/tags/tribler-dispersy-1.0.0rc1
$ cd tribler-dispersy-1.0.0rc1
$ rm *.db *.log
$ ./dispersy-test
```

2.8 Summary

This report documents the second release of the QLectives Platform, that can be used to base the development of different distributed communities. For that, this report describes the core element of the platform, called Dispersy, which stands for Distributed Permission System. This system is a major step towards a general software platform for integrating the living labs' applications, QMedia and QScience into a common ground.

The work on the QLectives Platform builds on the results of QLectives Deliverable D4.1.1 and lays the foundations for experimentation with the algorithms developed in Stream 2, in particular with those reported in Deliverables 2.1.1 and 2.1.2. Moreover, the implementation described in this report is the base of the development of QMedia version 2, described in Deliverable D4.3.2. The release of the QLectives Platform version 3.0 is scheduled for February 2012.

Bibliography

[1] M. Balakrishnan, K. Birman, A. Phanishayee, and S. Pleisch. Ricochet: Lateral error correction for time-critical multicast. In *NSDI 2007: Fourth Usenix Symposium on Networked Systems Design and Implementation*, 2007.

[2] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah. Gossip algorithms: Design, analysis and applications. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1653–1664. IEEE, 2005.

[3] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah. Randomized gossip algorithms. *IEEE Transactions on Information Theory*, 52(6):2508–2530, 2006.

[4] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

[5] R. Delaviz, N. Andrade, and J.A. Pouwelse. Improving Accuracy and Coverage in an Internet-Deployed Reputation Mechanism. In *IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*, pages 1–9. IEEE, 2010.

[6] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.

[7] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory, 1999.

[8] L. Gong. JXTA: A network programming environment. *Internet Computing, IEEE*, 5(3):88–95, 2002.

[9] S.M. Hedetniemi, S.T. Hedetniemi, and A.L. Liestman. A survey of gossiping and broadcasting in communication networks. *Networks*, 18(4):319–349, 1988.

[10] G. Seroussi. Elliptic curve cryptography. In *Information Theory and Networking Workshop, 1999*, page 41. IEEE, 2002.

[11] Y. Vigfusson, K. Birman, Q. Huang, and D.P. Nataraj. GO: Platform support for gossip applications. In *IEEE Ninth International Conference on Peer-to-Peer Computing, P2P'09*, pages 222–231. IEEE, 2009.

[12] S. Voulgaris and M. Van Steen. Epidemic-style management of semantic overlays for content-based searching. *Euro-Par 2005 Parallel Processing*, pages 1143–1152, 2005.

[13] S. Voulgaris, M. van Steen, and K. Iwanicki. Proactive gossip-based management of semantic overlay networks. *Concurrency and Computation: Practice and Experience*, 19(17):2299–2311, 2007.