Tariq Elahi—Ed. (KUL)
George Danezis (UCL)
Panos Louridas (GRNET)
George Tsoukalas (GRNET)
Aggelos Kiayias (UEDIN)
Harry Halpin (GH)
Benjamin Weggenmann (SAP)

Initial requirements, design, and prototype

Deliverable D4.1

October 31st, 2016
PANORAMIX Project, # 653497, Horizon 2020
http://www.panoramix-project.eu

Horizon 2020
European Union funding
for Research & Innovation

Revision History

Revision	Date	Author(s)	Description
0.1	2016-05-30	TE (KUL)	Initial draft
0.2	2016-06-07	TE,RG (KUL)	Final editing and review
1.0	2016-06-07	AK (UEDIN)	Final version and submission to the EC
1.1	2016-10-16	TE (KUL)	Revision after first periodic review
1.2	2016-10-19	TE (KUL)	Restructure document after consultation with expert
1.3	2016-10-19	HH (GH)	Input for messaging use-cases
1.4	2016-10-19	PL (GRNET)	Input for e-voting use-cases
1.5	2016-25-19	PL (GRNET)	Review and comments.
1.6	2016-26-19	HH (GH)	Review and edits to background, messaging, and requirements sections.
2.0	2016-10-31	AK (UEDIN)	Revised final version and submission to the EC

Executive Summary

This deliverable details the initial requirements, design and prototype of the PANORAMIX system. First we describe the requirements process for our mix-net system and the three use-cases, namely e-voting, surveys and messaging. Then, we provide the initial design, the preliminary API and we finish the document with an overview of the development tools and practices.

We note that this is one of the deliverables that required significant effort in order to achieve convergence of opinion amongst the consortium partners. Mix-nets are complex objects and while previous works provide various high level and theoretical abstractions, producing a unified document that encompasses the various different aspects and use cases of mix-nets was a challenging task. However, we feel that we are on track with this initial iteration of the requirements, design, and prototyping plan to reach the high-level goals of this work package.

This document identifies three key ingredients to our strategy: (i) decomposition of the PANORAMIX design into the core functionality of the common architecture as well as the functionality of the three use-cases found in chapter 2, (ii) a modular design to the core platform that will accomodate a wide range of properties, and hence use-cases described in chapter 3, and (iii) an agile development methodology with close integration with users for quick turn around time described in chapter 4.

While the details of specific requirements, design choices, and API will be refined as the project progresses and captured in subsequent documents, we anticipate that the adopted overall strategy will keep us within the bounds of the project high-level objectives.

Contents

1. Introduction

The primary aim of the PANORAMIX project is to make progress on the state of the art of mix networking in terms of their privacy and security properties, ease of use, and wide-spread real-world adoption. These outcomes are to be realised through a common framework that will be publicly available to the community.

This document outlines the initial requirements, design, prototyping plan for the PANORAMIX project. In chapter 2 we describe our requirements methodology and use it to arrive at a set of objectives and requirements (the Requirements Specification). Chapter 3 discusses the design space and the use-cases that we will focus on (Design Document). Chapter 4 provides the initial version of API of PANORAMIX code base (under development in WP , while chapter 5 provides details of the developmental methodology, tools, and prototyping plan that will be followed (Implementation).

It is expected that this document will be revised as project partner knowledge and practice grows. As outlined in the Description of WP4, each year there will be an update of the document as outlined, with an updated list of Requirement Specification, an updated Design Document that shows how we met the requirements, an Implementation that meets as many of the requirements (as possible, given research and work constraints) based on the latest iteration of the design. Since this is the first version of the deliverable, there is no Task Progress Report, but each new version will include a Task Progress Report to clearly show how progress is being made on the core PANORAMIX mix-net infrastructure.

This work is the initial version that collects the requirements of the core-infrastructure that *all* use-cases will depend on. Individual use-cases and requirements that differ between use-cases are done in each work package deliverable for that use-case. In terms of WP6, the privacy-enhanced statistics specific requirements and use-cases from SAP are protected by customer confidentiality agreements, as are their own internal requirements in terms of their private cloud and the systems administrators thereof. For messaging-specific use-cases and requirements, see Deliverable 5.1 *D5.1: Requirements and User Interface Design* for e-voting specific use-cases and requirements, see Deliverable *D7.1 Applying mix-nets to Email Document* for messaging specific use-cases and requirements. Thus, the purpose of this deliverable is not to aim at the requirements for every specific use-case, but to aim at the use-cases and requirements for the core infrastructure they will all share in terms of mix networking. In this regard, the aim of this document is to consolidate the initial requirements from mix-net operators, developers of mix-net enabled applications, and the administrators of PANORAMIX-enabled installations.

2. Initial Requirements

2.1 Process

There are two different traditions in computer science and engineering where security engineering is addressed. The first is where security is a *non-functional requirement*, i.e. security in software engineering, where security is not the goal of the system under design [Lau02]. The second is where security is a functional requisite, i.e. security in security engineering, where security is the goal of the system [FSK11].

The requirements and design process in the software engineering case typically proceeds by identifying stakeholders, their system goals, the resultant requirements, finally leading to a system design. In contrast, in the second case of security engineering, the requirements and design process typically proceeds by identifying security and privacy properties that we want to enable, identifying threat models, adversaries and their capabilities, and finally a design (and sometimes security proofs) that addresses all of the above.

It is important to note that the provenance of the requirements are coming from both of cases: The first in which the stakeholders are given priority, and the second case where the system security properties are given priority. In the first case, it is the business logic and users of the system that drive the core functionality of the system (recall that security is not a core functionality here). In the second case, it is the security experts, the academic literature in the field, the existing protocol designs, and standards that define much of what the security requirements should or ought to be.

This contrast highlights an important point which is that the very people who may be the actual end-users of security systems may also not be fit to provide input about aspects of the system such as security and other highly technical features. This is because the end-users, not being experts, are unable to articulate their security requirements or possibly appreciate the impact of their choices on their own (often implicit) security and privacy assumptions as well as the system in general [WT99, SBKH06]. This is the reason why we elicit requirements from expert sources in matters of security and the highly technically-skilled system operations who will be evolved in PANORAMIX deployments.

Adhering rigidly to either process can lead to undesirable outcomes; such as systems that are deployed and used widely but that are not secure and a patchwork of fixes to vulnerabilities that are exposed daily, see e.g., [SFK+12], or systems that are secure but so user-unfriendly that they are never fielded, as is the case for many modern mix-net messaging systems being produced out of research [VDHLZZ15]. Again, security properties cannot simply be assumed, and one of the reasons for the failures in e-voting deployment is precisely the lack of thinking through the security properties of each component. As put by Ron Rivest, " Almost all proposed cryptographic voting protocols *assume* that a voter has a secure computing platform that will faithfully execute her portion of the protocol." Since we want both usability and high security for

PANORAMIX, our requirements analysis will have to take on board both security requirements and more traditional requirements engineering.

PANORAMIX seeks the middle ground between these two methods and we propose a hybrid approach by picking the best from both the above traditions. Our proposed approach is as follows:

1. Identify the overarching goals and objectives of the PANORAMIX project. These justify the hybrid approach since we want secure systems that are also easy to use and deploy.
2. Identify stakeholders or sources of requirements provenance. In this case, the main stakeholders will be mix designers, application designers that use mix-nets, and systems administrators. Note that there are some of stakeholders who while not directly evolved in designing or using mix networking, are important even if they never actually touch the system, e.g. outside security experts, standards, laws, etc.
3. Review the state of the art, and how it is deficient in addressing the overarching goals of the PANORAMIX project.
4. Identify the high-level requirements from the goals, keeping in mind the disparate sources of requirements coming from the different use-cases and the high-level properties that are needed *in general* by each use case.
5. We then drill down from the high-level requirements to requirements that are specific to the identified use-cases of e-voting, surveys and statistics, and messaging. These requirements are elicited from interviews with technically skilled mix-net designers, systems administrators, and application developers. Note that we to not present consumer (i.e. end-user) requirements, but the requirements from the designers, application developers, and systems administrators in our partners, as their end-user requirements are detailed in other deliverables (D5.1 and D7.1). It is from interviews with these groups that the requirements in this work package are derived.
6. Finally we analyse the requirements to arrive at a systems-level design in terms of its scope, principles, and dimensions. The aim of this analysis is to better address the needs of fulfilling the overarching goals of the project as well as fulfill the requirements of the individual use-cases.
7. The design dimensions allow for evaluating and analysing conflicts that arise between requirements, oftentimes called trade-offs, and a natural requirement prioritisation mechanism. We collect the trade-offs that have occurred so far and discuss them and their resolution via an approach that allows the core PANORAMIX system to be extensible by addressing common requirements first and allow extensibility so that it can be adapted to requirements that differ between use-cases.

2.2 High-level Project Objectives and Requirements

The PANORAMIX project is concerned with three main objectives: 1) **improving** the security and privacy properties of the state-of-the-art of mix networking, 2) promoting mix-nets **adoption** in general as well as for specific use-cases of e-voting, surveys and statistics, and messaging, and 3) making mix-nets **easy** to design, deploy, and use by non-experts.

In the above we have made the implicit assumption that an enhanced (PANORAMIX) mix-net is a viable solution to the security and privacy, adoption, and ease-of-use shortcomings of the current state-of-the-art for general and use-case specific scenarios. It will be the PANORAMIX

project's overarching goal to succeed where previous solutions have failed by creating a design that ensures that the deficiencies in security and privacy, adoption, and ease-of-use are addressed.

2.3 High-level Stakeholder Objectives and Requirements

There are three key constituencies of the community that the PANORAMIX project targets in particular: (1) designers and implementers of anonymity systems; (2) designers and implementers of systems that utilise anonymous communications; (3) administrators of infrastructure on which anonymity systems are run. A key goal for the common PANORAMIX framework, and code to support it, is to provide very clear and tangible benefits to these three constituencies. Therefore, in this section, we go through high-level hypotheses that we will test via interviews with the stake-holders identified in Section 2.6.

For the first constituency, mix designers, the benefit will exhibit itself in 1) the reduction of lines of code necessary, 2) the reduction in testing complexity and time, 3) the relative simplicity and cleanliness of the mix code, and 4) the provision of significant functionality.

For the second, the project will provide value through 1) the reduction in complexity to interface with the mix network, 2) a simple API and abstraction of all crypto and security critical configuration into safe defaults.

Finally, administrators will find a richness of the tools available for the full operational life cycle: 1) installation, 2) automated deployment, 3) configuration, 4) monitoring, 5) logging, alerts, 6) resource management and caps, 7) update, 8) packaging, and 9) deinstallation.

While not stakeholders *per se*, domain experts, the academic and industrial literature, standards that exist around securings online communications and the Internet in general, as well the existing software libraries and systems implementations are also sources of input. We shall identify particular sources of requirements when we discuss the core PANORAMIX functionality as well as the individual use-cases below.

However, it is instructive to first look at the state-of-the-art to inspire further discussion about requirements and designs which we do next.

2.4 State-of-the-art

The mix-net literature is deep and extensive. We conduct a survey of the state-of-the-art mix-nets that provide client and application programming interfaces (API) for message based mix network (or remailers as they are known in the context of e-mail communication in *cypherpunk* circles). We also review the API and client interfaces of Tor, The Onion Router, as an example of a mature project offering a clean interface to non-cryptographers and even non-programmers. It is our aim to draw upon and extract from these works the useful and effective features that will then be leveraged by the PANORAMIX framework in its API and design approach.

2.4.1 Mixminion

Mixminion [DDM03] is a Type 3 anonymous remailer protocol, supporting anonymous replies. Mixminion uses *Single-Use Reply Blocks* (or SURBs) to allow anonymous replies.

The Mixminion client provides interfaces to store and query information about known mixes,

generating path of remailers, encoding messages into packet bodies and creating SURBs, managing the sending and queuing process, decoding the received packets and reassembling the fragmented messages.

Mixminion supports three types of encoded messages: binary messages, anonymous reply messages (send to SURBs addressed to the sender) and single packets ("fragments") from large anonymous messages.

Two key resources to understand the command line interface (CLI) and API of Mixminion are:

- The mixminion user manual: http://mixminion.net/manpages/mixminion.1.pdf
- The mixminion client API interface (source): https://github.com/mixminion/mixminion/blob/master/lib/mixminion/ClientAPI.py

Generating SURBs For each forward packet, the user can create one or more single-use reply blocks associated with a particular identity (CLI `--identity`) so that others can reply to the sender's anonymous messages. Later, when you receive replies to a given SURB, you will be able to tell which identity the SURB was generated for. The user can specify the number and lifetime of the generated SURBs.

Message sending Mixminion breaks a message into packets, encodes the packets, and sends them into the mix network. To send a message (which can be read from a standard input or file) the user has to specify a destination address or use one or more SURBs. The user can send a message immediately (`-send`) or put the message into the queue (`-queue`) and the delivery is not attempted until you call of `mixminion flush`, which sends all the packets from the queue. The user can clean, control and inspect the list of messages.

Decoding The command `decode` allows to extract the contents of an encoded message. If the message is fragmented into multiple packets, the packets will be stored until the whole message has been received, at which point you can retrieve the original message with `mixminion reassemble`. If the packet is encrypted to a reply block, you will be prompted for a passphrase.

Servers and paths The user can specify the path of remailers (`--path`) for the packets or SURBs. By default, Mixminion picks, or recommends, a random series of mixes (around 4-5) as your path (specified in the configuration file). The user can decide to use a randomly chosen path, but can specify the number of servers in the path. The path can also consist of randomly chosen servers or specific servers determined by the user. The forward path has to have at least two remailers, whereas the reply or SURBs at least one. No mix appears twice in a row and the last mix is configured to deliver messages to the chosen destination. An unrecommended server is not used unless such a server is specifically requested.

Mixminion stores a configuration file **$HOME/.mixminionrc** which contains all its options. This file can be modified by the user to set up relevant options. In the internal files (which should not be ordinarily modified) Mixminion stores information about:

- secret keys used to decode messages sent to the SURBs. These keys are encrypted with the users passphrase.
- outgoing queued packets,
- metadata for a single queued packet,
- a packet that has been successfully delivered, and is waiting to be overwritten and removed,

- messages waiting reassembly,
- database of digests for SURBs which have been used already, to prevent repeat use. Entries are removed from this database once the corresponding SURBs are expired,
- recently downloaded directories (to speed up servers)

The Mixminion Client implements also the 'password manager' that asks the user for passwords as needed to decrypt resources, and remembers passwords that have already been asked.

2.4.2 Mixmaster

Mixmaster [MCPS03] is an anonymous remailer software implementing type 1 and 2 remailers[1], which allows users to send messages as emails anonymously. However, Mixmaster does not support replies or anonymous recipients. For transport, Mixmaster uses SMTP. Users and remailers own their MIXPASS - the passphrase used to protect your "nyms" and PGP keys.

The key CLI and API documents for Mixmaster are:

- The Mixmaster remailer user manual: `http://man.cx/mixmaster(1)`

Message sending The message, which we want to send is read from a standard input or from a file. If the message is read from the file, it is expected that the message will be complete with all mail headers. If the message is read from a command line, the sender can specify the destination address and other options via the command line. We can specify both a single destination address `--to` and the destination newsgroup `--post-to`.

Mixmaster allows the client to add subject and header line to the message header and attachments to the forwarded message. In terms of the received messages, user can send a single reply to a message or a group reply. The user can control the reliability of the messages to be sent by increasing the number of copies of the message which should be send. She can also increase protection against traffic analysis by sending dummy messages. The client can also configure the minimal reliability and maximal latency of the remailers which she wants to use in the chain. The **SENDMAIL** stores path to the sendmail program, which is responsible for sending the messages (the description can be found in http://man.cx/sendmail(1)).

Servers and paths The user can specify the chain of remailers or can use a chain of four random remailers selected by the Mixmaster. The information about available remailers is stored by Mixmaster is several files, containing list of all reliable remailers as well as a list of the remailers, which should not be used in randomly generated remailer chains. In a separate file are stored the public keys of the remailers. All of these files can be overridden by the users by setting the corresponding option in the configuration file `mix.cfg`. The configuration file stores all set options for the client and remailers. The file is set by default, however, the individual configuration file should be created when setting up a remailer.

The remailer actions, send and receive, can be triggered manually or automatically (`--daemon`). Each of the options might be overridden from the command line. The remailer can also generate a new key for itself or update other remailer keys. Each remailer key has some limited lasting time. By default it is set to 13 month, however the expiration date can be set manually. When the remailer key is expired, Mixmaster will continue to decrypt messages encrypted to an expired

[1] Refer to the survey for a taxonomy: Danezis, George, Claudia Diaz, and Paul Syverson. "Systems for anonymous communication." Handbook of Financial Cryptography and Security, Cryptography and Network Security Series (2009): 341-389.

key for `KEYGRACEPERIOD` period of time after the expiration. A new key will be generated and advertised `KEYOVERLAPPERIOD` period of time before the expiration of the key.

Mixmaster stores also several remailer files with configuration etc., which are read by each of the remailers if needed.

The **internal** Mixmaster files store information about

- Remailer public and secret keys (in separate files),
- Public Diffie-Hellman parameters used for El-Gamal key generation
- Public DSA[2] parameters used for DSA key generation,
- Message pool files and directory,
- Logs for already processed messages and for gathering statistics.

Cryptographic tools: Mixmaster uses the following cryptographic tools: SSLeay/OpenSSL cryptographic library, IDEA encryption algorithm, OpenPGP encryption format (Mixmaster does not call any external encryption program).

2.4.3 Sphinx

Sphinx [DG09] is a cryptographic message format used to relay anonymised messages within a mix network. Each message consists of a header and the payload. Sphinx provides both a clean cryptographic interface, and has also been implemented. However unlike Mixminion and Mixmaster it is a protocol definition, not a complete system.

The Key references that discuss the interface and API are:

- The sphinx code level API: https://github.com/DonnchaC/sphinx/blob/master/SphinxClient.py
- The sphinx paper has a function by function description, in Section 3, of inputs and outputs to build the mix format: http://www.cypherpunks.ca/~iang/pubs/Sphinx_Oakland09.pdf

Message sending To send a forward message, user first creates the encapsulated header and encodes a message by deriving all session keys, wrapping the message in multiple layers of encryption, and calculating the correct message authentication codes for each stage.

To create a header each user specifies the destination address and a sequence of remailers (path). First, sender generates a random element (satisfying the decisional Diffie-Hellman assumption), which will be used as a seed for encryption. Using this element each mix will derive a secret that is shared with the original sender of the message. Other keys used for encryption and MAC generation are extracted from this shared secret.

To avoid linking the messages traversing through the path, after each mixing step the element is blinded to make it indistinguishable from any other output element. Thus, the sender has to compute all blinding factors and following elements while creating the forward packet. Sender uses the computed elements to create all forward headers. Each header contains the element

[2]DSA refers to the digital signature algorithm, http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

and the encryption of next mix address, MAC and header for the next remailer. After creating the header, the sender prepares the encapsulated forward message.

Generating SURBs To build the anonymous reply, the intended receiver of the reply builds a mix header addressed back to herself with no payload. The reply-header is included in a message to give anyone the ability to reply.

To build the reply header to herself, the sender selects the remailer path and computes the sequence of shared keys and the mix header (as for the forward packet, but now the destination address is the same as the senders address). The shared keys are hashed using the random key and stored by the sender. Thus, when the sender receives the replay, she can look up in the table for the keys which will be used to decrypt the message. The information about the first mix node, the header and the key used for shared keys hashing is send to the nymserver over a secure channel, to be indexed under the sender's pseudonym (this can be done using: encryption with the nymserver's public key, signing with the pseudonym's private key, and sending the message to the nymserver using the Sphinx forward channel).

Decoding and routing After receiving a message, the remailer uses the element from the cyclic group and its private key to extract a set of shared session keys. Next, it checks the MAC of the message. Later, the header of the message is decrypted using the blinded elements and using those elements mix decrypts the payload of the message. The remailer extracts the routing information and next MAC and forwards the resulting message to the next destination.

When the receiver wants to send the replay, she attaches the replay-header to the replay message. This message is routed and processed in the same manner as the forward message, leading to simplicity of implementation and larger anonymity sets.

2.4.4 Tor

Tor is the most mature anonymization protocol based on onion routing—which is based on nested encryption to provide sender and recipient anonymity. [DMS04] Despite no mixing taking place at the onion routers (OR), the Tor protocol allows the flexibility of user selected paths over ORs, and thus it can be used as a basis for a good API and CLI for some type of mixing. Key references include

- The Tor controller spec: https://gitweb.torproject.org/torspec.git/tree/control-spec.txt
- The Stem tor controller library API docs: https://stem.torproject.org/api.html

Tor control protocol

The Tor control protocol (TC) is used by other programs (such as front-end user-interfaces) to communicate with locally running Tor processes. TC is a bidirectional message-based protocol in which the client (or controller) and server send typed messages to each other over the underlying stream (may be implemented via TCP, TLS-over-TCP or Unix-domain socket). For security, the stream should not be accessible by untrusted parties. Servers respond to client messages in the order messages are received. The protocol is loosely based on SMTP.

How does a controller tells Tor about a particular OR? There are four possible formats:

- Fingerprint -- The router whose identity key hashes to the fingerprint. This is the preferred way to refer to an OR.
- Fingerprint ~ Nickname – The router whose identity key hashes to the given fingerprint, but only if the router has the given nickname.
- Fingerprint=Nickname – The router whose identity key hashes to the given fingerprint, but only if the router is Named and has the given nickname.
- Nickname -- The named router with the given nickname, or, if no such router exists, any router whose nickname matches the one given. This is not a safe way to refer to routers, since Named status could under some circumstances change over time.

Configuration TC allows the user to run several commands to obtain the information about the configuration, ask for authentication or get information about the data that are not stored in the Tor configuration file, for example: policy, router status, network status, etc. Tor provides a few special configuration options for use by the controllers. These options can be set and examined by the SETCONF and GETCONF commands, but are not saved to disk by SAVECONF.

For more details visit https://gitweb.torproject.org/torspec.git/tree/control-spec.txt.

Authentication If the client wants to be authenticated, she sends the request to the server to authenticate. Before the client has authenticated, no command other than `PROTOCOLINFO`, `AUTHCHALLENGE`, `AUTHENTICATE`, or `QUIT` is valid. If the control port is open and no authentication operation is enabled, Tor trusts any local user that connects to the control port. There are two ways to authenticate a client:

- `CookieAuthentication` option - if is true, Tor writes a "magic cookie" file named "control_auth_cookie" into its data directory. To authenticate, the controller must demonstrate that it can read the contents of the cookie file:
- `HashedControlPassword` option - it must contain the salted hash of a secret password. The salted hash is computed according to the S2K algorithm in RFC 2440 (OpenPGP), and prefixed with the s2k specifier. This is then encoded in hexadecimal, prefixed by the indicator sequence "16:".

Managing circuits and stream The client can contact the server to build new circuit or to add new server to extend an existing circuit with that ID according to the specified path. When the request is to build a new circuit the controller has the option of providing a path for Tor to use to build the circuit. If it does not provide a path, Tor will select one automatically from high capacity nodes according to `path-spec.txt`. The client can also change the circuit purpose or close the circuit entirely. Moreover, the client can inform the server that the specified stream should be associated with the specified circuit. Each stream may be associated with at most one circuit, and multiple streams may share the same circuit. Streams can only be attached to completed circuits (that is, circuits that have sent a circuit status "BUILT" event or are listed as built in a GETINFO circuit-status request). The client can also close a stream or change the exit address on the specified stream.

Replies TC supports replies, which follow the same 3-character format as used by SMTP, with the first character defining a status, the second character defining a subsystem, and the

third designating fine-grained information. The replies inform about the success, failures, error etc.

Stem

Stem is a Python controller library for Tor. Stem supports the Tor relay and bridge descriptors, which contains the infrequently changing information about a Tor relay (contact information, exit policy, public keys, etc).

Authentication Stem supports the following authentication methods to ensure commands to the controller are genuine:

- NONE - No authentication required,
- PASSWORD - Password required, (as Tor's HashedControlPassword option),
- COOKIE - Contents of the cookie file required, (as Tor's CookieAuthentication option),
- SAFECOOKIE - Need to reply to a hmac challenge using the contents of the cookie file,
- UNKNOWN - Tor provided one or more authentication methods that are not recognise by Stem, probably something new.

For more detailed information API information visit the Stem Documentation https://stem.torproject.org/api.html.

2.5 Common mix-net platform requirements

We now identify the common requirements that the PANORAMIX platform needs to provide in general and for the use-cases we have outlined above.

Assumptions. We make the assumption that the different requirements of the three use-cases of e-voting, surveys and statistics, and messaging (which are discussed immediately following) provide the needed coverage of the design space. We assume that these use-cases will cover the features and requirements for many classes of applications that the project should be able to support. It is also assumed that the PANORAMIX framework will accommodate the complete life-cycle of a mix-net, from bootstrapping, to expansion or contraction, to decommissioning. The only dependence will be that infrastructure is available, *e.g.* servers and network connectivity between them, and that the operators of the mix-net have an appropriate level of access.

2.5.1 Privacy

mix-nets provide privacy for communications over insecure network channels. There are many mix-net designs in the literature some of which have been deployed, however, they all have privacy-enhancing properties in common. There are three types of privacy properties that the mix-net can provide: (1) unlinkability, (2) anonymity, and (3) pseudonymity. [DD08]

Unlinkability provides privacy against a third party network observer from learning if two particular parties are communicating with one another. Furthermore, it also ensures that an observer

is unable to tell if different messages are related to the same participant, in essence, preventing the profiling of users.

Anonymity provides privacy where one or both end-points of the communication channel learn the other's identity. Sender-anonymity and receiver-anonymity are two flavors of the above where the sender remains anonymous or the recipient remains anonymous, respectively.

Pseudonymity is a relaxation of anonymity where the participant desires only to protect their real-world identity, but wants to preserve their online persona, or profile, that is revealed to all observers.

2.5.2 Security

The principle adversary that mix-nets protect against is the global passive adversary (GPA). The adversary's aim is to learn the identity of either one or both of the participants of the communications that it observes. This adversary can observe all communications that flow into and out of the mix-net. The adversary is also expected to have access to auxiliary data sources and other background knowledge about its intended targets to facilitate the identification of the mix-net participants. In the presence of such auxiliary information (which may be revealing non-trivial information about the parties engaged) the objective is that observing the mix-net does not allow the inference of any additional information.

More powerful, active, adversaries can manipulate the mix-net to cause patterns of traffic to emerge and potentially leak information about the identities of the participants of the network. They may be able to control or corrupt some portion of the mix-net, either as operators of adversarial relays or through coercion of honest relays. They may also be able to inject, drop, and manipulate network traffic between participants and infrastructure of the mix-net.

The aim of the PANORAMIX framework is to address both types of adversaries to the degree that this is possible depending on the requirements of each application.

2.5.3 Core PANORAMIX library

A key goal is ease of use for developers. Therefore, it must hide as much of the cryptographic details from the developer as possible as studies have shown that cryptographic primitives are hard even for skilled developers to use correctly [EBFK13]. This means the PANORAMIX framework would provide at least the following functionality:

1. Offer a network-based API and client libraries for server and web browser environments,
2. Encapsulate from the developer and users raw cryptographic material (*e.g* keys, signatures, etc.),
3. Encapsulate from the developer and maintainer the cryptographic workflow (*e.g.* signing, verifying, de/encrypting, authenticating, etc.),
4. Streamline and encapsulate network management (*e.g.* establishing connections between network nodes, marshaling messages, resending, etc.),
5. Produce a turn-key mix-net bootstrapping mechanism (*e.g.* instantiating the mix-net for use),
6. Ensure security of the bootstrapping, operation, and maintenance of the mix-net (*e.g* access control for access to mix-net servers to enact configuration changes).
7. Expose some of its low-level communications functions to API client applications that can

then re-use those functions to implement protocols among their different actors without having to re-implement a separate communication stack.

2.6 Use-cases

We now describe the requirements of the three use-cases that the PANORAMIX project aims to enable.

2.6.1 e-voting (WP5)

E-voting requires a high burden of proof to be provided in order to verify that elections have been carried out fairly and with integrity. The e-voting use case involves Zeus [TPLT13], [3] [4] GRNET's e-voting system that was derived from Helios, a web-based open-audit e-voting system [Adi08, BGP11].[5] By using the technology developed in the PANORAMIX project, we seek to enhance the security, the performance, and the usability of Zeus, thus producing a more efficient generation of an already functional e-voting system.

Zeus has been in constant development in a very agile manner. When it was first deployed, it was as a requirement for legally binding elections in the governing bodies of universities in Greece. We adopted the Helios e-voting system as a starting point and the overarching requirement was to keep the overall experience and user interface as simple as possible, because most of the voters are not computer scientists. The first batch of 22 elections was carried out with the first version of Zeus. Input from users, that is, voters and election administrators, was mostly in the form of questions on particular aspects of the system. The vast majority of users had not had experience with real e-voting applications before Zeus, so there was no baseline for comparison.

In this context, we understood that Zeus had been a success because, first, all elections were carried out successfully, and second, users came back to us and asked us to use the system again, for future elections. When GRNET developed Zeus, the understanding was that it would be used for the first batch of 22 elections mentioned above, following which it would be decommissioned. The sole reason that it is in still in use, and its user base is growing, is because users asked for it.

Development and requirements elicitation progressed in a more organised manner from that point onwards. We have been eliciting user requirements in two ways:

(a) We maintain an organised helpdesk, where all user issues and questions are aggregated and answered. We triage the material arriving on the helpdesk and thereby elicit those issues that can be framed as requirements.

(b) Some users have periodically given us more detailed feedback, in the form of detailed lists of requirements and features that they would like to see in Zeus. These are always answered, and most of their suggestions are indeed implemented in Zeus.

As mentioned, Zeus development is very agile, which means that we are able to roll out new production versions in very short periods (from a couple of weeks down to a day). So, new features can be shown quickly to the interested users, who can verify that they respond to their needs. The helpdesk has an important role in the evolution, as they are the voices of the users to the development team. Note that in particular elections, members of our helpdesk sit along the

[3] http://zeus.grnet.gr

[4] http://github.com/grnet/zeus

[5] For more information regarding the requirements and user interface design of the e-voting platform, please refer to Deliverable 5.1.

election administrators and are therefore able to pass back to the development team feedback from first-hand on the field use of the system.

In terms of the design dimensions e-voting requires a mix-net capable of unidirectional message transmission, static routing and topology, highly robust and provides high levels of auditability, and can tolerate high latency.

Security

Verifiable Reliable Anonymous Private Messaging. Voting is essentially a procedure where voters send messages to the committee of trustees. These messages must be:

- **Verifiable** meaning that senders must be convinced that their message is delivered.
- **Reliable** meaning that exactly all sent messages will be received.
- **Anonymous** meaning that senders cannot be linked to the messages they submitted.
- **Private** meaning that only the recipient can read and publish the messages.

Making it easy for independent experts to write their own verification software helps create a strong argument and subsequent practice in favor of the service. The mixing component should facilitate this.

Distribution of trust An election or any other voting is as trustworthy as the committee of trustees that oversees it. Independent institutions, computer system administration, and all major competing interests in the election can be part of the committee.

Therefore, a mix-net must be easily composable by the contributing servers of parties that do not share infrastructure, or even good faith among them. The distributed parties must also be able to audit the mix-net operation.

Cryptography and Key Management The e-voting application requires proofs of the mixing integrity. This typically means a re-encryption mix-net must be used but alternatives may also be considered, especially if they are part of the same framework and API.

As part of the general verifiability requirement it is desirable that a vote can be proven to be included in the mix, and therefore in the counted results, without the need for full access to the proof data, for both security and efficiency reasons.

The mix-net must provide an interface for key generation, encryption, decryption, signing, and safe-keeping separate from the service runtime so that trustees can use them in their own controlled computers.

Performance

mix-net Computational and Storage Complexity Ideally, the computation of a mix proof and its verification would be able to process over 10 ciphertexts per second per CPU core, producing a proof that is no more than ten times the total volume of the mixed ciphertexts.

The computation and verification of proofs must be scalable to many CPUs and machines.

Available bit size for plaintexts Ideally, the bit size for messages should be unlimited, and configured at mix-net creation. 256-bits is an absolute minimum.

With a 256-bit plaintext a ballot can:

- Rank any subset of 57 candidates in order of preference.
- Provide a 53-character write-in vote in only 26 uppercase latin letters and space.
- Provide a 16-character write-in vote in UTF-16 format.

2.6.2 Surveys & statistics (WP6)

At SAP, the primary target for exploitation are SAP internal development units. Therefore, requirements were elicited from several internal stakeholders and owners of products that fit within the big data scenario described in WP6, including e.g. anomaly/threat detection and the evaluation of vehicle telematics data.

From there, we gathered requirements towards the PANORAMIX mix-net that support its easy integration and usage in a wide variety of SAP products. In terms of the design dimensions survey and statistics applications require a mix-net capable of unidirectional message transmission, static routing and topology, is highly robust and provides low levels of auditability, and can tolerate high latency. The requirements can be categorised as concerning the mix-net itself, and the sender and the receiver of messages.

Mix-net specific

- Setup/Deployment should be as simple as possible. (e.g. install on servers, store configuration file or run setup script and enter parameters).
- Keys related to the mix-net should be managed by the mix-net (i.e. Provide means for users to obtain necessary public keys, e.g. via API)
- Should be easy to install and run on common server systems (e.g. common Linux distribution on x86/amd64 hardware, with the possibility of running on other systems, such as Windows or Mac as well).

Sender specific

- Provide easy-to-use API to employ mix-net from sender
  - Basic function to have the mix-net send a message/record to a receiver
  - Provide ways to indicate if/ensure that a message has been delivered successfully, e.g.
    - Provide a “return code” indicating success or failure (in an effortless way)
    - Have the mix-net resend a message without acknowledgment
- Provide generic interface to interact with the mix-net, e.g. based on sockets
  - Interaction with mix-net should be straight-forward for major programming languages, e.g. Java, Python, and C/C++
  - Allow easy integration of mix-net into existing/common software frameworks

Receiver specific

- Provide easy-to-use API to employ mix-net from receiver
  - Receiving messages should be transparent (polling, etc., if necessary, should be hidden from user)

  - Allow replying to sender (bidirectional transmission; optional)
- Provide generic interface to interact with mix-net, e.g. based on sockets
  - Interaction with mix-net should be straight-forward for major programming languages, e.g. Java, Python, and C/C++.
  - Allow easy integration of mix-net into existing/common software frameworks

2.6.3 Messaging (WP7)

Messaging systems are a new frontier for mix networking, and an important one given the interest in meta-data protection. In terms of the design dimensions messaging applications require a mix-net capable of bidirectional message transmission, dynamic routing and topology, is reasonably robust and tolerates low levels of auditability and low latency.

These requirements were elicited from a group of systems administrators and open-source developers from various providers of secure messaging products. The coders and systems administrators involved in this discussion include: Elijah Sparrow (Thoughtworks), Micah Anderson (Riseup.net), Ruban Pollan (Greenhost), Nick Merrill (Calyx), Harry Halpin (Greenhost), Kali Kaneko (Greenhost), and Max Wiehle (Merlinux) as well as a representative from the standards community, Daniel Kahn Gilmour of the IETF. These requirements have received wide if informal review from the larger community that is hard to quantify over e-mail. Also members of the secure messaging community were e-mailed and discussed, as represented by the developer of OpenWhisper Systems Moxie Marlinspike, Trevor Perrin (independent), and Vincent Breitmoser of K-9 mail (an Android e-mail client). Vincent believed these requirements are generic to messaging and can work on mobile as well as desktop. Moxie Marlinspike and Trevor Perrin believe that e-mail is difficult and thus focus on secure messaging and they were interested in mix networking but remain more skeptical. Thus, the total number of people interviewed for messaging was eleven.

The composition of those people chosen for requirements elicitation is as follows: Of these, six were developers of applications that use mix networking Kaneko and Wiehle working on Bitmask, Vincent working on K-9 mail, and Marlinspike and Perrin on Signal. Merrill, Anderson, Sparrow, and Halpin were all representatives of possible PANORAMIX systems administrators. As he was interested in standards in terms of mix-nets, Daniel Kahn Gilmour was the only person who would could count as a mix-net designer.

- Users expect to be able to access their data across multiple devices with little delay and have the data backed up to redundant cloud storage.
- If key authentication is difficult, then there is low effective confidentiality for any user who might be subject to an active attack. Since existing systems of public key authentication for messages are either very difficult for users or require a central authority, the confidentiality of existing messaging systems is often low in practice.
- The system should be resistant to social network analysis (in particular, of metadata and routing information). In particular, this is where a general purpose mix-net is necessary.
- A client of the messaging service should be able to register a new device with their service provider. This process should make use of the authentication, key generation and establishment, and secure communication features of the mix-net.
- An email recipient should be able to receive emails to their inbox. The “last mile” step of retrieving emails to the user’s device may be carried by their email client.

- The client should be able to synchronise the data on the server-side across all of their devices. This functionality is not email-specific.
- A user should be able to send email to a user registered with the same service provider (different providers in a federated set up) using a mixing strategy described and as provisioned by the service provider.
- Allow for (or at least not hinder) the effective control of spam and email abuse.
- Clients do not need to be online in order to receive messages.

Messaging has a set of requirements that are the opposite of a traditional mix networking system. The messaging system will build on top of the open source email solution LEAP[6] that will solve the automatic public key authentication and high data availability requirements, as well as email delivery and registering new devices. PANORAMIX is necessary in order to deal with unmappability requirement, including the sending of email across providers that are both PANORAMIX-enabled and not PANORAMIX enabled. LEAP assumes a network of trusted providers. SMTP-based email is by itself a bidirectional protocol and the requirements of messaging require some degree of "low" latency, although not as low as in use with instant messaging. Due to the use of "To" and "Reply" headers, bidirectionality is absolutely required for messaging. LEAP does assume a network of trusted LEAP-enabled providers that can provide a higher-level of security than typical email providers, as given in detail by Elijah Sparrow of Thoughtworks. Thus, a static topology may be possible that maps to this trusted network of email providers, and this will make dealing with issues around spam and email abuse possible. This static topology may also allow static routing, although dynamic routing would be much preferred. However, as email is an open protocol and the network of trusted providers may change over time, a dynamic topology with dynamic routing is preferable. In this regard, messaging would prefer a BDD mix network.

The most important requirement from messaging is that *clients* who send and receive messages may not always be online. This requirement was intuitively obvious to Greenhost and also supported by other possible PANORAMIX-enabled providers such as systems administrators Elijah Sparrow and Micah Anderson. While traditional e-voting systems and previous approaches to mix networking assume that a client (the input and output node) is always online and we would expect providers and mix nodes to stay online, in real messaging systems users gain and lose connectivity very often. For example, a mobile phone may go into an area without coverage, or a laptop may be disconnected from wifi. While there are still some desktop clients that are always online, we would expect the majority of devices using the mix network to be mobile or laptop computers whose connectivity to the mix-net cannot be assured. It is crucial that this requirement be addressed by PANORAMIX. This requirement holds for the mobile case as well.

We would expect the network of trusted email providers to either themselves serve as or delegate an authority node and a membership node, and for each email provider in the trusted network to serve as a relay node, as well as being able to both input messages from their users and output messages. This is very different from a design where messages are sent into an entirely different mix network full of independent relays with a single input and output node. In a real-world scenario, these input and output nodes could be attacked. Nonetheless, a design similar to Tor may be suitable if there are many different relay nodes with multiple input and output nodes. Also, we a hybrid design that adds additional non-email provider relay nodes could also be useful if email providers are unable to devote the computational resources to provide PANORAMIX mix network functionality themselves.

Decentralization is one point of disagreement. Moxie Marlinspike noted that in order for Sig-

[6] *http://leap.se*

nal to use PANORAMIX, he is worried about decentralization over a number of entities that Open Whisper systems doesn't control due to issues of updatime and availability of (possibly unreliable) third-parties: "if the basic premise is that some other entity which won't collude with OWS needs to run a service forever that matches the same scale, uptime, and performance guarantees as Signal, that's probably not going to work." However, other systems administrators, such as Nick Merrill and Elijah Sparrow, were must more confident of the ability of decentralization to work. Sparrow even considered decentralization to be hard requirement, as he would not want a centralised entity to be subverted and so possibly damage privacy and security properties via backdoors that cannot be verified by independent third parties.

Mobile and Desktop Requirements

The first version of the messaging application, to be built by Greenhost on the LEAP software and involving the larger network of coders and providers who are preparing to use LEAP, will focus on Desktop use with the PANORAMIX e-mail client. This requirement comes from discussions with the systems administrators at Greenhost as well as other systems administrators such as Elijah Sparrow of Thoughtworks who are not affiliated with the project. Compabilitity with other clients such as Thunderbird clients may also explored, as discussions with end-users show this is a popular mail user-agent for checking email, although the codebase is officially abandoned by Mozilla. Desktop deployment has a number of advantages, including easy deployment of the fundamental Python libraries that LEAP is built from and a lack of metadata being revealed by the client, as some systems administrators and expert users (in particular, Elijah Sparrow of Thoughtworks) felt that a mobile platform was inherently insecure and privacy-invasive. In contrast, it is very hard for mobile clients to prevent leaking metadata around location, so we expect early privacy-conscious adopters to focus on desktop clients rather than a more mass-market mobile application.

On top of the email messaging use case, a third-party use case has been identified that can leverage the underlying messaging mix-net and use it as a platform for mobile messaging. The mobile messaging application will likely have to deal with even lower latency requirements as well clients being offline even more frequently than in the desktop email messaging case. In detail, Moxie Marlinspike (developer of Signal, a encrypted messaging application) noted that for Signal "the p75 for all messages is usually around 2 seconds, but when two parties are in active chat it's sub-second. Latency imposed by our server itself is p99 40ms. I think the 100ms to 300ms range would be something we could accommodate." In contrast, e-mail messaging requirements are hypothesised by Elijah Sparrow (and in-line with Greenhost) to be of the order of up to 5 seconds, with a larger possible delay. Elijah Sparrow also expressed concern that there may also be very specific issues to the platform, such as Android, the may arise in the course of development. This requirements come from discussions of Greenhost with mobile application developers Trevor Perrin and Moxie Marlinspike of OpenWhisper Systems, who work on the encrypted messaging app Signal on Android as well as designed the encrypted messaging systems of Facebook, Google, and WhatsApp, as well as earlier discussions with Mobile Vikings developer Raf Degens.

2.7 Conflict Between Requirements

In general, there is core consensus on the requirements in terms of privacy and security properties. E-voting has more of an emphasis on verifiability and anonymity, but both messaging and e-voting need privacy. Anonymity does raise concerns over spam in messaging, but a number of possibilities to deal with this are under development (See D7.1. for details). In particular,

also e-voting has much more well-understood computational and performance requirements. Messaging will likely need even lower latency, but this is still an open research question (see D3.1 work on Hornnet for one possible solution). There is a broad consensus on the privacy requirements between messaging and e-voting.

There are two points of conflict. The first is that messaging would need to allow the client to go offline, while in e-voting systems all nodes in the mix-net can be assumed to be online. Note that this is a *new* requirement for mix networking in general that was arrived at via requirements elicitation. Therefore, new research is needed to provide a solution. The second is the requirement for the mix network to be decentralised, i.e. dynamic. Note this requirement should not be easily ignored, as it was also phrased by the IETF representative Daniel Kahn Gilmour as being important for future standardization of PANORAMIX, as well as large-scale deployments for messaging like Thoughtworks. However, in general, this feature could hurt reliability and is technically more complex to deploy. Therefore, we are moving this feature to research in WP3 to be addressed in D3.2.

2.8 Legal Requirements

The legal requirements are of extreme importance. Although PANORAMIX is still in process of hiring a general Data Protection lawyer (currently we are in negotiations with Axel Arnbak of the firm de Brauw, Blackstone and Westbroek), we have initial requirements from the Data Protection authorities in Greece and in the Netherlands (via "Bits of Freedom" that work with Greenhost). We expect to formalise and detail these requirements in the next iteration of this deliverable, i.e. D4.2.

It is clear that the PANORAMIX mix-net infrastructure must obey both the existing national-level Data Protection acts, the Data Protection Directive (95/46/EC) and, more importantly, upcoming General Data Protection Regulation regulation (GDPR, EU 2016/679). Due to its nature as a privacy-enhancing technology that gives user's privacy and increased security when using applications such as e-voting and messaging, the project has assumed that an increase in user privacy would in general fulfill the Data Protection Regulation. In detail, we aim the design of PANORAMIX to be in compliance with the Regulation in the following manner:

- PANORAMIX by nature does not keep personal data outside of the nodes, as when it is on any member of the PANORAMIX network it is kept in an encrypted form and then deleted after shuffling and the messages are relayed. Therefore, *notice requirements* re private messages will only be have to be done by the messaging client and possibly the input and output nodes, on a per-application basis. The membership nodes will not have access to unencrypted data and retain even the encrypted data only for a short period of time due to latency requirements.
- The responsibility for gathering data for testing and infrastructure purposes will be explicitly communicated to users via an "opt-in" *consent* form that will be developed in co-ordination with a Data Protection lawyer in order to guarantee the requirements in Article 4 and Article 7 are followed.
- A Data Protection Officer will be assigned by mix-net co-ordinators and applications where this is deemed relevant. Note that the case of GRNET PANORAMIX-enhanced e-voting a data protection officer will be important to have, and the Greek government is already in good contact over this with GRNET. In fact, in the external advisory board, the PANORAMIX project has included Prof. Antonis Symvonis, a member of the Green Data Protection Authority. Greenhost will continue to work with "Bits of Freedom" and hopefully hire an independent Data Protection lawyer (i.e. Axel Arnbak, attorney at

de Brauw, Blackstone and Westbroek) to help with the larger PANORAMIX infrastructure. SAP already is investing in the legal infrastructure needed for compliance with the Regulation.

- *Data Breaches* (as detailed by Article 32) are always a cause for concern. In terms PANORAMIX, data breaches also can cause only limited damage to member nodes in the network as all data is encrypted and the nodes do not have the keys to decrypt the data, so therefore a breach at a single node or even group of nodes would not be a cause for concern. The only concern would again be output nodes, where data is decrypted. Yet since the messages are anonymised, it may be difficult for the output node to communicate with the sender that a breach has occurred. In WP3.2, we are developing new research to allow output nodes to maintain privacy but still inform the end-user if data has been output, and therefore this same technique could be used to inform the user of a breach at an output node. Breaches should also be handled by the end-user Messaging Client, but such a breach can by nature only effect an individual user, not multiple users as would be the case at a breach in any other part of the network.
- The *right to erasure* given by Article 17 is enforced by default in the network, as all data is deleted after it is sent. Messaging clients may keep data as may mix-net co-ordinators for authentication purposes, and therefore must comply to the right to erasure. This will be further detailed in WP4.2.
- As data protection requires *data portability* (given by Article 18), PANORAMIX is currently working with the IETF to use current open standards and if needed, create new standards. By virtue of having a single API that can be used by multiple code-bases, PANORAMIX should allow an end-user to move from one mix-net to another easily.

As all the use-cases are based in Europe (messaging in the Netherlands, SAP in Germany, and e-voting in Greece), all initial use-cases will fall within the scope of the Data Protection Directive as exists today and Regulation as will be adopted in 2018. In terms of the new regulation, a single-set of rules will help PANORAMIX, as we imagine the distributed nature of nodes in a mix-nets means that having a "one-stop shop" in terms of rules will make it easier for us to ensure that the network as a whole, which may cross European borders, will be compliant. We will work with Data Protection controllers and Supervisory Authorities to assure compliance as the project develops. This work will fall on the Mix-net co-ordinators who run any input and output nodes, as assisted by the mix-net verifier and the developer of PANORAMIX-enhanced messaging clients. This future work will be detailed in WP4.2, which will contain a larger analysis of the Data Protection requirements. We refer to Deliverable D1.4 for further discussion on legal aspects of the project.

3. Initial Design

3.1 Considerations for a General Purpose Mix-net

In order to accommodate the wide-range of applications and operational parameters (i.e. the requirements captured above) it is imperative that the PANORAMIX framework, through its API, be as general-purpose as possible, while also providing an advantage over the current state-of-the-art. As noted earlier, the PANORAMIX framework will be easy to design for, deploy and manage, and integrate with mix-net enabled applications. These goals will address the needs/requirements of the three constituents (designers, implementers, and deployers) that compose the target audience of the framework and promote real-world uptake and impact.

In order to accomodate these aims we now identify and describe the parameters that distinguish common mix-net operational and design scenarios, that will enable the PANORAMIX platform to meet the requirements established above.

3.1.1 Design dimensions

The mix-net API is designed to be generic enough to handle a range of different mix-net designs. We now identify the five main design dimensions that together produce distinctly different mix-net designs and scenarios.

1. **Direction:** This is the direction messages flow in the network. The possible types are bidirectional and unidirectional. Bidirectional networks allow messages to flow to and from the edges of the mix-net on the same connection. Unidirectional networks only allow messages to flow in one direction through the mix-net.
2. **Routing:** This is the high level scheme the network will use to route messages through the network. The possible types we consider are static and dynamic. Static routing means that messages flow between nodes according to the routes established at the time the mix-net is created. All messages flow through the network in exactly the same route(s), and intermediate nodes do not make any decisions on which direction to route the messages. Dynamic routing means that messages flow between nodes according to the routes selected at the time the message is sent (either by the sender, or intermediate nodes). There is no routing relationship between messages as they can all take different routes, and intermediate nodes may be called upon to make routing decisions on which direction to route the messages. The key distinction is that static routing is set up at the time of network creation (or first use) and dynamic routing is set up at the time the message is being sent through the network.
3. **Topology:** This is the shape of the network in terms of the network links between the nodes. The possible types are static and dynamic. Static topology allows direct communication between nodes as decided when the network is created (or time of first

use). Nodes can be set up into many shapes (e.g. star, clique, mesh, etc.) but once these have been defined, there are no changes possible during run time; this implies that new nodes can not be added to the network during run time. Dynamic topology allows direct communication between nodes as decided during run time.Here, new nodes can be added during run time.

4. **Robustness (Mixing strategy):** This is the rule set that each individual relay in the mix-net applies to every unit of traffic that transits through it. The mixing strategy can be complex, or simple, with the express aim of disrupting any traffic patterns that an adversary can observe to attack users. The higher the obfuscating potential the more robust the design is to privacy and security violations. While there are many mechanisms, their effect on traffic patterns can be abstracted to timing and order.

    - **Timing:** Messages are held in a queue, buffer, or similar storage data structure for some time period before it is sent on to the next destination. The intended effect is that the observer is not able to correlate a message as it enters a relay and as it exits the relay based on time. This timing may be fixed or can be governed by the size of the queue, the rate of incoming messages, or other events. Timing is always additive, meaning that the mix-net can never output a message faster than a conventional network would.

    - **Order:** The order of incoming messages is permuted before being sent to the next desintation. The intended effect is that an observer is not able to correlate messages based on their ordinal position from the incoming and outgoing traffic. Again, there may be a simple fixed permutation, or a more complex operation (for example based on a randomizing element), to reorder the messages and obfuscate any order that the observer could detect. The implication here is that there must be a (sufficient) set of messages for the permutation to act on to be effective.

5. **Auditability:** Auditability refers to the ability of a mix-net to provide evidence that its operation is according to specifications. This evidence usually comes in the form of auxiliary information that is produced during the mixing process and is stored in a publicly accessible service. In the literature this usually is described as a public bulletin board, formally an append-only communication channel with memory that any participant can access freely. Auditability entails two different aspects: verifiability and accountability. Verifiability means that a one of the participants or third party can check if the mix-net has operated according to the defined mixing strategy. Accountability means that if there is any wrong doing in the operation of the mix-net then the misbehaving node can be identified. Systems can provide high and low levels of auditability. Highly auditable systems provide both verifiability and accountability, while low systems do not provide either of these means.

6. **Latency:** This is the time lag introduced as a consequence of the mixing strategy at the mix-net relays. Depending on the particular timing and order obfuscation mechanism employed this latency may be high or low relative to a conventional network without mixing components. This measures the performance of the mix-net as experienced by the end-users of the mix-net enabled application. This dimension needs to be tuned to the end-user expectations in order to be acceptable.

These dimensions can produce many different types of network flavors, such as a unidirectional, dynamic routing, static topology mix-net for example. However, not all type combinations produce naturally useful networks, such as networks with static routing coupled with dynamic topology, which seems to be a contradictory combination.

The project partners, in discussion, have decided that looking at the extreme cases will allow

for the API to be able to, at least in principle, accommodate a wide range of useful and likely to occur mix-net designs.

One extreme is the unidirectional, static routing, static topology (or USS for short) mix-net and on the other extreme is the bidirectional, dynamic routing, dynamic topology (BDD) mix-net. As a process to hone in on an appropriate API for the PANORAMIX API we will ensure that it can accommodate these two extreme network types. Also, as a consequence of this set up, we recognise that the USS version is a simpler instance of the more robust BDD version. This means that we should design the API to the BDD network type, and ensure that it still makes works for the USS network type.

While a unidirectional mix-net type can accommodate all of the use-cases, a bidirectional variant will provide additional benefits in the messaging use-case (WP7). We prioritise the USS type over the BDD type where conflicts may arise as the code-base is developed and tested.

3.1.2 Privacy-performance trade-offs and conflict resolution

We now qualitatively describe conflicts between the requirements in refernce to the design dimensions above and how they affect the privacy and performance properties of mix-nets. Our aim is to identify potential conflicts and reasonable resolutions in this early phase of the project.

Robustness versus Latency

Recall that the robustness of a mix-net depends on the mixing strategy that it adopts and has an effect on the timing and ordering of messages. A weakly robust system does not introduce a lot of delay between a message being received and sent onwards nor does it attempt to permute the message order (since there may not be sufficient messages to permute). The performance in such a setting would be considered high, since there is almost no difference between the latency in the mix-net as compared to a conventional network. However, the observer will not have a difficult time in correlating incoming and outgoing messages and thus linking the communicating parties together, and perhaps also learning their identity. A strongly robust system introduces substantially more delay between message arrival and departure and also uses a more complex permutation to obfuscate the ordering of the messages, which itself may also introduce a delay due to a computational bottle-neck. Now, the observer has a more difficult time in correlating messages which raises the robustness, hence privacy, of the system, but at the cost of higher latency and lower performance.

To resolve this conflict we shall allow the use-case of e-voting to prioritise robustness over latency while the messaging and surveys and statistics use-cases shall prioritise latency over robustness.

Auditability, Privacy, and Latency

Auditability as a property comes in direct conflict with privacy, since verifying that mix-net servers properly forward the incoming traffic depends on the operations that are performed in processing the traffic (that includes batching, decrypting and reordering the messages that are submitted). To mitigate, but not entirely remove the private information leakage, privacy-preserving tools have been utilised in the literature to facilitate auditability such as zero-knowledge (ZK) proofs of shuffles. More specifically, a non-interactive ZK proof in general is a specially crafted string that in the context of certain other auxiliary information ensures that a certain communication transcript conforms to publicly known, pre-agreed and efficiently computable predicate. However, these methods come at the cost of higher latency, since nodes

need to compute and transmit sometimes large amounts of data that scales with the total number of nodes in the mix-net.

To resolve this conflict the e-voting use-case shall prioritise Auditability over privacy and latency concerns.

Unidirectional versus Bidirectional

Unidirectional message delivery maintains all of the privacy properties of pseudo-/anonymity and unlinkability. Since there is no way to send a message back to the sender, it is also impossible to learn anything more than what the mixing strategy already allows. In contrast, it is more onerous to enable bidirectional communication between two parties. This is because in order to send a message back the sender, who wishes to retain their anonymity, must provide an address where the message is to be sent. This must be carefully done in order to prevent an observer being able track the reply back to the original sender. Also, the return address must not reveal the identity of the original sender. These considerations lead to a more complex message structure, as well as network addressing scheme that can introduce overheads that cause latency to increase and potentially cause a loss of privacy. However, the bidirectional channel can enable all sorts of interactive applications. Indeed, the vast majority of applications on the Internet are bidirectional.

Globally, the PANORAMIX project will prioritise unidirectional traffic over bidirectional since we have established that all of the use-cases are served by unidirectional traffic flow. Only the messaging use-case will additionally benefit from bidirectional traffic flow.

3.1.3 From general framework to specific use-cases

We now illustrate how the specific use-cases fit within the dimensions of the PANORAMIX framework we have thus far described. This is useful to see since the use-cases (1) have distinct privacy and performance envelopes, (2) showcase the wide range of applications that the PANORAMIX framework can support, and (3) are relevant to the needs for privacy-preserving communications today. These use-cases will be developed into applications and are outputs of work packages 5, 6, and 7, respectively.

Table 3.1: Comparison of use-cases across the dimensions of robustness, latency, and direction.

	Robustness	Latency	Directional	Auditable
E-voting (WP5)	High	High	Uni	High
Surveys and statistics (WP6)	High	High	Uni	Low
Messaging (WP7)	Low	Low	Bi	High

Table 3.1 provides a comparison of the use-cases across the dimensions of robustness, latency, direction, and auditability. We see that messaging is very different from the other two providing a polar opposite set of characteristics to the other two. The distinction between e-voting and surveys and statistics is that e-voting needs to be auditable.

These conflicts can be resolved by building an initial system that follows the prioritization listed above. In particular, the core PANORAMIX infrastructure must have high latency and high robustness, as high latency and robustness are needed for e-voting and statistics, but do not harm messaging. They only harm messaging if they come at a cost of not allowing messaging clients to go offline. However, if the messaging clients use a traditional client-server infrastructure (as the e-mail use case in D7.1 indeed follows), then this requirement can be

addressed by allowing clients to not have the same properties as servers. Therefore, the servers in a messaging mix network can be stable and use a USS mix-net, and a BDD style network with the ability to dynamically add new servers and for servers to dynamically add clients can be added to the core later. This will be reported on in future WP4 deliverables (WP4.2 and WP4.3) and since it involves hard research questions, these questions will be tackled in WP3's future deliverables (D3.2 and D3.3). Likewise, as auditability is only a requirement for voting, it can be added later as an optional feature or package that does not need to be used by the "core" infrastructure also used by surveys and messaging. In conclusion, the requirement conflict resolution determines that there will a core USS high latency and high robustness PANORAMIX infrastructure that can be *extensible* to take into account the auditability of e-voting and dynamic nature of messaging.

3.2 Proposed General Mix-net Framework

We now describe the components of a general mix-net framework that allows one to realise many possible mix-nets as defined by the dimensions we have described above. We emphasise on the general-purposeness of the design we propose, since this approach allows the most flexibility and chances of success in meeting all of the requirements and needs of the stakeholders. We decompose the mix network design into: 1) the types of network nodes present, and 2) the user roles that will interact with it.

3.2.1 Network nodes types

We have distilled the types of network nodes into their potential roles. In order to keep the design as generic as possible, we assume that the each role may be undertaken by a single node, or many. Also, a node might undertake one or more of the roles. Figure 3.1 depicts a typical arrangement of the nodes for a general mix-net.

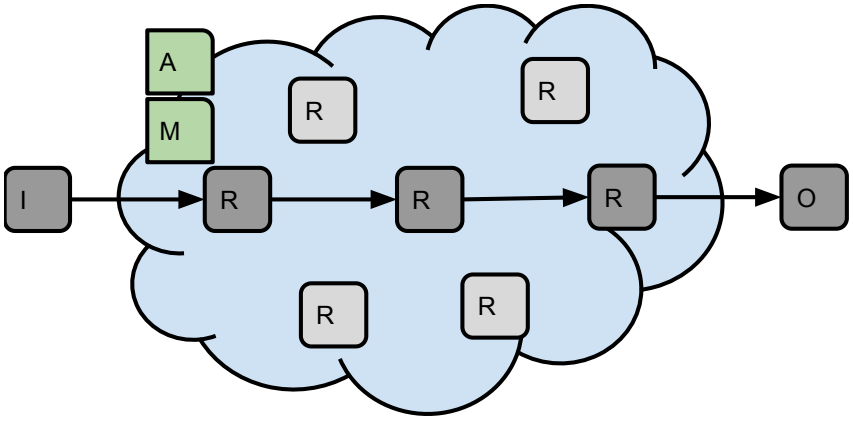

Figure 3.1: A general design for a multipurpose mix-net.

We have identified the following five types:

1. **Relay node [R]:** This node performs the mixing operations. Its input is (list of) message(s) and the output is a mixed (list of) message(s). The input can come from other relays or an Input node (which is described below). The output is to other relays or to the Output node (which is also described below). A special case of the relay node is the Entry node [E] which is the first hop of the mix-net. The Input node sends messages to

the Entry node. Another special case of the relay node is the Exit node [X] which is the last hop of the mix-net. The Exit node sends messages to the Output node.

2. **Input node [I]:** As input this node receives messages to send into the mix-net from the application that uses it directly. It may perform house-keeping operations, but these are to be kept hidden from the users/application. It outputs the messages to the relay(s) in the mix-net.
3. **Output node [O]:** This node receives the final output of the mix-net. We assume that this is the final state of messages and that the application must now handle message delivery to the end user.
4. **Authority node [A]:** Trust is a key component in the setup and operation of the mix-net. Authoritative nodes provide that trust basis. They will provide information about the network, and other operational objects, that allow the nodes to operate in a secure manner. This node might not be present where trust is implicitly placed on the mix-net and the maintainer.
5. **Membership node [M]:** Since mix-nets are open to abuse, we anticipate that they may be gated and only authenticated users/apps will be able to use them. The membership node provides the mechanism for issuing credentials as well as validating credentials. Other credential related operations are also provided (e.g. revocation, reissue, generate new, etc).

3.2.2 User roles

Four roles have been identified:

1. **Mix-net maintainer:** This is the entity that sets up the mix-net and may also operate the physical devices that compose the mix-net although this is not mandatory. They need not be a user of the mix-net in any capacity.
2. **App developer:** This is the entity that develops applications that will make direct use of the mix-net that is set up by the maintainer.
3. **App extender:** This entity develops application that will make indirect use of the mix-net with another app that directly interfaces with the mix-net, which was produced by the app developer above.
4. **App user:** This entity makes use of the applications that the app developer (or extender) produce.

Mix-net maintainer

In order to attain the high-level goals where the details are encapsulated from the user, for the mix-net maintainer it is decided that a wizard-driven setup makes sense.

Mix-net configuration and initialization

In the static case the process to setup the mix-net can be facilitated through a wizard that guides the user to pick the appropriate settings such that upon completion the correct configuration files are produced.

For instance the maintainer may specify the following:

- The number of servers and their individual IP/DNS. $[E] \rightarrow R_1 \rightarrow R_2 \ldots R_m \rightarrow [O]$.

- The output is a set of configuration files to be executed in the respective servers.
- The membership server $[M]$ has a master secret-key and can generate membership secret-keys for applications and a group public-key. It may also be able to revoke such keys by appropriately changing the group public-key that may contain a revocation list.
- The input point server $[I]$ can authenticate applications and collect inputs for the mix-net.
- The exit node server $[E]$ receives mixed values and forwards them according to instructions.

The effect of this process will be to setup the mix-net and provide the maintainer pointers (e.g. URIs) to the entry points and exit points of the mix-net that can be provided to applications.

A key consideration is how is the cryptographic material is managed and kept secure.

Maintenance

Mix-net Maintainer uses an interface to control mix-net (e.g., the master secret-key controls, membership list, revocation of application keys).

App developer and extender

The app developer will be provided access to the mix-net for testing purposes during the development phase and then once the application goes live, access will be granted for actual use. The access controls are not fleshed out as yet. Points of consideration are if the app itself will manage access rights, or will it delegate the control to the mix-net maintainer. In either case it will need to provide this function in a secure and privacy preserving manner.

App user

The user of the mix-net enabled application will not directly interact with the mix-net, and indeed may not even be aware that a mix-net exists on the back-end. As such, the app user is no expected to affect the functionality of the mix-net, other than indirectly through message volume and timing of when messages are sent.

4. Preliminary API

4.1 Mix-net API architecture

Having described the design of the general-purpose mix-net in the previous chapter, we now describe the API that will support it. We separate the mix-net interface into two categories, messaging and mixing. This is a natural split since ultimately these two functions are the core of any mix-net.

4.1.1 Messaging Interface

A mix-net's ultimate goal is to anonymously deliver messages to known recipients, i.e. hide the sender identity. When everything is set up, this is a simple interface. PANORAMIX intends that the default capabilities of its framework will allow application developers and operators interact only with this simple Messaging Interface.

The messaging interface is just concerned with the involves

4.1.2 Roles and their Components

Role	**Component**
Coordinator	Controller
Contributor	Mix Server
Correspondent	Messaging Client
Auditor	Mix Verifier

Mix-net Coordinator The mix-net Coordinator is the person responsible for the operation of the centralised infrastructure of the mix-net. They are the ones who allow applications to use the mix-net, or allow mix nodes to participate in the mixing.

Mix-net Controller In order to create new mix-nets, the Coordinator needs to start a mix-net Controller server. The Controller acts as a registry for mix networks. For each mix-net, a unique URL is registered which serves as a single point of contact, were both mix nodes and clients connect to.

The Coordinator is responsible for advertising the mix-net URL in order to bootstrap the network.

Mix Contributor Every person that wishes to take part in the mix-net, must set up a Mix Server, initialised by the mix-net URL and suitable parameters.

A mix contributor is part of the trust network of the application. Users rely on Contributor's honesty and compliance, to not reveal or otherwise exploit any knowledge of the senders' identities.

Therefore, it is important that Contributors are institutionally independent from each other and that their Mix Servers belong to separate administrative domains.

Mix Server Upon initialization, a Mix Server creates an encryption key, if not provided, whose private part is known only to the Contributor. The public part is registered with the Controller.

Correspondent Correspondents send and receive messages through a mix-net with a simple interface using Mix-net Client software. Cryptographic and communication details are hidden from the correspondents, who rely on the application and mix-net experts to set everything up.

Messaging Client A Messaging Client is initialised with a mix-net URL and appropriate parameters. Upon launch, it offers a local REST gateway with a simple send/receive interface. The Messaging Client handles all cryptographic and communication internally.

Because Messaging Clients receive plaintext communications, they must be part of the Correspondent's local infrastructure. For example, within the same computer, or an institution's network.

4.1.3 Mixing Interface

The Mixing Interface is a lower-level interface that expresses all necessary details for initializing cryptographic, network, and storage functions.

First of all, the mix-net architecture and algorithms have to be expressed in the Mixing Interface language. This common language will enable the same mixing solution to be used from multiple applications, and will allow the same application to be deployed over mixes with different qualities. Furthermore, it makes it easy to incrementally tweak and enhance mix-net solutions.

The first application to be built on top of the Mixing Interface is the Messaging Interface itself. PANORAMIX aims to support as many common cases as possible within the Messaging Interface, so that applications can easily deploy and use anonymous communications without the need of security and crypto experts.

However, some applications will need be developed by experts and will need more control over the mix-net operations in order to closely integrate mixing into their application protocol, and even program extensions to the mixing algorithm itself.

Expert application developers will still need to deploy and connect to the basic components of Controller, Mix Server, and Messaging Client, only now they will have access to a much more detailed API.

The application will be responsible to make everything work together. The Messaging Interface provides a default way to run things, and applications may use it as a starting point.

4.2 Internal API

The Work Package pulls technologies from WP3 to build a product that may be customised to serve the purposes of the use cases of WP5, WP6, and WP7. That is, the mix-net in PANORAMIX must support different use cases involving different technologies and configurations.

It would also need to be simple and robust enough so that it can be deployed in production and attain a lifecycle beyond the end of the project as a positive contribution to the community. It is expected that new extensions and use cases will be added during its lifetime.

In order to support this internal development, and to additionally offer more control to advanced and demanding applications, an internal API has been designed that abstracts mix-net operations.

This interface brings three system layers together:

The production service machinery which includes server software, tools for deployment, configuration, monitoring, and scaling.

The message transport which allows cryptographically secured communications among all nodes (controllers, contributors, clients, etc.) It is expected that different use-cases will require different transports (e.g. e-mail, AMQP).

The cryptographic workflow which implements the actual mixing algorithms working with generally structured data, oblivious to application specific details or communications concerns.

The following sections describe a high-level design of the API.

4.2.1 REST Introduction

We model the intermediate interface REST API, meaning that there is a hierarchical namespace which catalogues collections of resources. E.g.:

`PANORAMIX-index.eu/peers_by_authority/EU/GR/GRNET/df6b80...dc1e150`

Each collection hosts resource entities of the same type. Each resource entity has a number of attributes and accepts reading and writing on these attributes. E.g.:

```
panmix info PANORAMIX-index.eu/peers_by_authority/EU/GR/GRNET/df6b80...dc1e150
{
  'resource-type': 'peer',
  'resource-origin' 'grnet.gr/panmix/peers_by_id/df6b80...dc1e150
  'peer-id': 'df6b80...dc1e150',
  'peer-owners': ['5891b5b...522d5df0', ...]
  'peer-inbox': ...,
  'peer-outbox': ...,
  'peer-cycle': 177,
  'peer-type': 'ZEUS-MIXNET-V3',
  ...
}
```

Resource entities may accept type-specific action verbs that trigger one-way state transitions, such as a peer mixing its inbox:

```
panmix alias @grnet PANORAMIX-index.eu/peers_by_authority/EU/GR/GRNET/df6b80...dc1e150
panmix action MIX @grnet ...
```

For simplicity and generality we will skip anything HTTP-specific since it's just a technicality and focus instead on what the resource types are, what are their attributes, their actions and their semantics.

It is important to note that although the namespace can be indexed down from a global entry point, subnamespaces may linked into it and be actually hosted elsewhere. Moreover, important processing is not actually performed by the namespace service, but by accessing clients.

4.2.2 Resource Entities

The internal API has the following resource types: peer, key, peer-owner, peer-cycle, peer-inbox, peer-outbox, message, negotiation, contribution, consensus.

Peers

The peer is the basic entity in the interface. It is controlled by an owner set of persons or other peers by means of a cryptographic private key shared among the owners.

The peer has an inbox and an outbox. The inbox receives messages encrypted with the peer's public key. The messages are not processed immediately but they pile up for a while and then processed in scheduled intervals called cycles.

At the end of the cycle, the inbox is processed in its entirety performing one of the following actions:

1 CONSUME: All messages in the inbox are decrypted and consumed by the peer. The consumption may place arbitrary messages in the outbox.

2 PROCESS: All messages in the inbox are shuffled, decrypted, and placed in the outbox.

3 DECRYPT: A PROCESS but with a null shuffle.

4 SHUFFLE: A PROCESS but with a null decryption.

It is important to note that the processing is not performed by the API server which only hosts publishable data, but by the clients of the owners who can authorise themselves with the peer's private key.

The messages of the outbox may be to arbitrary destinations. A separate Transport service is responsible to route messages to other peers' inboxes.

Negotiation and Consensus

An important feature of the interface is that peer ownership can be nested and shared among physical or virtual persons. Another important goal is that there is always a cryptographically secure audit trace of actions authorised by peer owners.

This need extends beyond the functions specified by the interface itself, namely the creation of peers and the processing of cycles.

We aim peers to be a powerful primitive on which applications may build their protocols by creating new peer and message types and new actions. Such applications will need a way to safely authorise and log their state transitions.

For this purpose we provide a generic negotiation and consensus primitive whose data is hosted in the API namespace.

Anyone may start a negotiation by submitting an initial document to be agreed upon. The newly created negotiation has an unpredictable identifier so that it may be shared in private among a set of participants. Each participant can view the whole history of the negotiation and then upload their own document as a contribution to it. All contributions are signed by the corresponding contributor's private key. Note that participants may be PANORAMIX peers.

Participants may go through several rounds of contributions before they can agree. The API service automatically detects when the last signed contribution from all contributors is the exact same document and automatically freezes the negotiation and the concluded document and cryptographically indexes them as a consensus.

The identifier of the consensus commits to the entire agreed document and all negotiating participants signatures and can be used as both a description of an action (e.g. create a peer, an election) and the authorization for it.

4.3 API documentation

4.3.1 Overview

The mix-net API is based around *peers* who exchange *messages*. Each peer opens a *cycle* to accept messages in its *inbox*. When sufficient messages are collected in the inbox, the peer retrieves the messages, processes them and posts them to its *outbox*. An external posting mechanism is responsible to send the outbox messages to their recipients.

4.3.2 Peers

A peer is any participant to the mix-net, either a mix-net contributor, a correspondent, an auditor, or any other stakeholder. A peer must be registered to the mix-net controller using a cryptographic identifier.

List Peers

Returns a list containing information about the registered peers.

URI	Method	Description
`/peers`	GET	List peers

Example reply:


```
[{"status": "READY", "owners": [], "name": "peer1", "peer_type":
  "GATEWAY", "size_max": 10, "key_type": 1, "size_min": 3,
  "consensus_id": None, "key_id": "13C18335A029BEC5"}]
```

Create a Peer

Create a new peer with the specified parameters; see the example below. Alternatively, the payload can contain a key *"consensus_id"*, indicating a prescription to create a peer agreed upon by multiple peers through a negotiation.

URI	Method	Description
/peers	POST	Create a peer

Example request:

```
{"payload": {"operation": "create", "resource": "peer",
             "key_data": "public key", key_id: "13C18335A029BEC5",
             "key_type": 1, "name": "peer1", "peer_type": "GATEWAY",
             "size_min": 3, "size_max": 10},
"signature": "payload signature with private key"}
```

Example reply:

```
"13C18335A029BEC5"
```

Get peer info

Get info for a single peer.

URI	Method	Description
/peers/<peer_id>	GET	Get info for a peer

Example reply:

```
{"status": "READY", "owners": [], "name": "peer1", "peer_type":
 "GATEWAY", "size_max": 10, "key_type": 1, "size_min": 3,
 "consensus_id": None, "key_id": "13C18335A029BEC5", "key_data"}
```

4.3.3 Cycles

A peer handles messages in cycles. A correspondent sends messages to a specified cycle and can query how many messages are currently in a cycle.

List peer cycles

URI	Method	Description
/peers/<peer_id>/cycles	GET	List peer cycles

Example reply:

```
[{"peer_id": "13C18335A029BEC5", "id": 1,
  "status": "OPEN", "size_current": 0}]
```

Create a peer cycle

URI	Method	Description
/peers/<peer_id>/cycles	POST	Create a peer cycle

Example request:

```
{"payload": {"operation": "create", "resource: "cycle",
             "peer_id": "13C18335A029BEC5"},
 "signature": "payload signature"}
```

Get cycle info

URI	Method	Description
/peers/<peer_id>/cycles/<cycle_id>	GET	Get info for a cycle

Example reply:

```
{"peer_id": "13C18335A029BEC5", "id": 1,
 "status": "OPEN", "size_current": 0}
```

Update a cycle

Update a cycle, for example to mark it as CLOSED, i.e. not accepting any new messages.

URI	Method	Description
/peers/<peer_id>/cycles/<cycle_id>	POST	Update a cycle

Example request:

```
{"payload": {"operation": "action", "action": "close", "resource": "cycle",
             "peer_id": "13C18335A029BEC5", "cycle_id": 1},
 "signature": "payload signature"}
```

4.3.4 Messages

Messages are posted to a cycle's inbox of a specified peer. Once a sufficient number of messages are collected, the peer retrieves the inbox messages, processes them and uploads the transformed messages to outbox. The actual processing is not part of this API, but of the application logic.

List inbox/outbox messages

URI	Method	Description
/peers/<peer_id>/cycles/<cycle_id>/[in,out]box	GET	List messages

Example inbox reply:

```
[{"box": 0, "cycle_id": 1, "id": 1, "sender": "orig_sender1",
  "data": "encrypted message 1", "recipient": "this_peer"},
 {"box": 0, "cycle_id": 1, "id: 2, "sender": "orig_sender2",
  "data": "encrypted message 2", "recipient": "this_peer"}]
```

Example outbox reply:

```
[{"box": 1, "cycle_id": 1, "id": 3, "sender": "this_peer",
  "data": "decrypted message a", "recipient": "next_peer_a"},
 {"box": 1, "cycle_id": 1, "id": 4, "sender": "this_peer",
  "data": "decrypted message b", "recipient": "next_peer_a"}]
```

In this example, we assume that processing has shuffled the messages in order to hide the connection between encrypted messages (1, 2) and decrypted messages (a and b).

Send a message to inbox/outbox

URI	Method	Description
/peers/<peer_id>/cycles/<cycle_id>/[in,out]box	POST	Send a message

Example request:

```
{"payload": {"operation": "create", "resource": "message",
             "sender": "FC650CA0F7749FF0", "recipient": "13C18335A029BEC5"
             "peer_id": "13C18335A029BEC5", "cycle_id": 1,
             "box": 0, "data": "encrypted message"},
 "signature": "payload signature"}
```

Example reply:

```
"1"
```

4.3.5 Negotiations and Consensus

Negotiation is a mechanism that allows peers to agree upon a common text after rounds of amendments. The final text is signed by all participating peers. A text can be a prescription for an action that requires consensus of all stakeholders.

When a negotiation completes successfully, a consensus id is computed by hashing the negotiation data. This id can be provided to any operation that requires a consensus to proceed, e.g. in order to create a new peer with multiple owners.

Initiate a negotiation

The peer who starts a new negotiation is given a hard-to-guess negotiation id. The peer can then invite other peers to the negotiation by communicating them the id.

URI	Method	Description
/negotiations	POST	Initiate a negotiation

Example reply:

```
"long_negotiation_id"
```

Get negotiation details

URI	Method	Description
/negotiations/<negotiation_id>	GET	Get negotiation details

Example reply:

```
{"id": "neg_id", "status": "OPEN", "hash": None, "contributions": []}
```

Contribute to negotiation

Contribute a signed text to a negotiation. If all peers participating so far sign the same text, then the negotiation completes successfully and the consensus id is produced. No more contributions are accepted.

URI	Method	Description
/negotiations/<negotiation_id>	POST	Contribute to negotiation

Example request:

```
{"payload": {"resource": "negotiation", "operation": "action",
             "action": "contribute", "negotiation_id": "neg_id",
             "text": "contribution text", "signature": "text signature"},
 "signature": "payload signature"}
```

Get consensus details

Get the agreed upon text along with its signatures for the given consensus_id.

URI	Method	Description
/consensus/<consensus_id>	GET	Get consensus details

Example reply:

```
{"id": "consensus_id", "negotiation_id": "neg_id", "text": "agreed text",
 "signings": {"peer1_id": "text signature1",
              "peer2_id": "text signature2"}}
```

5. Development tools and practices

5.1 Programming Environments

For each software that is released by PANORAMIX the development team maintains a version controlled repository, and an environment to run tests, build distribution packages, and deploy for functional and integration testing. These may be separate environments or a unified one. The environment or environments where the release is deployed for integration and / or testing constitute the official supported environments recommended to users.

The environments are preferably automatically created from instructions within the development repository itself, and there is a distinct maintainer for them responsible for providing reference environments for testing and continuous integration automation.

The developer spawns instances of the environments as needed during the development of new features. When the feature is complete, the feature is then tested in environment instances belonging to the continuous integration infrastructure to verify integration. When new features require changes or additions to the reference environments, for example the deployment of additional software dependencies or configuration parameters, these changes have to be accepted by the maintainer of the environment, so that they can update continuous integration tools to use the new versions instead.

5.2 Iteration plan

Development is planned in two different scales. The long-term goals are documented and used to define separate clusters of features. Then each cluster of features is developed in short-term iterations between two and four weeks. These iterations are broken down in daily work items that are tracked in a `phabricator` installation. After each sort iteration an internal demonstration is set up. Based on the demonstration, the work is evaluated and feedback is incorporated in the planning of the next iteration. Iterations may also offer refinements or revisions of the long-term goals as development goes on and more experience is gathered.

The short term iterations always deliver results that are continuously integrated with the testing, building, and deployment environments. This makes it easy for different contributors within the same or different teams to join in development, and it allows to determine the maturity of a feature implementation.

5.3 Code repository

In the repository (`git`), there is a main development branch, which is always functional and integrated. Each release has its own branch for maintenance. Each feature is forked from the development branch and then merged back. There are two conditions for code merging into the development branch. First, it must have been reviewed by different developers, and second it must have been discussed in a technical demonstration where its overall design and implementation is evaluated.

5.4 Prototyping methodology

During the technical demonstration of a new feature, the developers present the designs they have employed the practices and patterns they used, and elaborate on the challenges they encountered. Their peers will get to know the new code and form an opinion on its implementation effectiveness, maintainability, and what are the next steps. Additionally, people responsible for system architecture or high-level design will judge how well the new code integrates with the whole project's backbone, if there has been enough code reuse, or if there was introduction of unnecessary dependencies.

5.5 Software release and version control

Once software is released externally, both the main codebase and the accompanying environments are published in the project's website and in appropriate distribution places such as Github or PyPI. In order to prepare contribution for PANORAMIX software, third parties are expected to follow a similar workflow within their own teams making use of the published integration and testing environments. After a contribution is ready, it is published in the projects development mailing list and the software owners review it with the same process as if it was coming from their own team.

5.6 Documentation

Documentation is very important and must be part of building a software package for distribution. It is therefore part of the continuous integration cycle and it has to be demonstrated along with functionality during reviews of new features. Deploying the software also deploys the full documentation for all users, auditors, and / or developers. The standard tool used for documentation for both design and API reference is Sphinx, allowing publishing both as HTML for web or PDF for reports.

Bibliography

[Adi08] Ben Adida. Helios: Web-based open-audit voting. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, pages 335–348, Berkeley, CA, USA, 2008. USENIX Association.

[BGP11] Philippe Bulens, Damien Giry, and Olivier Pereira. Running mixnet-based elections with helios. In *Proceedings of the 2011 Conference on Electronic Voting Technology/Workshop on Trustworthy Elections*, EVT/WOTE'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association.

[DD08] George Danezis and Claudia Diaz. A survey of anonymous communication channels. Technical report, Technical Report MSR-TR-2008-35, Microsoft Research, 2008.

[DDM03] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a Type III Anonymous Remailer Protocol. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 2–15, May 2003.

[DG09] George Danezis and Ian Goldberg. Sphinx: A compact and provably secure mix format. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P 2009)*, pages 269–282. IEEE Computer Society, May 2009.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[EBFK13] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. ACM, 2013.

[FSK11] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography engineering: design principles and practical applications*. John Wiley & Sons, 2011.

[Lau02] Soren Lauesen. *Software requirements: styles and techniques*. Pearson Education, 2002.

[MCPS03] Ulf Möller, Lance Cottrell, Peter Palfrader, and Len Sassaman. Mixmaster Protocol — Version 2. IETF Internet Draft, July 2003.

[SBKH06] Steve Sheng, Levi Broderick, Colleen Alison Koranda, and Jeremy J Hyland. Why johnny still cant encrypt: evaluating the usability of email encryption software. In *Symposium On Usable Privacy and Security*, pages 3–4, 2006.

[SFK+12] Sebastian Schrittwieser, Peter Frühwirt, Peter Kieseberg, Manuel Leithner, Martin Mulazzani, Markus Huber, and Edgar R Weippl. Guess who's texting you? evaluating the security of smartphone messaging applications. In *NDSS*. Citeseer,

2012.

[TPLT13] Georgios Tsoukalas, Kostas Papadimitriou, Panos Louridas, and Panayiotis Tsanakas. From Helios to Zeus. In *2013 Electronic Voting Technology Workshop / Workshop on Trustworthy Elections, EVT/WOTE '13, Washington, D.C., USA, August 12-13*, 2013.

[VDHLZZ15] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 137–152. ACM, 2015.

[WT99] Alma Whitten and J Doug Tygar. Why johnny can't encrypt: A usability evaluation of pgp 5.0. In *Usenix Security*, volume 1999, 1999.