PloneVote High-level Design

Lázaro Clapp

8 January 2011

Contents

1 Introduction

The purpose of this document is to provide a high-level overview of the design of the PloneVote system. To do this, we seek to divide the whole system into major components, which exist at the level of an application module or subsystem. We endeavor to describe the concerns and basic functions of each such component and, more importantly, describe all interactions between the different components of the system.

We start by further detailing the notion of component in subsection 2.1. The rest of section 2 deals with describing different criteria around which it might be useful to classify the different PloneVote components: by concern (2.2), by execution environment (2.3) and by how much their implementation is a requirement for the system (2.4).

In section 3, we start by briefly mentioning and describing each component, showing its place within each of the above three classifications. In subsection 3.2, we present a diagram of the PloneVote high-level architecture, specifying the dependencies between components (a notion we also define there). Subsection 3.3 is a note and discussion on the concept of communication and storage formats as it applies to the PloneVote system and on our chosen classification of formats into opaque and transparent formats.

Finally, section 4 lists all the PloneVote components in order, providing, for each such component: a brief description, a list of component-level concerns, a summary of its interactions and dependencies with other components and a list of the opaque and transparent formats the component owns and defines.

Beyond what is specified in this document, it should be possible to design, code and test each component as a single entity, referencing only this high-level design document, the requirements document and the designs of those other components with which the component being designed closely interacts with or directly depends upon.

2 Classification of components

2.1 About components

For the purposes of this document, a component is a software module or package that acts as a high-level building block for our system. Components communicate with one another through well defined interfaces and can be distributed independently of one another (although they may still need communication with other components to provide an useful function, or define themselves in terms of other components). A component may be as large as a library containing multiple classes, or as small as a single JavaScript function inside a script file. The division of the PloneVote system into components is done in part arbitrarily, our main objective in identifying components is to partition PloneVote into modules that can be developed independently and that have an interaction model as loosely coupled as possible.

The goal of this high-level design document is to separate the PloneVote system into components, describe briefly what system concerns each component is responsible for, catalogue all component interactions and briefly explain what those interactions are like. A component level design document will precede the development of each component, finalizing the details of its external interface and most of its constituent parts. The design document of each component should be possible to understand by referencing only the PloneVote requirements document, this high-level design document and the design document for components that directly interact with the described component.

We define as components only the pieces of the system which we will need to build for the current prototype of the system. That is, existing infraestructure, such as the Plone CMS or modules from the python standard library, are not named as components in this document.

PloneVote components can be classified around three independent criteria: their main concern, the environment in which they execute and the extent to which the component is an mandatory part of the system. We briefly describe these classifications in the following sections.

2.2 Components by major concern

Each component by itself is responsible for solving a number of system concerns at the right level of granularity. However, each component is also a part of the solution to one of the three main orthogonal concerns of PloneVote: election security, ballot and vote counting scheme, and election process. For the purposes of this document, we further separate the ballot scheme and the vote counting scheme as concerns, even though they are a lot more closely related to one another than the three concerns in the original division are. Any component of PloneVote is primarily related to, and can be filed under, one of the following concerns:

- Election Security: The component primary function is to implement part of the Election Security Protocol. Either the component implements part of the cryptographic machinery or API for the operations of the ESP, or it allows users to easily access such functionality (e.g. the component of the desktop client which allows an auditor to perform a verifiable ballot shuffle).

- Ballot Definition: The component primary function is to define, configure or manipulate a ballot or vote casting scheme. This includes any component that provides an interface to cast or manipulate specific types of ballots as its primary function.

- Vote Counting: These components are primarily concerned with defining a scheme for counting votes or with supporting vote counting schemes in general. Components with this concern are likely to have strong references to one or more components concerned with Ballot Definition.

- Election Process: The component primary function relates to running the election or supporting steps of the election process that are not directly part of the Election Security Protocol. This category also includes any components providing infrastructure to other components, where such infrastructure is not part of the three previous concerns, such as logging, event notification and client/server communication.

2.3 Components by execution environment

Another way of classifying PloneVote components is by the environment in which they run. There are three distinct environments in which code related to the system may be executed: Plone module, desktop application and within a web browser. We also consider a fourth environment for code that will need to be portable between Plone and desktop environments:

- Python Library: These are components which must run in any standard python environment, both on the server, within a Plone context, as well as called from the desktop client in any of the supported client platforms. These components are expected to rely only on the standard python library and other portable python modules.

- Plone Product: These are components which execute inside the Plone CMS on the server. They may depend on portable python modules, the Plone CMS, the Zope server and DB, and any portable Plone or Zope extensions. Ideally, this part of PloneVote should be fully operating system independent (as long as the target OS can run Plone), but only Linux compatibility is truly required for the current prototype.

- Desktop Client: These are components that run as part of the trustee or auditor desktop client. All desktop client components should be written in python and use GTK as a portable GUI library for user interaction. Desktop client code should run without issue on current versions of Linux (including Ubuntu latest normal and LTS releases), Windows (including XP, Vista and 7) and Mac OS X.

- Web Client: These are components that run fully inside a web browser as part of the client code for vote casting. Web client components should be, whenever possible, developed using standard JavaScript (ECMAscript), (X)HTML and CSS definitions and APIs. As a minimum, the web client should be compatible with the current version of Mozilla Firefox. Compatibility with Google Chrome, Apple Safari and Microsoft Internet Explorer (7+, and 6 if possible) is also desired.

2.4 Components by requirement

A final taxonomy of components centers around their importance or urgency for PloneVote as a whole, comprising three categories:

- Core Requirement: The component is an intrinsic part of PloneVote, at least as it is currently envisioned. These are components that define PloneVote and are not expected to be switched out in the foreseeable future. They must be implemented for the first prototype to be considered complete and should be carefully designed, coded and tested. The library supporting basic cryptographic functions for the Election Security Protocol is an example of a core component.

- Prototype Requirement: The component is required for the current PloneVote prototype, despite not being an irreplaceable in the final system. This include the specific ballot and vote counting schemes selected for implementation (multivote) as well as some support modules that are needed for PloneVote to function as required for the prototype but could be replaced at a later date without fundamentally changing the system (such as the particular notification system used by the server).

- Optional: The component is optional for the first prototype. That is, the prototype could be considered complete without the component. In this document, we list only components related to the preferential ballot scheme as optional. Even if we do not end up implementing preferential ballots in this iteration, considering a second ballot and vote counting scheme during design can help us keep extensibility in mind while designing the parts of PloneVote that should be able to accommodate diverse voting schemes in the future.

3 PloneVote High-level architecture

3.1 Summary of PloneVote components

We list and briefly describe each component considered as part of the current PloneVote prototype. Table 1 shows how all components distribute across the three classifications given on the previous section, with rows representing concern categories (2.2), columns representing execution environment (2.3), and font decoration representing requirement distinctions (2.4): bold for core requirements, normal for prototype requirements and italics for optional components. A more in depth view of each component is given in section 4.

1. PloneVote Crypto Library: A python library providing the cryptographic primitives and operations used by both the PloneVote server and the desktop client to implement the Election Security Protocol.

2. Ballot scheme base module: A python library defining the required interfaces and general operations for dealing with ballot schemes in the abstract. This module should include all the mechanisms and definitions that are shared by all possible ballot schemes. Specific ballot scheme extensions to PloneVote will hook into this component.

Components	Python Library	Plone Product	Desktop Client	Web Client
Election Security	- **PloneVote crypto library**		- **Trustee client operations** - **Auditor client operations**	- **JS encryption routine(s)**
Ballot Definition	- **Ballot scheme base module** - Multivote ballot scheme extension - *Preferential ballot scheme extension*			- JS multivote ballot wizard - *JS preferential ballot wizard*
Vote Counting	- **Vote counting base module** - Multivote counting extension, - *Schulze-STV counting extension*			
Election Process	- Candidate list representation model	- **SCE base module** - **Candidate/ voter list tools** - Server-to-client communicator - Event logging subsystem - Notification subsystem	- **Base client (GUI)** - Client-to-server communicator	- **JS vote casting client**

Table 1: Classification of PloneVote components

3. Multivote ballot scheme extension: The particular extension module implementing multivote as a ballot scheme. This should include only the python side (server and desktop client) of the ballot scheme implementation, while the "JavaScript multivote ballot wizard" component below (18) implements the user interface for casting a multivote from the web client.

4. Preferential ballot scheme extension: The particular extension module implementing preferential voting as a ballot scheme. This should include only the python side (server and desktop client) of the ballot scheme implementation, while the "JavaScript preferential ballot wizard" component below (19) implements the user interface for casting a preferential ballot vote from the web client.

5. Vote counting base module: A python library defining the required interfaces and general operations for dealing with vote counting algorithms for particular ballot schemes. This module should include all the mechanisms and definitions that are shared by all possible vote counting schemes. Specific vote counting scheme extensions to PloneVote will hook into this component.

6. Multivote counting extension: The particular extension module implementing multivote counting as a vote counting scheme, applicable to multivote ballots.

7. Schulze-STV counting extension: The particular extension module imple-

menting Schulze-STV counting as a vote counting scheme, applicable to preferential ballots.

8. SCE base module: The base Plone product coordinating the server side of PloneVote and the phases, steps and configuration of an Standard Candidate Election as defined in the requirements document. This component also provides most of the server side user interface related to creating, configuring and interacting with an election.

9. Candidate/voter list tools: A Plone product for creating, editing and finalizing candidate and voter lists to be used on an SCE.

10. Candidate list representation model: A small python module providing a representation model for candidate lists and the ability to store and parse those lists. The main function of this module is to provide a bridge between the many PloneVote components (in the server and desktop) that need to read candidate lists and the list tools above, used to generate them (9).

11. Server-to-client communicator: A Plone product exposing the actions required by the desktop client (such as uploading and downloading important election files with the appropriate checks, or performing specific changes to the election) as a web service. Essentially, this module provides a proxy web service API to a safe subset of the PloneVote server API, which is used by external desktop clients. Ideally, it should allow authenticated log-in to Plone by the desktop client on behalf of its user.

12. Client-to-server communicator: A python library for use by the desktop client to interact with the PloneVote server exposed web services. Essentially, this module provides a proxy local API to the web service API created by the server-to-client communicator component (11). Ideally, it should allow authenticated log-in to Plone by the desktop client on behalf of its user.

13. Base client (GUI): The base graphical user interface and shared functionality of the desktop trustee and auditor clients. This component uses the client-to-server communicator component to interact with the server and display lists of running elections. Trustee and auditor operation components (14 and 15) can register and provide particular client operations that depend on election state.

14. Trustee client operations: Defines and implements all actions specific to the trustee desktop client, extending the base client with the required functionality to perform trustee actions: key pair set-up, partial decryption, etc.

15. Auditor client operations: Defines and implements all actions specific to the auditor desktop client, extending the base client with the required

functionality to perform auditor actions: verifiable shuffling, election verification, etc.

16. JavaScript vote casting client: The basic code and interface of the PloneVote vote casting web client, coded in JavaScript, (X)HTML and CSS. This component should coordinate loading the correct client ballot wizard (for example, 18 or 19), encrypting the user vote, showing the vote receipt and uploading the encrypted vote to the server.

17. JavaScript encryption routine(s): The JavaScript code required for performing public key encryption on cast votes before uploading them to the server. This component should also contain the code for creating vote receipts.

18. JavaScript multivote ballot wizard: The JavaScript GUI and encoder for recording a multivote.

19. JavaScript preferential ballot wizard: The JavaScript GUI and encoder for recording a preferential vote.

20. Event logging subsystem: The PloneVote component concerned with offering logging facilities to the rest of the system, logging messages about each particular election and about the operation of the server side components of PloneVote in general.

21. Notification subsystem: The PloneVote component concerned with delivering notifications to users, by means of both messages in the Plone site and e-mail.

3.2 Component dependencies

For the purposes of this document, we say that a component depends on another if the first component needs to know any details about the classes, methods, actions or formats defined by the second component. This includes modules directly referencing one another, which is the more direct and usual notion of dependency. However, it also includes more subtle relationships, such as a component needing to know the structure of a certain file format produced by another module, which may not even execute in the same computer or environment. A guideline for how we think about dependencies is to say that component A depends on component B if component A cannot or should not be written before we have a detailed design of component B.

In figure 1, we can see all the components of the current PloneVote prototype (including optional components) and their dependency relationships. An arrow pointing from component A to component B indicates that A depends on B. When writing the component level design and code for each specific component, we should keep the diagram in mind and not start coding a component before we have at least the design document for all the components it depends on.

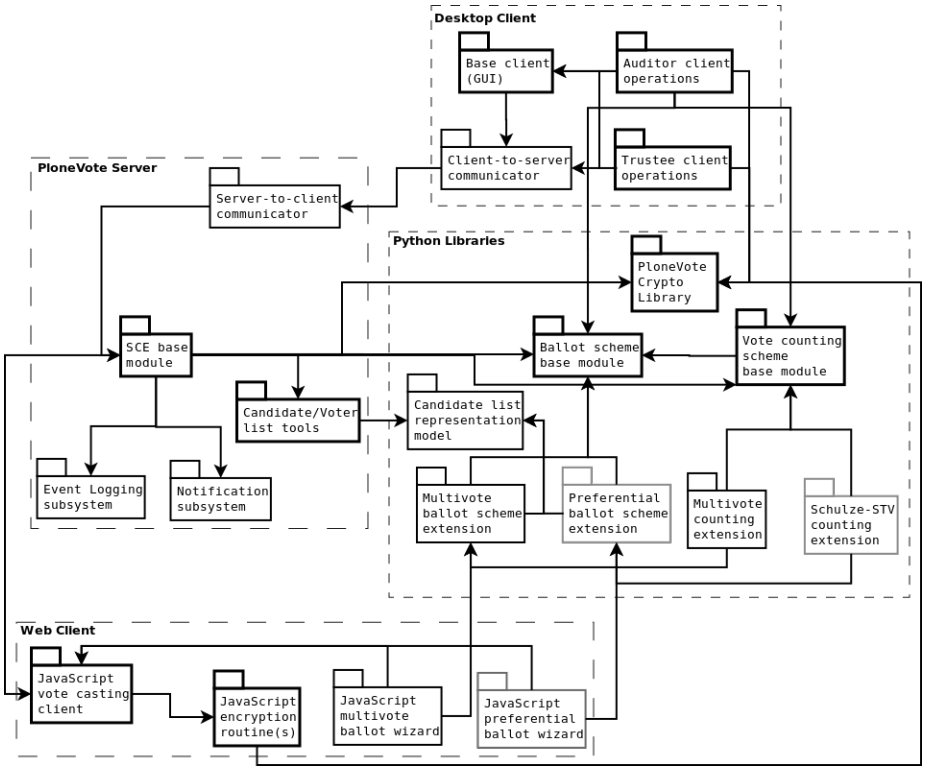

Figure 1: PloneVote High-Level Architecture Diagram

3.3 On communication and storage formats

Being a system that spans three completely different execution environments (Plone server, desktop application and web client), much of the communication in PloneVote will be conducted through specially formated files (mostly XML dialects), specially encoded text, or data streams. Additionally, the system requires storing many different kinds of important data, from lists of election candidates and voters, to proofs required for verification. In order to have a workable and well-defined design for the system, it is important to carefully specify the file formats used by each PloneVote component, for both storage and communication. Of particular importance, are those files (or other forms of persistent data, such as Zope objects[1]) which are read and/or written by two or more distinct components, thus acting as means of communication between those components, either directly or indirectly.

As we discuss each PloneVote component in some detail in section 4 of this document, and later, as we write each component's full design document, we will make a distinction between two types of file or data storage formats:

- *Opaque or private formats* are those which are used exclusively within one PloneVote component. That is, only one component reads, writes or manipulates these files. These formats may be stored or moved around by other components, which treat them as simple text streams or generic files (for example, partial decryptions flow all the way from the desktop client to the server, crossing 4 to 6 modules depending on how one does the count, but only the PloneVote crypto library needs to understand or operate over them). An opaque format is, thus, one such that only one component cares about its internal semantics. We call this component the owner of the format. As the name would seem to indicate, an opaque format is a private part of its owner's implementation and not part of said component's public interface. This makes it easier to change this kinds of formats in the future, as the change would presumably only affect the format's owner and not any other component in the system.

- *Transparent or public formats* are those which must be read, written and/or modified by more than one component at the semantic level. These formats are usually used for communication between components and thus are part of the components' public interface. A transparent format may have an "owner", a component that by design or by convention we have chosen as the authority on how that format is defined. However, changing a public format means changing every component which needs to operate on it at the semantic level, that is, any component to which the format is transparent.

[1] In fact, in the following discussion, we should take the word "format" to mean any specially structured data which can flow in or out of a component, for communication or persistent storage. This includes not only files, but also: structured text strings, data streams, Zope objects, etc.

We also speak of formats being transparent or opaque to a particular component which handles files, streams or objects encoded in that format. A format is transparent to a certain component if the component may, at some point, care about the semantics of the format or its internal syntax, and opaque otherwise. An opaque format, in its general meaning, is one which is transparent only to one component, its owner. A transparent format is one which is transparent to at least two distinct components.

In the following discussion and on the design documents for each component, we will endeavor to point out whether a format is opaque or transparent the first time we mention said format. Clearly, transparent formats are more delicate to design than opaque formats, if only because of the broader impact changing the format may have on the system's design or code.

4 List of PloneVote components

4.1 PloneVote Crypto Library

4.1.1 PloneVote Crypto Library description

The PloneVote Crypto Library module is a pure python library implementing all primitive cryptographic operations required by the rest of the PloneVote system, with the exception of the JavaScript routines required at vote casting time to encrypt the vote and obtain a receipt. The objective is to abstract the complexities of all cryptographic operations behind an easy to use interface for the server and desktop client(s). Verification operations for all verifiable cryptographic operations are also part of this library.

4.1.2 PloneVote Crypto Library concerns

The following are all concerns of the PloneVote Crypto Library:

- (**P0**) Generating a private and partial public key pair. Also, encoding those keys for transport or storage.
- (**P0**) Constructing a k of n threshold encryption scheme by combining n partial public keys into a combined public key.
- (-[2]) Encrypting arbitrary text with the combined public key.
- (**P0**) Generating a secure receipt of an encrypted text (possibly a simple hash).
- (**P0**) Generating a partial decryption, using a single private key, of a ciphertext encrypted with the combined public key.

[2]This actually only needs to be possible from the JavaScript encryption routine(s), but it would be useful for tests to also implement it in the PloneVote Crypto Library.

- (**P0**) Providing a verifiable proof of correctness of the above partial decryption.
- (**P1B**) Checking a proof of correct partial decryption without access to the private key used for said decryption.
- (**P0**) Combining k partial decryptions of the same ciphertext encrypted with a k of n threshold encryption scheme, restoring the plaintext.
- (**P0**) Verifiably shuffling a collection of ciphertexts, producing a proof of correct shuffling together with the shuffled collection.
- (**P0**) Checking a proof of correct shuffling (that is, asserting equivalence between the initial and shuffled collections).

4.1.3 PloneVote Crypto Library dependencies

The PloneVote Crypto Library depends on no other components

The following components directly depend on the PloneVote Crypto Library:

- *SCE base module*: The SCE base module directly uses the following PloneVote Crypto Library operations:
  - Constructing a threshold encryption scheme.
  - Generating receipts.
  - (Possibly) Checking a proof of partial decryption.
  - Combining partial decryptions into a full decryption.
  - Checking a proof of correct shuffling.
- *Trustee client operations*: The trustee client operations component directly uses the the following PloneVote Crypto Library operations:
  - Generating a private and partial public key pair.
  - Constructing a threshold encryption scheme (to verify the correctness of the combined public key generated by the server).
  - Generating a partial decryption (with proof).
  - (Possibly) Checking proofs of partial decryption and/or correct shuffling.
- *Auditor client operations*: The auditor client operations component directly uses the the following PloneVote Crypto Library operations:
  - Generating receipts (to check those generated by the server).
  - Performing a verifiable shuffle.

  - Checking a proof of correct shuffling.
  - Checking a proof of partial decryption.
  - Combining partial decryptions into a full decryption (to verify the combination done by the server).

- *JavaScript encryption routine(s)*: The JavaScript encryption module must be able to operate with the public key format of the PloneVote Crypto Library in order to encrypt votes, generating the same format of ciphertext than this library. It must also create the same format of vote receipts.

4.1.4 PloneVote Crypto Library formats

The PloneVote Crypto Library defines many formats that need to be stored and moved between execution contexts (server and client). However, many of these formats are opaque, needing only be understood, used or verified by the PloneVote Crypto Library itself.

- *Partial public key* (**opaque**): The format used to store and transmit a partial public key.
- *Private key* (**opaque**): The format used to store and transmit the private key.
- *Combined public key* (**transparent**): The format used to store and transmit the combined public key.
- *PloneVote Crypto ciphertext* (**transparent**): The format of a text encrypted with PloneVote Crypto, could include an embedded public key and pre-encryption padding to prevent attacks based on knowing the length of the plaintext.
- *PloneVote Crypto receipt* (**transparent**): The format of the cryptographic vote receipt, possibly a hash.
- *Verifiable partial decryption* (**opaque**): The format of a partial decryption, embedding its own proof of correctness.
- *Proof of shuffling* (**opaque**): The format of a proof of correct shuffling, it may or may not include the collection of ciphertexts before and after shuffling.

4.2 Ballot scheme base component

4.2.1 Ballot scheme base description

The ballot scheme base component is a python library that defines the general interfaces, operations and extension points used by ballot scheme extensions. This module should include all the mechanisms and definitions that are shared by all possible ballot schemes. Specific ballot scheme extensions to PloneVote will hook into this component.

4.2.2 Ballot scheme base concerns

The following are all concerns of the ballot scheme base component:

- (**P0**) Keeping a registry of available ballot schemes in the system, allowing for ballot scheme extensions to register themselves into it and for other python code to query registered extensions.
- (**P0**) Retrieving information about registered ballot scheme extensions, including: an unique identifier; name, description and metadata of the ballot scheme module; possible configuration options (with enough information to populate a Plone form as a configuration interface); whether the ballot scheme expects a list of candidates to populate itself or has a custom definition form; and the corresponding JavaScript ballot wizard, if registered with the extension.
- (**P0**) Providing an abstract interface to the operations of each ballot scheme extension, including: creating the ballot template with the selected configuration (including candidate options), parsing an unencrypted vote into a particular subclass of an abstract vote object (which can later be counted by vote counting extensions) and validating a vote object against a ballot template.

4.2.3 Ballot scheme base interactions

Ballot scheme base depends on no other components

The following components directly depend on ballot scheme base:

- *Multivote ballot scheme extension*: The multivote ballot scheme extension must register itself as a ballot scheme with the ballot scheme base component. The multivote ballot scheme extension must also define all the information about its particular ballot scheme required by the ballot scheme base component, and particular instances of the abstract ballot template and abstract vote object classes or interfaces defined in it.
- *Preferential ballot scheme extension*: The preferential ballot scheme extension must register itself as a ballot scheme with the ballot scheme base component. The preferential ballot scheme extension must also define all the information about its particular ballot scheme required by the ballot scheme base component, and particular instances of the abstract ballot template and abstract vote object classes or interfaces defined in it.
- *Vote counting base module*: The vote counting base module must be able to query the ballot scheme base component for registered ballot schemes, in order to associate particular vote counting extensions with particular ballot schemes. Both components must, at the very least, use the same ids for ballot schemes.

- *SCE base module*: The SCE base module needs to query the ballot scheme base component for registered ballot schemes, in order to select one such scheme for an election. Furthermore, it needs to query the specific ballot scheme information in order to present its configuration options to the election administrator. Additionally, the SCE base module should create a ballot template for the election, previous to the voting phase, and also parse submitted votes after decryption but before sending them to the correct vote counting extension.
- *Auditor client operations*: The auditor client operations component needs to verify the ballot template of the election, parse the decrypted votes and perform a vote count by using this module in combination with the vote counting base module and the particular extensions used for the election.

4.2.4 Ballot scheme base formats

The only transparent data format used by the ballot scheme base component is a ballot scheme identifier. Most other components that communicate with it do so directly, by implementing its interfaces and/or using its public API.

- *Ballot scheme unique identifier* (**transparent**): An unique identifier for each ballot scheme extension, used to register and refer to the extension in this and other components.
- *Ballot scheme registry* (**opaque**): A database or index of registered ballot scheme extensions.

4.3 Multivote ballot scheme extension

4.3.1 Multivote ballot scheme description

This is the module implementing the multivote ballot scheme extension for PloneVote. Much of what this module must do has to do with fulfilling the interfaces required by the ballot scheme base component.

4.3.2 Multivote ballot scheme concerns

The following are all concerns of the multivote ballot scheme extension:

- (**P0**) Registering multivote as a ballot scheme option in the ballot scheme base component's registry.
- (**P0**) Presenting basic metadata upon query, as well as the fact that this scheme takes a list of candidates to populate the ballot template.
- (**P0**) Defining the format for storing and transmitting multivote ballots and votes, as well as the particular classes for a multivote ballot template and multivote vote object. Allowing both those kinds of objects to be automatically constructed from the ballot scheme base component.

- (**P1A**) Allowing a multivote ballot template to be populated from a candidate list.
- (**P1B**) Presenting the following configuration options for a multivote scheme, recording them in the ballot template and allowing them to be validated for a particular multivote vote object:
  - Minimum number of votes
  - Maximum number of votes
  - (**P2**) Whether or not a blank vote is permitted

4.3.3 Multivote ballot scheme interactions

Multivote ballot scheme depends on:

- *Ballot scheme base component*: The multivote ballot extension must register with and adhere to the interfaces and APIs of the ballot scheme base component. This includes using the same identifier format, supporting metadata and configuration information queries, and creating particular classes of the ballot template and vote object abstract definitions provided by the ballot scheme base component.
- *Candidate list representation model*: The multivote ballot extension must be able to iterate through a list of candidates stored in the candidate list representation model and use it to populate a ballot template.

The following components directly depend on multivote ballot scheme:

- *Multivote counting extension*: The multivote counting extension must be able to interact with vote objects defined by the multivote ballot scheme component and perform a vote count over them. Note that it needs not be able to read the multivote vote format, as it can use it through the multivote ballot scheme extension's classes. The multivote counting extension should also be aware of the multivote ballot scheme extension's id in the ballot scheme registry, so that it can reference it and register itself as providing a vote count scheme for said ballot scheme.
- *JavaScript multivote ballot wizard*: The JS multivote ballot wizard must be able to parse the multivote ballot template format (see below) in order to display a ballot to the user for vote casting. It must then be able to record the user preferences into a multivote vote format compatible with the one used by the multivote ballot scheme extension.

4.3.4 Multivote ballot scheme formats

- *Multivote ballot template format* (**transparent**): A format storing the ballot of a particular multivote election. This format contains the configuration options (min, max, blank allowed, etc) and list of candidates or

options for which votes may be cast. Files in this format are created by the multivote ballot scheme extension and read by the JavaScript multivote ballot wizard in order to present the vote casting interface.

- *Multivote vote format* (**transparent**): A format storing a single multivote vote. Files in this format are created by the JavaScript multivote ballot wizard at vote casting time and then read, validated and exposed as vote objects (for counting) by the multivote ballot scheme extension.

4.4 Preferential ballot scheme extension

4.4.1 Preferential ballot scheme description

This is the module implementing the preferential ballot scheme extension for PloneVote. Much of what this module must do has to do with fulfilling the interfaces required by the ballot scheme base component.

4.4.2 Preferential ballot scheme concerns

The following are all concerns of the preferential ballot scheme extension, note that, except for the priorities, they are similar to those for multivote:

- (**P3**) Registering preferential voting as a ballot scheme option in the ballot scheme base component's registry.
- (**P3**) Presenting basic metadata upon query, as well as the fact that this scheme takes a list of candidates to populate the ballot template.
- (**P3**) Defining the format for storing and transmitting preferential ballots and votes, as well as the particular classes for a preferential ballot template and preferential vote object. Allowing both those kinds of objects to be automatically constructed from the ballot scheme base component.
- (**P3**) Allowing a preferential ballot template to be populated from a candidate list.
- (**P3**) Presenting the following configuration options for a preferential vote scheme, recording them in the ballot template and allowing them to be validated for a particular preferential vote object:
  - Minimum number of votes
  - Maximum number of votes
  - Whether or not a blank vote is permitted

4.4.3 Preferential ballot scheme interactions

The preferential ballot scheme extension depends on:

- *Ballot scheme base component*: (See description for multivote).
- *Candidate list representation model*: (See description for multivote).

The following components directly depend on the preferential ballot scheme extension:

- *Schulze-STV counting extension*: The Schulze-STV counting extension must be able to interact with vote objects defined by the preferential ballot scheme component and perform a vote count over them. It should also be aware of the preferential ballot scheme id in the ballot scheme registry, and able to register itself as a counting scheme for that ballot scheme.

- *JavaScript preferential ballot wizard*: The JS preferential ballot wizard must be able to parse the preferential ballot template format in order to display a ballot to the user for vote casting. It must then be able to record the user preferences into a preferential vote format compatible with the one used by the preferential ballot scheme extension.

4.4.4 Preferential ballot scheme formats

- *Preferential ballot template format* (**transparent**): A format storing the ballot of a particular preferential vote election. Contains configuration options and the list of candidates, and will be parsed by the JS preferential ballot wizard to generate the vote casting interface.

- *Preferential vote format* (**transparent**): A format storing a single preferential vote. Files in this format are created by the JS preferential ballot wizard at vote casting time and then read, validated and exposed as vote objects (for counting) by the preferential ballot scheme extension.

4.5 Vote counting base component

4.5.1 Vote counting base component description

The vote counting base component is a python library which defines all the required interfaces and general operations for dealing with vote counting algorithms for particular ballot schemes. Specific vote counting extensions to PloneVote will hook into this component.

4.5.2 Vote counting base component concerns

- (**P1B**) Defining a general interface for a vote counting strategy or method, which takes a list of validated votes in a certain ballot scheme, verifies that the type of the vote objects is correct, and performs a count.

- (**P1B**) Providing a mechanism for registering vote counting schemes, together with the ballot scheme(s) they support, as well as querying the list of all vote counting schemes for those compatible with a given ballot scheme.

- (**P1B**) Determining a general way for vote counting extensions to present the results of the counting process to the user (possibly as simple as having them return an (X)HTML file).

4.5.3 Vote counting base component interactions

The vote counting base component depends on:

- *Ballot scheme base component*: The vote counting base module must be able to query the ballot scheme base component for registered ballot schemes, in order to associate particular vote counting extensions with particular ballot schemes.

The following components directly depend on the vote counting base component:

- *Multivote counting extension*: The multivote counting extension must be capable of registering itself with the vote counting base component as a vote counting extension. It must also provide the interfaces required by this component to produce a count for multivote ballots and output the count results in the expected format.

- *Schulze-STV counting extension*: The Schulze-STV counting extension must be capable of registering itself with the vote counting base component as a vote counting extension. It must also provide the interfaces required by this component to produce a count for preferential ballots and output the count results in the expected format.

- *SCE base module*: The SCE base module needs to query the vote counting base component for registered vote counting schemes compatible with a particular ballot scheme, in order to select a counting scheme for an election. During step 16 of the election, the SCE base module must also use this module to obtain a correct count, in an user readable format, for the collection of decrypted votes. This must be possible without the SCE base module having any hard-coded knowledge about the existing ballot schemes or vote counting extensions.

- *Auditor client operations*: The auditor client operations component will use this component to produce a vote count outside of the server and show the results to the user.

4.5.4 Vote counting base component formats

- *User results display format* (**transparent**): This component prescribes a particular format for reporting the results of a counting process to the user, most likely a web page format, which must be used by all vote counting extensions. This format needs not be treated programmatically once created, just shown to the user as a results page. Individual vote

counting extensions may create more structured output formats for the results if required.

4.6 Multivote counting extension

4.6.1 Multivote counting extension description

Implements multivote counting for multivote ballots as a vote counting extension.

4.6.2 Multivote counting extension concerns

- (**P1B**) Defining an strategy or method for counting the results from a collection of multivote votes, compatible with the interfaces and protocols defined by the vote counting base component.
- (**P1B**) Creating a results report for the above vote count, generated in the *user results display format* defined by the vote counting base component.

4.6.3 Multivote counting extension interactions

The multivote counting extension depends on:

- *Vote counting base component*: The multivote counting extension must define a way of counting multivote votes that is compatible with the interfaces specified in the vote counting base component. It must also register itself with this component as a vote counting scheme applicable to multivote ballots.
- *Multivote ballot scheme extension*: The multivote counting extension must be able to count collections of multivote votes. For that, it should be able to retrieve and understand ballot scheme specific information from the ballot template and vote object classes defined by the multivote ballot scheme extension. It should also know the multivote ballot scheme id, in order to register itself correctly with the vote counting base component.

No other component directly depends on the multivote counting extension.

4.6.4 Multivote counting extension formats

The multivote counting extension introduces no new file or data formats.

4.7 Schulze-STV counting extension

4.7.1 Schulze-STV counting extension description

Implements the Schulze-STV counting algorithm for preferential ballots as a vote counting extension.

4.7.2 Schulze-STV counting extension concerns

- (**P3**) Defining an strategy or method for counting the results from a collection of preferential votes, using the Schulze-STV preferential vote counting algoritm. This strategy or method must be compatible with the interfaces and protocols defined by the vote counting base component.
- (**P3**) Creating a results report for the above vote count, generated in the *user results display format* defined by the vote counting base component.

4.7.3 Schulze-STV counting extension interactions

The Schulze-STV counting extension depends on:

- *Vote counting base component*: The Schulze-STV counting extension must define a way of counting preferential votes (using the Schulze-STV algorithm) that is compatible with the interfaces specified in the vote counting base component. It must also register itself with this component as a vote counting scheme applicable to preferential ballots.
- *Preferential ballot scheme extension*: The Schulze-STV counting extension must be able to count collections of preferential votes. For that, it should be able to retrieve and understand ballot scheme specific information from the ballot template and vote object classes defined by the preferential ballot scheme extension. It should also know the preferential ballot scheme id, in order to register itself correctly with the vote counting base component.

No other component directly depends on the Schulze-STV counting extension.

4.7.4 Schulze-STV counting extension formats

The Schulze-STV counting extension introduces no new file or data formats.

4.8 SCE base module

4.8.1 SCE base module description

The SCE base module is the Plone product coordinating the server side of PloneVote and the phases, steps and configuration of an Standard Candidate Election as defined in the requirements document. This component also provides most of the server side user interface related to creating, configuring and interacting with an election.

4.8.2 SCE base module concerns

- (**P1B**) Defining an SCE election object/content and its state within the Plone CMS.

- (**P1B**) Controlling the phases, steps and transitions (manual or time triggered) for an SCE election.
- (**P1A**) Providing the programmatic mechanisms and server-side user interface for creating and configuring the basic properties of an SCE election. It should be possible to configure at least the following parameters of an election: name, description, ballot scheme, vote counting scheme, Election Security Protocol settings, type of step transition triggers (timed or manual) and corresponding times.
- (**P2**) Querying the possible settings for the ballot scheme of an election and providing an interface for the election administrator to configure those settings at election creation time.
- (**P1A** to **P2**) Presenting election information to voters, candidates and other users of the system in the form of an election page that may or may not support customization.
- (**P1A** and **P1B**) Presenting the various server-side user interfaces for working with the election, including: election commission set-up and review tools, access to the voter web client, final vote casting and recording, own vote verification and the results page. The exception are the interfaces for creating a list of candidates and a list of voters, which are part of the candidate/voter list tools.
- (**P1A** and **P1B**) Storing and keeping track of all objects and files associated with the election, including: partial and combined public keys, lists of voters, list of candidates, ballot template, cast votes, verifiably shuffled vote collections, proofs of shuffling, partial decryptions (with proofs), results and vote receipts.
- (**P1B**) Tracking which users are allowed to vote and whether they have voted already or not.
- (**P1A**) Creating and sending notifications related to election state and events.
- (**P3**) Configuring custom messages for election notifications.

4.8.3 SCE base module interactions

The SCE base module depends on:

- *PloneVote Crypto Library*: The SCE base module directly uses the following PloneVote Crypto Library operations in order to follow the server part of the Election Security Protocol:
  - Constructing a threshold encryption scheme.
  - Generating receipts.

  - (Possibly) Checking a proof of partial decryption.
  - Combining partial decryptions into a full decryption.
  - Checking a proof of correct shuffling.

- *Ballot scheme base module*: The SCE base module needs to query the ballot scheme base component for registered ballot schemes, in order to select one such scheme for an election. Furthermore, it needs to query the specific ballot scheme information in order to present its configuration options to the election administrator. Additionally, the SCE base module should create (and store) a ballot template for the election, previous to the voting phase, and also parse submitted votes after decryption but before sending them to the correct vote counting extension.

- *Vote counting base module*: The SCE base module needs to query the vote counting base component for registered vote counting schemes compatible with a particular ballot scheme, in order to select a counting scheme for an election. During step 16 of the election, the SCE base module must also use this module to obtain a correct count, in an user readable format, for the collection of decrypted votes.

- *Candidate and voter list tools*: The SCE base module needs to invoke (and communicate with) the candidate and voter list tools in order to create, review and finalize the voter and candidate lists used for the election. It will use the finalized voter list created by the tool to populate the authorized voter list (see formats bellow) for the election, and the finalized candidates list to populate the ballot template[3].

- *JavaScript vote casting client*: The SCE base module should be able to deploy the JavaScript vote casting client to the voter's browser and accept the encrypted votes generated by the client as originating from the user.

- *Event logging subsystem*: The SCE base module uses the event logging subsystem to log all information related to the election and its events (see the Requirements Document for information about what should be logged).

- *Notification subsystem*: The SCE base module uses the notification subsystem to deliver relevant site and/or e-mail notifications to users regarding the running election(s).

[3]The process will internally use the candidate list representation model and invoke the particular ballot scheme being used in the election. However, this information should be abstracted away from the SCE base module in the form of an opaque (to the SCE module) object that is generated by the candidate and voter list tools and accepted by the ballot scheme base module.

The following components directly depend on the SCE base module:

- *Server-to-client communicator*: The server-to-client communicator essentially works by exposing actions of the SCE base module through a web service API for use by the trustee/auditor client. The SCE module must accept, verify and store formats uploaded through the server-to-client communicator. It must also respond to commands and honor authenticated channels created by the communicator.
- *JavaScript vote casting client*: The JavaScript vote casting client should be able to submit an encrypted vote to the SCE base module on the server for voter authorization and vote casting (recording).

4.8.4 SCE base module formats

- *Authorized voter list* (**opaque**): A list of users authorized to vote in the election and whether or not they have voted already, used to verify voter eligibility just before a vote is successfully cast and stored for the election.

4.9 Candidate/voter list tools

4.9.1 Candidate/voter list tools description

The candidate/voter list tools are a Plone product for creating, editing and finalizing candidate and voter lists to be used in an SCE. It provides the ability of automatically generating a list of Plone users matching an specified criteria and then manually editing said list.

4.9.2 Candidate/voter list tools concerns

- (**P1A**) Generating an initial list of users from a set of criteria regarding user metadata (including properties from the Faculty/Staff Directory Plone product).
- (**P1A**) Allowing specific users to be manually added or removed from the list.
- (**P2**) Requiring auditable messages justifying why users were added or removed from a list, as well as storing the list of all changes made over an user list and the initial selection criteria.
- (**P3**) Allowing (for the candidate list, configurable via an option) entries to be defined on the list that do not correspond to any existing Plone user.
- (**P1B**) Finalizing the lists, publishing them as files and web pages, and storing them within the system.
- (**P2**) Optionally requiring candidature confirmation for the candidates (as an option available when creating the list)

- (**P3**) Adding candidate or profile information to the entries of candidate list on behalf of the candidates themselves.
- (**P3**) Supporting the ability to complain about the candidate/voter lists. This might require interaction with the notification subsystem.
- (**P1A**) Providing the server-side user interface for all the previous (implemented) operations.

4.9.3 Candidate/voter list tools interactions

The candidate/voter list tools depend on:

- *Candidate list representation model*: The candidate/voter lists tools must be able to write the list of candidates to the candidate list representation model, so that the different ballot schemes may be able to read this list without directly communicating with the candidate/voter lists tools or having to understand the more complex criteria-based user list format (see formats below)

The following components directly depend on the candidate/voter list tools:

- *SCE base module*: The SCE base module needs to invoke (and communicate with) the candidate and voter list tools in order to create, review and finalize the voter and candidate lists used for the election. It will use the finalized voter list created by the tool to populate the authorized voter list for the election, and the finalized candidates list to populate the ballot template.

4.9.4 Candidate/voter list tools formats

- *Criteria-based user list format* (**opaque**): A format for storing a full user list created with this tools, including audit information if implemented. It should record: the initial selection criteria (**P1A**), the full list of users matching that criteria (**P1A**), all added and removed users (**P1A**), the audit messages and timestamps for each removal or addition (**P2**), added non-user entries (**P3**), candidate profile information (**P3**).

4.10 Candidate list representation model

4.10.1 Candidate list representation model description

The candidate list representation model is a small python module providing a representation model for candidate lists and the ability to store and parse (write and read) those lists. The main function of this module is to provide a bridge between the PloneVote components that need to read candidate lists and the candidate/voter list tools used to generate them. It keeps the candidate list format opaque, by offering a programmatic API to write to and read from it.

4.10.2 Candidate list representation model concerns

- (**P1A**) Defining and persistently storing a list of candidates.
- (**P1A**) Providing an interface for creating and adding entries to such a list of candidates.
- (**P1A**) Iterating through the list of candidates, retrieving information about each such candidate. This information should at least include a name and picture for each candidate.

4.10.3 Candidate list representation model interactions

The candidate list representation model depends on no other components

The following components directly depend on the candidate list representation model:

- *Candidate/voter list tools*: The candidate/voter lists tools must be able to write the list of candidates to the candidate list representation model, so that the different ballot schemes may be able to read this list without directly communicating with the candidate/voter lists tools or having to understand the more complex criteria based user list format (see formats below)
- *Multivote ballot scheme extension*: The multivote ballot scheme extension uses the candidate list representation model to populate the entries of its ballot template from a candidate list object.
- *Preferential ballot scheme extension*: The preferential ballot scheme extension uses the candidate list representation model to populate the entries of its ballot template from a candidate list object.

4.10.4 Candidate list representation model formats

- *Candidate list format* (**opaque**): The persistent file format in which fixed lists of candidates are stored. This does not include audit information or selection criteria, just a list of users and possibly some profile information.

4.11 Server-to-client communicator

4.11.1 Server-to-client communicator description

The server-to-client communicator is a Plone product exposing the actions of the PloneVote server components (notably the SCE base module) required by the desktop client, in the form of a web service. Essentially, this module provides a proxy web service API to a safe subset of the PloneVote server API, which is used by external desktop clients. Ideally, it should allow authenticated log-in to Plone by the desktop client on behalf of its user.

4.11.2 Server-to-client communicator concerns

- (**P1A**) Providing a web service interface to the PloneVote system that allows listing all ongoing elections and their current state (phase and steps started/concluded). All actions allowed by the following concerns should be exposed through that web service interface.
- (**P2**) Allowing a desktop client or some other web service consumer to create a securely authenticated session with the Plone CMS system.
- (**P3**) Allowing an authenticated user, who was selected as an election trustee for a particular election by the election administrator, to accept that assignment (during step 2 of the SCE).
- (**P2**) Allowing an authenticated election trustee to upload their partial public key for an election, provided he is indeed listed as a trustee for the election and step 3 of the SCE is ongoing ("Election commission key set-up").
- (**P1B**) Allowing all partial public keys and the combined public key for an election to be downloaded. This should be available to any user after key set-up.
- (**P1B**) Allowing all sets of encrypted votes (the original set and all verifiable shuffles) to be downloaded by any user after the start of the counting phase of an election.
- (**P2**) Allowing the client to leave a mark on the server indicating that it is currently performing a verifiable shuffle. This mark can be read by other clients before starting their own shuffles.
- (**P1B**) Allowing a new verifiable shuffle for the last available set of encrypted votes to be uploaded, together with proof of correct shuffling. The server must then verify that the shuffle was done honestly and correctly.
- (**P1A**) Allowing a verifiable partial decryption of the last shuffled ballot set to be uploaded.
- (**P1B**) Allowing full download of all materials required to verify the election security, including: all ballot sets, all proofs of shuffling, all verifiable partial decryptions, the ballot template, the list of voters having voted and the required details of the election's configuration (ballot and vote counting scheme, for example).

4.11.3 Server-to-client communicator interactions

The server-to-client communicator depends on:

- *SCE base module*: The server-to-client communicator mostly works by exposing actions of the SCE base module through a web service API for

use by the trustee/auditor client. The SCE module must accept, verify and store formats uploaded through the server-to-client communicator. It must also respond to commands and honor authenticated channels created by the communicator.

The following components directly depend on the server-to-client communicator:

- *Client-to-server communicator*: The client-to-server communicator needs to be able to send commands to and upload files through the server-to-client communicator. As a local proxy to all actions allowed by the server-to-client communicator component, the client-to-server communicator must understand the full public web service API of the server side component.

4.11.4 Server-to-client communicator formats

- *Web service exchange formats* (**transparent**): The server-to-client communicator and the client-to-server communicator must exchange data and instructions through the web. This communication would most likely depend on some standard format, such as one of the various commonly used XML-based web service exchange formats. Both this format as well as the specifics of the web service API, must be agreed upon by both communicator components early in their design.

4.12 Client-to-server communicator

4.12.1 Client-to-server communicator description

The client-to-server communicator component is a python library for use by the desktop client to interact with the PloneVote server exposed web services. Essentially, this module provides a proxy local API to the web service API created by the server-to-client communicator component.

4.12.2 Client-to-server communicator concerns

The concerns of the client-to-server communicator are closely tied to those of the server-to-client communicator, since it provides the same services to the local desktop client as the server-side component exposes through the web. We can summarize the client-to-server communicator responsibilities as follows:

- (**Varies**) Provide a locally transparent API to all actions exposed through the web by the server-to-client communicator component.

4.12.3 Client-to-server communicator interactions

The client-to-server communicator depends on:

- *Server-to-client communicator*: The client-to-server communicator needs to be able to send commands to and upload files through the server-to-client communicator. As a local proxy to all actions allowed by the server-to-client communicator component, the client-to-server communicator must understand the full public web service API of the server-side component.

The following components directly depend on the client-to-server communicator:

- *Base client (GUI)*: The base PloneVote desktop client uses the client-to-server communicator component to retrieve information about all elections running in a particular PloneVote server and their current state.
- *Trustee client operations*: The trustee client operations module uses the client-to-server communicator in order to:
  - Upload a partial public key to the server after being authenticated as an election trustee (if authentication is provided)
  - Download all partial public keys and the combined public key for key set-up verification (anonymously)
  - Download the last shuffled set of encrypted votes
  - Upload a partial decryption of the previous encrypted votes.
  - (Possibly) Download all shuffled sets of encrypted votes and proofs to verify shuffling before decryption.
- *Auditor client operations*: The auditor client operations module uses the client-to-server communicator in order to (anonymously):
  - Download sets of encrypted votes for shuffling
  - Announce an in-progress verifiable shuffling
  - Upload a shuffled set of votes with a proof of correct shuffling
  - Download all materials required to verify an election

4.12.4 Client-to-server communicator formats

- *Web service exchange formats* (**transparent**): For communication with the server-to-client communicator component. See 4.11.4.

4.13 Base client (GUI)

4.13.1 Base client description

The base client component implements the core graphical user interface and shared functionality of the desktop trustee and auditor clients. This component uses the client-to-server communicator component to interact with the server and display the list of running elections.

The trustee and auditor client operations can register actions with the base client in certain election states. When viewing a particular running election, the user is presented the available actions. Upon selecting an action, a wizard is loaded into the base client where workflow is controlled by the particular component that defined the action (trustee or auditor client operations), which guides the user in performing said action.

4.13.2 Base client concerns

- (**P1A**) Providing a graphical interface desktop client that can connect anonymously to a PloneVote server and list all ongoing elections, showing their current phase and step.
- (**P1A**) Allowing trustee and auditor client operations components (and possibly other future components) to register possible actions given an election's state.
- (**P1A**) Listing available actions to the user, allowing them to select any such action and loading the appropriate interface elements and code from the component defining the election.
- (**P2**) Providing a shared interface for client log-in and log-out to the server, using the client-to-server communicator ability to start and track and authenticated session with the server.
- (**P3**) Allowing components to register menu operations and windows, besides election actions. This would be needed, for example, to implement sophisticated key pair management in the trustee client operations module.

4.13.3 Base client interactions

The base client depends on:

- *Client-to-server communicator*: The base client uses the client-to-server communicator component to retrieve information about all elections running in a particular PloneVote server and their current state. It may also use it to establish an authenticated session with the server.

The following components directly depend on the base client:

- *Trustee client operations*: The trustee client operations module must register its actions with the base client and perhaps use it to authenticate with the server.
- *Auditor client operations*: The auditor client operations module must register its actions with the base client.

4.13.4 Base client formats

The base client introduces no new transparent formats, though it may save some internal configuration in the same machine where it executes.

4.14 Trustee client operations

4.14.1 Trustee client operations description

The trustee client operations module defines and implements all actions specific to the trustee desktop client and registers those actions with the base client. Trustee client actions/operations include: key pair set-up, public key set-up verification and partial decryption.

4.14.2 Trustee client operations concerns

- (**P1A**) Creating a private and partial public key pair (using PloneVote Crypto) and storing both in the local machine. The client should prompt for this operation to be realized before uploading a partial public key for an election for the first time.
- (**P3**) Providing the option of protecting a private key with a passphrase and symmetric encryption.
- (**P3**) Adding a menu item and window to the base client to manage private and partial public keys. Allowing for keys to be saved to external files.
- (**P3**) Implementing a trustee appointment confirmation/denial action and registering it with the base client. This action should be available during step 2 of the SCE ("Election Commission Selection") and requires authentication to the server.
- (**P1A**) Implementing a key set-up action and registering it with the base client. This action should be available during step 3 of the SCE ("Election Commission Key Set-up").
- (**P2**) Allowing the partial public key to be uploaded to the server as part of the key set-up action. This requires authentication to the server as an election trustee.

- (**P1B**) Implementing a threshold scheme verification action and registering it with the base client. This action should be available during step 4 of the SCE ("Election Commission Key Verification").
- (**P1A**) Implementing a vote decryption action and registering it with the base client. This action should be available during step 15 of the SCE ("Vote Decryption"). This action consists on downloading the last shuffled set of votes, creating a partial decryption and uploading that decryption to the PloneVote server.

4.14.3 Trustee client operations interactions

The trustee client operations module depends on:

- *PloneVote Crypto Library*: The trustee client operations component directly uses the the following PloneVote Crypto Library operations:
  - Generating a private and partial public key pair.
  - Constructing a threshold encryption scheme (to verify the correctness of the combined public key generated by the server).
  - Generating a partial decryption (with proof).
  - (Possibly) Checking proofs of partial decryption and/or correct shuffling.
- *Client-to-server communicator*: The trustee client operations module uses the client-to-server communicator in order to:
  - Upload a partial public key to the server after being authenticated as an election trustee (if authentication is provided)
  - Download all partial public keys and the combined public key for key set-up verification (anonymously)
  - Download the last shuffled set of encrypted votes.
  - Upload a partial decryption of the previous encrypted votes.
  - (Possibly) Download all shuffled sets of encrypted votes and proofs to verify shuffling before decryption.
- *Base client*: The trustee client operations module must register its actions with the base client and perhaps use it to create authenticate with the server.

No other component directly depends on the trustee client operations module.

4.14.4 Trustee client operations formats

Besides possibly storing preferences, the trustee client operations module must be able to store the private and partial public keys generated by PloneVote Crypto as persistent local files:

- *Armored key file* (**opaque**): A file format for storing private and partial public keys, possibly with passphrase protection.

4.15 Auditor client operations

4.15.1 Auditor client operations description

The auditor client operations module defines and implements all actions specific to the auditor desktop client and registers those actions with the base client. Auditor client actions/operations include: verifiable ballot shuffling and election verification.

4.15.2 Auditor client operations concerns

- (**P1B**) Implementing a verifiable shuffling action and registering it with the base client. This action should be available during step 14 of the SCE ("Verifiable Shuffling"). Marking that a shuffling action has commenced and testing that the shuffle was accepted by the server are part of this concern.

- (**P1B**) Implementing an election verification action and registering it with the base client. This action should be available during step 17 of the SCE ("Election Verification and Validation"). This action includes all verifications implemented in the client, at the very least including: verifying all proofs of shuffling, verifying all proofs of partial decryption and their combination into a full decryption, and performing a recount (it could also include own vote verification).

4.15.3 Auditor client operations interactions

The auditor client operations module depends on:

- *PloneVote Crypto Library*: The auditor client operations component directly uses the the following PloneVote Crypto Library operations:

  - Generating receipts (to check those generated by the server).
  - Performing a verifiable shuffle.
  - Checking a proof of correct shuffling.
  - Checking a proof of partial decryption.
  - Combining partial decryptions into a full decryption (to verify the combination done by the server).

- *Ballot scheme base module*: The auditor client operations component needs to verify the ballot template of the election, parse the decrypted votes and perform a vote count by using this module in combination with the vote counting base module and the particular extensions used for the election. The extension modules need to be distributed with the client,

but are loaded indirectly through the ballot and vote counting scheme base modules, so the auditor client operations module does not depend upon any particular ballot scheme or vote counting extension.

- *Vote counting scheme base module*: (see "*Ballot scheme base module*" above)
- *Client-to-server communicator*: The auditor client operations module uses the client-to-server communicator in order to (anonymously):
  - Download sets of encrypted votes for shuffling
  - Announce an in-progress verifiable shuffling
  - Upload a shuffled set of votes with a proof of correct shuffling
  - Download all materials required to verify an election
- *Base client*: The auditor client operations module must register its actions with the base client.

No other component directly depends on the auditor client operations module.

4.15.4 Auditor client operations formats

The auditor client operations module introduces no new storage or communication formats.

4.16 JavaScript vote casting client

4.16.1 JavaScript vote casting client description

The JavaScript vote casting client comprises the basic code, resources and interface of the PloneVote web client, coded in JavaScript, (X)HTML and CSS. This component should coordinate loading the correct client ballot wizard, encrypting the user vote, showing the vote receipt and uploading the encrypted vote to the server.

4.16.2 JavaScript vote casting client concerns

- (**P2**) Verifying browser support for the web client, including the necessary cryptographic routines. Ideally, this should be done by testing JavaScript API and DOM elements, not by browser detection. It should also be verified that JavaScript is enabled in the browser.
- (**P1A**) Downloading all required components and resources for the web client, including the ballot template, the correct ballot wizard and the encryption routine(s).
- (**P1A**) Presenting an offline client for the user to create their vote using the ballot wizard, verify the text of the vote, encrypt it, get a voting receipt and upload the vote to the server.

4.16.3 JavaScript vote casting client interactions

The JavaScript vote casting client depends on:

- *SCE base module*: The JavaScript vote casting client should be able to submit an encrypted vote to the SCE base module on the server for voter authorization and vote casting (recording).
- *JavaScript encryption routine(s)*: The JavaScript vote casting client will call the JavaScript encryption routine(s) module to encrypt a vote before casting it to the server and to generate a vote receipt on the user's browser.

The following components directly depend on the JavaScript vote casting client:

- *SCE base module*: The SCE base module should be able to deploy the JavaScript vote casting client to the voter's browser and accept the encrypted votes generated by the client as originating from the user.
- *JavaScript multivote ballot wizard*: The JavaScript multivote ballot wizard will be passed to the vote casting client by the server as a parameter and run as a subroutine of the vote casting client. This component must then be aware enough of the vote casting client to operate in the context of said client and return control to it (together with the plaintext vote) once the user has finished creating their vote. It must also obtain the ballot template from the vote casting client.
- *JavaScript preferential ballot wizard*: (Same interaction as that of the multivote ballot wizard above).

4.16.4 JavaScript vote casting client formats

The JavaScript vote casting client introduces no new storage or communication formats.

4.17 JavaScript encryption routine(s)

4.17.1 JavaScript encryption routine(s) description

The JavaScript encryption routine(s) component encapsulates the code required to perform public key encryption on cast votes before uploading them to the server. This component should also contain the code for creating vote receipts.

4.17.2 JavaScript encryption routine(s) concerns

- (**P1B**) Encrypting an arbitrary vote plaintext in a manner compatible with the PloneVote Crypto Library and using a combined public key provided by the server.

- (**P1B**) Generating a vote receipt in a manner compatible with the PloneVote Crypto Library

4.17.3 JavaScript encryption routine(s) interactions

The JavaScript encryption routine(s) component depends on:

- *PloneVote Crypto Library*: The JavaScript encryption module must be able to operate with the public key format of the PloneVote Crypto Library in order to encrypt votes, generating the same format of ciphertext than this library. It must also create the same format of vote receipts.

The following components directly depend on the JavaScript encryption routine(s) component:

- *JavaScript vote casting client*: The JavaScript vote casting client will call the JavaScript encryption routine(s) module to encrypt a vote before casting it to the server and to generate a vote receipt on the user's browser.

4.17.4 JavaScript encryption routine(s) formats

The JavaScript encryption routine(s) component introduces no new storage or communication formats. However, it must be able to manipulate the combined public key format defined by PloneVote Crypto Library component and create an encrypted vote in the PloneVote Crypto ciphertext format.

4.18 JavaScript multivote ballot wizard

4.18.1 JavaScript multivote ballot wizard description

The JavaScript multivote ballot wizard is the component tasked with providing a graphical interface for defining and encoding a multivote vote within the user's web browser. It must be able to read a multivote ballot template and expose a simple visual way of selecting the candidates for whom the user wishes to vote. This component should also check that the created vote is within the limits specified in the ballot template (for example, that the selected candidates fall between the minimum and maximum specified).

It is important that the JavaScript multivote ballot wizard be as easy to use as possible and allow the user to clearly view and review how they will be voting before encryption occurs.

4.18.2 JavaScript multivote ballot wizard concerns

- (**P1A**) Providing a graphical user interface, within the web browser, for visualizing, creating and encoding a multivote vote given the ballot template.

- (**P2**) Verifying that the created vote falls within the limits defined in the ballot template.

4.18.3 JavaScript multivote ballot wizard interactions

The JavaScript multivote ballot wizard depends on:

- *Multivote ballot scheme extension*: The JavaScript multivote ballot wizard must be able to parse the multivote ballot template format in order to display a ballot to the user for vote casting. It must then be able to record the user preferences into a multivote vote format compatible with the one used by the multivote ballot scheme extension.
- *JavaScript vote casting client*: The JavaScript multivote ballot wizard will be passed to the vote casting client by the server as a parameter and run as a subroutine of the vote casting client. The component must then be aware enough of the vote casting client to operate in the context of said client and return control to it (together with the plaintext vote) once the user has finished creating their vote. It must also obtain the ballot template from the vote casting client.

No other component directly depends on the JavaScript multivote ballot wizard.

4.18.4 JavaScript multivote ballot wizard formats

The JavaScript multivote ballot wizard introduces no new storage or communication formats. However, it must be able to parse a multivote ballot template, as defined by the multivote ballot scheme extension, and encode the created vote into a multivote vote format also compatible with the one used by the multivote ballot scheme extension.

4.19 JavaScript preferential ballot wizard

4.19.1 JavaScript preferential ballot wizard description

The JavaScript preferential ballot wizard is the component tasked with providing a graphical interface for defining and encoding a preferential vote within the user's web browser. It must be able to read a preferential ballot template and expose a simple and visual way of selecting and ordering the candidates for whom the user wishes to vote. This component should also check that the created vote is within the limits specified in the ballot template (for example, that the selected candidates fall between the minimum and maximum specified in the ballot template).

It is important that the JavaScript preferential ballot wizard be as easy to use as possible and allow the user to clearly view and review how they will be voting before encryption occurs.

4.19.2 JavaScript preferential ballot wizard concerns

- (**P3**) Providing a graphical user interface, within the web browser, for visualizing, creating and encoding a preferential vote given the ballot template.
- (**P3**) Verifying that the created vote falls within the limits defined in the ballot template.

4.19.3 JavaScript preferential ballot wizard interactions

The JavaScript preferential ballot wizard depends on:

- *Preferential ballot scheme extension*: The JavaScript preferential ballot wizard must be able to parse the preferential ballot template format in order to display a ballot to the user for vote casting. It must then be able to record the user preferences into a preferential vote format compatible with the one used by the preferential ballot scheme extension.
- *JavaScript vote casting client*: The JavaScript preferential ballot wizard will be passed to the vote casting client by the server as a parameter and run as a subroutine of the vote casting client. The component must then be aware enough of the vote casting client to operate in the context of said client and return control to it (together with the plaintext vote) once the user has finished creating their vote. It must also obtain the ballot template from the vote casting client.

No other component directly depends on the JavaScript preferential ballot wizard.

4.19.4 JavaScript preferential ballot wizard formats

The JavaScript preferential ballot wizard introduces no new storage or communication formats. However, it must be able to parse a preferential ballot template, as defined by the preferential ballot scheme extension, and encode the created vote into a preferential vote format also compatible with the one used by the preferential ballot scheme extension.

4.20 Event logging subsystem

4.20.1 Event logging subsystem description

The event logging subsystem is the PloneVote component concerned with offering logging facilities to the rest of the Plone products which are part of the system, logging messages about each particular election and about the operation of the server side components of PloneVote in general.

For a list of the minimal information that should be logged, see section 3.4.4 of the PloneVote Requirements Document. Keep in mind that the logging subsystem is not statically aware of any of the information logged by PloneVote, it

just provides a generic interface for logging messages. This component may simply proxy logging requests to the Plone logging mechanisms, or else implement separate log files for use by PloneVote.

4.20.2 Event logging subsystem concerns

- (**P2**) Providing and implementing a general interface for logging system messages.
- (**P3**) Ensuring that logs are append-only and not overwritable by the PloneVote system itself.

4.20.3 Event logging subsystem interactions

The event logging subsystem depends on no other components.

The following components directly depend on the event logging subsystem:

- *SCE base module*: The SCE base module uses the event logging subsystem to log all information related to the election and its events.

4.20.4 Event logging subsystem formats

- *Log formats* (**opaque**): The event logging subsystem uses either a single or multiple log files to persistently store messages logged through it.

4.21 Notification subsystem

4.21.1 Notification subsystem description

The notification subsystem is the PloneVote component charged with delivering notifications to its users. This subsystem should provide per-user notifications, in the form of messages visible upon log-in to the Plone site hosting PloneVote, as well as via e-mail. The notification subsystem does not define the events for which it offers notification, it merely provides a generic interface for programmatically sending messages to users.

4.21.2 Notification subsystem concerns

- (**P2**) Providing a generic interface for delivering notifications to users.
- (**P2**) Showing pending notifications in the Plone site upon user log-in, and allowing the user to acknowledge or dismiss the notifications.
- (**P2**) Sending out notifications as mails to the user's e-mail account.
- (**P3**) Allowing notifications to specify priority and supporting either a global or a per-election configuration option to select how messages of different priorities are delivered (e.g. high importance notifications are

delivered by site and e-mail, while low importance notifications are shown only on the site). Possibly have per-user preferences regarding e-mail notifications as well.

4.21.3 Notification subsystem interactions

The notification subsystem depends on no other components.

The following components directly depend on the notification subsystem:

- *SCE base module*: The SCE base module uses the notification subsystem to deliver relevant site and/or e-mail notifications to users regarding the running election(s).

4.21.4 Notification subsystem formats

The notification subsystem may define internal formats for storing notifications visible inside the Plone site, but it does not otherwise define any transparent formats.