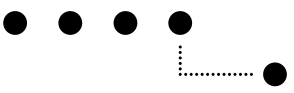

Berner Fachhochschule
Technik und Informatik

Selectio Helvetica

Prozesse und Implementation

von
Severin Hauser

Master Thesis

Fachbereich Informatik
Departement für Technik und Informatik
Berner Fachhochschule

21. Januar 2011

Advisor : Prof. Dr. Eric Dubuis
Experte : Prof. Dr. Andreas Steffen

Abstract

Baloti ist ein Webportal für in der Schweiz lebende Ausländer/Innen. Damit diese Erfahrungen sammeln können mit nationalen Abstimmungen, werden die Abstimmungen durch Baloti simuliert. Dazu wird ein elektronisches Abstimmungssystem benötigt. Diese Arbeit stellt die Konkretisierung, Selectio Helvetica, eines Protokolls für ein Abstimmungssystem vor, das sich an den spezifischen Anforderungen von Baloti orientiert. Damit dieses Protokoll implementiert werden kann, wurden im Rahmen dieser Arbeit aus den kryptographischen Funktionen die notwendigen Prozesse modelliert. Anhand dieser Prozesse war es anschliessend möglich eine Architektur und die benötigten Schnittstellen zu definieren. Am Ende wird eine Implementation basierend auf diesen Schnittstellen vorgestellt.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Baloti ist ein Webportal für in der Schweiz lebende Ausländer/Innen. Ziel dieses Portal ist es, ihnen die politischen Prozesse der Schweiz näher zu bringen und sie mit dem Ablauf der Wahlen und Abstimmungen vertraut zu machen. In diesem Zusammenhang möchte Baloti den Ausländer/Innen ermöglichen ihre Meinung zu nationalen Abstimmungen auf dem Portal kundzutun. Für diese Aufgabe soll ein elektronisches Abstimmungssystem verwendet werden. Dieses Abstimmungssystem soll möglichst viele Anforderungen erfüllen, die auch an eine reale Abstimmung gestellt werden.
Die Arbeitsgruppe Swissvote der Berner Fachhochschule wurde mit der Aufgabe betraut, ein solches System zu definieren und anschliessend eine dazu passende Implementation zu erstellen. Dafür wurde ein bestehendes kryptographisches Protokoll für Abstimmungssysteme von Spycher und Haenni [SH10] konkretisiert und an die Bedürfnisse von Baloti angepasst. Im Zuge dieser Arbeit war es das Ziel aus diesem angepassten Protokoll, Selectio Helvetica, Prozesse für eine mögliche Implementation zu erstellen. Dabei wird in dieser Arbeit auch das zugrundelegende Protokoll beschrieben. Ziel dieser Beschreibung ist es jedoch nicht, die Eigenschaften des Protokolls zu diskutieren, sondern dem Leser ein grundsätzliches Verständnis des Protokolls und dessen kryptographischen Funktionen zu vermitteln, so dass er den Ablauf innerhalb der entwickelten Prozesse verstehen kann.
Anhand der Prozesse wurde anschliessend die Architektur und die Schnittstellen für eine Implementation erarbeitet. Dabei wurde auf eine möglichst modulare und leicht anpassbare Konstruktion geachtet. Anschliessend wurde auf Basis der Architektur mit der Implementation einzelner Komponenten begonnen.
In Kapitel 2 wird die Ausgangslage und Zeitstruktur dieser Arbeit beschrieben. In Kapitel 3 werden die Rollen und Phasen des Protokolls vorgestellt und einige kryptographische Funktionen erklärt, welche im Protokoll verwendet werden.
In Kapitel 4 folgt mit Selectio Helvetica Light eine vereinfachte Version des Protokolls, welche für eine erste Implementation verwendet wurde.
In Kapitel 5 wird dann schliesslich das Protokoll Selectio Helvetica vorgestellt, wie es für Baloti implementiert werden soll. In Kapitel 6 wird die Architektur vorgestellt und die dazugehörigen Bausteine und Schnittstellen beschrieben. In Kapitel 7 findet man Implementationsdetails zu den einzelnen Artefakten. In Kapitel 8 schliesslich wird ein Fazit über diese Arbeit gezogen und mögliche Entwicklungen in der Zukunft aufgezeigt.

2 Ausgangslage

Als diese Master Thesis im März 2010 startete stand ursprünglich die Implementation des Protokolls TrustVote [KDH10] auf dem Plan. Leider fanden die Autoren des Protokolls im Mai einige Schwachstellen in TrustVote, welche eine Implementierung wenig sinnvoll erscheinen liessen.
Glücklicherweise hat meine Projektgruppe in dieser Zeit von einem Projekt des Zentrum für Demokratie Aarau erfahren. Dieses Projekt, Baloti, hat zwei Hauptziele. Einerseits möchte man mittels Baloti den in der Schweiz lebenden Ausländer das Konzept unserer Abstimmungen näher bringen und sie über die Abläufe während der Abstimmung informieren. Anderseits geht es auch darum heraus zu finden, wie dieser Teil der Bevölkerung zu den nationalen Vorlagen steht.
In diesem Zusammenhang hat sich das Zentrum für Demokratie für die Verwendung eines elektronischen Abstimmungssystems entschieden. Da ihnen das nötige Fachwissen fehlt um ein solches System im Alleingang zu entwickeln und betreiben, haben wir uns angeboten sie bei diesem Schritt zu unterstützen. Sie haben Interesse bekundet und so begannen wir die Zusammenarbeit. Dies war für unser Team eine sehr spannende Ausgangslage. Weil unsere Ressourcen sehr limitiert waren, war schnell klar, dass es sinnvoll ist, das Thema dieser Master Thesis anzupassen. Ab Mai 2010 also arbeitete ich im Team um die Entwicklung des Projekts Selectio Helvetica, eines elektronischen Abstimmungssystems für Baloti.
Meine Aufgabe innerhalb des Projekts war es einerseits das verwendete Protokoll in implementationsfähige Prozesse abzubilden und andererseits eine Vorlage für die Implementation zu erstellen.
Nach einigen Abklärungen war klar, dass wir das von Spycher und Haenni entwickelte Protokoll [SH10] als Basis für das elektronische Abstimmungssystem verwenden wollten. Allerdings gab es einige Baloti spezifische Anforderungen, welche beim Design des Protokolls und bei der späteren Implementation unbedingt eingehalten werden mussten. Beim Protokoll musste es möglich sein, dass ein Benutzer sich jeder Zeit registrieren und dann sofort abstimmen kann. Dies widersprach dem SH10 Protokoll, bei dem von einem festen Wählerregister vor dem Beginn der Abstimmung ausgegangen wird. Ausserdem wollte Baloti bestimmen, wer sich registrieren kann.
Auf technischer Ebene war die Anforderung, dass die Implementation des Voters auf standard Webtechnologien basieren muss, damit die Einstiegshürde tief gehalten werden kann und sich die Implementation gut in den Webauftritt von Baloti einfügt.
Ausserdem benötigte Baloti bereits im September eine erste Implementation, damit die zu diesem Zeitpunkt stattfindenden nationalen Abstimmungen begleitet werden konnten. Das ergab folgenden Zeitplan für dieses Projekt.

2010				
Mai	**Juni**	**Juli**	**August**	**September**
Protokoll anpassen	Prototypen erstellen	Implementation SH Lite	Implementation SH Lite	Betrieb SH Lite

2010			2011	
Oktober	**November**	**Dezember**	**Januar**	**Februar**
Spezifikation SH	Implementation SH	Master Thesis	Master Thesis	Implementation SH

Abbildung 2.1: Zeitplan

3 Protokollgrundlagen

3.1 Rollen

3.1.1 Administration

Die Administration möchte eine Abstimmung ausrichten. Sie bestimmt alle Daten rund um die Abstimmung. Dazu zählt auch, dass sie eine Liste der stimmberechtigten Voter bestimmt und entscheidet, welche Trustees für diese Abstimmung vertrauenswürdig sind. Die Administration ist auch verantwortlich für die Bereitstellung aller öffentlichen Daten. Ausserdem will die Administration am Ende der Abstimmung das Resultat berechnen können.

$$\mathcal{A}$$

3.1.2 Trustee

Der Trustee vertritt die Interessen einer Gruppe von Votern. Im Zuge dieser Aufgabe stellt er sicher, dass bei einer Abstimmung alles korrekt abläuft. Dies geschieht, indem der Trustee bei den kryptographischen Funktionen mitarbeitet.
Pro Abstimmung gibt es immer mehrere Trustees. Dabei sollten möglichst verschiedene Interessengruppen vertreten sein.

$$\mathcal{T} = T_1, \ldots, T_n$$

3.1.3 Voter

Der Voter möchte seine Stimme für eine bestimmte Abstimmung abgeben. Dabei ist für ihn wichtig, dass er sicher sein kann, dass seine Stimme gezählt wird und niemand weiss wie er gestimmt hat.
Am Ende der Abstimmung will der Voter kontrollieren, dass richtig gezählt wurde.

$$\mathcal{V} = V_1, \ldots, V_m$$

3.2 Phasen

Die Protokoll wurden in dieser Arbeit in unterschiedliche Phasen unterteilt. Dies soll helfen, die Übersicht über die Protokolle zu verbessern, und vereinfacht die Beschreibung der notwendigen Prozesse.

3.2.1 Erstellung der VoterRoll

In dieser Phase wird eine Liste der stimmberechtigten Voter, die sogenannte *voter roll*, vorbereitet. Ausserdem werden in dieser Phase die kryptographischen Basiswerte gesetzt, welche auch in den späteren Phasen verwendet werden.

3.2.2 Erstellung der Abstimmung

In dieser Phase erstellt die Administration eine neue Abstimmung und fordert die Trustees auf, die notwendigen kryptographischen Schritte zu berechnen.

3.2.3 Abstimmen

In dieser Phase können die Voter ihre Stimme für eine Abstimmung abgeben.
Falls ein Voter noch nicht registriert ist, dann kann er sich in dieser Phase auch registrieren.

3.2.4 Auszählen

In dieser Phase wird die Korrektheit der erhaltenen Stimmen kontrolliert und diese anschliessend ausgezählt

3.3 Kryptographische Funktionen

Die hier aufgeführten kryptographischen Funktionen werden im Protokoll von Selectio Helvetica verwendet. Da diese zum Teil sehr komplex sind und die Prozesse des Protokolls unlesbar aufgeblasen hätten, werden diese hier getrennt aufgeführt. Dabei wird folgendes Vorgehen gewählt: Zuerst werden die mathematischen Schritte und deren Sinn erklärt. Anschliessend folgt der Prozess in einer Abbildung. Dazu werden spezielle Punkte im Prozess erwähnt.
Wo es möglich war, wurden die bekannten Rollen aus dem Protokoll verwendet, um auch die Aufgabenverteilung innerhalb des Protokolls klar zu machen.

3.3.1 Verteilte Berechnung des Generators

Dieses Protokoll ermöglicht einer Gruppe von Trustees einen Generator verteilt zu berechnen. Bei in den folgenden Protokollen verwendeten Generatoren wird davon ausgegangen, dass diese mit dem hier beschriebenen Protokoll erstellt werden.

- Die Administration wählt zuerst folgende Werte aus:
  - Primzahl q
  - Primzahl p, wobei $p = 2q + 1$
- Jeder Trustee T_i wählt einen Generator g_i aus G_q, wobei G_q die Untergruppe von Z_p^* mit q Elementen ist.
- Anschliessend berechnet T_i ein *commitment* c_i für seinen Generator g_i und publiziert diesen Wert bei der Administration.

$$c_i = hash(g_i)$$

- Nachdem alle Trustees ihr *commitment* publiziert haben, werden die Generatoren publiziert.
- Die Administration kontrolliert ob alle Generatoren zu den dazugehörigen *commitments* passen. Alle Generatoren, für die dies zutrifft, werden in die Menge R aufgenommen.
- Die Administration berechnet den gemeinsamen Generator für alle gültigen Generatoren R.

$$g = \prod_{R} g_i$$

In Abbildung 3.1 ist der Prozess beschrieben, um einen verteilten Generator zu erstellen. Wie im Prozess ersichtlich ist, ist es wichtig, dass die Administration zuerst alle *commitments* erfragt, bevor sie die Generatoren verlangt. Ausserdem ist auch ersichtlich, dass die Administration ungültige Generatoren nicht in R aufnimmt, sondern verwirft. So wird sicher gestellt, dass keine ungültigen Generatoren im Schlussresultat auftauchen.

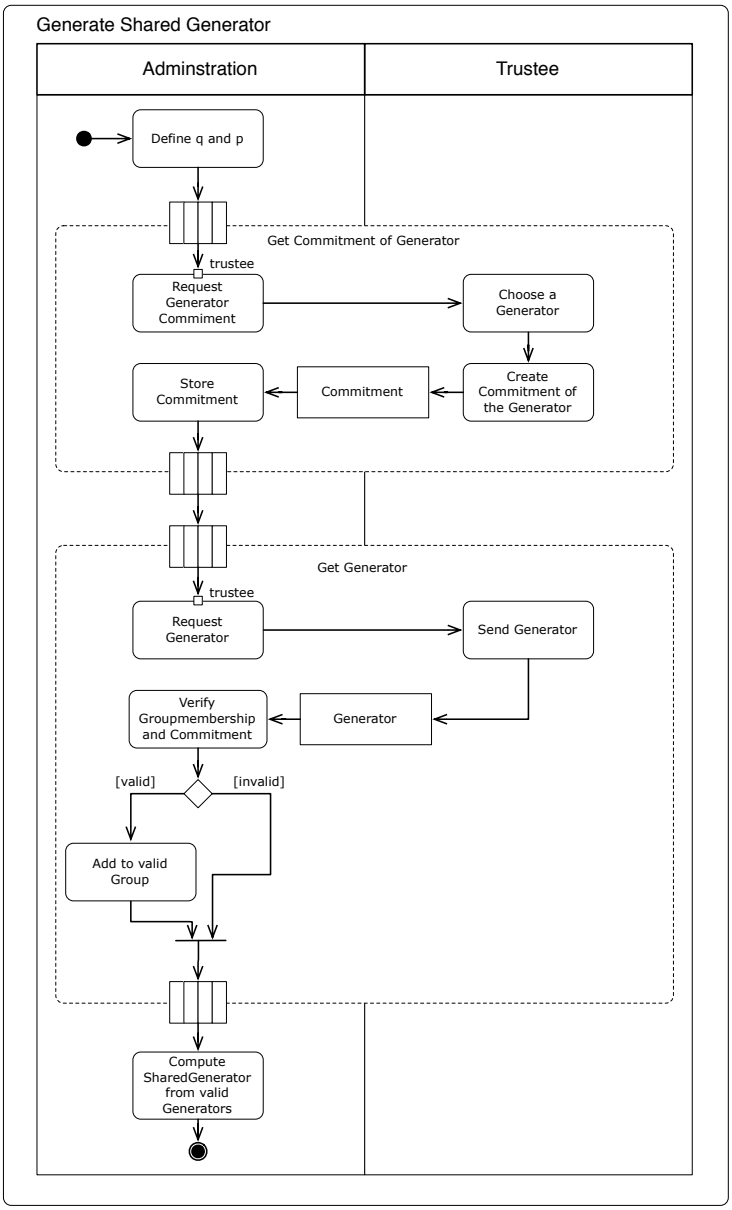

Abbildung 3.1: Verteilter Generator

3.3.2 Verteilter privater Schlüssel

Die Basis für dieses Protokoll beschreibt Pederson in [Ped91] im Detail. Es erlaubt einer Gruppe von Trustees, ein gemeinsames privates/öffentliches El-Gamal Schlüsselpaar zu erstellen, wobei niemand aus der Gruppe den privaten Schlüssel komplett kennt. Erst wenn eine genügend grosse Gruppe zusammen kommt, kann der private Schlüssel berechnet werden. Die dazu benötigte Grösse der Gruppe wird Schwellenwert genannt und zu Beginn des Protokolls festgelegt.

Phase 1 - Erzeuge die Polynome und commitments

In dieser Phase wählt die Administration die Rahmenbedingungen für das Protokoll aus und lässt die Trustees ihren Teil des gemeinsamen privaten Schlüssels wählen. Dies geschieht, indem jeder Trustee für sich ein Polynom erstellt. Damit er das Polynom anschliessend nicht mehr ändern kann, muss er sich zu jedem Koeffizienten dieses Polynoms festlegen (sogenannte *commitments*).

- Zuerst wählt die Administration folgende Werte aus:
  - Primzahl q
  - Primzahl p, wobei $p = 2q + 1$
  - Generator g aus G_q, wobei $\#G_q = q$ und $G_q \leq Z_p^*$
  - Schwellenwert t, Dieser Wert gibt an, wie viele Trustees zusammen kommen müssen, um den privaten Schlüssel zu berechnen.
- Nun teilt sie diese Werte allen Trustees mit.
- Jeder Trustee T_i wählt nun ein Polynom mit dem Grad $t - 1$. Dafür wählt er t zufällige Werte als Koeffizienten aus.

$$a_{i,0}, \ldots, a_{i,t-1} \in Z_q$$

- Damit der Trustee das Polynom später nicht mehr ändern kann muss er sich nun auf alle Koeffizienten *commiten*, indem er den Generator g mit jedem Koeffizienten einzeln potenziert.

$$A_{i,k} = g^{a_{i,k}}$$

- Nun sendet der Trustee alle *commitments* $A_{i,0}, \ldots, A_{i,t-1}$ an die Administration.
- Die Administration veröffentlich alle erhaltenen *commitments*.

Wie in Abbildung 3.2 zusehen ist, ist der Prozess für Phase 1 sehr einfach. Wichtig ist hier, dass die Trustees ein Polynom von der richtigen Grösse erstellen. Ausserdem muss auch darauf geachtet werden, dass ein Trustee erst in Phase 2 übergeht, wenn er Phase 1 beendet hat.

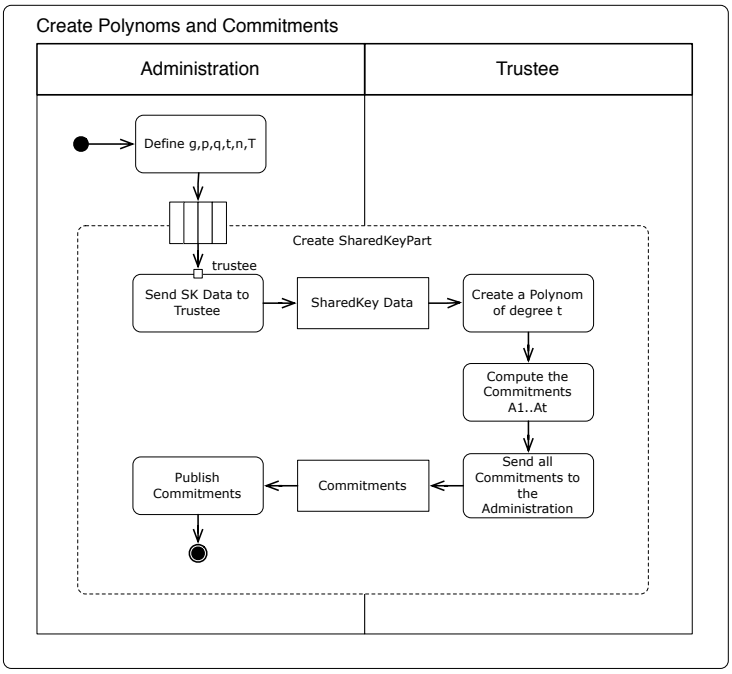

Abbildung 3.2: Verteilter privater Schlüssel Phase 1

Phase 2 - Sende Teilschlüsselinformationen

In dieser Phase tauschen die Trustees untereinander Informationen über ihre Polynome aus. Dies stellt sicher, dass eine Gruppe, deren Grösse den Schwellenwert überschreitet, auch den privaten Schlüssel berechnen kann. Damit hier niemand betrügen kann, werden die Informationen mit den in Phase 1 abgegebenen *commitments* verglichen.

$$T_i, T_j \in \mathcal{T} \text{ und } Q = \mathcal{T}$$

- Der T_i berechnet aus seinem Polynom die Teilschlüsselinformation s_{ij} für T_j.

$$s_{ij} = a_{i,0} + a_{i,1} \cdot j + \ldots + a_{i,t-1} \cdot j^{t-1}$$

- Der T_i sendet nun dieses s_{ij} an T_j.
- Der T_j überprüft das s_{ij}, indem er es mit den *commitments* von T_i vergleicht.

$$\prod_{k=0}^{t-1} A_{i,k}^{j^k} = g^{s_{i,j}}$$

- Wenn der Vergleich fehlgeschlagen ist, muss T_i dieses s_{ij} publizieren.
- Nachdem der T_i s_{ij} publiziert hat überprüft auch die Administration s_{ij} mit den *commitments* von T_i.

- Wenn der Vergleich fehlschlägt, disqualifiziert die Administration den Trustee T_i.

$$T_i \notin Q$$

Abbildung 3.3 zeigt den Prozess für Phase 2. Wie aus dem Prozess ersichtlich ist, informiert T_j die Administration, wie sein Vergleich ausgegangen ist. Dies ermöglicht dieser anschliessend, bei einem ungültigen s_{ij} , den Trustee T_i aufzufordern, dieses s_{ij} zu publizieren. Wenn das s_{ij} publiziert ist, kann die Administration dieses nun verifizieren und, falls es wirklich ungültig ist, den Trustee disqualifizieren.

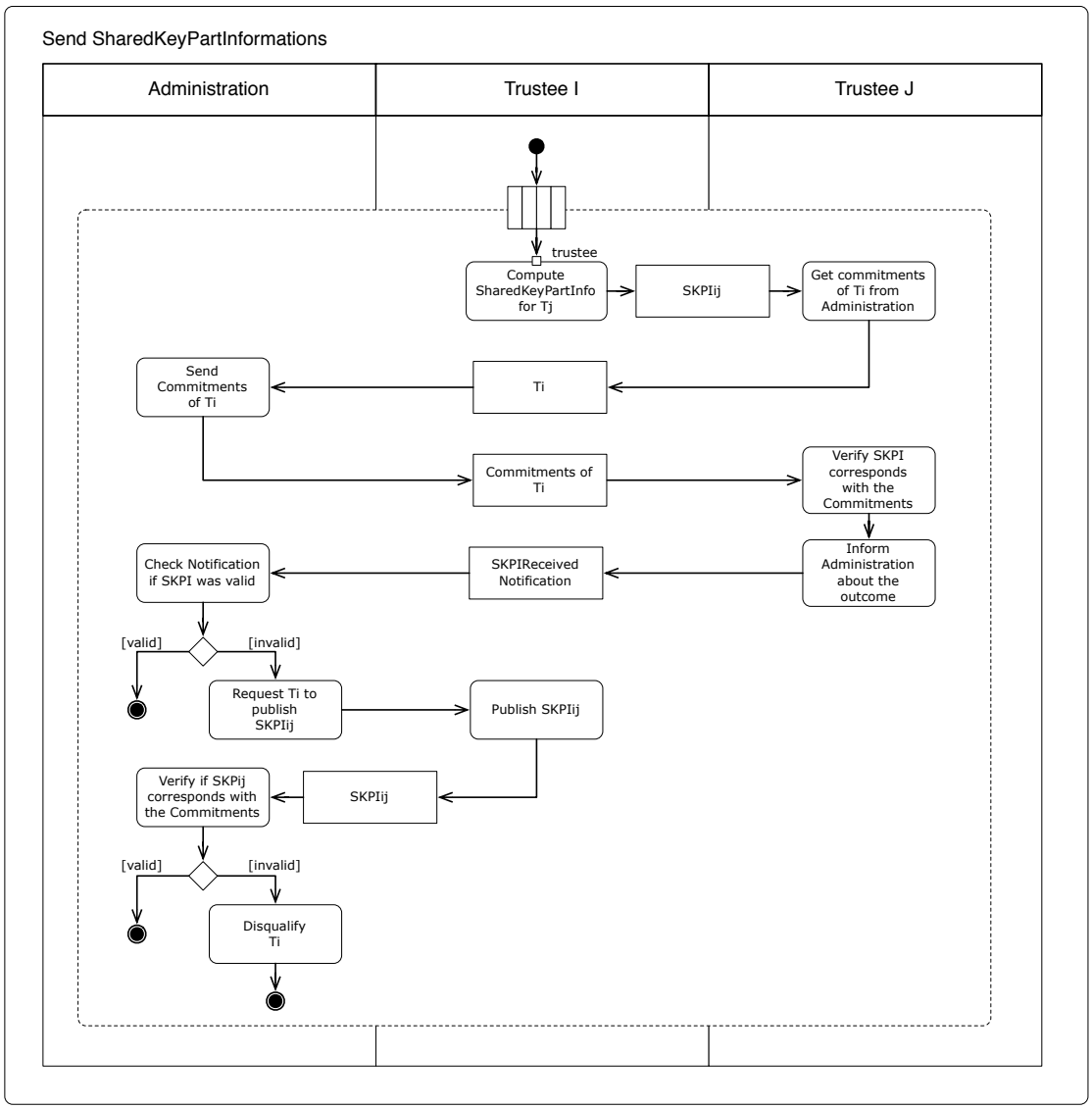

Abbildung 3.3: Verteilter privater Schlüssel Phase 2

Phase 3 - Berechne die privaten Teilschlüssel und den öffentlichen Schlüssel

In dieser Phase sind nur noch die Trustees dabei, welche nicht disqualifiziert wurden. Werte von Trustees, welche disqualifiziert wurden, werden nicht weiter verwendet. Die hier aufgeführten Schritte sind voneinander unabhängig und können in beliebiger Reihenfolge und parallel ausgeführt werden.

- Jeder Trustee berechnet mit Hilfe der erhaltenen Teilschlüsselinformationen seinen Teil des privaten Schlüssels.

$$s_j = \sum_{i \in Q} s_{i,j}$$

- Die Administration berechnet mit Hilfe der *commitments* für jeden Trustee einen öffentlichen Schlüssel zu seinem privaten Schlüssel.

$$h_j = \prod_{i \in Q} \prod_{k=0}^{t-1} A_{i,k}^{j^k}$$

- Ausserdem berechnet die Administration einen gemeinsamen öffentlichen Schlüssel aus den *commitments*.

$$h = \prod_{i \in Q} A_{i,0}$$

Da es sich hier um die reine Berechnung von mathematischen Formeln handelt, wurde für diese Phase kein Prozess erstellt.

Phase 4 - Berechne den gemeinsame privaten Schlüssel

In dieser Phase wird der gemeinsame private Schlüssel berechnet. Die hier verwendete Rolle *Client* soll zeigen, dass jeder den privaten Schlüssel berechnen kann, wenn er t Teile des privaten Schlüssels von den Trustees erhält.
Im Protokoll von Selectio Helvetica ist es jeweils ersichtlich, wer diese Rolle innehat.

- Der *Client* verlangt von allen Trustees deren privaten Teilschlüssel.
- Anschliessend vergleicht der *Client* die privaten Teilschlüssel mit den dazugehörigen öffentlichen Teilschlüssel.

$$\text{wenn } g^{s_i} = h_i \text{ dann } i \in Q$$

- Wenn der *Client* genügend gültige private Teilschlüssel erhalten hat, kann er nun den privaten Schlüssel berechnen.

$$s = \sum_{i \in Q} s_i \cdot \lambda_{i,Q} \text{ wobei } \lambda_{i,Q} = \prod_{l \in Q}^{i} \frac{l}{l-i}$$

Wie in Abbildung 3.4 zu sehen, ist der Prozesse für die Berechnung des privaten Schlüssels recht einfach. Um sicher zu gehen, dass das richtige Ergebnis heraus kommt, werden ungültige Teilschlüssel gar nicht erst gespeichert. Wichtig ist auch, dass der letzte Schritt nur erfolgreich ist, wenn der *Client t* gültige Teilschlüssel hat.

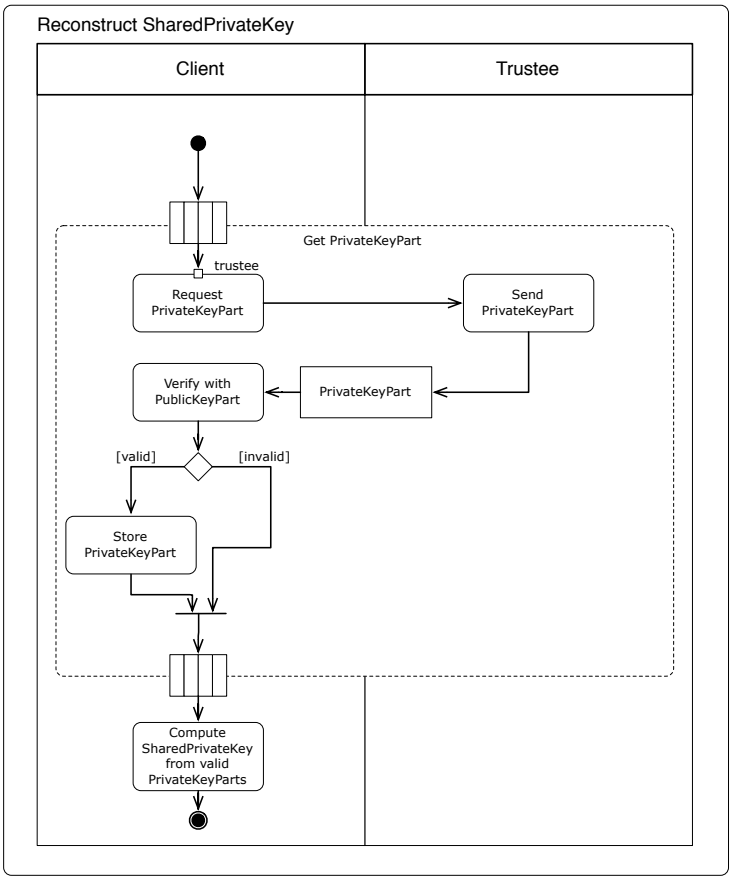

Abbildung 3.4: Verteilter privater Schlüssel Phase 4

3.3.3 Pseudonyme mischen

Dieses Protokoll wird in [SH10] beschrieben und erlaubt das Mischen und Anonymisieren einer Liste von öffentlichen ElGamal-Schlüsseln. Dabei bleibt der private Schlüssel immer gleich. Die neuen öffentlichen Schlüssel werden Pseudonyme genannt.
In Selectio Helvetica und Selectio Helvetica Light besitzt jeder Voter einen öffentlichen ElGamal-Schlüssel. Diese Technik erlaubt also, aus der Liste der stimmberechtigten Voter eine Liste mit Pseudonymen zu erstellen.
In diesem Protokoll wird auch eine Anlehnung von Randomized Partial Checking (RPC) aus [JJR02] verwendet, damit der Trustee beim Erstellen der Pseudonyme nicht betrügen kann.

Erstellen der commitments für den Beweis

Bei RPC wird von der Person, für die der Beweis erstellt wird, eine *challenge* benötigt. In diesem Protokoll ist dies nicht eine einzelne Person, sonder alle anderen Trustees. Damit keiner der Trustees seinen Teil der *challenge* an die Teile der anderen Trustees anpassen kann, müssen sich die Trustees vorgängig zu ihren Werten *commiten*. Die *challenge* besteht aus mehreren Permutationen, wobei jeder Trustee genau eine beisteuert.

- Zuerst wählt die Administration folgende Werte aus:
  - Primzahl q
  - Primzahl p, wobei $p = 2q + 1$
  - Generator g aus G_q, wobei $\#G_q = q$ und $G_q \leq Z_p^*$
  - Liste S der Voter Identitäten, welche ein Pseudonym erhalten sollen. $S_k \in G_q$
  - Generatoren $g_1, \ldots, g_k, f$, wobei k = Anzahl Einträge in S
- Jeder Trustee T_i erstellt für jeden anderen Trustee T_j jeweils eine Permutation $\phi_{i,j}$ des Vektors $v = \{0, \ldots, 0, 1, \ldots, 1\}$, wobei die Länge k des Vektors gleich der Grösse der Liste S ist und genau $\frac{k}{2}$ 0 und $\frac{k}{2}$ 1 enthalten sind.

$$\phi_{i,1} \ldots \phi_{i,n} \text{ für } i \neq j$$

- Nun berechnet er für alle diese Permutationen jeweils ein *commitment*. Damit das *commitment* nicht durch kombinatorische Versuche geöffnet werden kann, wird in der Berechnung noch einen Zufallswert $r_{i,j} \in Z_q$ eingebettet.

$$c_{i,j} = g_1^{\phi_{i,j}(1)} \cdot \ldots \cdot g_k^{\phi_{i,j}(k)} \cdot f^{r_{i,j}}$$

- Am Ende publiziert er alle berechneten *commitments* auf der Administration.

$$c_{i,1} \ldots c_{i,n}$$

Wie in Abbildung 3.5 zu sehen ist, ist der Prozess für die Erstellung der *commitments* sehr einfach.

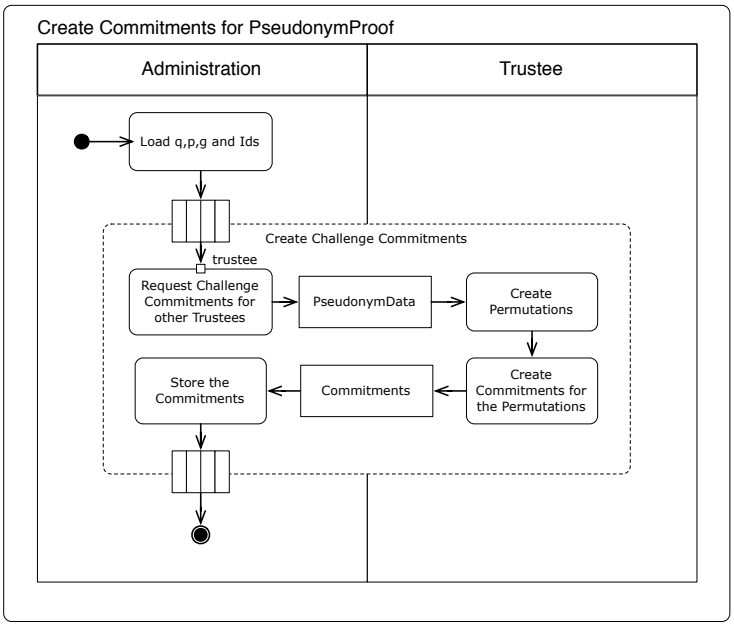

Abbildung 3.5: Pseudonyme Commitments erstellen

Erstelle Pseudonyme

In dieser Phase werden die Pseudonyme erstellt. Dabei wird die Pseudonymisierung in 2 Schritten erstellt, damit der Trustee anschliessend mit RPC beweisen kann, dass er beim Mischen nicht betrogen hat. Die folgende Schritte werden iterativ für alle $\mathcal{T}$ angewendet, wobei $S_1 = S$ und $g_1 = g$.

- Die Administration sendet S_i und g_i an Trustee T_i.
- Der Trustee wählt daraufhin 2 zufällige Zahlen (α_1, α_2) und 2 zufällige Permutationen (π_1, π_2).
- Danach erstellt er mit α_1 und der Permutation π_1 aus der Liste S_i zuerst das Zwischenresultat M_i .

$$M_{i,k} = S_{i,\pi_1(k)}^{\alpha_1} \text{ und } m_i = g_i^{\alpha_1}$$

- Aus dieser Liste M_i erstellt er mit α_2 und der Permutation π_2 das Resultat S_{i+1}

$$S_{i+1,k} = M_{i,\pi_2(k)}^{\alpha_2} \text{ und } g_{i+1} = m_i^{\alpha_2}$$

- Der Trustee publiziert M_i, m, S_{i+1} und g_{i+1} auf der Administration.
- Die Administration fordert nun alle anderen Trustees auf ihre Permutationen $\phi_{j,i}$ für T_i und das passende $r_{j,i}$ zu publizieren.

- Die Administration kontrolliert, dass die publizierten Permutationen zu den *commitments* passen. Falls eine Permutation nicht zu ihrem *commitment* passt, wird diese nicht für die *challenge* verwendet.

- Die gültigen Permutationen kombiniert die Administration zur finalen *challenge*.

$$\phi_i = \phi_{1,i} \circ \ldots \circ \phi_{n,i}$$

- Nun schickt die Administration dem Trustee T_i die *challenge* ϕ_i.

- Der Trustee verwendet nun die *challenge* um den Beweis zu erstellen, dass er korrekt gerechnet hat.
  - Für jede Stelle k von ϕ_i welche eine 0 enthält beweist er die Verbindung von $M_{i,k}$ und $S^{\alpha_1}_{i,\pi_1(k)}$ auf.

$$ZKP[(\alpha_1) : M_{i,k} = S^{\alpha_1}_{i,\pi_1^{-1}(k)} \wedge m = g_i^{\alpha_1}]$$

  - Für jede Stelle k von ϕ_i welche eine 1 enthält beweist er die Verbindung von $S_{i+1,k}$ und $M^{\alpha_2}_{i,\pi_2(k)}$ auf.

$$ZKP[(\alpha_2) : S_{i+1,k} = M^{\alpha_2}_{i,\pi_2(k)} \wedge g_{i+1} = m_i^{\alpha_2}]$$

- Nun sendet T_i seinen Beweis an die Administration.

- Die Administration überprüft den Beweis.
  - Wenn der Beweis korrekt ist und $i = n$, publiziert die Administration $\hat{S} = S_{i+1}$ als finale Pseudonyme mit Generator $g_p = g_{i+1}$.
  - Wenn der Beweis korrekt ist und $i < n$, fordert sie den nächsten Trustee T_{i+1} auf, eine Pseudonymisierung zu erstellen mit S_{i+1} und g_{i+1}.
  - Wenn der Beweis nicht korrekt ist und $i = n$, publiziert die Administration $\hat{S} = S_{i-1}$ als finale Pseudonyme mit Generator $g_p = g_{i-1}$.
  - Wenn der Beweis nicht korrekt ist und $i < n$, fordert sie den nächsten Trustee T_{i+1} auf, eine Pseudonymisierung zu erstellen mit S_i und g_i.

Wie in Abbildung 3.6 zu sehen, ist der Prozess relativ kompliziert. Wichtig ist, dass die Trustees ihre Permutationen für die *challenge* nicht publizieren, bevor der betroffene Trustee seine Pseudonymisierung erstellt hat.

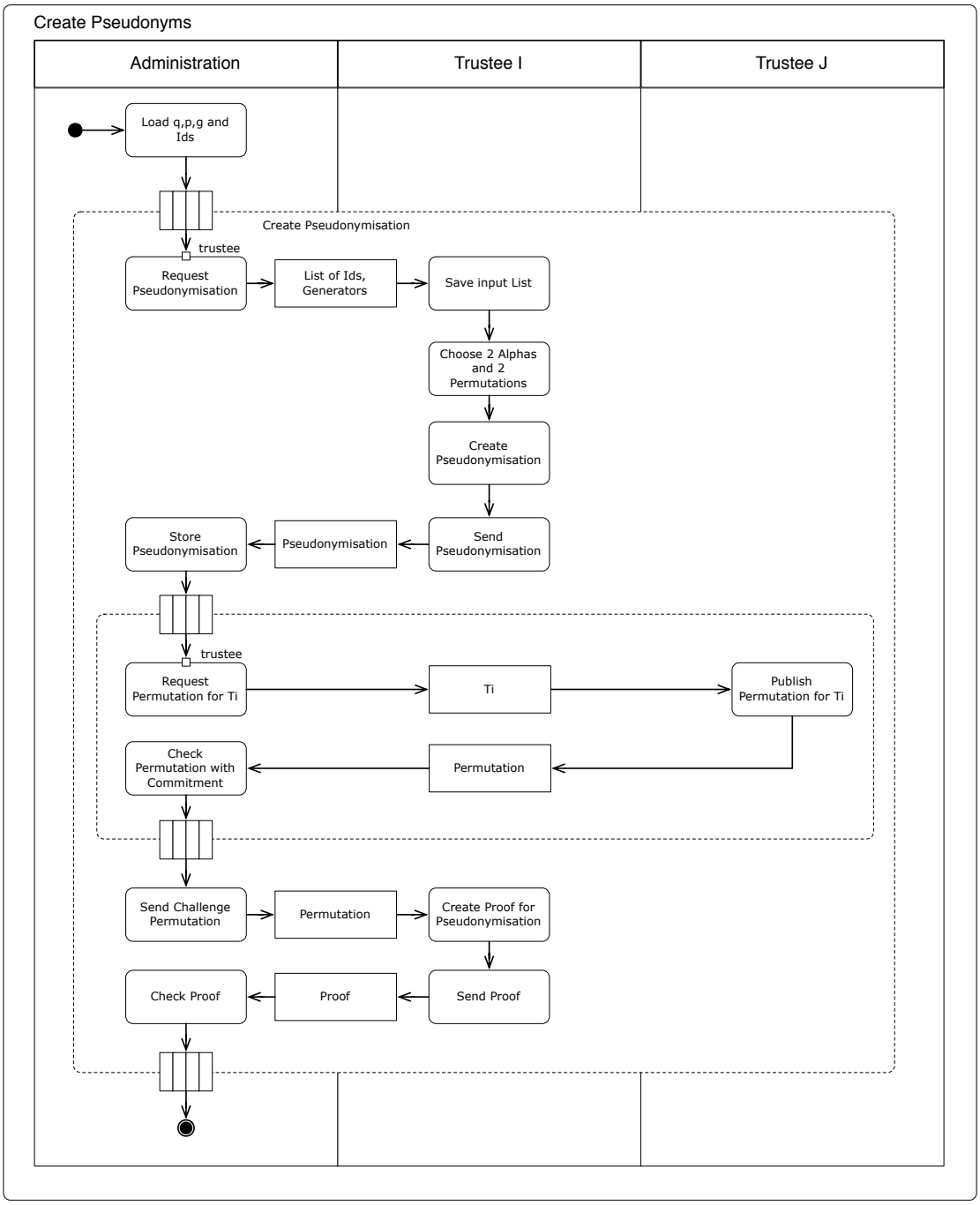

Abbildung 3.6: Pseudonyme erstellen

4 Selectio Helvetica Light

Selectio Helvetica Light basiert auf dem Protokoll von Spycher und Haenni [SH10] und wurde auf die Bedürfnisse von Baloti angepasst. Dabei wurde die Implementation vereinfacht, indem verteilte kryptographische Funktionen durch organisatorische Massnahmen ersetzt wurden. Grund dafür war, dass Selectio Helvetica Light bereits im September 2010 lauffähig sein musste. Dadurch war der Zeitraum für die Implementation sehr beschränkt. Gleichzeitig sollte das Protokoll aber immer noch sicher sein.
Ausserdem musste das Protokoll angepasst werden, so dass man auch spontan an einer Abstimmung teilnehmen kann, ohne in der initialen *voter roll* zu sein. Dies wird erreicht, indem ein genügend grosses Set an Voter-Einträgen vorgeneriert wird. Wenn sich nun jemand während der Abstimmung registriert, wird ihm einer dieser vorgenerierten Einträge vergeben.
Ein weiteres Problem ist die Länge des privaten Schlüssels des Voters. Es ist nicht realistisch, dass sich ein Voter eine derart lange Zahl merken kann. Darum werden sogenannte *votingcodes* verwendet. Jeder *votingcode* ist zehn Zeichen lang und stellt einen Link auf einen privaten Schlüssel dar. So muss sich der Voter nur seinen *votingcode* merken anstelle des privaten Schlüssels.

4.1 Phasen

4.1.1 Erstellung der VoterRoll

- Die Administration wählt die Basiswerte für die kryptographischen Funktionen (q, p, g) aus.
- Anschliessend generiert sie für die maximale Anzahl Voter ein ElGamal Schlüsselpaar und speichert die Schlüssel getrennt.
- Nun generiert die Administration zu jedem privaten Schlüssel einen *votingcode*.

4.1.2 Erstellung der Abstimmung

- Die Administration erstellt ein ElGamal-Schlüsselpaar für die Verschlüsselung der Stimmen. Dabei wird ein neuer Generator g_e verwendet.
- Anschliessend erstellt sie einen zufälligen Werts α und eine Permutation π. Mit Hilfe dieser Werte erstellt sie für jeden Voter in der *voter roll* ein Pseudonym und mischt diese.

- Nun berechnet sie mit Hilfe des zuvor gewählten α und dem Generator g einen neuen Generator $g_p = g^{\alpha}$ für diese Abstimmung. Dieser wird später von den Votern verwendet um ihr Pseudonym zu berechnen.

4.1.3 Abstimmen

Zuerst wird hier der gesamte Ablauf für den Voter aufgezeigt. Dieser Teil ist sehr allgemein gehalten. Anschliessend wird im Detail aufgezeigt, was bei der Stimmabgabe passiert.

Ablauf für den Voter

In Abbildung 4.1 wird der gesamte Ablauf skizziert, falls der Voter abstimmen möchte. Falls er sich bereits bereits früher registriert hat und seinen *votingcode* noch kennt, kann er direkt bei Schritt 5 einsteigen.

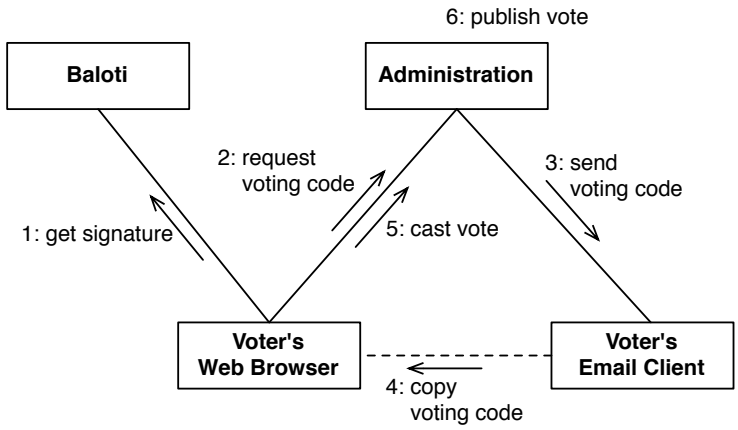

Abbildung 4.1: SH Light Ablauf

1. Zuerst verlangt der Voter von Baloti eine Signatur für seine Email Adresse. Baloti prüft die Email Adresse und wenn alles in Ordnung ist, schickt Baloti eine gültige Signatur zurück.
2. Nun sendet der Voter dieses Signatur an die Administration.
3. Die Administration kontrolliert die Signatur und schickt dann eine Email an die angegebene Adresse mit einem freien *votingcode*.
4. Der Voter kopiert den *votingcode* aus der Email in die Abstimmungs-Webseite.
5. Nun kann der Voter seinen Wahlzettel ausfüllen und verschlüsselt abschicken.
6. Die Administration publiziert den erhaltenen verschlüsselten Wahlzettel.

Detailierte Abgabe der Stimme

Hier wird gezeigt was im Detail abläuft, wenn der Voter seine Stimme abgeben will (Abb. 4.2).

- Zuerst füllt der Voter seinen Stimmzettel aus und gibt seinen *votingcode* an.
- Nun fordert er von der Administration seinen privaten Schlüssel und den öffentlichen Schlüssel der Abstimmung an.
- Der Voter muss nun seine Stimme für die Abstimmung kodieren.
- Anschliessend kann er seiner Stimme mit dem öffentlichen Schlüssel der Abstimmung verschlüsseln.
- Nun muss er die verschlüsselte Stimme mit seinem privaten Schlüssel signieren.
- Ebenfalls mit seinem privaten Schlüssel berechnet er sein Pseudonym für diese Abstimmung.
- Nun schickt er alle Daten an die Administration.
- Die Administration kontrolliert das Pseudonym und prüft ob die Signatur zu diesem Pseudonym passt.
- Anschliessend wird die verschlüsselte Stimme und das Pseudonym publiziert.

4.1.4 Auszählen

Am Ende der Abstimmung publiziert die Administration den privaten Schlüssel der Abstimmung. Somit kann jeder die Stimmen entschlüsseln und auszählen. Weil zu den entschlüsselten Werten nur die Pseudonyme assoziiert sind, kennt man die wahre Identität des Voters nicht.

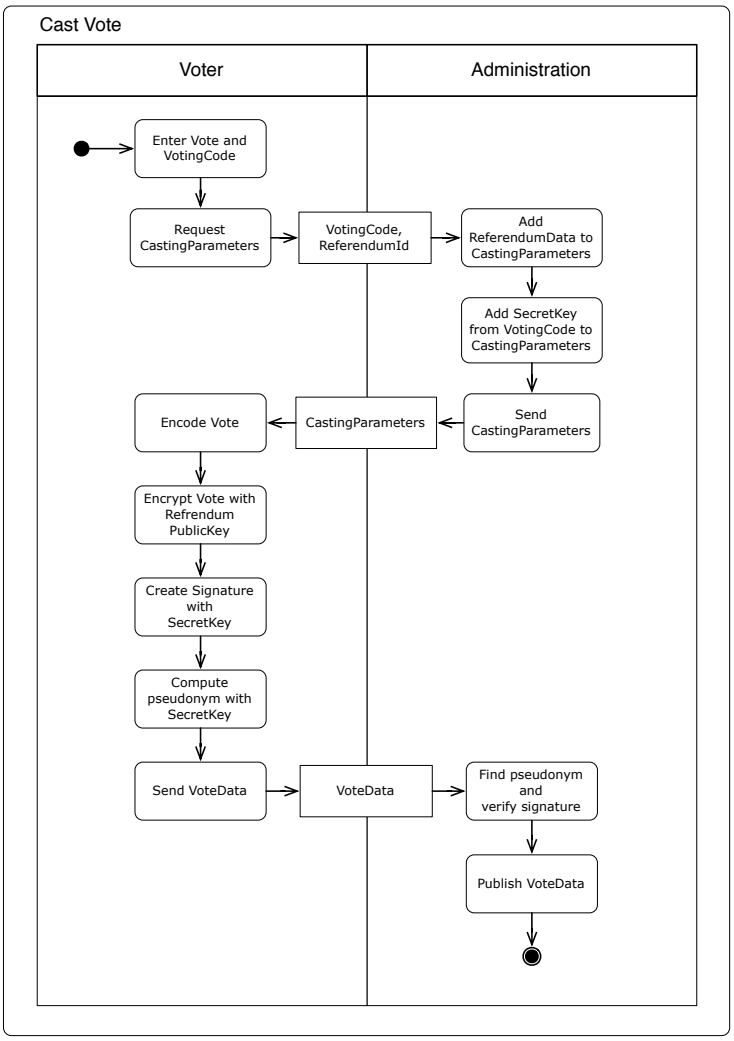

Abbildung 4.2: SH Light Stimme abgeben

5 Selectio Helvetica

Selectio Helvetica ist eine Konkretisierung des SH10 Protokolls [SH10]. Im Gegensatz zu SH10 kennt Selectio Helvetica keinen papierbasierten Kanal. Es gibt also keine Abstimmung auf Papier. Die Eigenschaft der *coercion resistance* ist darum nicht gegeben. Ausserdem unterstützt Selectio Helvetica genau wie Selectio Helvetica Light die spontane Teilnahme eines Voter während der Abstimmung.
Im Gegensatz zu Selectio Helvetica Light werden hier jedoch alle Eigenschaften des Protokolls über kryptographische Funktionen erreicht und nicht über organisatorische Massnahmen. Die Ermöglichung der spontanen Teilnahme der Voter verfolgt das gleiche Konzept wie bei Selectio Helvetica Light. Es werden Voter-Einträge vorgeneriert und bei Bedarf an hinzukommende Benutzer vergeben. Allerdings kann bei Selectio Helvetica der Benutzer selber einen *votingcode* wählen und bekommt nicht einen festen per Mail zugestellt.

5.1 Erstellung der VoterRoll

- Die Administration wählt die Basiswerte für die kryptographischen Funktionen (q, p).
- Anschliessend wählen die Trustees gemeinsam einen Generator g, welcher zu q und p passt.
- Nun wählt die Administration für jeden Voter, der generiert werden soll, eine VoterId. Diese Id dient nur zur eindeutigen Identifizierung des Voters solange die *voter roll* noch nicht vollständig aufgebaut ist.
- Nun handeln die Administration und die Trustees mit Hilfe des Protokolls "Verteilter privater Schlüssel" für jeden Voter ein Schlüsselpaar aus.
- Schlussendlich kann die Administration für alle Voter die Id mit dem öffentlichen Schlüssel h_v ersetzen und die *voter roll* publizieren.

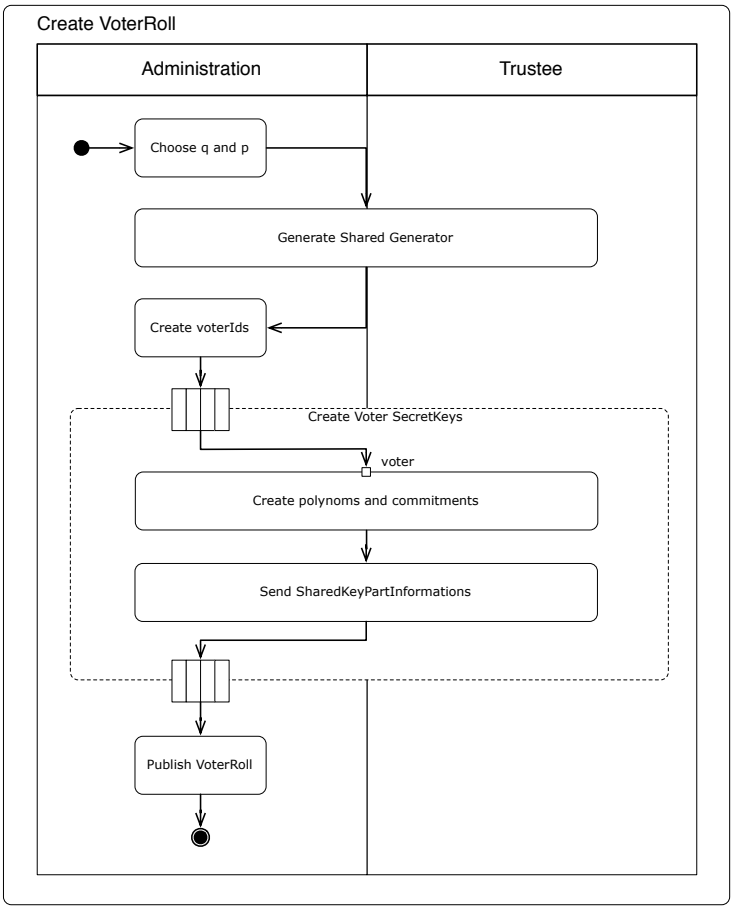

Abbildung 5.1: SH Erstellung der voter roll

5.2 Erstellung der Abstimmung

In dieser Phase müssen zwei unabhängige Schritte ausgeführt werden (Abb. 5.2). Erstens muss ein Schlüsselpaar erstellt werden für die Verschlüsselung der Stimmen. Zweitens müssen die Pseudonyme erstellt werden für alle Voter.

- Zuerst wählt die Administration die Grunddaten für die Abstimmung, wie Start- und Enddatum, Abstimmungsidentifikator und welche Trustees mitmachen sollen an dieser Abstimmung.

- Nun fordert die Administration die Trustees auf, beim Erstellen des Schlüsselpaars und der Pseudonyme mitzumachen.

- Erstellen des Schlüsselpaars:

  - Zuerst muss ein neuer Generator g_e erstellt werden, damit das Abstimmungs-Schlüsselpaar keine Verbindung zur *voter roll* aufweist.
  - Anschliessend kann nach "Verteilter privater Schlüssel" ein Schlüsselpaar für die Abstimmung erstellt werden. Dabei resultiert der öffentliche Schlüssel h_A für die Abstimmung.

- Erstellen der Pseudonyme:
  - Zuerst müssen die notwendigen Generatoren erstellt werden $(g, g_1, \ldots, g_k, f)$.
  - Nun erstellen die Trustees iterativ ihre Pseudonymisationen und beweisen, dass sie nicht betrogen haben.
  - Am Ende ergibt sich die Permutation $\hat{S}$ und der dazugehörige Generator g_p.

- Am Schluss publiziert die Administration die Pseudonyme $\hat{S}$, den Generator g_p für die Pseudonyme, den Generator g_e für die Verschlüsselung der Stimmen, den öffentlichen Schlüssel h_A für diese Abstimmung, die beteiligten Trustees $\mathcal{T}$ und die Abstimmungsfragen.

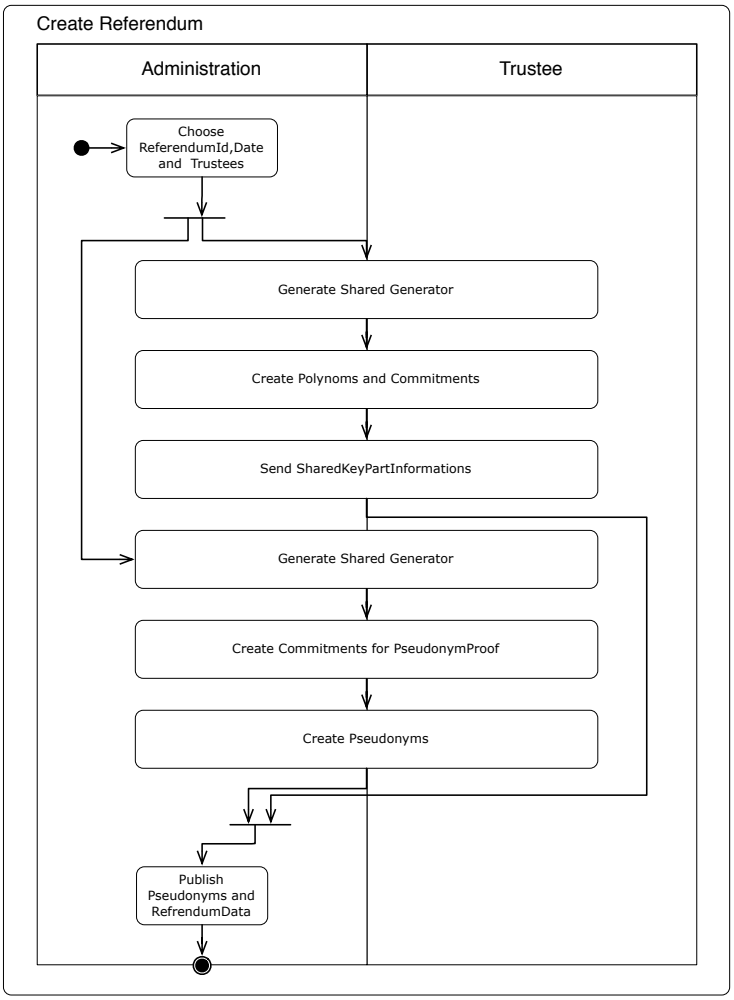

Abbildung 5.2: SH Erstellung der Abstimmung

5.3 Abstimmen

Hier wird der komplette Ablauf für einen neuen Voter aufgezeigt. Bereits registrierte Voter können natürlich den Registrierungsprozess in Abb. 5.3 überspringen.

5.3.1 Registriere Voter

In Abbildung 5.3 ist der Prozess, welcher ein Voter durchlaufen muss, um sich zu registrieren, aufgeführt. Wichtig ist, dass die Autorisierung von Baloti kommt. Dies ist Sache zwischen Baloti und dem Voter und in diesem Prozess nicht im Detail erfasst.

- Wie bei Selectio Helvetica Light muss sich der Voter, um sich registrieren zu können, von Baloti seinen Email Adresse signieren lassen.

- Nun kann er die Signatur und seine Email Adresse an die Administration schicken.
- Diese kontrolliert die Signatur. Wenn diese gültig ist, schickt die Administration dem Voter ein Authentication Token zurück. Dieses Authentication Token besteht aus einer freien VoterId, der Email Adresse des Voters und einer Signatur der Administration.
- Der Voter kann jetzt einen *votingcode* wählen. Diesen *votingcode* verwendet er später um seinen privaten Schlüssel von den Trustees zu erhalten.
- Nun kodiert der Voter seinen *votingcode* für jeden Trustee. Sinn dahinter ist, dass sein *votingcode* für jeden Trustee unterschiedlich ist. Damit wird verhindert, dass ein Trustee sich für den Voter ausgeben kann.
- Anschliessend schickt er den kodierten *votingcode* und das Authentication Token an den Trustee.
- Der Trustee kontrolliert das Authentication Token.
- Anschliessend speichert er den *votingcode* bei der im Authentication Token enthaltenen VoterId.

5.3.2 Stimme abgeben

Dieser Prozess ähnelt sehr dem Prozess von Selectio Helvetica Light. Wie in Abb. 5.4 zu sehen ist, besteht der einzige Unterschied darin, dass der Voter seinen privaten Schlüssel von den Trustees erhält und nicht von der Administration.

- Der Voter gibt seine Stimme und den *votingcode* ein.
- Anschliessend fordert er die Informationen über die Abstimmung von der Administration an.
- Die Administration sucht die Informationen über die Abstimmung und sendet sie zurück an den Voter.
- Dieser berechnet für jeden Trustee den encodeten *votingcode* um sich zu authentifizieren.
- Nun fordert der Voter von den Trustees über "Verteilter privater Schlüssel" seinen privaten Schlüssel s_v an.
- Der Voter kodiert und verschlüsselt sein Stimme mit dem öffentlichen Schlüssel h_A der Abstimmung.

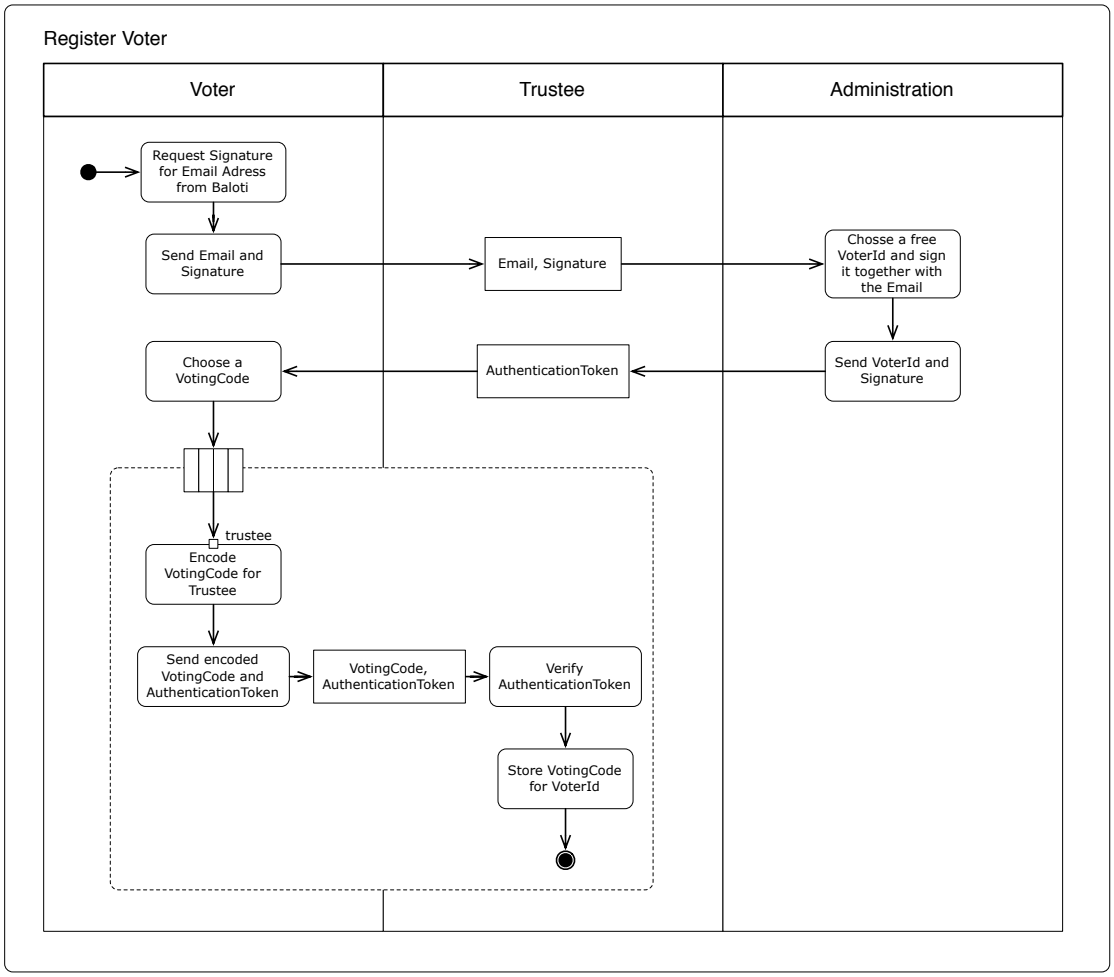

Abbildung 5.3: SH Voter Registration

- Anschliessend berechnet er eine Signatur und das Pseudonym mit seinem privaten Schlüssel.

$$\text{pseudonym} = g_p^{s_v}$$

- Jetzt kann er seine Stimme und die anderen nötigen Daten an die Administration schicken.
- Die Administration kontrolliert, ob die Signatur gültig ist und zum Pseudonym passt.
- Schlussendlich publiziert die Administration die Stimme beim passenden Pseudonym.

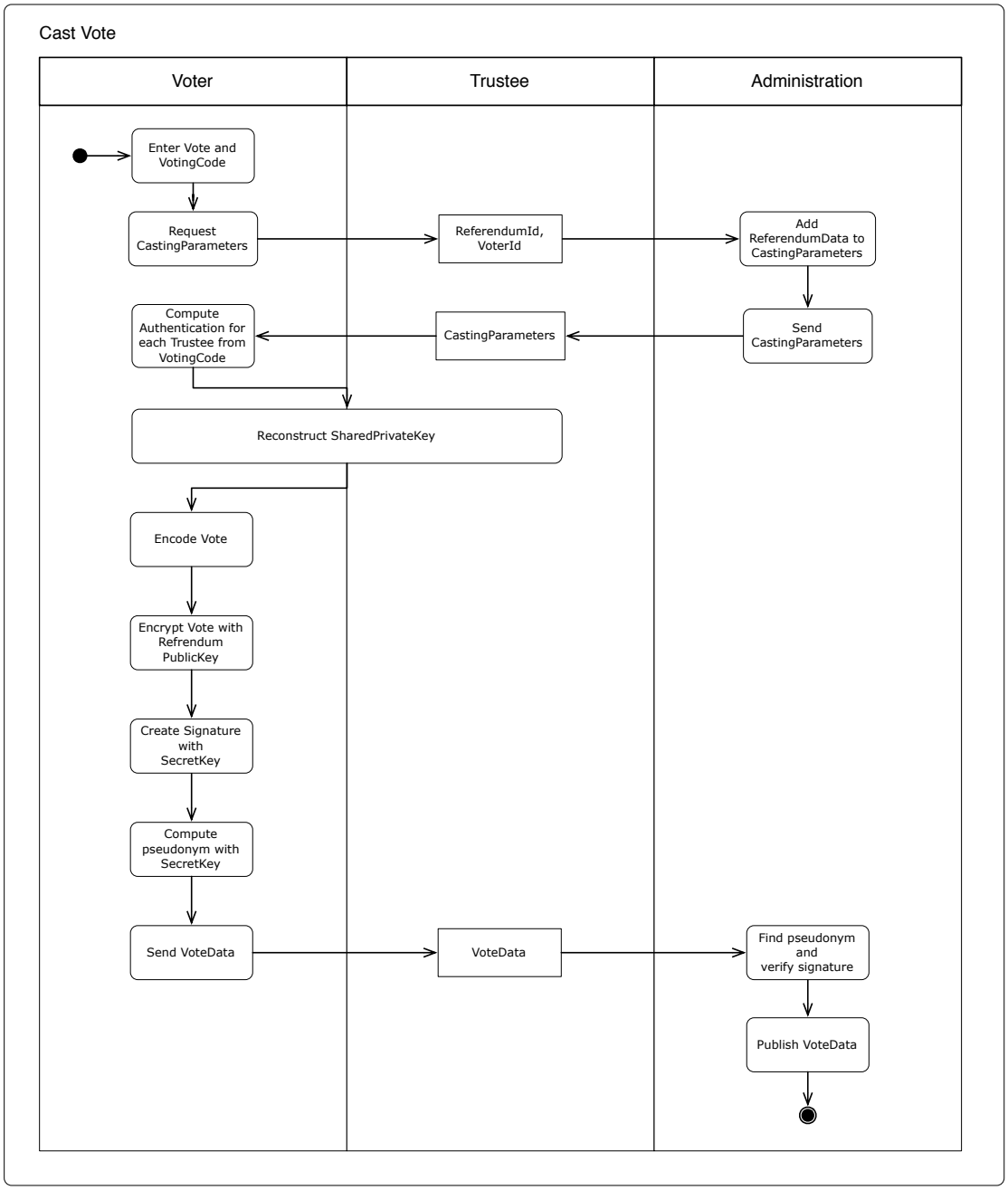

Abbildung 5.4: SH Abstimmen

5.4 Auszählen

Die Abbildung 5.5 zeigt den Prozess um die Stimmen auszuzählen.

- Zuerst berechnet die Administration mit den Trustees den privaten Schlüssel s_A für die Abstimmung.
- Nun kann die Administration alle erhaltenen Stimme entschlüsseln und auszählen.
- Am Ende publiziert sie das Resultat, die entschlüsselten Stimmen und den privaten Schlüssel der Abstimmung. Mit dem privaten Schlüssel kann jeder die Stimmen selbst entschlüsseln und nachzählen.

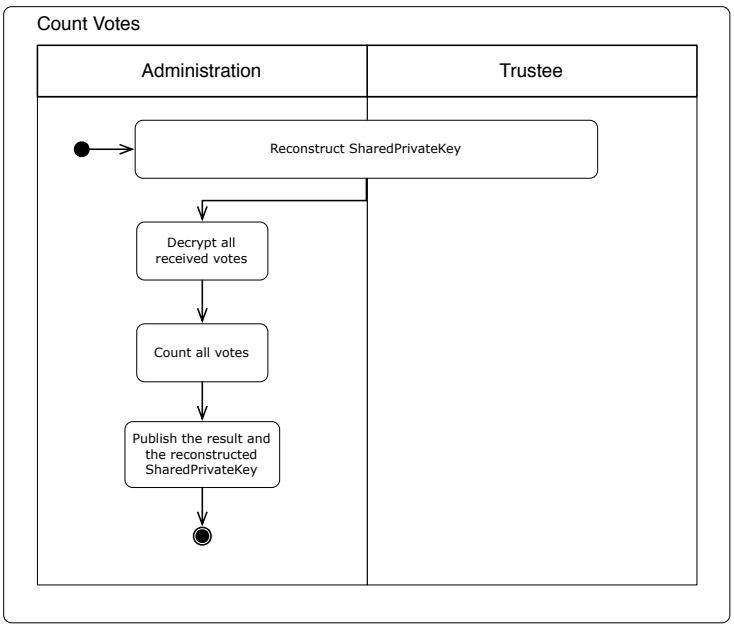

Abbildung 5.5: SH Auszählen

6 Architektur

6.1 Verteilansicht

Wie in der Abbildung 6.1 zu sehen ist, werden für die Implementation von Selectio Helvetica entsprechend der drei Rollen drei Artefakte benötigt. Die Artefakte Administration und Trustee werden beide auf einen JavaEE Server bereitgestellt, wobei vom Trustee mehrere getrennte Instanzen benötigt werden. Deshalb wird das Artefakt auch auf mehreren Server bereitgestellt.
Der Voter wird mittels just-in-time Bereitstellung von der Administration an jeden beliebigen Browser ausgeliefert.

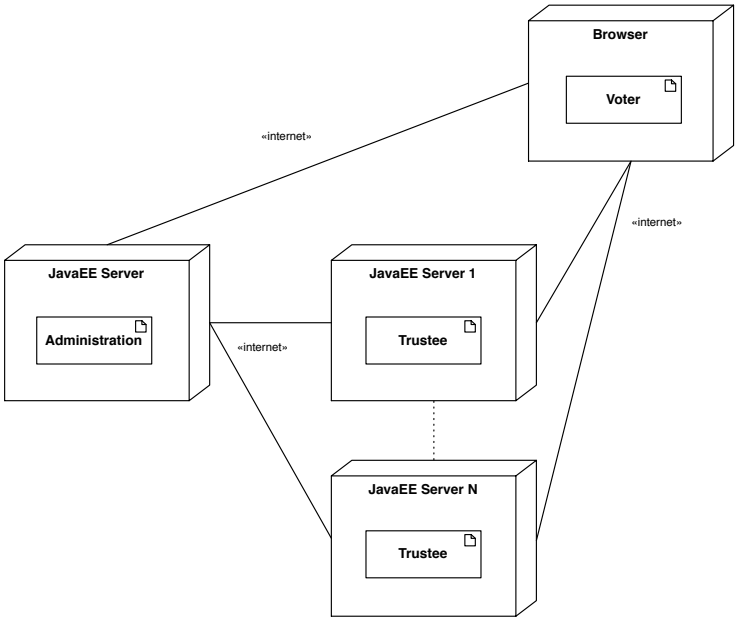

Abbildung 6.1: SH Verteilansicht

6.2 Bausteine und Schnittstellen

Wie in der Verteilansicht beschrieben, besteht Architektur von Selectio Helvetica aus drei Artefakten. Wenn man diese nun als Bausteine modelliert, ergibt sich das Diagramm in Abbildung 6.2, dass auch die notwendigen Schnittstellen zwischen den Bausteinen

aufzeigt. Die Schnittstellen wurden nach Phasen und Konsument voneinander getrennt. Die Kommunikation zwischen Administration und Trustee findet über Webservices statt. Auch die Trustees untereinander kommunizieren mit Webservices. Die Kommunikation mit dem Voter erfolgt immer über den Austausch von JSON-Objekten. Die Transferobjekte werden hier nicht im Detail beschrieben. Detaillierte Informationen zu den Transferobjekten findet man in den WSDL-Dateien der Webservices oder den JSON-Objekten.

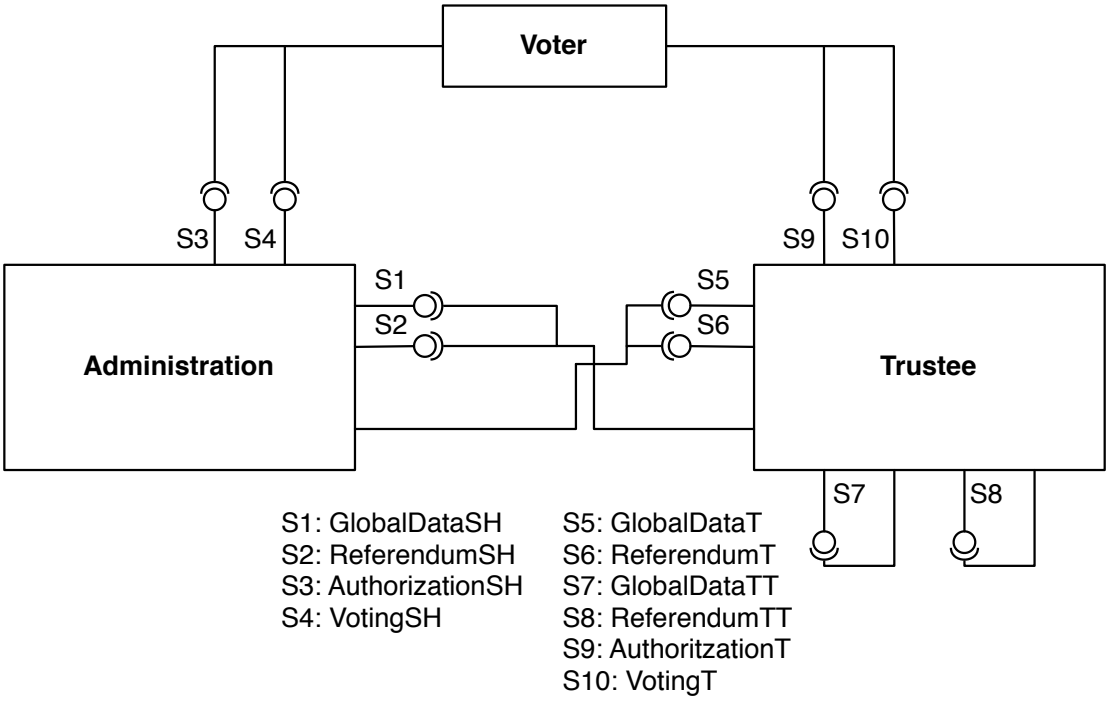

Abbildung 6.2: SH Bausteine und Schnittstellen

6.2.1 Kommunikationssicherheit

Bei den verwendeten Kryptoprotokollen ist wichtig, vor allem auch bei einem Betrugsfall, dass man beweisen kann, dass man alles korrekt berechnet hat. Dazu gehört auch, dass man beweisen kann von wem man welche Daten erhalten hat.
Aus diesem Grund werden alle Daten, welche versendet werden, vom Sender signiert. Der Empfänger muss diese Signaturen überprüfen und speichern. Somit kann sicher gestellt werden, dass die Beweiskette immer komplett ist.
Darum befinden sich in den Schnittstellen zu diesen Protokollen immer Einträge für eine Signatur, welche nicht aus den Protokoll-Prozessen ersichtlich ist. Signiert wird immer ein einzelner String, welcher aus allen Daten zusammengesetzt wird. Dabei ist die Reihenfolge immer gleich der Ordnung in den WSDL Dateien und bei Listen immer

mit aufsteigender Sortierung. BigInteger werden als Base36 kodierten String eingefügt. In den Schnittstellenbeschreibungen wird nicht weiter auf diese Signatur eingegangen.

6.2.2 Fehlerbehandlung

Jede der Schnittstellen kann im Fehlerfall eine Fehlermeldung werfen. Diese sind hier nicht weiter dokumentiert, aber in den WSDL und JSON Dateien ersichtlich. Diese Fehlermeldung besteht jeweils aus einem Fehlercode und einer passenden Fehlermeldung.

6.2.3 Baustein Administration

Die Administration sorgt für die Koordinierung der Abstimmungen. Dabei stellt sie auch allen anderen Bausteinen die öffentlichen Daten bereit. Wie in der Abbildung 6.2 zu sehen ist, besitzt die Administration vier Schnittstellen, gekennzeichnet durch die Endung SH. S1 und S2 dienen der Kommunikation mit den Trustees. S3 und S4 sind für die Kommunikation mit den Votern.

6.2.4 Baustein Trustee

Der Trustee sorgt für die Berechnung der kryptographischen Funktionen. Da mehrere Instanzen von diesem Baustein benötigt werden, muss der Trustee auch mit sich selber kommunizieren können. Darum auch die Schnittstellen S7 und S8.

6.2.5 Baustein Voter

Da der Voter per just-in-time Bereitstellung verteilt wird, kann nicht garantiert werden, dass allfällige Schnittstellen erreichbar wären. Aus diesem Grund besitzt der Voter keine eigenen Schnittstellen, sondern konsumiert nur Schnittstellen von Administration und Trustee.

6.2.6 Schnittstelle GlobalDataSH

Diese Schnittstelle ermöglicht den Trustees bei der Erstellung der *voter roll* ihre Arbeit auf der Administration zu publizieren und die Daten der anderen Trustees abzurufen.

addCommitments

Diese Operation erlaubt den Trustees *commitments* zu allen Voter-Polynomen zu publizieren.
Eingehend:

- referendumId - Identifikator der Abstimmung.
- trusteeId - Identifikator des Trustees, welcher die *commitments* schickt.
- List(Commitment) - Liste der *commitments* und VoterIds.
- Signature

getCommitments

Diese Operation erlaubt den Trustees die *commitments* von anderen Trustees abzufragen.
Eingehend:

- referendumId - Identifikator der Abstimmung.
- trusteeId - Identifikator des Trustees, dessen *commitments* man erhalten möchte

Ausgehend:

- List(Commitment) - Liste der *commitments* des angegebenen Trustees.
- Signature

SKPIreceived

Diese Operation ermöglicht den Trustees, der Administration mitzuteilen, wenn sie ein Set von $s_{i,j}$ erhalten haben.
Eingehend:

- SKPIreceivedNotification - Notifikation, dass der Trustee ein Set von $s_{i,j}$ erhalten hat.
- Signature

publishSKPI

Diese Operation erlaubt dem Trustee die $s_{i,j}$, welche er für einen anderen Trustee berechnet hat, zu publizieren.
Eingehend:

- SKPI - Das Set von $s_{i,j}$, welches veröffentlicht werden soll.
- Signature

6.2.7 Schnittstelle ReferendumSH

Diese Schnittstelle wird von den Trustees benutzt um auf der Administration Daten zu einer bestimmten Abstimmung abzulegen oder nachzufragen. Siehe auch Kapitel 5.2.

addReferendumCommitments

Diese Operation erlaubt den Trustees *commitments* für den Abstimmungsschlüssel zu publizieren.
Eingehend:

- referendumId - Identifikator der Abstimmung.
- trusteeId - Identifikator des Trustees, welcher die *commitments* schickt.
- List(Commitment) - Liste der *commitments*.
- Signature

getReferendumCommitments

Diese Operation ermöglicht den Trustees die *commitments* für den Abstimmungsschlüssel von den anderen Trustees abzufragen.
Eingehend:

- referendumId - Identifikator der Abstimmung.
- trusteeId - Identifikator des Trustees, dessen *commitments* man erhalten möchte

Ausgehend:

- List(Commitment) - Liste der *commitments* des angegebenen Trustees.
- Signature

addGeneratorCommitments

Mit dieser Operation können die Trustees sich zu ihren Teilen der verteilten Generatoren, welche für diese Abstimmung verwendet werden, *commiten*.
Eingehend:

- referendumId - Identifikator der Abstimmung.
- generatorCommitments - Für jeden benötigten Generator wird ein *commitment* gesendet.
- Signature

SKPIreceived

Diese Operation erlaubt den Trustees, der Administration mitzuteilen, wenn sie ein $s_{i,j}$ von einem anderen Trustee erhalten haben.
Eingehend:

- SKPIreceivedNotification - Notifikation, dass der Trustee ein $s_{i,j}$ erhalten hat.
- Signature

publishSKPI

Diese Operation erlaubt den Trustees, ein $s_{i,j}$, welches sie für einen anderen Trustee berechnet haben, zu publizieren.
Eingehend:

- SKPI - Das $s_{i,j}$, welches veröffentlicht werden soll.
- Signature

addPeusdonymList

Die Operation erlaubt den Trustees, eine Pseudonymisation, welche sie für eine bestimmte Abstimmung berechnet haben, auf der Administration zu publizieren.
Eingehend:

- Pseudonymisation - Die Pseudonymisation, welche der Trustee erstellt hat.
- Signature

submitProof

Diese Operation erlaubt den Trustees, ihre Beweise, dass sie nicht betrogen haben bei der Pseudonymisation, der Administration mitzuteilen.
Eingehend:

- Proof - Der Beweis, dass der Trustee bei der Pseudonym-Erstellung nicht betrogen hat.
- Signature

6.2.8 Schnittstelle AuthorizationSH

Diese Schnittstelle erlaubt es dem Voter ein AuthenticationToken zu erstellen um sich zu registrieren.

getAuthenticationToken

Diese Operation ermöglicht jeder Person mit einer gültigen Signatur von Baloti ein Authentication Token zu erhalten.
Eingehend:

- mailAdress - EMail Adresse mit welcher sich der Voter registrieren möchte.
- Signature - Signatur der EMail Adresse ausgestellt von Baloti.

Ausgehend:

- voterId - Identifikator, welcher diesem Voter zugewiesen wurde.
- mailAddress - EMail Adresse des Voters
- Signature - Signature der Administration von voterId und mailAddress.
- trustees - Alle Trustees, welche an der *voter roll* beteiligt sind.

6.2.9 Schnittstelle VotingSH

Über diese Schnittstelle stimmt der Voter ab.

getCastingParams

Diese Operation erlaubt es allen Votern von der Administration die Informationen zu einer Abstimmung abzufragen.
Eingehend:

- referendumId - Identifikator der Abstimmung.

Ausgehend:

- castingParameters - Alle kryptographischen Daten, welche vom Voter benötigt werden um seine Stimme abzugeben.

- trustees - Alle Trustees, welche an der *voter roll* beteiligt sind.

castVote

Diese Operation erlaubt allen Votern ihre Stimme für eine bestimmte Abstimmung abzugeben.
Eingehend:

- referendumId - Identifikator der Abstimmung.
- pseudonym - Das Pseudonym des Voters für diese Abstimmung.
- encryptedVote - Die verschlüsselte Stimme des Voters.
- Signature - Signatur vom Voter für die verschlüsselten Stimme.

6.2.10 Schnittstelle GlobalDataT

Diese Schnittstelle erlaubt der Administration den Trustee zu informieren, wenn eine neue *voter roll* erstellt werden soll und die nötigen Schritte auszuführen.

createGenerator

Diese Operation erlaubt es der Administration, den Trustee aufzufordern, sich an der Erstellung eines verteilt berechneten Generators zu beteiligen.
Eingehend:

- generatorId - Identifikator des Generators.
- Signature

Ausgehend:

- Commitment - *commitment* des Generators.
- Signature

getGenerator

Diese Operation ermöglicht es der Administration einen Generator, zu welchem sich der Trustee *commitet* hat, anzufordern.
Eingehend:

- generatorId - Identifikator des Generators.

Ausgehend:

- Generator - Generator für den Identifikator.
- Signature

newVoters

Diese Operation erlaubt es der Administration, den Trustee zur Beteiligung an der Erstellung einer *voter roll* aufzufordern.
Eingehend:

- VoterIds - Identifikatoren für die zu erstellenden Voters.
- generator - Generator, welcher verwendet werden soll.
- Signature

requestPublicationOfSKPIs

Diese Operation erlaubt der Administration, diesen Trustee zu bitten, ein Set von $s_{i,j}$ zu publizieren.
Eingehend:

- voterIds - Alle Voter für die das $s_{i,j}$ publiziert werden soll.
- trusteeId - Identifikator des Trustees T_j,für den die $s_{i,j}$ für die Voter bestimmt sind.
- Signature

6.2.11 Schnittstelle ReferendumT

Diese Schnittstelle dient dazu, dass die Administration den Trustee über neue Abstimmungen informieren und ihm die dafür nötigen Daten schicken kann.

newReferendum

Diese Operation erlaubt der Administration, dem Trustee mitzuteilen, wenn es eine neue Abstimmung gibt. Hiermit fordert die Administration den Trustee auch auf sich bei der Erstellung der Generatoren zu beteiligen.
Eingehend:

- Referendum - Alle nötigen Daten für die Abstimmung.
- Signature

getGenerators

Diese Operation erlaubt es der Administration die Generatoren, zu welchen sich der Trustee *commitet* hat, anzufordern.
Eingehend:

- referendumId - Identifikator des Abstimmung.

Ausgehend:

- Generators - Generatoren für die Abstimmung.
- Signature

createPseudonymisation

Über diese Operation kann die Administration den Trustee auffordern, eine Pseudonymisierung für diese Abstimmung zu erstellen.
Eingehend:

- referendumId - Identifikator der Abstimmung.
- identities - Identitäten, für welche Pseudonyme erstellt werden sollen.
- Signature

getPermutation

Diese Operation erlaubt der Administration den Trustee aufzufordern seinen Teil der *challenge* für einen bestimmten anderen Trustee zu publizieren.
Eingehend:

- referendumId - Identifikator der Abstimmung.
- trusteeId - Identität des Trustees für welche die *challenge* bestimmt ist.
- Signature

Ausgehend:

- permutation - Permutation für den geforderten Trustee.
- Signature

createProof

Mit dieser Operation fordert die Administration vom Trustee den Beweis zu seiner Pseudonymisierung zu berechnen.
Eingehend:

- referendumId - Identifikator der Abstimmung.
- permutation - Die Permutation, welche als die *challenge* verwendet werden soll.
- Signature

createReferendumKey

Diese Operation erlaubt der Administration, den Trustee aufzufordern, sich bei der Berechnung des El-Gamal Schlüsselpaars für die Abstimmung zu beteiligen.
Eingehend:

- referendumId - Identifikator der Abstimmung.
- encryptionGenerator - Generator g_e, welcher verwendet werden soll.
- Signature

requestPublicationOfSKPI

Diese Operation erlaubt der Administration, den Trustee aufzufordern ein $s_{i,j}$ zu publizieren.
Eingehend:

- referendumId - Identifikator der Abstimmung.
- trusteeId - Identifikator des Trustees T_j für den das $s_{i,j}$ bestimmt ist.
- Signature

getSecretKeyPart

Mit dieser Operation fordert die Administration den Trustee auf, seinen Teil der privaten Schlüssels auf der Administration zu publizieren.
Eingehend:

- referendumId - Identifikator der Abstimmung.
- Signature

Ausgehend:

- skp - Teilschlüssel des Trustees für diese Abstimmung.
- Signature

6.2.12 Schnittstelle GlobalDataTT

Diese Schnittstelle dient der Kommunikation der Trustees untereinander, damit sie die verteilten privaten Schlüssel für die Voter erstellen können.

addSKPIs

Diese Operation ermöglicht einem anderen Trustee diesem Trustee seine $s_{i,j}$ mitzuteilen.
Eingehend:

- trusteeId - Identifikator des Trustees.
- voterSKPIs - Eine Liste bestehend aus VoterIds und dazugehörigen $s_{i,j}$.
- Signature

6.2.13 Schnittstelle ReferendumTT

Diese Schnittstelle dient der Kommunikation der Trustees untereinander, damit sie einen verteilten privaten Schlüssel für die Abstimmung erstellen können.

addSKPI

Diese Operation erlaubt einem anderen Trustee diesem Trustee sein $s_{i,j}$ mitzuteilen.
Eingehend:

- referendumId - Identifikator des Abstimmung.
- trusteeId - Identifikator des Trustees.
- SKPI - $s_{i,j}$
- Signature

6.2.14 Schnittstelle AuthorizationT

Diese Schnittstelle wird verwendet um neue Voter zu registrieren.

setVotingCode

Über diese Operation setzt der Voter seinen VotingCode für diesen Trustee.
Eingehend:

- authToken - Authentication Token.
- votingCode- VotingCode, welcher der Voter für diesen Trustee verwenden will.

Ausgehend:

- accepted - Ein Boolean, welcher anzeigt, ob der Trustee das Authentication Token akzeptiert hat.

6.2.15 Schnittstelle VotingT

Diese Schnittstelle wird von den Votern während der Abstimmung verwendet um an ihren privaten Schlüssel zu kommen.

getSecretKey

Diese Operation erlaubt es einem Voter mit Hilfe seines VotingCodes den Teil seines privaten Schlüssel zu erhalten, welcher von diesem Trustee gespeichert wird.
Eingehend:

- votingCode - VotingCode, welcher der Voter für diesen Trustee definiert hat.

Ausgehend:

- sKP - Teil des privaten Schlüssel für den Voter, der vom Trustee gespeichert wird.

6.3 Laufzeitansicht

Um die Übersicht über die Laufzeitansicht zu verbessern, sind die Sequenzdiagramme in die Phasen des Protokolls unterteilt.

6.3.1 Erstellung der VoterRoll

Generiere verteilten Generator

Wie in Abbildung 6.3 zu sehen ist, informiert die Administration den Trustee, dass sie einen neuen Generator benötigt. Dieser antwortet mit einem *commitment*. Wenn die Administration alle *commitments* hat, fordert sie vom Trustee den Generator an.

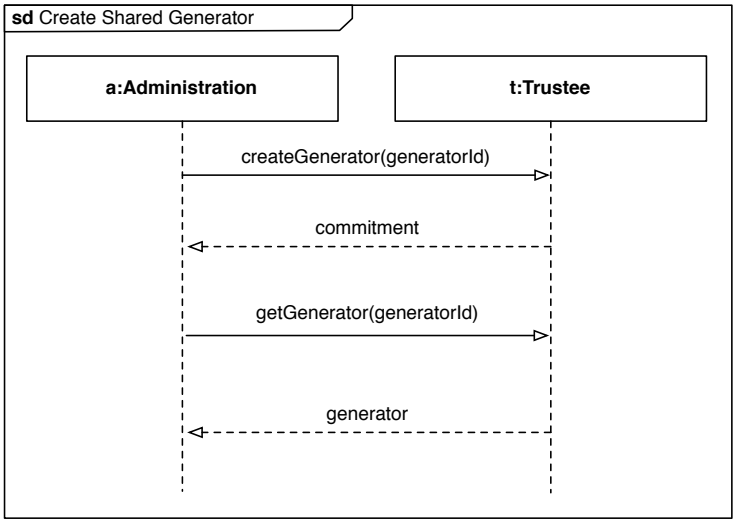

Abbildung 6.3: SH Sequenz Generiere einen verteilten Generator

Erstellung der VoterRoll

Bei der Erstellung der *voter roll* (Abb. 6.4) muss zuerst ein verteilter Generator erstellt werden. Anschliessend teilt die Administration den Trustees mit für welche VoterIds ein verteilter privater Schlüssel erstellt werden soll. Zuerst erstellen alle Trustee für jeden Voter ein Polynom und senden die passenden *commitments* an die Administration. Die Trustees berechnen nun untereinander die nötigen verteilten privaten Schlüsseln und informieren die Administration am Ende, wie die Berechnung abgelaufen ist. Falls ein Trustee versucht zu betrügen, fordert die Administration ihn auf seine $s_{i,j}$ aufzudecken.

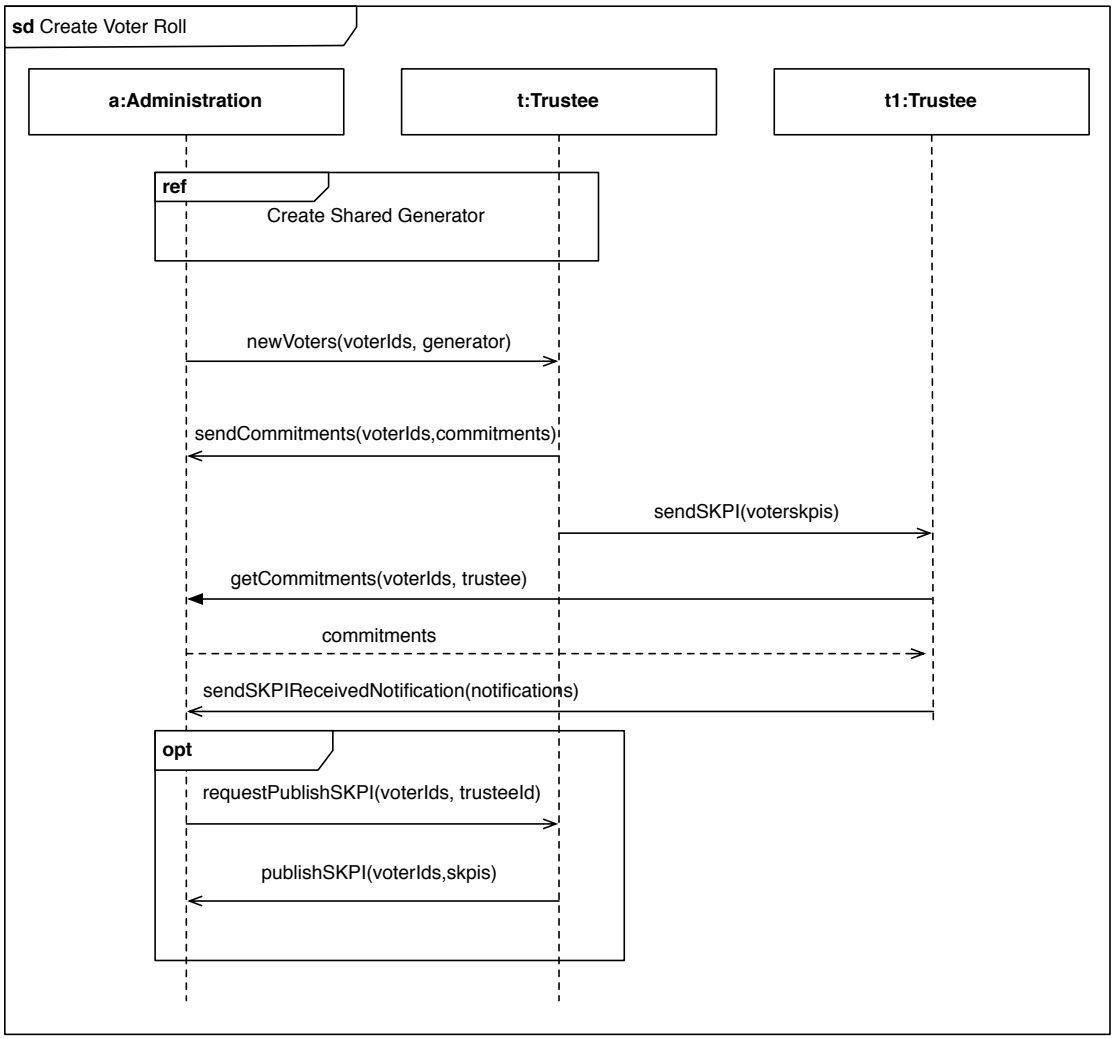

Abbildung 6.4: SH Sequenz Erstelle die *voter roll*

6.3.2 Erstellung der Abstimmung

In Abbildung 6.5 sieht man das Sequenzdiagramm für die Erstellung einer neuen Abstimmung. Wie aus der Abbildung ersichtlich ist, wird zuerst der Trustee über die neuen Abstimmung informiert. Mit dieser Information geht auch die Aufforderung einher, beim Erstellen von neuen Generatoren mit zu machen. Wie im Diagramm 6.6 zu sehen ist,

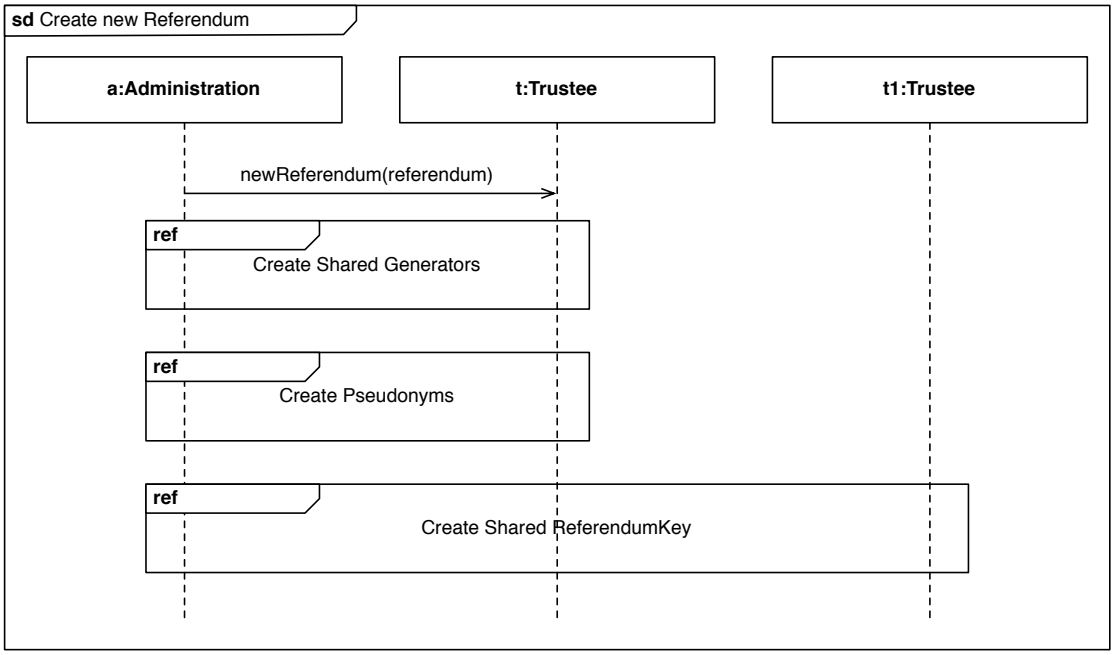

Abbildung 6.5: SH Sequenz Neue Abstimmung

antwortet der Trustee mit ensprechenden Informationen. Anschliessend müssen die Pseudoynme für die Voter erstellt werden. Dies ist detailliert beschrieben in Abbildung 6.7. Sobald der Trustee an der Reihe ist, seinen Teil beizutragen, teilt ihm dies die Administration mit. Anschliessend berechnet der Trustee seine Pseudonmyisierung und schickt diese zurück an die Administration. Diese fordert die anderen Trustees auf, ihre Permutationen für die *challenge* zu publizieren. Wenn sie alle Permutationen erhalten hat, erstellt sie die *challenge* und schickt diese dem Trustee. Der Trustee erstellt auf Grund der *challenge* den Beweis. Diesen schickt er nun an die Administration und diese publiziert den Beweis. Nun muss noch ein verteilter privater Schlüssel für die Abstimmung erstellt werden. Wie in Abbildung 6.8 zusehen ist, schicken zuerst die Trustees ihre *commitments* an die Administration und verteilen anschliessend ihre $s_{i,j}$ unter den anderen Trustees. Wenn ein Trustee ein $s_{i,j}$ erhält, informiert er die Administration darüber. Wenn etwas nicht in Ordnung war, fordert die Administration den betroffenen Trustee auf das $s_{i,j}$ zu publizieren.

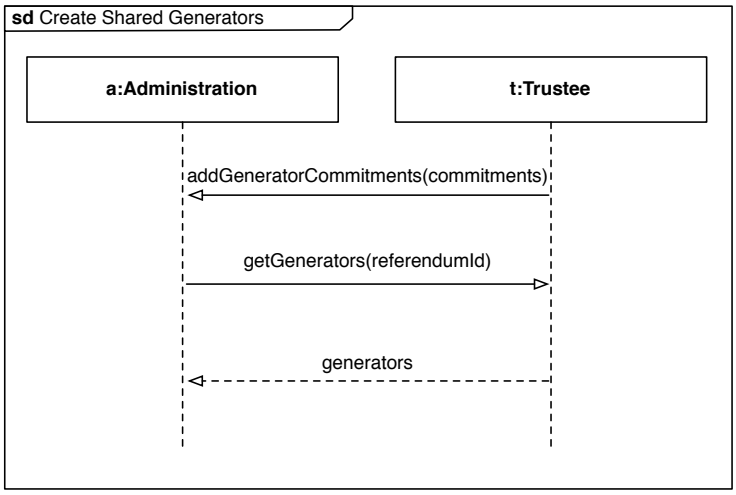

Abbildung 6.6: SH Sequenz Generiere mehrere verteilte Generatoren

6.3.3 Abstimmen

Registriere Voter

Bevor ein Voter abstimmen kann, muss er sich registrieren. Der dazu nötige Ablauf ist in Abbildung 6.9 dokumentiert. Zuerst benötigt der Voter ein Authentication Token von der Administration. Mit diesem Authentication Token kann er sich nun bei den Trustees mit seinem VotingCode registrieren.

Stimme abgeben

Wenn ein Voter abstimmen will, muss er sich an den Ablauf in Abbildung 6.10 halten. Zuerst benötigt er die Daten zu der Abstimmung. Diese ruft er von der Administration ab. Anschliessend fordert er von den Trustees seinen verteilten privaten Schlüssel an. Nun kann er seine Stimme erstellen und schlussendlich an die Administration abschicken.

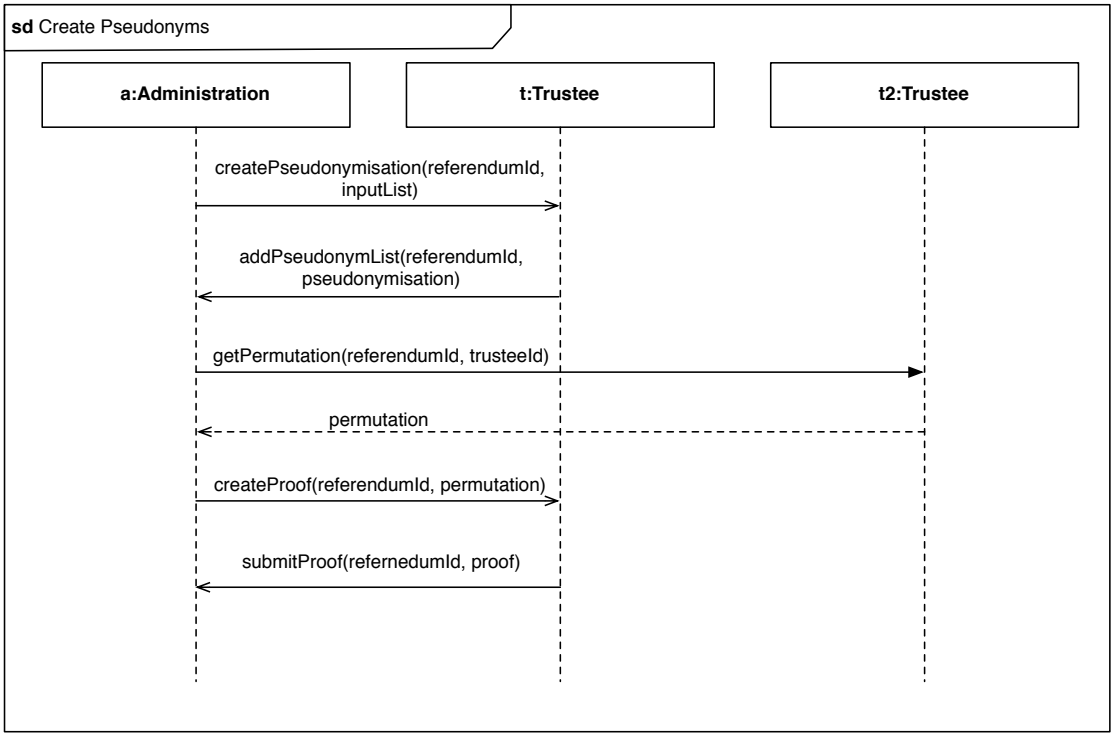

Abbildung 6.7: SH Sequenz Pseudonyme

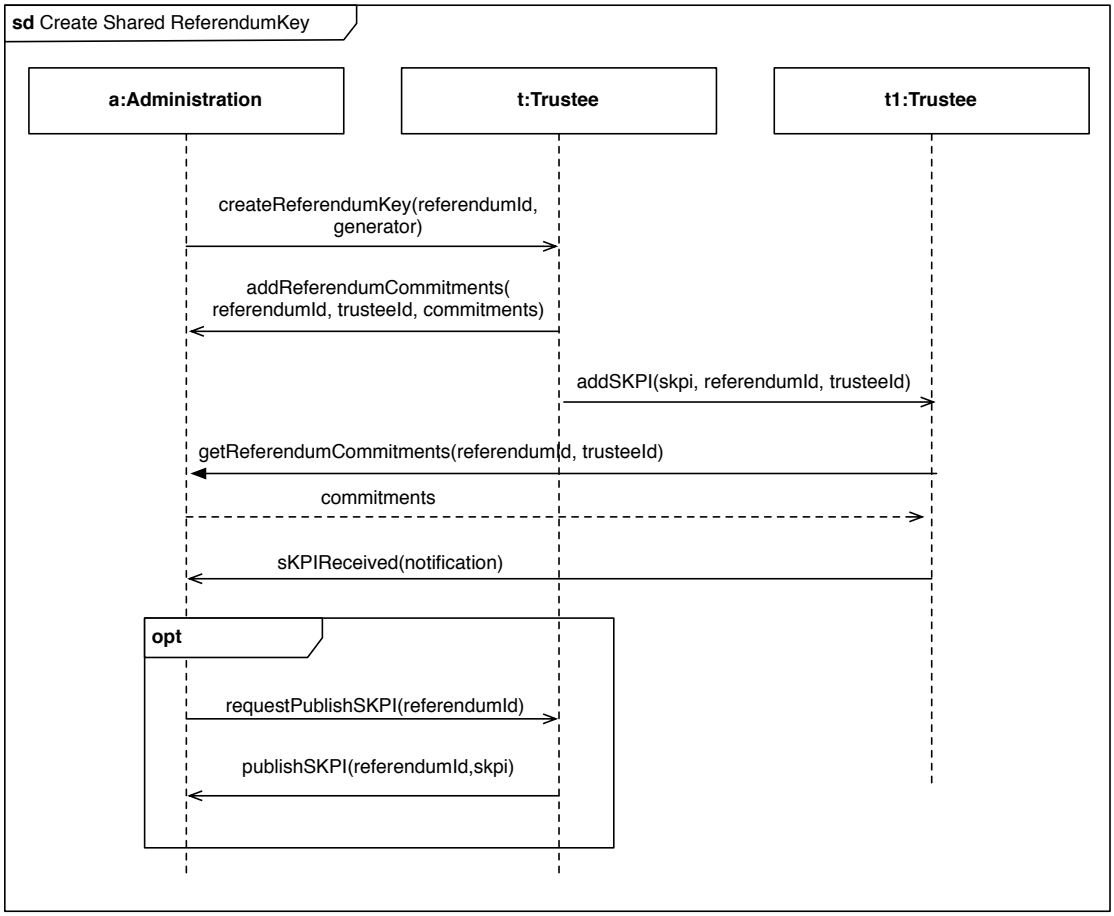

Abbildung 6.8: SH Sequenz Abstimmungsschlüssel

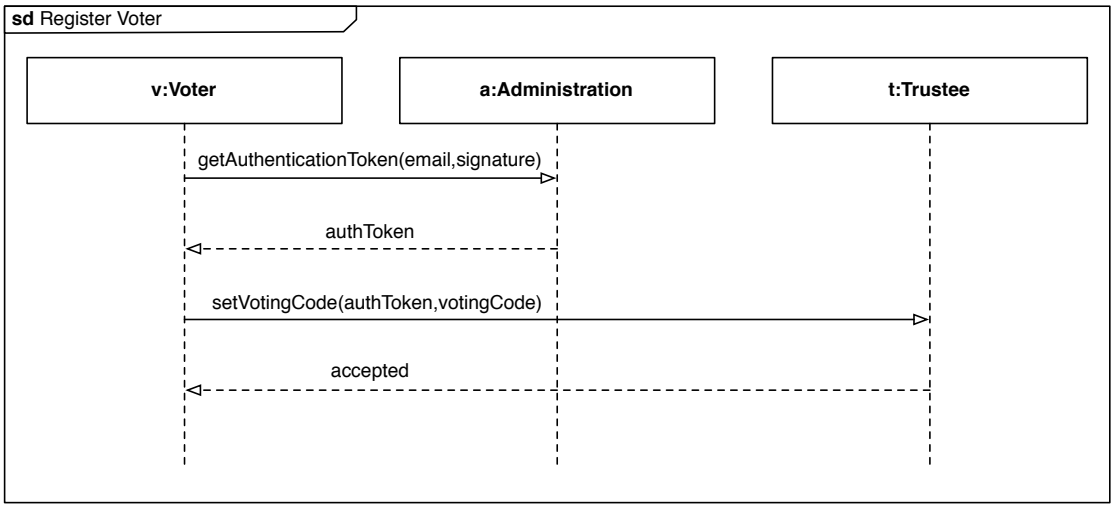

Abbildung 6.9: SH Sequenz Registriere neuen Voter

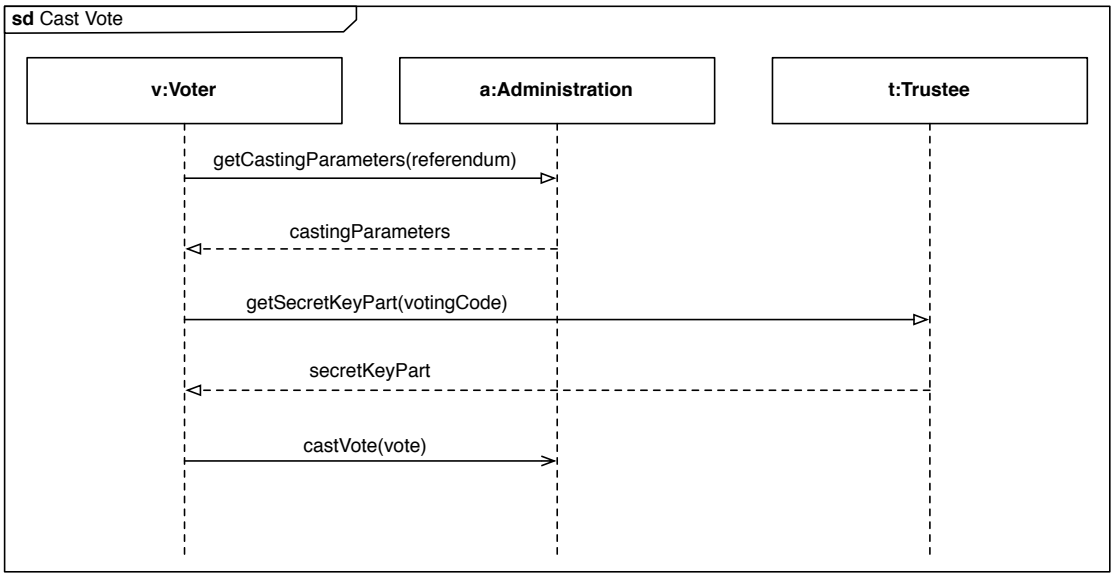

Abbildung 6.10: SH Sequenz Abstimmen

6.3.4 Auszählen

Damit die Stimmen ausgezählt werden können, muss der private Schlüssel der Abstimmung berechnet werden. Wie in Abbildung 6.11 zu sehen, fragt dafür die Administration alle Trustee für ihren Teilschlüssel an. Anschliessend kann der private Schlüssel berechnet und veröffentlicht werden.

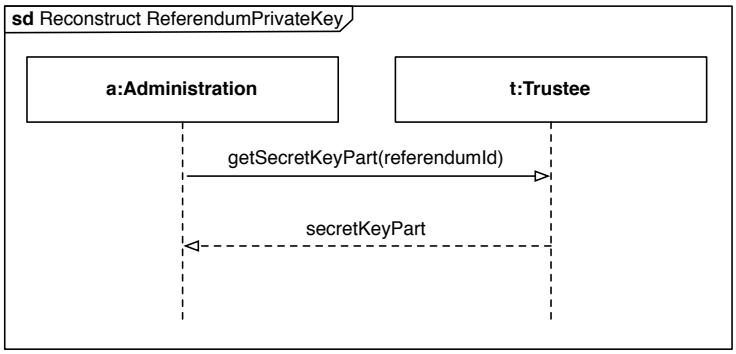

Abbildung 6.11: SH Sequenz Privater Schlüssel herstellen

6.4 Allgemeine technische Konzepte

6.4.1 Datenbank

Das Datenbankmodell wird durch die Java Persistence API(referenz) abstrahiert. Dies erlaubt die Datenstruktur direkt in Java zu modellieren und erhöht die Flexibilität bei der Verwendung der Datenbank Software.
Auf Grund der Grössenbeschränkung von BigIntegern in SQL, werden BigInteger in diesem System in der Datenbank als Strings gespeichert. Dabei wird für die Umwandlung die Basis 36 verwendet. In Java erfolgt die Verwendung der Basis mittels Character.MAX_RADIX.

6.4.2 Voterblöcke

Durch die Aufteilung der Voter in verschiedene Blöcke lässt sich beim Erstellen der Pseudonyme einen markanten Performancegewinn realisieren. Darum findet man in der Implementierung von Selectio Helvetica diese Aufteilung der Voter, welche nicht im Protokoll dokumentiert sind.
Diese Anpassung des Protokolls hat keine Auswirkungen auf die Sicherheitseigenschaften solange die Blockgrösse genügend gross gewählt wird.

7 Implementation

Leider war es auf Grund der knappen Zeitspanne nicht mögich eine komplette Implementation zu erstellen. Hier werden die fertigen Teile genannt und interessante Aspekte aufgeführt.

7.1 Allgemeine Implementationsdetails

7.1.1 Speicherung von BigInteger als String

Wie bereits im Kapitel über die Architektur erwähnt wurde, werden die verwendeten BigInteger als Strings gespeichert, weil ansonsten das Mapping von JPA ungenügend ist. Der folgende Codeauszug zeigt, wie die Lösung in der Implementation aussieht. Dabei ist wichtig, dass allfällige Annotationen für JPA bei der Variable gesetzt werden und nicht bei den Methoden, da ansonsten JPA trotzdem BigInteger verwendet in der Datenbank. Wie in dem Codeauszug zu sehen ist, werden alle betroffenen Variablen mit einem entsprechenden Kommentar versehen.

```
1     private String generator; //BigInteger
2
3     public BigInteger getGenerator() {
4         return new BigInteger(generator,Character.MAX_RADIX);
5     }
6
7     public void setGenerator(BigInteger generator) {
8         this.generator = generator.toString(Character.MAX_RADIX);
9     }
```

7.1.2 Testen

Java Klassen

Normale Java Klassen sollen mit JUnit getesten werden. Wichtig sind hier vor allem die Klassen welche die kryptographischen Funktionen bereit stellen. Die Testabdeckung der Java Klassen soll 100% erreichen.
Ziel der Tests ist es die Korrektheit der mathematischen und logischen Funktionen zu garantieren.

EJB und WebServices

Das Testen von EJBs und WebServices ist nicht sehr einfach. Getestet wird mit einem embedded Glassfish und einer embedded Derby Datenbank. Um WebService-Aufrufe aus den EJB zu testen, müssen Mock-WebServices implementiert werden. Hier wird eine Testabdeckung von 20% angestrebt, da die Entwicklung der Tests viel Zeit beansprucht und meistens keinen grossen Nutzen bringen.

Integrationstests

Die Integrationstest sind wichtig um die vielen verschiedenen Schnittstellen und Abläufe zu testen. Ziel ist ein Integrationstest für eine komplette Abstimmung. Leider war im Zuge dieser Arbeit keine Zeit einen solchen Testablauf zu definieren und dokumentieren.

7.2 SH Adminstration

Bei der Administration ist das Mapping für die Datenbank komplett implementiert. Ausserdem sind alle benötigten kryptographischen Funktionen vorhanden. Zusätzlich wurde der Prozess für die Erstellung einer Abstimmung teilweise implementiert.

7.2.1 Systemdesign

Wie in der Abbildung 7.1 zu sehen ist, wurden die einzelnen Teile soweit als möglich von einander separiert. Jeder der grossen Blöcke repräsentiert eine eigene Bibliothek. Auf der Präsentationsebene wurden die einzelnen Komponenten nach verwendeter Technologie aufgetrennt. Der Gedanke dahinter war, dass sich so einzelne Technologien einfacher und sauberer ersetzten lassen.
Die WebService-Clients wurden auf Integrationsebene auch in eine Bibliothek gepackt. Dies weil hier sehr viel Code automatisch generiert wird und möglichst verhindert werden soll, dass dies zu unerwünschten Nebeneffekten führt.

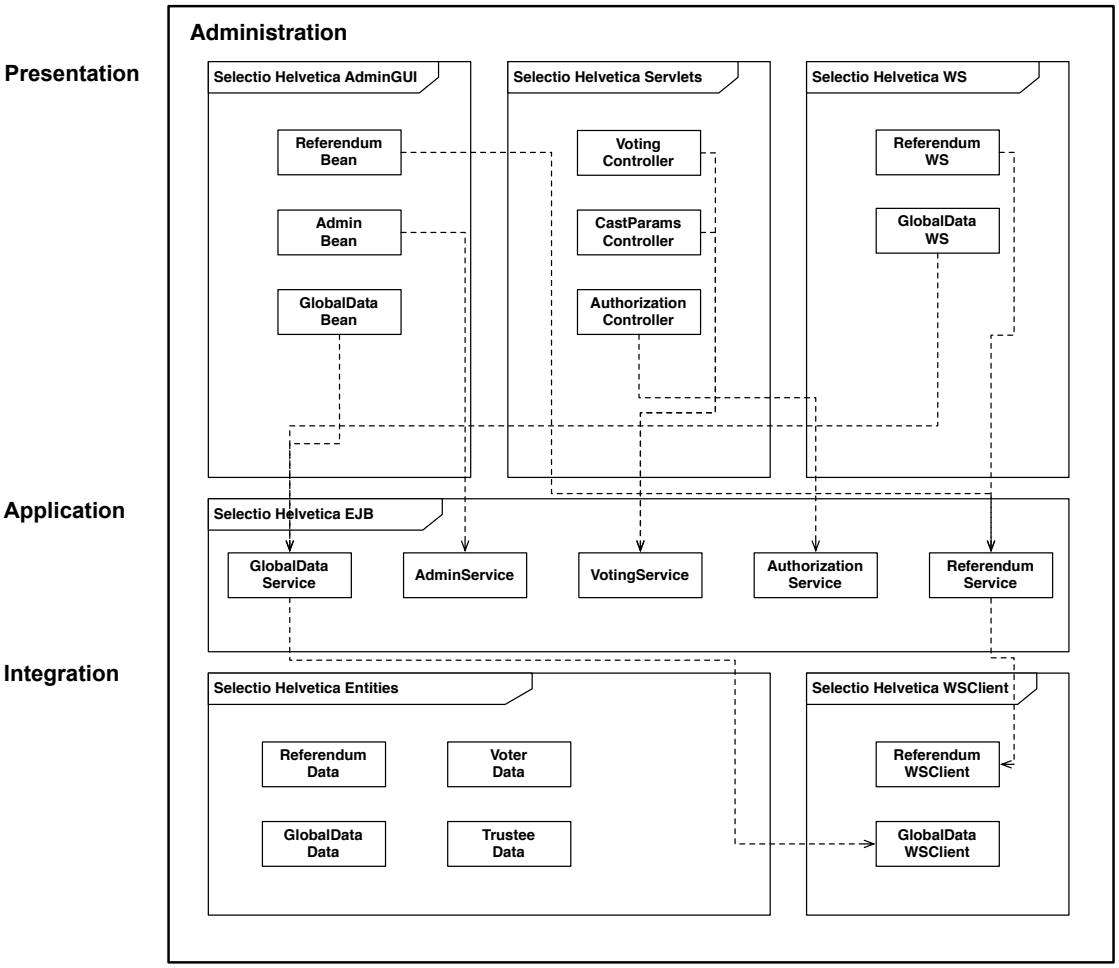

Abbildung 7.1: SH Administration Design

7.2.2 Datenmodell

Wie in Abbildung 7.2 zu sehen, besteht das Datenmodell aus 16 verschiedenen Entitäten. Es ist im Grunde ein relativ einfaches Datenbankmodell. Einzig diese Doppelverbindung von ReferendumData und TrusteeData kann etwas verwirrend sein. Diese Konstellation ergab sich, weil zwei unabhängige Prozesse jeweils beide Daten nutzen. Hier folgt nun eine genaue Beschreibung aller Entitäten. Die Auflistung erfolgt über die Relationen von links nach rechts und von oben nach unten.

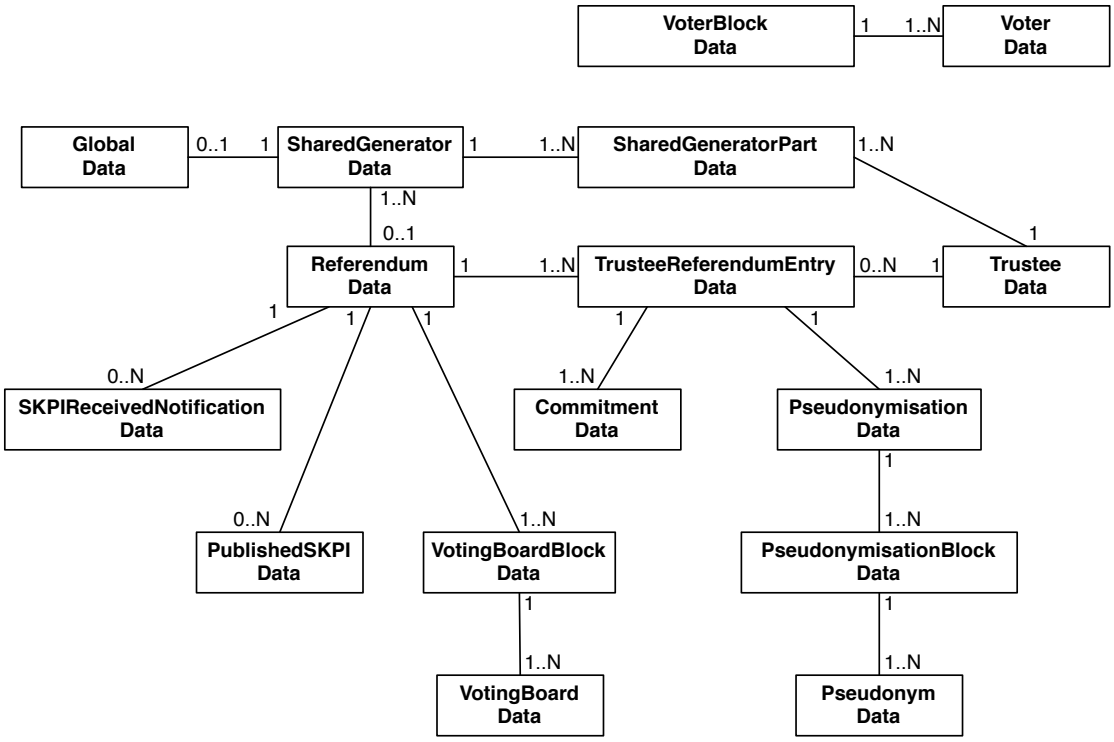

Abbildung 7.2: SH Administration Datenmodell

VoterBlockData

- *blockNumber - Integer* Nummer des Blocks.
- *voters - List(VoterData)* Die Voter, welche in diesen Block gehören.

VoterData

- *publicId - BigInteger* Identifikation des Voters.
- *emailAddress - String* Email Adresse des Voters.

GlobalData

- *blockSize* - *Integer* Gibt an wieviele Voter in einen Block zusammen gefasst werden.
- *modulus* - *BigInteger* Für die kryptographischen Funktionen verwendete Primzahl p.
- *subgroupOrder* - *BigInteger* Für die kryptographischen Funktionen verwendete Primzahl q.
- *globalGenerator* - *SharedGenerator* Generator g um die *voter roll* zu erstellen.
- *securityParameter* - *Integer* Gibt die Bit-Länge an, die neu generierten BigInteger haben muss.

SharedGeneratorData

- *generator* - *BigInteger* Zahl, welche den kompletten Generator repräsentiert.
- *generatorParts* - *List(SharedGeneratorPartData)* Eine Liste aller Teile für diesen Generator.
- *gIndex* - *Integer* Index dieses Generators.

SharedGeneratorPartData

- *generatorPart* - *BigInteger* Teilstück des Generators.
- *commitment* - *BigInteger* Das vom Trustee abgegebene *commitment* für dieses Teilstück.
- *trustee* - *TrusteeData* Trustee, welcher dieses Teilstück berechnet hat.

ReferendumData

- *referendumId* - *String* Identifikation der Abstimmung.
- *startDate* - *Date* Startdatum der Abstimmung.
- *endDate* - *Date* Enddatum der Abstimmung.
- *publicKey* - *BigInteger* Öffentlicher Schlüssel der Abstimmung.
- *encryptionGenerator* - *SharedGeneratorData* Generator g_e um die Votes zu ver- und entschlüsseln.
- *pseudonymGenerator* - *BigInteger* Generator g_p um die Pseudonyme zu berechnen.
- *pseudonymProofGenerators* - *List(SharedGeneratorData)* Alle Generatoren, welche für den Beweis der Pseudonyme benötigt werden.
- *threshold* - *Integer* Gibt an wie viele Trustees benötigt werden um den privaten Schlüssel zu berechnen.
- *trustees* - *List(TrusteeReferendumEntryData)* Alle Trustees, die zu dieser Abstimmung eingeladen sind.
- *notifications* - *List(SKPIReceivedNotificationData)* Alle Notifikationen der Trustees, welche für diese Abstimmung von einem anderen Trustee ein SKPI erhalten haben.

- *publishedSKPIs - List(PublishedSKPIData)* Alle SKPI die für diese Abstimmung publiziert wurden.

TrusteeReferendumEntryData

- *eIndex - Integer* Index für diesen Trustee während dieser Abstimmung.
- *disqualified - Boolean* Gibt an ob der Trustee für die Abstimmung disqualifiziert werden musste.
- *trustee - TrusteeData* Referenz auf den Trustee.
- *commitments - List(CommitmentData)* Alle *commitments*, welche die Administration von diesem Trustee erhalten hat.
- *pseudonymisation - PseudonymisationData* Die Pseudonymisierung, welche die Administration von diesem Trustee erhalten hat.

TrusteeData

- *trusteeId - String* Identifikator für diesen Trustee.
- *url - String* Basis URL unter der dieser Trustee erreichbar ist.
- *contact - String* Email Adresse unter welche der Verantwortliche erreicht werden kann.

SKPIReceivedNotificationData

- *senderId - String* Identifikator des Trustees, welcher das $s_{i,j}$ gesendet hat.
- *receiverId - String* Identifikator des Trustees, welcher das $s_{i,j}$ erhalten hat und die Administration informiert.
- *valid - Boolean* Gibt an, ob der Empfänger das $s_{i,j}$ akzeptiert hat oder nicht.

CommitmentData

- *commitment - BigInteger* Das *commitment* für eine bestimmte Stelle aus dem Polynom des Trustees.
- *cIndex - Integer* Gibt an, zu welcher Stelle des Polynoms dieses *commitment* gehört.

PseudonymisationData

- *pseudonymGeneratorA1 - BigInteger* Der Generator m
- *pseudonymGeneratorA2 - BigInteger* Der Generator g_{i+1}
- *pseudonymisationBlocks - List(PseudonymisationBlockData)* Alle Datenblöcke zu dieser Pseudonymisation.

PublishedSKPIData

- *senderId - String* Identifikator des Trustees, welcher das $s_{i,j}$ gesendet hat.
- *receiverId - String* Identifikator des Trustees für welchen das $s_{i,j}$ bestimmt war.

- *skpi - BigInteger* Das publizierte $s_{i,j}$.

VotingBoardBlockData

- *vbIndex - Integer* Nummer des Blocks.
- *votingBoardData - List(VotingBoardData)* Alle Einträge für diesen Block.

PseudonymisationBlockData

- *pdIndex - Integer* Nummer des Blocks. Dient dazu, dass der Block bei Bedarf einfacher dem originalen VoterBlock zugeordnet werden kann.
- *inputList - Map(PseudonymData)* Die Liste S_i, welche der Trustee erhalten hat zum Pseudonymisieren.
- *middleList - Map(PseudonymData)* Die Liste M_i.
- *outputList - Map(PseudonymData)* Die Liste S_{i+1}.

VotingBoardData

- *pseudonym - BigInteger* Das Pseudonym.
- *encryptedVote - Embed(ELGamalCipher)* Verschlüsselte Stimme für dieses Pseudonym.
- *voteSignature - Embed(ELGamalSignature)* Signatur der verschlüsselten Stimme.
- *vote - String* Die entschlüsslte Stimme.

PseudonymData

- *psIndex - Integer* Index des Pseudonyms im Block.
- *pseudonym - BigInteger* Das Pseudonym.

7.3 SH Trustee

Beim Trustee ist das Datenbankmodell komplett. Ausserdem sind alle notwendigen Komponenten für die kryptographischen Operationen vorhanden.

7.3.1 Systemdesign

Wie in Abbildung 7.3 zu sehen ist, wurde auch beim Trustee auf der Präsentationsebene die verschiedenen Technologien in eigene Bibliotheken gepackt. Dies erlaubt diese einfach und sauber zu ersetzten. Auf der Integrationsebene wurde auch die WebService-Clients wieder separiert. Dies weil, wie auch bei der Administration, hier viel Code automatisch generiert wird. Damit die Übersicht besser gewahrt werden kann und es zu keinen Konflikten kommt, wurden diese vom restlichen Code separiert.

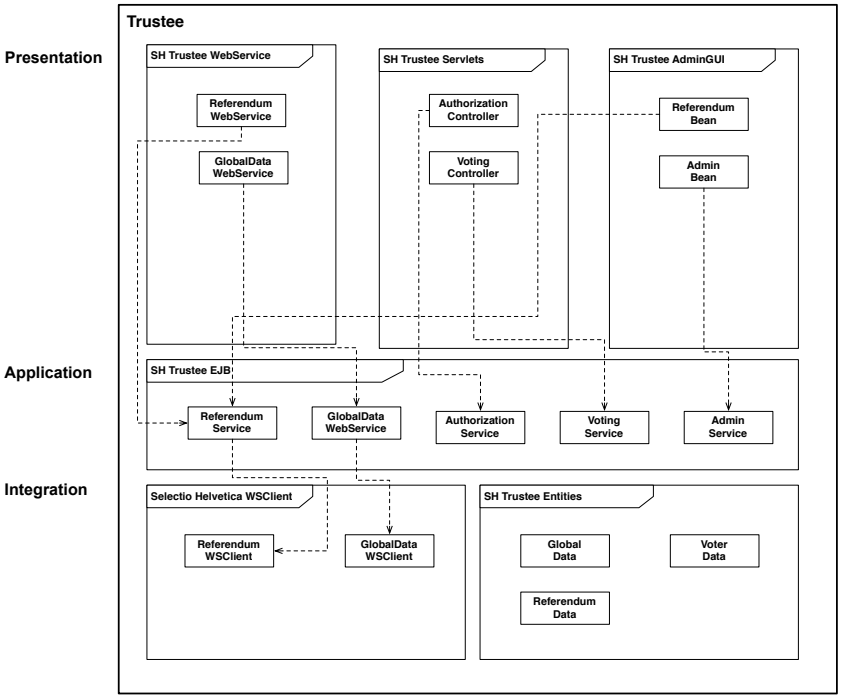

Abbildung 7.3: SH Trustee Design

7.3.2 Datenmodell

Wie in Abbildung 7.4 zu sehen, besteht das Datenmodell aus 11 verschiedenen Entitäten. Wichtig ist anzumerken, dass SharedGenerator keine Verbindung zwischen GlobalData und ReferendumData darstellt, sondern lediglich zur einen oder anderen Entität gehört. Hier folgt eine genaue Beschreibung dieser Entitäten. Die Auflistung erfolgt über die Realationen von oben nach unten und von links nach rechts.

GlobalData

- *blockSize* - *Integer* Gibt an wie viele Voter in einen Block zusammen gefasste werden.
- *modulus* - *BigInteger* Für die kryptographischen Funktionen verwendete Primzahl p.
- *subgroupOrder* - *BigInteger* Für die kryptographischen Funktionen verwendete Primzahl q.
- *globalGenerator* - *BigInteger* Generator g um die *voter roll* zu erstellen.
- *globalGeneratorPart* - *BigInteger* Teilstück von g, welches von diesem Trustee definiert wurde.
- *generatorCommitment* - *BigInteger commitment* zum Teilstück vom Generator.

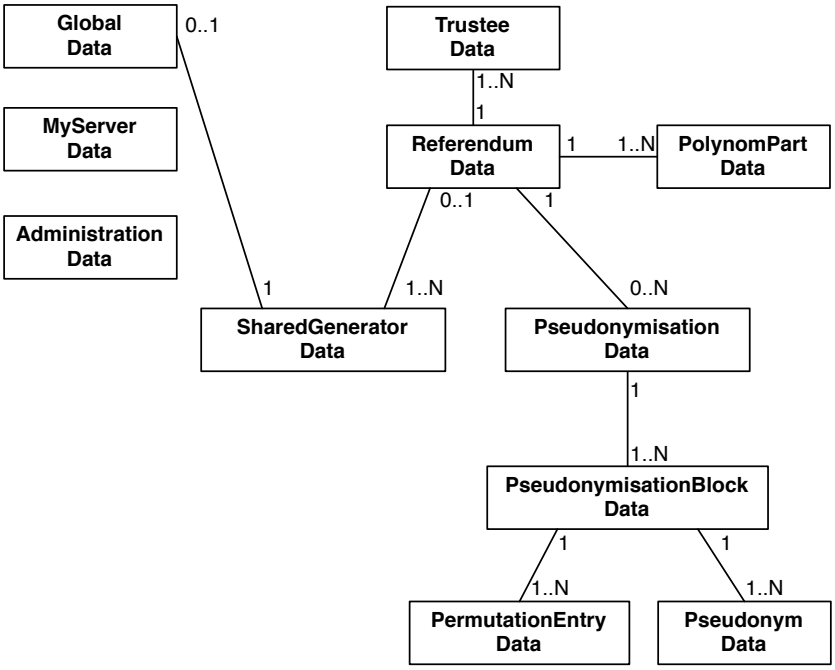

Abbildung 7.4: SH Trustee Datenmodell

- *securityParameter* - *Integer* Gibt die Bit-Länge an, die neu generierten BigInteger haben muss.

MyServerData

- *myId* - *String* Der Identifikator dieses Trustees.
- *myURL* - *String* Die URL unter welcher dieser Trustee erreichbar ist.

AdministrationData

- *URL* - *String* Die URL unter welcher die Administration erreichbar ist.

SharedGeneratorPartData

- *gIndex* Index des Generators für den dieses Teilstück gedacht ist.
- *generatorPart* - *BigInteger* Teilstück des Generators.
- *commitment* - *BigInteger* Das vom Trustee abgegebene *commitment* für dieses Teilstück.
- *generator* - *BigInteger* Der gemeinsame Generator.

TrusteeData

- *trusteeId - String* Identifikator des Trustees.
- *URL - String* Die URL unter der Trustee erreichbar ist.
- *tdIndex - Integer* Der Index dieses Trustees für die Abstimmung.
- *receivedSKPI - BigInteger* Das $s_{i,j}$ das von diesem Trustee empfangen wurde.
- *sKPI - BigInteger* Das $s_{i,j}$, welches an diesen Trustee gesendet wurde.

ReferendumData

- *referendumId - String* Identifikation der Abstimmung.
- *startDate - Date* Startdatum der Abstimmung.
- *endDate - Date* Enddatum der Abstimmung.
- *trustees - List(TrusteeData)* Alle anderen Trustees, an dieser Abstimmung beteiligt sind.
- *threshold - Integer* Gibt an wie viele Trustees benötigt werden um den privaten Schlüssel zu berechnen.
- *tIndex - Integer* Der Index dieses Trustees für diese Abstimmung.
- *polynom - Map(PolynomPartData)* Das Polynom, welches dieser Trustee ausgewählt hat.
- *pseudonymisation - PseudonymisationData* Die Pseudonymisation, welcher der Trustee für diese Abstimmung erstellt hat.

PseudonymisationData

- *pseudonymGeneratorA1 - BigInteger* Zufallswert α_1
- *pseudonymGeneratorA2 - BigInteger* Zufallswert α_2
- *pseudonymisationBlocks - List(PseudonymisationBlockData)* Alle Datenblöcke zu dieser Pseudonymisation.

PseudonymisationBlockData

- *pdIndex - Integer* Nummer des Blocks. Dient dazu, dass der Block bei Bedarf einfacher dem originalen VoterBlock zugeordnet werden kann.
- *inputList - Map(PseudonymData)* Die Liste S_i, welche der Trustee erhalten hat zum Pseudonymisieren.
- *middleList - Map(PseudonymData)* Die Liste M_i.
- *outputList - Map(PseudonymData)* Die Liste S_{i+1}.

PseudonymisationBlockData

- *pdIndex - Integer* Nummer des Blocks. Dient dazu, dass der Block bei Bedarf einfacher dem originalen VoterBlock zugeordnet werden kann.

- *inputList - Map(PseudonymData)* Die inputList S_i, welche der Trustee erhalten hat zum Pseudonymisieren.
- *middleList - Map(PseudonymData)* Die middleList M_i.
- *outputList - Map(PseudonymData)* Die outputList S_{i+1}.
- *permutation1 - Map(PermutationData)* Die Permuation π_1, welche S_i mit M_i verbindet.
- *permutation2 - Map(PermutationData)* Die Permuation π_2, welche M_i mit S_{i+1} verbindet.

PermutationEntryData

- *oldIndex - Integer* Alte Position.
- *newIndex - Integer* Neue Position.

PseudonymData

- *psIndex - Integer* Index des Pseudonyms im Block.
- *pseudonym - BigInteger* Das Pseudonym.

PolynomPartData

- *secretPart - BigInteger* Zufälliger Wert $a_{i,k}$
- *commitment - BigInteger* Das aus $a_{i,k}$ errechnete *commitment* $A_{i,k}$.
- *pIndex - Integer* Gibt die Position k innerhalb des Polynoms an.

7.3.3 Erstellen einer Permutation

In der Implementation wird eine Permutation von einer Liste von PermutationEntry's repräsentiert. Jedes PermutationEntry besitzt, wie in Listing 7.1 zu sehen ist, zwei Werte. Dabei steht der erste Wert für die Position in der alten Liste und der zweite für die Position in der neuen Liste.

```
1 public class PermutationEntry {
2     private Integer oldIndex;
3     private Integer newIndex;
4     // Getter/Setter omitted
5   }
```

Listing 7.1: PermutationEntry

Im unten stehenden Listing 7.2 ist zu sehen, wie diese Wertepaare erstellt werden. Zuerst wird eine Liste der möglichen neuen Positionen erstellt (newIndexes) . Nun wird iterativ für jede alte Position (oldIndex) eine neue Position (newIndex) zufällig ausgewählt. Diese wird der alten Position zugewiesen und anschliessend aus dem Pool der neuen Positionen entfernt.

```
1 public class PermutationGenerator {
```

```
2      public static Map<Integer,PermutationEntry> createPermutation(Integer
           size) {
3          List<Integer> newIndexes = new ArrayList<Integer>();
4          for (int i = 0; i <= size-1;i++) {
5              newIndexes.add(i);
6          }
7
8          Map<Integer,PermutationEntry> permutations = new
               HashMap<Integer,PermutationEntry>();
9          for (int oldIndex = 0; oldIndex <= size-1;oldIndex++) {
10             PermutationEntry permutationEntry = new PermutationEntry();
11             permutationEntry.setOldIndex(oldIndex);
12             Integer newIndex;
13             if (size-(1+oldIndex) > 0 ) {
14             newIndex = newIndexes.remove((new
                   SecureRandom()).nextInt(size-(oldIndex)));
15             } else {
16                 newIndex = newIndexes.remove(0);
17             }
18             permutationEntry.setNewIndex(newIndex);
19             permutations.put(permutationEntry.getOldIndex(),
                   permutationEntry);
20         }
21         return permutations;
22         }
23 }
```

Listing 7.2: PermutationGenerator

7.4 SH Voter

Beim Voter gibt es nur eine Implementation in JavaScript für Selectio Helvetica Light. Der Prozess ist eigentlich in vielen Punkten gleich, also könnte die Implementation zu grossen Teilen wiederverwendet werden. Allerdings hat JavaScript bei grossen BigIntegern zurzeit Performanceprobleme. Aus diesem Grund eignet sich die Implementation nur bedingt für Selectio Helvetica. Geplant ist die Umstellung der kryptographischen Berechnungen auf Java.
Das Design und die Strukturen für die Implementation bestehen jedoch schon und werden hier nun aufgezeigt.

7.4.1 Systemdesign

Wie in Abbildung 7.5 zu sehen ist, besteht die Implementation aus einer Klasse, welche durch interne Funktionsaufrufe den Ablauf kontrolliert. Die beiden öffentlichen Funktionen dienen der Auslösung des Prozesse. Dabei werden als Input einerseits ein Objekt mit allen notwendigen Daten erwartet und andererseits eine Rückruf-Funktion (callback), welche mit der Antwort (RequestStatus) umgehen kann.
Für die Kommunikation mit der Administration und den Trustees werden HTTP-Requests

verwendet. Diese verwenden ebenfalls Rückruf-Funktionen. Dies zeigt sich in den internen Funktionen, welche mit "process" beginnen. Ausserdem kann beim Vergleich mit dem Prozess in Kapitel 5.3 festgestellt werden, dass die internen Funktionen den einzelnen Aktivitäten des Prozesses entsprechen.

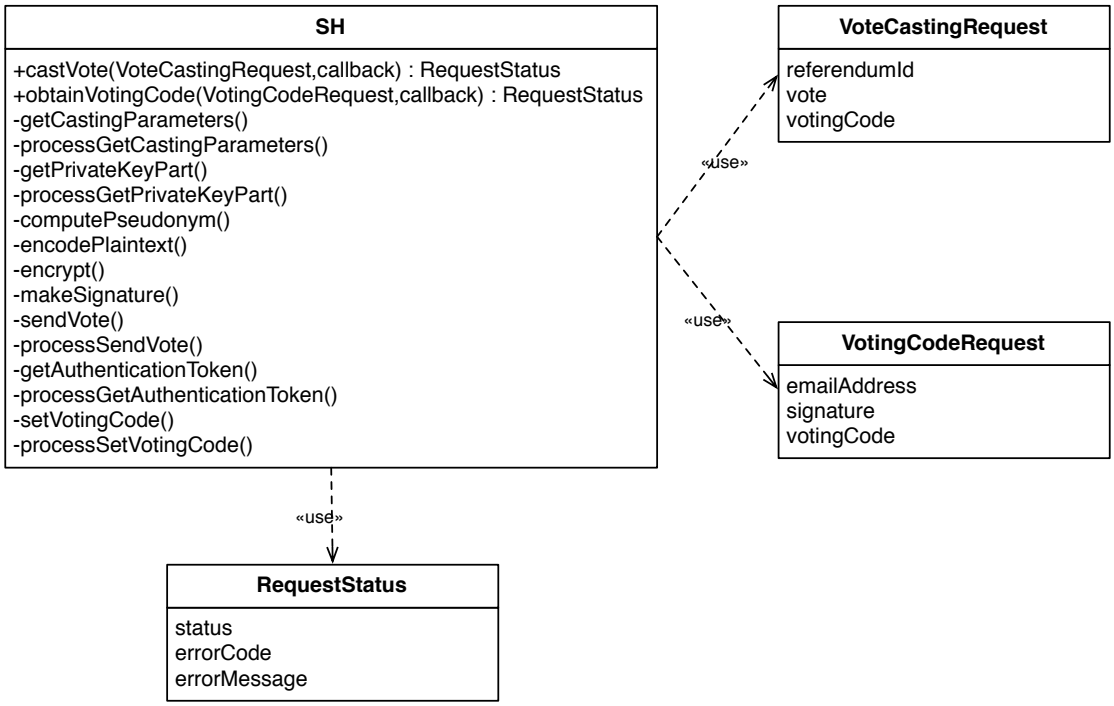

Abbildung 7.5: SH Voter Klassendiagramm

8 Fazit und Ausblick

Die Entwicklung von Prozessen, welche mehrere verschiedene Akteure einbeziehen, ist eine komplizierte Angelegenheit. Dies weil, die korrekte Synchronisierung der verschiedenen Akteure meistens sehr Fehler anfällig ist. Dadurch müssen die Prozesse sehr robust konzipiert sein. Vor allem, wenn wie bei den hier verwendeten Kryptoprotokollen, da nicht davon ausgegangen werden kann, dass alle Akteure ihre Rolle erfüllen. Genau diese Anforderungen sind es aber auch, die diese Arbeit sehr spannend und auch fordernd macht. Da ich bereits in einem vorhergehenden Projekt Erfahrungen in dieser Richtung sammeln konnte und dadurch eine Vorstellung hatte, wie das Resultat aussehen könnte, konnte ich trotz der kurzen Zeitspanne das Protokoll in funktionierende Prozesse umwandeln. Durch den engen Zeitrahmen hatte man kaum Zeit innerhalb der Arbeitsgruppe die Entscheidungen und Resultate ausgiebig zu besprechen und analysieren. Dadurch haben sich im Verlauf des Projekts auch ein, zwei kleine Fehler eingeschlichen, welche später korrigiert werden mussten.
Trotz der kurzen Zeitspanne, war es mir möglich innerhalb dieser Arbeit die Grundlagen für die Implementation zu entwerfen und die wichtigsten Komponenten zu implementieren.
Für die Zukunft gibt es noch viele Arbeiten, mögliche Anpassungen und Erweiterungen, welche ins Auge gefasst werden könnten. Einerseits sicher eine komplette Implementation des Protokolls und der beschriebenen Architektur, damit gezeigt werden kann, dass die Prozesse im Sinne des Protokolls funktionieren. Des Weiteren könnte man die Prozesse und die Implementation so anpassen, dass die Registrierung eines Voters ohne Baloti funktioniert. Dadurch wäre die Implementation auch für andere Projekte einsetzbar.
Auf Ebene des Protokolls gibt es die Möglichkeit, dass man der Administration das Publizieren von öffentlichen Daten abnimmt und dies an ein Bulletinboard überträgt. Es wäre sogar ein Szenario möglich, in welcher die Trustees miteinander so ein Bulletinboard betreiben. Die Administration hätte dann nur noch die minimal notwendige Kontrolle über die Daten der Abstimmung. Was dem Konzept des Protokolls von öffentlichen Daten noch besser entsprechen würde.
Ausserdem könnte man neben den Prozessen auch noch Zustandsdiagramme für Selectio Helvetica im Gesamten und die Akteure im Einzelnen erstellen. Dies würde in Kombination mit den Prozessen das Protokoll weiter konkretisieren und bei der Implementierung erweiterte Testszenarien ermöglichen.

Glossar

coercion resistance
Bei coercion resistance handelt es sich um eine Erweiterung der Eigenschaft "Privatheit". Dabei wird gefordert, dass man den Erpresser auch täuschen kann. Details siehe [JCJ05].

commitment
In der Kryptographie erlaubt das commitment-Schema sich zur Verwendung eines Wertes zu verpflichten ohne das dieser Wert offengelegt wird.

Derby
Ein auf Java basiertes relationales Datenbank-Managment-System.

EJB
Komponente innerhalb der JavaEE Umgebung.

ElGamal
ElGamal ist ein asymmetrischer Verschlüsselungsalgorithums beruhend auf dem mathematischen Problem des diskreten Logarithmus.

Generator
In der Gruppentheorie verwendeter Begriff für eine Untergruppe von Elementen, welche die gesamte Gruppe aufspannen.

Glassfish
GlassFish ist ein Open-Source-Projekt eines Java EE Servers von Sun Microsystems.

JavaEE
Steht für Java Enterprise Edition. Bezeichnet die Spezifikation für die transaktionsbasierte Ausführung von Java-Anwendungen.

JSON
JavaScript Object Notation. Eine kompaktes Datenformat in menschen-lesbarer Form für den Austausch zwischen Anwendungen.

JUnit
Ein Framework zum Testen von Java-Programmen mittels Unit-Testing.

Mock
: Attrape eines Objekts, welches bei Tests verwendet wird um die Umgebung des zu testenden Codes nachzubilden.

Pseudonym
: Ein Pseudonym ist ein öffentlicher Schlüssel aus einem ElGamal Schlüsselpaar. Berechnet wird ein Pseudonym, indem bei einem bekannten öffentlichen Schlüssel der Generator verändert wird.

Webservices
: Definition eines XML-basierten, plattformunabhängigen Netzwerkschnittstelle für Anwendungen.

WSDL
: Web Services Description Language. Eine Sprache, welche es erlaubt Webservices auf Basis von XML zu beschreiben.

Literaturverzeichnis

[JCJ05] Ari Juels, Dario Catalano, and Markus Jakobsson. Coercion-resistant electronic elections. In *Proceedings of the 2005 ACM workshop on Privacy in the electronic society*, WPES '05, pages 61–70, New York, NY, USA, 2005. ACM.

[JJR02] Markus Jakobsson, Ari Juels, and Ronald L. Rivest. Making mix nets robust for electronic voting by randomized partial checking. In *Proceedings of the 11th USENIX Security Symposium*, pages 339–353, Berkeley, CA, USA, 2002. USENIX Association.

[KDH10] R. König, E. Dubuis, and R. Haenni. Why public registration boards are required in e-voting systems based on threshold blind signature protocols. In R. Krimmer and R. Grimm, editors, *EVOTE'10, 4th International Workshop on Electronic Voting*, number P-167 in Lecture Notes in Informatics, pages 255–266, Bregenz, Austria, 2010. Gesellschaft für Informatik E.V.

[Ped91] Torben Pryds Pedersen. A threshold cryptosystem without a trusted party. In *Proceedings of the 10th annual international conference on Theory and application of cryptographic techniques*, EUROCRYPT'91, pages 522–526, Berlin, Heidelberg, 1991. Springer-Verlag.

[SH10] O. Spycher and R. Haenni. A novel protocol to allow revocation of votes in a hybrid voting system. In *ISSA'10, 9th Annual Conference on Information Security – South Africa*, Sandton, South Africa, 2010.