ELSEVIER

Future Generation Computer Systems 19 (2003) 69–85

FGCS
FUTURE GENERATION COMPUTER SYSTEMS

www.elsevier.com/locate/future

A secure and available electronic voting service for a large-scale distributed system

Gianluca Dini*

Dipartimento di Ingegneria della Informazione: Elettronica, Informatica, Telecomunicazioni, Via Diotisalvi 2, 56126 Pisa, Italy

Received 29 October 2001; received in revised form 31 May 2002; accepted 27 June 2002

Abstract

We present a secure and available electronic voting service suitable for a large-scale distributed system such as the Internet. The proposed service is based on replication and tolerates both benign and fully arbitrary failures. If enough servers are correct, service availability and security are ensured despite the presence of faulty servers and malicious voters. A voter that is affected by a crash failure can vote after recovery. The proposed service satisfies common voting requirements including voter eligibility and privacy, and tally accuracy. In addition, the service satisfies a further important requirement, namely tally verifiability without any intervention of voters. Anyone, including an external observer, can easily be convinced that the voting outcome is fairly computed from the ballots that were correctly cast. It follows that the proposed voting scheme strengthens the security properties of the electronic voting procedure, and simplifies the interaction of voters with the electronic voting system.
© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Distributed systems; Applied cryptography; Security and privacy; Voting protocols

1. Introduction

Electronic voting (e-voting) refers to a process whereby people can cast their votes through a large-scale communication system, such as the Internet, from any place where they can get network access [41]. The problem of deploying an e-voting service over a large-scale distributed system is not new, and has been extensively investigated [1–4,7,8,11,14,15,20,23,25,30,34]. However, over the last year there has been a reinforced interest in e-voting as a way to make voting more convenient and, hopefully, increase participation in elections. E-voting is seen as a logical extension of electronic applications in commerce, education and government [19].

An e-voting service is commonly expected to satisfy a number of security criteria. However, in an open, large-scale distributed system no individual server can be permanently and completely trusted because it may be compromised by an external attacker or a corrupted insider. If the server is compromised, no security criteria can be satisfied.

A traditional approach to build a secure e-voting service consists in distributing trust to a set of servers, and assigning each of them the responsibility of a specific voting operation, e.g. validation of voters, collection of votes, their counting, or their verification. If enough servers are uncompromised, security requirements can be ensured. Several voting schemes have been conceived according to this approach

* Tel.: +39-50-568-549; fax: +39-50-568-522.
E-mail address: g.dini@iet.unipi.it (G. Dini).

0167-739X/02/$ – see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S0167-739X(02)00109-7

[4,11,14,15,20,23,25,30,34]. Although these systems meet basic requirements for an effective deployment in a large-scale system, the traditional approach presents intrinsic limitations in the level of service availability that can be provided. If any of the servers suffers of a crash failure, the voting operations associated with that server, and thus the whole voting service, become unavailable. During the voting process, even benign failures occurring rarely as well as very short periods of unavailability are unacceptable because they may affect the vote of thousands of voters and alter the overall elections outcome [27].

In an open, large-scale distributed system, voters cannot be trusted either. Voting systems are commonly designed to tolerate malicious voters: unauthorised voters are prevented from voting, and eligible voters can vote at most once. However, the possibility that the behaviour of a voter may deviate from specifications due to benign failures such as crashes should also be taken into account. Consider Sensus [11], for example. If the voting platform from which the voter casts crashes after contacting the validator but before having received the validation certificate, then the voter will be never able to vote, even after recovery. In fact, another execution of validation would be interpreted as an attempt of the voter to vote more than once and thus not allowed.

In this paper, we present an e-voting service that is both available and secure, and that tolerates benign failures of voters. We achieve security and availability by following a replication-based approach. We distribute trust to a set of servers and let them share the responsibility of implementing the voting operations. If enough servers are correct, service availability and security can be ensured despite the presence of faulty servers acting against the voting process, both individually and in collusion with one another or with malicious voters. Furthermore, we achieve tolerance to crashes of voters by allowing the voter validation and vote casting operations to be repeated across crashes and recoveries without violating security requirements.

The proposed e-voting service supports common security requirements: only eligible voters should be able to vote; an eligible voter should not vote more than once; no one should be able to determine how any individual voted. Furthermore, in contrast to e-voting schemes such as [4,11,25,34], the proposed e-voting service guarantees voting accuracy and verifiability without requiring any intervention of voters [9,10,16,33,43]. That is, any given observer can independently verify that no invalid vote has been added to the final tally and no valid vote has been altered or deleted. This has several advantages. The resulting service is more convenient for voters. Furthermore, the task of counting votes is simplified since no cooperation of voters is required. Finally, the overall service security increases since no security control is delegated to voters.

The implementation of the e-voting service is founded on *dissemination quorums*, a quorum construction technique that tolerates server failures ranging from benign to fully arbitrary, and that can be used when a replicated service is a repository of self-verifying information [28]. The implementation of any given voting operation involves a quorum of servers. As a quorum may be small—e.g. composed of $O(\sqrt{n})$ out of n servers—and neither server-to-server nor voter-to-voter interaction is required, the e-voting service may be efficiently deployed in a large-scale system. Forms of replication have been employed also by other e-voting systems [1,23]. However, they limit the use of replication to implement just one voting operation, namely, vote casting. Furthermore, in [23], the proposed degree of replication is insufficient to cope with compromised servers.

The paper is organised as follows. In Section 2 we list the requirements the proposed e-voting service meets. In Section 3, we describe the model of distributed system for which the service is conceived. In Section 4 we briefly introduce dissemination quorums. In Sections 5–7 we describe the e-voting service in two steps. In Section 5 we present the basic version of the service and in Section 6 we argue that the proposed e-voting service satisfies the requirements in Section 2. Then, in Section 7 we present an extension to the e-voting service aimed at tolerating crashes of voters. Finally, in Section 8 we present an early, fully operational prototype of the e-voting service and discuss its performance.

2. Electronic voting requirements

In this section, we introduce the security requirements of the e-voting service. Prior to the beginning

of the voting process, elections organisers specify the list of the *eligible* voters.

- *Eligibility*—Only eligible voters are allowed to vote.
- *Uniqueness*—Every eligible voter can vote only once.
- *Privacy*—No one can know the voting strategy of any given voter.
- *Availability*—A voter eventually succeeds in casting a vote.

In any moment, a voter may decide to abstain and thus cast no ballot. However, no one must be able to exploit that abstention and vote in the place of the voter.

- *Abstention*—If any given voter abstains no one can vote in the place of that voter.

  When the casting period is over, the final election tally is worked out. To be counted in the final tally, a vote must be *valid*, i.e., it must be unaltered and come from an eligible voter.
- *Accuracy*—It is impossible either for an invalid vote to be counted in the final tally, or for a valid vote to be either altered or eliminated.

  If the final tally contains any mistake, any given observer is able to detect it and give any third party, an *arbiter*, enough evidence for correction of the mistake.
- *Verifiability*—Anyone can independently verify that votes have been correctly counted.

  This set of requirements is an obvious desiderata that, no matter which other plausible properties an e-voting service may have, cannot be dispensed with. Most contemporary e-voting services strive for satisfying these requirements [11,25,30,36]. However, it should be noted that, there are further reasonable requirements that could be desirable for an e-voting service. For instance, several systems support the *Recasting* requirement that allows a voter to change his idea and recast before the deadline for casting ballots expires [4,14,25,34]. The requirement that servers cannot learn intermediate results is a further example. A comprehensive discussion of requirements for an e-voting service can be found in [19]. However, we would like to emphasise that the central issue of this paper, the replication-based implementation of an e-voting service, is largely orthogonal to this aspect.

3. System model

We consider a distributed asynchronous system composed of a set of computing *processes*, n of which are *servers* and the remainder of which are *clients*. A process is *correct* if it always follows the specified protocols. A *faulty* process, however, may deviate from its specification in any fashion whatsoever, i.e., "Byzantine" failures are allowed. Faulty processes may act maliciously and deliberately against the service, possibly teaming up and acting in collusion.

All processes communicate through *reliable* point-to-point channels. This means that a correct process receives a message from another correct process if, and only if, the other process sent it. However, asynchrony implies that there is no bound on message transmission times, and on process relative speeds. Therefore, the sender of a message can learn whether the intended receiver has indeed received the message when the sender receives the receiver's acknowledgement of that message.

When processes desire anonymity and untraceability, they communicate through *anonymous* channels [5]. Anonymity means that it is impossible to determine the sender from which a given message comes from. Untraceability means that any two messages from the same sender cannot be linked to each other. Practical solutions to anonymous communication have recently been proposed [17,24,37] as well as solutions to robust anonymous channels [13,21,22].

Our protocols make use of cryptographic algorithms. We denote by $E_k(m)$ the encryption of a given message m with key k. Whether the encryption algorithm is symmetric or not will be clear from the context. We assume that any given process P has a public–private key pair, where K_p is the public key and K_p^{-1} is the corresponding private one. Key K_p is assumed to be known to all processes, whereas the corresponding private key K_p^{-1} is known only to P. We denote by $\sigma_{K_\mathrm{p}^{-1}}(m)$ the digital signature of a message m with P's private key K_p^{-1}. When there is no danger of confusion, we write $\sigma_\mathrm{p}(m)$.

Blinding is a cryptographic technique used when we want processes to digitally sign messages without seeing their contents [6]. Blind signatures are a three-step process. In the first step, the originator of a given message m computes the *blinded form* of the message using a properly randomly chosen number f

called the *blinding factor*. We denote by $b_f(m)$ the result of this computation. In the next step, the signer, say P, signs the blinded form of the message and produces $\sigma_p(b_f(m))$, the blinded *signature* of m. Finally, the originator removes the blinding factor f and retrieves the original message signed by the signer, i.e., $\sigma_p(m)$. Anyone can verify that $\sigma_p(m)$ has been produced by P. However, anyone who does not know the secret blinding factor f, including the signer, cannot associate $b_f(m)$ with either the message m or with the act of signing it, i.e., $\sigma_p(m)$.

For reasons of security and efficiency, it is often preferable to sign a *message digest* of a message rather than the message itself. A message digest function h has the properties that the message digest $h(m)$ for any input m can be computed efficiently, but it is computationally infeasible to find any pair m and m' such that $h(m) = h(m')$, or to compute m given $h(m)$ [29].

We allow for a very strong adversary that can coordinate faulty processes. However, we assume that the adversary (and the faulty processes it controls) are computationally bound so that (with very high probability) it is unable to subvert the cryptographic techniques mentioned above. For example, the adversary cannot produce a valid signature of a non-faulty process, compute the information summarised by a digest from the digest, or find two messages with the same digest. Finally, we assume that cryptographic keys, blinding factors, and other numbers that must be unpredictable and unique are adequately chosen randomly from an adequately large space to prevent random collisions or the disclosure of secrets by cryptanalytic attacks.

4. Dissemination quorum systems

Let us consider a service replicated over a set S of n servers. A *quorum system* $\mathcal{Q} \subseteq 2^S$ is a non-empty set of subsets of S every pair of which intersects, i.e.,

Q-Intersection : $\quad \forall Q_1, Q_2 \in \mathcal{Q} : Q_1 \cap Q_2 \neq \emptyset$

Each $Q \in \mathcal{Q}$ is called a *quorum*.

Quorum systems have been traditionally used to build replicated services where servers are subject to benign failures such as crash failures. Informally, quorum systems increase availability and performance because a quorum can act on account of the whole system. Property Q-Intersection guarantees that the system remains consistent.

In a large-scale distributed system where failures may represent malicious acts of an adversary, Property Q-Intersection alone is not sufficient to guarantee consistency because it cannot prevent the intersection of any two quorums from containing only faulty servers. Recently, Malkhi and Reiter [28] have proposed the *Byzantine quorum systems*, which extend the notion of quorum to replicated services subject to arbitrary (Byzantine) failures. *Dissemination quorums* are a variation of Byzantine quorums that can be used when the replicated service is a repository of *self-verifying* information. In this kind of service, compromised servers may fail to propagate self-verifying information received from a (correct) client but cannot alter it without being detected.

In a large-scale distributed system in the presence of malicious attacks, servers may not fail independently and have correlated probabilities of being captured. We use the notion of *fail-prone* system to capture this non-uniform failure scenario [28]. Informally, a fail-prone system describes the failure scenario to which servers are subject by specifying the subsets of servers that may be simultaneously faulty. More formally, a fail-prone system $\mathcal{F} \subseteq 2^S$ is a non-empty set of subsets of S, none of which is contained in another, such that some $F \in \mathcal{F}$ contains all the faulty servers. A *dissemination quorum* (henceforth quorum) system $Q \subseteq 2^S$ for a fail-prone system $\mathcal{F}$ is a quorum system that satisfies the following properties:

Q-Consistency :

$$\forall Q_1, Q_2 \in \mathcal{Q} \Leftrightarrow \forall F \in \mathcal{F} : Q_1 \cap Q_2 \nsubseteq F$$

Q-Availability : $\quad \forall F \in \mathcal{F}, \exists Q \in \mathcal{Q} : F \cap Q = \emptyset$

Property Q-Consistency ensures that the intersection of any two quorums contains at least one correct server. This server guarantees that the self-verifying information received from any given client can be correctly disseminated. Property Q-Availability ensures that a quorum of correct servers always exists. In an asynchronous distributed system subject to failures, a client can learn whether an operation has completed with the correct servers of some quorum when the client has obtained responses from a full

quorum. Property Q-Availability guarantees that such a quorum always exists.

As an example of dissemination quorum system, let us consider the customary "threshold" assumption, according to which up to f servers can fail, with $n > 3f$. The fail-prone system $\mathcal{F}$ can be thus defined as $\mathcal{F} = \{F | F \subseteq S \wedge |F| = f\}$ and a dissemination quorum system $\mathcal{Q}$ for $\mathcal{F}$ can be defined as $\mathcal{Q} = \{Q | Q \subseteq S \wedge |Q| = \lceil (n + f + 1)/2 \rceil\}$. For $n = 3f + 1$, $|Q| = 2f + 1$. Note that the threshold assumption is based upon the hypothesis that servers fail independently. Quorum systems appropriate to situations where failures cannot be considered independent have been proposed by Malkhi and Reiter [28].

5. The e-voting service

The e-voting service supports a voting process that is structured in two phases: *Voting* and *Tallying*. In the voting phase, voters cast their votes. This operation consists of two steps. First of all, any given voter must be validated. This action consists in ascertaining whether a given voter is eligible, and, if this is indeed the case, issuing the voter with voting credentials. By means of these credentials and the chosen voting strategy the voter can later build a valid ballot and send it to the e-voting service.

When the voting phase is over, the tallying phase begins. In this phase, a principal called *tallier* retrieves ballots from the e-voting service and works out the *final tally*. Any given principal, an *observer*, can verify the accuracy of the final tally. This activity includes the observer building its own version of the tally and comparing it with the final tally. If the final tally contains any mistake, the observer is able to detect it and give any third party, an *arbiter*, enough evidence for correction of the mistake.

5.1. Votes, voting certificates, and ballots

To be counted in the final tally, a vote must be *valid*, i.e., it must be unaltered and come from an eligible voter. Checking whether a vote is valid is complicated by the fact that the identity of the voter must not be disclosed. The goal of the validation operation is just to issue credentials asserting that a given vote is unaltered and comes from an eligible voter without disclosing the identity of the voter. These credentials take the form of a "blinded" *voting certificate*. In the following we shall denote by C_v the voting certificate for vote v. The certificate structure will be discussed in the next sections.

A *ballot* $\beta = \langle \alpha, v, C_v \rangle$ is a triple composed of a voter's voting strategy v, the voting certificate C_v, and a random number α used to tag the vote. The ballot is a *self-verifiable* structure that allows anyone to verify that the voting strategy v is valid without knowing the identity of the voter.

Ballots are sent to e-voting service anonymously. Therefore, as a ballot contains no information linking a voter to the chosen voting strategy, voter privacy is ensured. To fulfil the uniqueness requirement, the e-voting service ensures that a voter can obtain at most one voting certificate. This implies that a voter cannot change his vote after he got the voting certificate. In fact, to do that, the voter should obtain another certificate, but this is not allowed. Finally, the e-voting service ensures that, if a valid vote has been collected, then it will be counted in the final tally. Such a ballot cannot be altered or eliminated without being detected by any given observer. How the e-voting service fulfils these guarantees is the argument of the next sections.

5.2. The replicated service

With reference to Fig. 1, the *e-voting service* is implemented as a replicated service over a set $S = \{S_1, \ldots, S_n\}$ of n *voting servers*. We denote by K_{s_i} and $K_{s_i}^{-1}$ the public and private key, respectively, of server S_i. We assume that a given fail-prone system $\mathcal{F}$ specifies the possible failure scenario to which servers are subject and that $\mathcal{Q}$ specifies a dissemination quorum system for $\mathcal{F}$. *Voting clients* represent the voting platforms from which voters interact with the service. Clients perform the protocols implementing the voting phase on behalf of voters.

In the voting phase, the interactions with the e-voting service take place according to the *Validation* and *Casting Protocols*. The former protocol allows an eligible voter to obtain a voting certificate for the chosen voting strategy, whereas the latter protocol allows the voter to actually cast a vote. These protocols will be described in Sections 5.3.1 and 5.3.2. Intuitively, they work as follows. In the validation

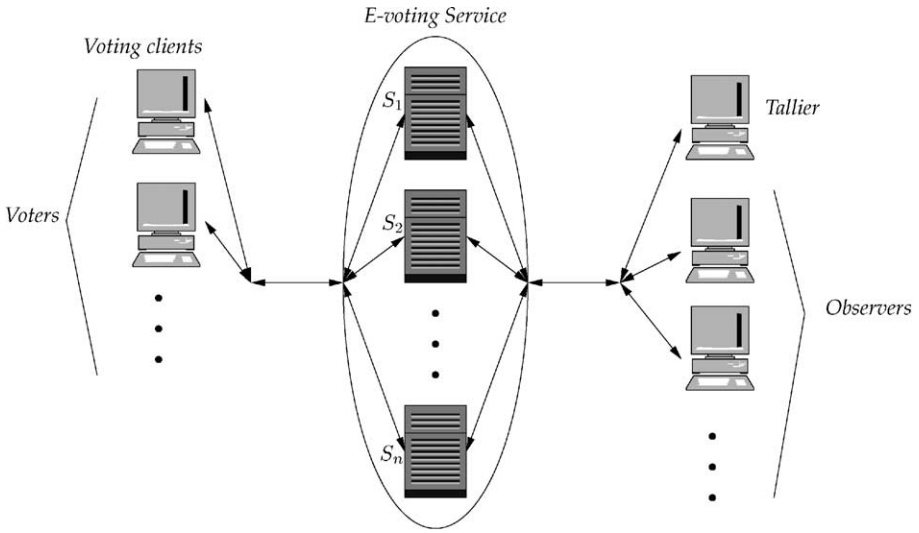

Fig. 1. The figure shows the replicated structure of the e-voting service.

protocol, a voter sends both identification information and the blinded form of the chosen voting strategy to every server in S until every server in a quorum signs it. It follows that the voting certificate C_v for vote v consists of the set of (digital) blind signatures of v by every server in a quorum. In the casting protocol, the voter anonymously sends the ballot to all servers in S until every server in a quorum collects it. We say that a ballot is *properly collected* by the e-voting service if it has been collected by every server in a quorum.

Ballots disseminated by voters to the e-voting service during the voting phase are gathered by a *tallier* and one or more *observers* during the tallying phase. These principals are responsible for working out the final tally (Section 5.4.1) and verifying its accuracy (Section 5.4.2), respectively. They interact with the e-voting service according to the *tallying protocol*. Intuitively, this protocol gathers ballots from every server in a quorum. Every properly collected ballot is thus received from at least one correct server. Invalid ballots are discarded, whereas the remaining valid ones are inserted in the final tally. The tallier carries out the tallying protocol to work out the final tally. Any given observer carries out the protocol to work out its own version of the tally, compare it with the version produced by the tallier, and ascertain that no valid ballot has been eliminated.

5.2.1. Data structures of servers

In order to support the voting phases, any given server $S_i \in S$ is equipped with stable storage [26] where it maintains the following data structures:

- The *Electoral Roll*, a trusted read-only data structure received from the elections organisers which specifies the set of eligible voters. Conceptually, the electoral roll can be considered as a table with one entry for every eligible voter which specifies the public key of that voter.
- The *Validation Table*, VT_i, which specifies the set of voters whose votes have been signed by server S_i. Conceptually, VT_i has one entry for each eligible voter, with $\mathrm{VT}_i[U]$ denoting the one reserved for voter U. Table entry $\mathrm{VT}_i[U]$ contains the value *true* if S_i has (blindly) signed the vote of U, and *false* otherwise. Initially, every table entry contains the *false* value.
- The *Collection Set*, CS_i, which contains the ballots collected by S_i. Initially, CS_i is empty.

5.3. The voting phase

Let U be an eligible voter (K_u^{-1}, K_u) be his key pair, and v his chosen voting strategy. The voter performs the voting phase from any given client. In the remainder of this section, and throughout Sections 5.4

and 6, we assume that voting clients behave correctly. We shall relax this assumption in Section 7, where we consider the possibility that clients are subject to crash failures. Furthermore, we use the terms client and voter interchangeably. When the difference is relevant, we shall highlight it.

5.3.1. The validation protocol

Initially, the voter randomly chooses an identification tag α and calculates the blinded form of $h(\alpha\|\nu)$, where $\|$ denotes the concatenation operator. As, in general, the blinding factor is required to satisfy certain properties with respect to the signature function,[1] then n blinding factors are chosen, one for each server, n blinded forms are calculated, one for each blinding factor. We denote by $\mathbf{f}$ the array of n blinding factors such that $\mathbf{f}[i]$ contains the blinding factor for server S_i. Furthermore, we denote by $\mathbf{b}$ the array of n blinded forms such that $\mathbf{b}[i]$ contains the blinded form of $h(\alpha\|\nu)$ by means of the blinding factor $\mathbf{f}[i]$, i.e., $\mathbf{b}[i] = b_{\mathbf{f}[i]}(h(\alpha||\nu))$.

Then, the voter performs the validation protocol, that consists of the following actions:

1. The voter signs $\mathbf{b}[i]$ and sends S_i a *validation request* message $\langle U, \mathbf{b}[i], \sigma_u(\mathbf{b}[i])\rangle$, $\forall i$, $1 \leq i \leq n$.
2. Upon receiving the validation request message, any given server S_i accesses the electoral roll to ascertain whether U is eligible and verify $\sigma_u(\mathbf{b}[i])$. If any of these checks fails, S_i drops the request message. Otherwise, S_i accesses $\mathrm{VT}_i[U]$ and takes alternative actions according to the contents of this entry:
   (a) If $\mathrm{VT}_i[U] = \mathit{false}$, then S_i signs $\mathbf{b}[i]$, sets $\mathrm{VT}_i[U]$ to *true*, and returns a *validation response* message $\langle S_i, \sigma_{s_i}(\mathbf{b}[i])\rangle$ to U.
   (b) If $\mathrm{VT}_i[U] = \mathit{true}$, then S_i returns a validation response message $\langle S_i, \perp\rangle$ to U, where $\perp$ denotes the null value.
3. Upon receiving a response message from a server S_i, the voter retrieves the signature $\sigma_{s_i}(h(\alpha||\nu))$, by removing $\mathbf{f}[i]$ from $\sigma_{s_i}(\mathbf{b}[i])$ (the message is dropped if it contains $\perp$), and verifies the obtained signature (the message is dropped, if the verification fails). Finally, the voter inserts the signature in the voting certificate C_ν and S_i in the set of responding servers. If this set forms a quorum, the voter terminates the registration protocol, and deletes $\mathbf{f}$ and $\mathbf{b}$.

According to step 3, the voting certificate C_ν released by the e-voting service for vote ν turns out to be the set of signatures $C_\nu = \{\sigma_{s_i}(h(\alpha||\nu))\}_{S_i \in Q}$, where $Q \in \mathcal{Q}$. The validity of a voting certificate should be limited to the current election only. There are several ways to achieve this goal. One way is to insert an expiration time in the voting certificate. An alternative way is to insert an election identifier in the certificate. The certificate will be valid only within the election specified by that identifier. For the sake of simplicity, but without loss of generality, we have chosen an alternative solution. We limit the validity of a certificate to one election by limiting the validity of the servers keys ($K_{s_i}^{-1}$, K_{s_i}, $\forall i$, $1 \leq i \leq n$) to that election. Every given server will be given a new pair of keys upon every new election.

5.3.2. The casting protocol

After validation, the voter builds a ballot $\beta = \langle \alpha, \nu, C_\nu\rangle$ and casts it according to the following actions:

1. The voter anonymously sends S_i a *ballot message* $\langle E_k(\beta), E_{K_{s_i}}(k)\rangle$, $\forall i$, $1 \leq i \leq n$, where k is a per-ballot key to be used in a symmetric encryption algorithm.
2. Upon receiving the ballot message, any given server S_i retrieves k and thus β, inserts β into CS_i, and returns a *ballot acknowledgement* message $\langle S_i, \sigma_{s_i}(h(k))\rangle$.
3. Upon receiving a ballot acknowledgement message, the voter verifies $\sigma_{s_i}(h(k))$ (if the check fails, the message is dropped), and inserts S_i in the set of acknowledging servers. If this set forms a dissemination quorum, the voter terminates the casting protocol and deletes quantities ν, α, β and C_ν.

According to step 2, any given server does not perform any check on the received ballot but stores it. The server checks neither whether the vote is valid, nor whether any vote tagged with the same identifier is already present in the local Collection Set. These checks would require cryptographic operations and accesses to secondary storage. Therefore, by avoiding them, we aim at improving both protocol throughput and latency. However, to enforce security requirements, the

[1] For instance, in a blinded signature scheme based on RSA [40], the blinding factor must be prime to the modulus.

presence of invalid and repeated votes will be checked later, off-line, in the tallying phase.

5.4. The tallying phase

5.4.1. Building the final tally

In the tallying phase, the tallier works out the final tally according to the *tallying protocol*, which consists of the following actions:

1. The tallier sends server S_i a *tallying request* message, $\forall i$, $1 \leq i \leq n$.
2. Upon receiving this request message, any given server S_i sends the tallier a *tallying response* message containing its Collection Set CS_i.
3. Upon receiving Collection Sets from every server in a quorum, the tallier works out the final tally as follows:
   (a) Initially, the tallier groups all the received ballots by identification tag. We denote by B_α the set of ballots tagged with α.
   (b) Then, for every set of ballots B_α, the tallier picks any valid ballot β_α contained therein, and inserts it in the final tally.

5.4.2. Verifying the tally accuracy

Any given external observer can verify the accuracy of the final tally and produce enough evidence of mistakes before the arbiter. The tally verification activity consists of the following actions:

1. The observer checks whether invalid ballots are present in the final tally. Invalid ballots are produced before the arbiter, which requires their removal from the final tally.
2. The observer checks whether repeated ballots—two or more ballots with the same identification tag—are present the final tally. Repeated ballots are produced before the arbiter, which requires that all but one of them are removed from the final tally.
3. The observer checks whether any valid ballot has been omitted. To this purpose, the observer works out *its own* version of the tally by performing the tallying protocol, compares it with the final tally produced by the tallier, and ascertains that any given ballot that is present in its own tally is also present in the final one. Omitted ballots are produced before the arbiter, which requires their insertion in the final tally.

6. Security analysis

In this section we argue that our protocols meet all of the requirements mentioned in Section 2. First, we discuss some of the functional properties of the protocols. Then, we argue that these properties guarantee that the protocols satisfies the requirements. Our arguments are informal, and are not intended to constitute a rigorous proof of security.

6.1. Functional properties of the validation protocol

Proposition 1. *Only eligible voters can obtain a voting certificate.*

Proof. In order to obtain a voting certificate, a voter needs signatures on the chosen vote from every server in a quorum. As signatures cannot be forged, the voter can obtain them through an execution of the validation protocol. Therefore, the rest of the proof consists in showing that only eligible voters can successfully perform that protocol. The proof is by contradiction. Suppose a not eligible voter U has successfully performed the validation protocol. This implies that every server in some dissemination quorum Q has signed the vote of U. However, this is impossible. Let $C \subseteq Q$ be the set of correct servers in Q. By Property Q-Consistency, $C \not\equiv \emptyset$, $\forall Q \in \mathcal{Q}$. As signatures cannot be forged, U cannot impersonate an eligible voter. Therefore, any correct server $S_i \in C$ detects U ineligibility at step 2 of the validation protocol and, consequently, does not (blindly) sign the vote of U. □

In practice, Proposition 1 means that any (malicious) principal that is not an eligible voter cannot validate itself, even colluding with faulty (compromised) servers. The principal fails to collect a voting certificate because it fails to collect signatures from correct servers.

Proposition 2. *An eligible voter can obtain at most one voting certificate.*

Proof. The proof is by contradiction. Suppose that an eligible voter U has obtained two valid certificates, namely C_ν and $C_{\nu'}$, for vote ν and ν', respectively. Also, suppose that ν has been signed by every server in a quorum Q, whereas ν' has been signed by every

server in a quorum Q'. In general, $Q \neq Q'$. It follows that both votes have been signed by every server in $Q \cap Q'$. However, this is impossible. Let C be the set of correct servers in $Q \cap Q'$. Property Q-Consistency ensures that $C \neq \emptyset, \forall Q,\ Q' \in \mathcal{Q}$. According to steps 2(a) and (b) of the validation protocol, every server in C signs either v or v', but not both. □

Proposition 2 implies that an eligible but malicious voter cannot obtain more than one voting certificate, even colluding with corrupt servers. Upon attempting to obtain the second certificate, the voter fails to receive signatures from enough correct servers. The proposition also implies that an eligible voter cannot change his vote after he got a voting certificate. In fact, to do that, an additional certificate would be needed.

Proposition 3. *It is not possible to link voters to their voting certificates.*

Proof. The proof descends directly from the properties of the blind signature scheme employed in the validation protocol. □

Proposition 4. *An eligible voter eventually completes the validation protocol.*

Proof. The voter can complete the validation protocol as soon as signatures on the chosen vote have been received from every server in a quorum. As communication is reliable (and the client that performs the protocol on the voter's behalf is correct), then the voter receives signatures from every correct server. Property Q-Availability ensures that these servers form a quorum. □

6.2. Functional properties of the casting protocol

Proposition 5. *It is not possible to link voters to their ballots.*

Proof. The proof descends from the following facts: voters interact with servers through anonymous channels; and, according to Proposition 3, ballots contain no information linking voters to their votes. □

Proposition 6. *A voter eventually completes the casting protocol.*

Proof. A voter can complete the casting protocol as soon as his ballot has been collected by every server in a quorum. As communication is reliable (and the client that performs the protocol on the voter's behalf is correct), then every correct server eventually collects and acknowledges the ballot. As a quorum of correct servers always exists (Property Q-Availability), then the ballot is eventually properly collected, and thus the casting protocol eventually completes successfully. □

6.3. Functional properties of the tallying protocol

Proposition 7. *An invalid ballot is never inserted in the final tally.*

Proof. Invalid ballots are discarded at step 3(b) of the tallying protocol. □

Proposition 8. *No two ballots tagged with the same identifier can be present in the final tally.*

Proof. For any given identifier, only one ballot tagged with that identifier is inserted in the final tally (step 3(b) of the tallying protocol). □

Proposition 9. *Every properly collected valid ballot is inserted in the tally.*

Proof. If a ballot is properly collected, then there exists a quorum $Q \in \mathcal{Q}$ such that every server in Q has collected the ballot. Let $Q' \in \mathcal{Q}$ be a quorum, in general $Q' \neq Q$, from which ballots are received at step 3 of the tallying protocol. Due to Property Q-Consistency, the ballot is received from at least one correct server in $Q \cap Q'$. If the ballot is valid, it is picked at step 3(b) and inserted in the final tally. □

6.4. Functional properties of tally verification

Proposition 10. *It is impossible that an invalid ballot is added to the final tally without being detected by any given external observer.*

Proof. An observer detects the presence of any invalid ballot at step 1 of tally verification. □

Proposition 11. *It is impossible that two or more valid ballots tagged with the same identification tag are present in the final tally without being detected by any given external observer.*

Proof. An observer detects the presence of any two ballots tagged with the same identification tag at step 2 of tally verification. □

Proposition 12. *It is impossible that a valid ballot is altered without being detected by any given external observer.*

Proof. If a valid vote is altered then it becomes invalid, and thus is detected by any given external observer (Proposition 10). □

Proposition 13. *It is impossible that a properly collected valid ballot is eliminated without being detected by any given external observer.*

Proof. An observer detects that a properly collected valid ballot has been removed at step 3 of tally verification. □

6.5. Correctness proof of the voting protocol

Proposition 14. *The voting protocol satisfies requirement eligibility.*

Proof. By Propositions 7 and 10, only valid votes are counted in the final tally. Therefore, the proof consists in showing that only eligible voters can build valid ballots. In order to build a valid ballot it is necessary a voting certificate. By Proposition 1, only an eligible voter can obtain a valid certificate. □

Proposition 15. *The voting protocol satisfies requirement uniqueness.*

Proof. The proof descends directly from the following considerations. First, Proposition 2 ensures that any given eligible voter can obtain at most one voting certificate. Thus the voter can cast only the vote related to it. Furthermore, Propositions 8 and 11 ensure that no duplicate of that ballot can appear in the final tally. □

Proposition 16. *The voting protocol satisfies requirement privacy.*

Proof. The proof descends directly from Propositions 3 and 5. □

Proposition 17. *The voting protocol satisfies requirement availability.*

Proof. The proof descends directly from Propositions 4 and 6. □

Proposition 18. *The voting protocol satisfies requirement abstention.*

Proof. As keys are kept secret, no principal can impersonate a given voter in the validation protocol. Therefore, no one can obtain a voting certificate, and thus vote, in the place of a voter that abstains. □

Proposition 19. *The voting protocol satisfies requirement accuracy.*

Proof. If the tallier is correct, then the proof descends directly from Propositions 7–9. In contrast, if the tallier is faulty and produces a wrong tally, Propositions 10–13 ensure that an external observer can detect the mistakes and obtain their correction. □

Proposition 20. *The voting protocol satisfies requirement verifiability.*

Proof. The proof descends directly from Propositions 10–13. □

6.6. Supporting non-coercibility

A voting service meets the *non-coercibility* requirement if voters cannot prove to any third party how they voted. If this requirement is met, vote-buying is impossible and a voter cannot be compelled to a cast a particular vote [2,18,35,38,42].

We argue that if voters are physically separated from outsiders, the proposed e-voting service meets the non-coercibility requirement too. The required separation can be achieved by placing voting clients into *voting booths*, for example. The proof descends from the following considerations. With

reference to a given voter U, a correct client discloses neither the voter's secrets (e.g., K_u^{-1}, ν, $\mathbf{f}$), nor the link between the voter and the identification tag α. Under these hypotheses, Propositions 3 and 5 ensure that it is not possible to link voters to their votes. As voters are physically separated from outsiders, they cannot see how voters actually vote.

7. On faulty clients

Our e-voting service meets all the requirements mentioned in Section 2 provided that clients are correct. In this section, we extend our treatment to address faulty clients in addition to faulty servers. To do that we classify faulty clients in either *compromised*, those that suffer a "truly Byzantine failure", or *honest*, those that suffer a crash failure.

In general, security requirements cannot be ensured in the presence of compromised clients. For instance, a compromised client can disclose, as well as modify, a voter's voting strategy, and even vote in the voter's place [41]. However, the proposed protocols limit damages that a compromised client can do to the overall voting process. A compromised client cannot cause any damage to properly collected votes. Intuitively, this is guaranteed by Propositions 9, 12 and 13, which ensure that, once a vote has been properly collected, it cannot be modified or eliminated, even colluding with faulty servers.

In the remainder of this section, we assume honest clients. It is worthwhile to note that uncompromised clients can be assumed under the realistic constraint that they are precinct voting platforms [19]. This means that voting clients are placed in public sites, and the election officials exert control on the hardware and the software of the voting platform as well as on the physical voting environment.

In its present form, the e-voting service is not able to tolerate honest clients. That is, a client crash may cause a voter to become unable to vote even after recovery. In the rest of this section, we discuss problems connected to crashes of clients during the validation and casting protocols, and present extensions to these protocols aimed at allowing a voter to vote across crashes and recoveries. Once again, we shall refer to user U that performs the voting phase from a given client and use the terms client and voter interchangeably but when the difference is relevant.

Let us consider the casting protocol first. Assume that a voter suffers a crash failure during an execution of that protocol just before the vote has been correctly propagated to every server in a quorum. As a consequence of the failure, quantities ν, α and C_α get lost. As Property 2 ensures that a voter can obtain at most one certificate, then the voter cannot obtain another certificate, and thus vote, even after recovery. A way to solve this problem is to equip clients with stable storage where to store quantities ν, α and C_ν. Upon crashing, all data are lost, except for the part that was recorded in stable storage. During recovery, the client reads quantities ν, α and C_ν from stable storage, repeats the voting protocol with these quantities. It should be noted that the e-voting service already supports a limited form of recasting that makes this solution feasible without compromising the accuracy of the final tally. In fact, although the e-voting service prevents a voter from voting twice or more times, however, the service permits the repeated casting of the *same* vote. Propositions 8 and 11 guarantee that no repetition of that vote will be present in the final tally, and thus that requirement uniqueness is met. From these considerations it follows that:

Proposition 21. *If a client remains correct for sufficiently long time, then a voter eventually completes the casting protocol.*

It should be noted that "sufficiently long time" must be taken as the time necessary to perform the casting protocol. In an asynchronous distributed system, no upper bound to this time can be specified.

Simply equipping clients with stable storage is not sufficient to allow them to perform the validation protocol across crashes and recoveries. Actually, a crash failure during an execution of that protocol may prevent a voter from ever obtaining a voting certificate, and thus vote, even after recovery. The reason for this is the following. Consider a client that suffers a crash failure during the execution of the validation protocol. Assume also that the crash failure occurs when the client has already contacted a given quorum Q but has not yet received all the corresponding response messages. Finally, assume that, after recovery, the execution of the validation protocol is repeated

and, as part of this execution, a new quorum Q' is contacted, where, in general, $Q \not\equiv Q'$. For Property Q-Consistency, $Q \cap Q'$ is not empty and contains at least one correct server, namely S_i. As S_i set $\mathrm{VT}_i[U]$ to value *true* during the first execution of the validation protocol (before the crash failure), the server returns a validation response message containing $\perp$ (step 2(b) of the validation protocol). Consequently, the voter will be never able to collect enough signatures to form a voting certificate.

A way to tolerate crash failures during the validation phase is to give an eligible voter the opportunity to repeat the validation protocol after recovery. However, as the voter submits blinded material to servers, such a repetition may cause a security leak. In fact, an eligible but malicious voter, pretending to suffer several crash failures, could execute the validation protocol repeatedly, each time with a different voting strategy. As servers cannot know what they are signing, the voter would succeed in obtaining several voting certificates. Consequently, the voter could vote several times, so violating the uniqueness requirement. A solution to this problem is to allow the voter to repeat the validation protocol with the *same* blinded material. This requires changes at both the voter and server side, as follows.

Prior to the beginning of the validation protocol, the quantities **f** and **b** are stored in stable storage, in addition to ν, α. These quantities will be used in any repetition of the validation protocol that follows a recovery from a crash failure.

On the server side, the validation table structure is modified. Any given table entry is now composed of two fields: the *blinded form field*, *f*, and the *blinded signature field*, *s*. With reference to entry $\mathrm{VT}_i[U]$, field *f* specifies the blinded form of the voting strategy ν of voter *U*, i.e., $\mathbf{b}[i] = b_{\mathbf{f}[i]}(h(\nu||\alpha))$; and field *s* specifies the corresponding (blinded) signature $\sigma_{s_i}(\mathbf{b}[i])$ with the server private key. Initially, both fields contain the null value $\perp$. Fig. 2 shows an extended version of the validation protocol that takes into account the new structure of the validation table. While steps 1 and 3 of the extended protocol are the same as the original protocol, step 2 has been modified to take advantage of the new table structure.

With reference to Fig. 2, assume that voter *U* carries out the validation protocol. The first time any given server S_i receives the validation request message, $\mathrm{VT}_i[U]$ contains $\langle \perp, \perp \rangle$ and therefore the server takes step 2(a). Assume now the voter suffers a crash failure. After recovery, the voter will repeat the validation protocol with the quantities retrieved from stable storage. It follows that S_i will receive a validation request message conveying quantity $\mathbf{b}[i]$ which is already present in the *f* field of $\mathrm{VT}_i[U]$. Thus S_i will take step 2(b). Finally, suppose the server receives a (malicious) validation request message conveying a quantity $b' \neq \mathbf{b}[i]$. As this quantity is different from

1. Initially, the voter signs $\mathbf{b}[i]$ and sends S_i a *validation request* message $\langle U, \mathbf{b}[i], \sigma_u(\mathbf{b}[i])\rangle, \forall i, 1 \leq i \leq n$.
2. Upon receiving the validation request message, any given server S_i accesses the electoral roll to ascertain whether U is eligible and verify $\sigma_u(\mathbf{b}[i])$. If any of these checks fails, S_i drops the request message. Otherwise, S_i accesses $VT_i[U]$ and takes alternative actions according to the contents of this entry:
   (a) If $VT_i[U] = \langle \perp, \perp \rangle$, then S_i signs $\mathbf{b}[i]$, stores the pair $\langle \mathbf{b}[i], \sigma_{s_i}(\mathbf{b}[i])\rangle$ into $VT_i[U]$, and returns a validation response message $\langle S_i, \sigma_{s_i}(\mathbf{b}[i])\rangle$ to U.
   (b) If $(VT_i[U] \neq \langle \perp, \perp \rangle) \wedge (VT_i[U].f = \mathbf{b}[i])$, then S_i returns U a response message $\langle S_i, VT_i[U].s\rangle$.
   (c) If $(VT_i[U] \neq \langle \perp, \perp \rangle) \wedge (VT_i[U].f \neq \mathbf{b}[i])$, then S_i returns U a response message $\langle S_i, \perp \rangle$.
3. Upon receiving a response message from a server S_i, the voter retrieves the signature $\sigma_{s_i}(h(\alpha||\nu))$, by removing $\mathbf{f}[i]$ from $\sigma_{s_i}(\mathbf{b}[i])$ (the message is dropped if it contains $\perp$), and verifies the obtained signature (the message is dropped, if the verification fails). Finally, the voter inserts the signature in the voting certificate C_ν and S_i in the set of responding servers. If this set forms a quorum, the voter stores C_ν in stable storage, deletes **f** and **b**, and terminates the Validation Protocol,

Fig. 2. The extended validation protocol. We use the "dot" notation to specify fields of a table entry, and $\langle f, s\rangle$ to denote the tuple contained therein.

quantity stored in the *f* field, S_i takes step 2(c). So doing S_i prevents malicious validations of two or more voting strategies for the same voter. From these considerations it follows that:

Proposition 22. *If a client remains correct for sufficiently long time, then an eligible voter eventually completes the validation protocol.*

Similarly to the casting protocol, "sufficiently long time" must be taken as the time necessary to perform the validation protocol. In an asynchronous distributed system, no upper bound to this time can be specified.

Propositions 21 and 22 imply that the extended e-voting service satisfies a weaker availability requirement than specified in Section 2:

Weak availability—If a client is correct for sufficiently long time, then a voter eventually succeeds in voting.

In practice, this means that, in order to vote, an honest client is not required anymore to be uninterruptedly correct for the joint execution of the validation and casting protocols. In contrast, a client may crash and recover, even repeatedly, so long as it remains correct for sufficiently long time.

8. Prototype

We have implemented a fully operational research prototype of our e-voting service using the protocols in Sections 5.2 and 7, in an effort to understand the factors that limit its performance. Our implementation used the Java programming language, and employed the Cryptix cryptography toolkit [12] for the basic cryptographic operations. In particular, the implementation used the DES [32] algorithm for symmetric encryption, the MD5 [39] algorithm to digest a message, and RSA [40], with 1024 moduli, for asymmetric encryption, digital signatures and blind signatures.

Table 1 and Fig. 3 show the approximate latency of the validation, casting, and tallying Protocols in the case of no failures. Table 2 shows the approximate throughput of the validation and casting protocols in the same case. Tests were performed on a cluster of personal computers interconnected by a 100Mb/s Ethernet. Every personal computer was based on a Pentium II 350 MHz with 512KB cache, 128MB of RAM, and 3.2GB of disk. From the software point of view, every personal computer ran the Windows NT Client 4.0 operating system. Personal computers that acted as voting servers used the MySQL [31] relational database management system to store the validation table and the Collection Set. There were four voting servers, which is the minimum number of servers required to tolerate the failure of one voting server, under the assumption of independent failures (Section 4).

Table 1
Latency of the validation and casting protocols averaged over 20 runs of 100 requests each[a]

Protocol	Latency (ms)	S.D. (%)
Validation	590	8
Casting	218.51	12.3

[a] The S.D. is expressed as a percentage of the corresponding average latency.

With reference to Table 1, latency labelled "Validation" was measured as the elapsed time between sending a validation request message to all servers and receiving a validation response message from every server in a quorum. This latency includes the latency of waiting for validation response messages from every server in a quorum—about 310 ms—and the latency of unblinding the contents of these messages—about 170 ms. The remaining 110 ms are ascribed to verification of blinded signatures and other computational overheads.

Latency labelled "Casting" was measured as the elapsed time between sending a ballot message to all servers and receiving a ballot acknowledgement message from every server in a quorum. This latency includes the latency of waiting for ballot acknowledgement messages from every server in a quorum—about 210.6 ms—and the latency of verifying signatures in these messages—about 8 ms.

Table 2
Throughput of the validation and casting protocols, averaged over 20 runs of 100 requests each[a]

Protocol	Throughput	S.D. (%)
Validation	3.28	1
Casting	4.17	2

[a] The S.D. is expressed as a percentage of the corresponding average latency.

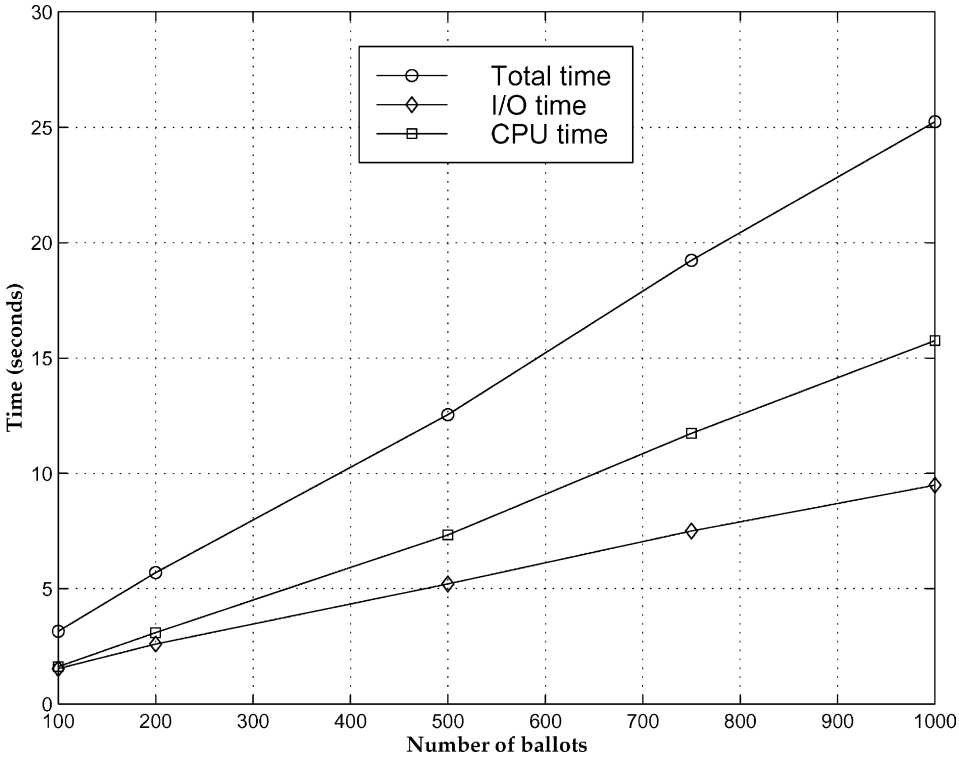

Fig. 3. Latency of the tallying protocol versus the number of ballots. Data points are the average value over 20 runs. Standard deviations were less than 7%, except for the I/O time for 200 ballots, which was 11.6%.

Latencies shown in Table 1 do not include the latency of preparing both the validation request message and the ballot messages. Preparation of these messages requires public key operations. In particular, preparation of validation request messages requires a signature for every server, whereas preparation of ballot messages requires a public key encryption of the per-ballot symmetric key for every server.[2] Preparation of these messages is part of an *initialisation phase* that precedes the actual execution of the protocols. In addition to message preparation, the initialisation phase includes the initialisation of the random numbers generator, and the random choice of both the blind factors and the per-ballot symmetric key. The initialisation phase takes about 5.5 s, five of which are mostly spent for the initialisation of the random number generator.

[2] The DES encryption of the ballot results largely negligible with respect to the other public key computations.

It should be noted that the latencies shown in Table 1 represent an underestimation because they do not include certain network and service overheads. First, we have carried out measurements so that no queueing at the servers could take place. Furthermore, measurements were carried on a fast local area network, through the TCP/IP reliable transport protocol, whose latencies resulted mostly negligible with respect to the computation latencies. Consequently, the shown latencies neither include queueing time at servers, nor the overhead due to anonymous communication over a wide area network [17,37]. In a large-scale distributed system, these quantities contribute to increase the overall latencies of the service.

Fig. 3 shows the latency of the tallying protocol that grows with the number of ballots cast. This latency was measured as the elapsed time at the tallier/observer between sending a tallying request message to all servers and working out a tally. This time

has been split into two components, "I/O time" and "CPU time". The I/O time includes the latency of the servers reading ballots from the Collection Set and sending them (in parallel) to the tallier/observer, and the latency of this principal receiving those ballots. The CPU time includes the latency of the tallier grouping those ballots, verifying them and selecting valid ones.

Table 2 shows the throughput of the validation and casting protocols. Each throughput value was measured as the service response rate when queues at the servers were not empty. In order to understand the factors that influence throughput we have measured the average service time of any given server in the validation and casting protocols. In the validation protocol, the average service time of any given server amounts to about 299 ms and is split into 200 ms of computation time, spent mainly performing the signature of blinded material, and 99 ms of I/O time, spent interacting with the MySQL DBMS. It follows that the e-voting service is able to process at most five validations per second. The service time of the casting protocol amounts to 210.6 ms and is split into 131.8 ms of computation time, mainly spent performing RSA decryption and digital signature, and 78.8 ms of I/O time, spent interacting with the DBMS. It follows that the e-voting service is able to process at most 7.6 casting operations per second. From these considerations, it follows that the service might benefit from a multi-threaded implementation aimed at masking the latencies introduced by the DBMS. In fact, we implemented a new, multi-threaded version of the voting server, which, as expected, produced an improved throughput of 4.32 validations per second, for an increase of about 37%, and 7.15 casting operations per second, for an increase of about 72%.

Cryptographic operations are a factor that strongly influences the performance of our voting protocols. These performance numbers are thus very sensitive to choices of cryptographic algorithms and key lengths. Moreover, they should improve substantially if more powerful server machines or special-purpose hardware to perform RSA were used. Finally, given the computation intensive nature of these operations, higher performance can be achieved by implementing the service in a compiled programming language such as C or C++.

9. Conclusion

We have presented a replicated e-voting service for a large-scale distributed system such as the Internet. Provided that enough servers are correct, the service ensures both security and availability despite the presence of faulty servers acting against the voting process, even in collusion with one another or with malicious voters. The service allows eligible voters that suffers a crash failure to successfully vote after recovery. Furthermore, the replicated structure of the service makes it possible to increase the security of the e-voting process by supporting verifiability of the voting outcome without requiring any intervention of voters. This simplifies the voting scheme and makes it more convenient for voters. Our implementation of the service suggests that this approach performs sufficiently well to be useful in a wide area setting.

Acknowledgements

The author is indebted to the anonymous referees for their comments and suggestions, which helped enormously to improve the paper. The author is also grateful to Tage Stabell-Kulø for his helpful comments on the first stages of the paper.

References

[1] R.S.-N.A. Baraani-Dastjerdi, J. Pieprzyk, Secure voting protocol using threshold schemes, in: Proceedings of the 11th IEEE Annual Computer Security Applications Conference, New Orleans, December 1995, pp. 143–148.

[2] J. Benaloh, D. Tuinstra, Receipt-free secret-ballot elections, in: Proceedings of the 26th Annual ACM Symposium on the Theory of Computing, Montréal, Que., Canada, May 23–25, 1994, pp. 544–553.

[3] C. Boyd, A new multiple keys cipher and an improved voting scheme, in: Proceedings of Advances in Cryptology—EUROCRYPT'89, Hounthalen, Belgium, April 10–13, 1989, pp. 617–625.

[4] C.-C. Chang, W.-B. Wu, A secure voting system on a public network, Networks 29 (2) (1997) 81–87.

[5] D. Chaum, Untraceable electronic mail, return address, and digital pseudonyms, Commun. ACM 24 (2) (1981) 84–88.

[6] D. Chaum, Security without identification: transaction systems to make the big brother obsolete, Commun. ACM 28 (10) (1985) 1030–1044.

[7] D. Chaum, Elections with unconditionally secrets ballots and disruption equivalent to breaking RSA, in: Proceedings of Eurocrypt'88, Davos, Switzerland, May 25–27, 1988, pp. 177–182.
[8] J.D. Cohen, M.J. Fischer, A robust and verifiable cryptographically secure election scheme, in: Proceedings of the 26th IEEE Annual Symposium on Foundations of Computer Science, October 1985, pp. 372–382.
[9] R. Cramer, R. Gennaro, B. Schoenmakers, A secure and optimally efficient multi-authority election scheme, in: Proceedings of the Advances in Cryptology—EUROCRYPT'97, Konstanz, Germany, May 11–15, 1997, pp. 103–108.
[10] R. Cramer, R. Gennaro, B. Schoenmakers, M. Yung, Multi-authority secret-ballot elections with linear work, in: Advances in Cryptology—EUROCRYPT'96, Saragossa, Spain, May 12–16, 1996, pp. 72–83.
[11] L.F. Cranor, R.K. Cytron, Sensus: a security-conscious electronic polling system for the internet, in: Proceedings of the 13th IEEE Hawaii International Conference on System Sciences, Maui, Hawaii, January 7–10, 1997, pp. 561–570.
[12] Cryptix, 2001. www.cryptix.com.
[13] Y. Desmedt, K. Kurosawa, How to break a practical mix and design a new one, in: Advances in Cryptology—EUROCRYPT 2000, Bruges, Belgium, May 14–18, 2000, pp. 557–572.
[14] G. Dini, Electronic voting in a large-scale distributed system, Networks 38 (1) (2001) 1–11.
[15] A. Fujioka, T. Okamoto, K. Otha, A practical secret voting scheme for large-scale elections, in: Proceedings of Advances in Cryptology—AUSCRYPT'92, Gold Coast, Qld., Australia, December 13–16, 1992, pp. 244–251.
[16] J. Furukawa, K. Sako, An efficient scheme for proving a shuffle, in: Advances in Cryptology—CRYPTO 2001, Santa Barbara, CA, USA, August 19–23, 2001, pp. 368–387.
[17] D.M. Goldschlag, M. Reed, P. Syverson, Onion routing for anonymous and private Internet connections, Commun. ACM 42 (2) (1999) 39–41.
[18] M. Hirt, K. Sako, Efficient receipt-free voting based on homomorphic encryption, in: Advances in Cryptology—EUROCRYPT'00, Vol. 1087, Lecture Notes in Computer Science, Springer, Berlin, May 2000, pp. 539–556.
[19] Report of the National Workshop on Internet Voting, March 2001, Internet Policy Institute. http://www.internetpolicy.org/research/e_voting_report.pdf.
[20] K.R. Iverson, A cryptographic scheme for computerized general elections, in: Proceeding of Advances in Cryptology—CRYPTO'91, Santa Barbara, CA, USA, August 11–15, 1991, pp. 405–419.
[21] M. Jakobsson, Flash mixing, in: Proceedings of the 18th ACM Symposium on Principles of Distributed Computing, Atlanta, GA, USA, May 3–6, 1999, pp. 83–89.
[22] M. Jakobsson, A. Juels, An optimally robust hybrid mix network, in: Proceedings of the 20th ACM Symposium on Principles of Distributed Computing, Newport, Rhode Islands, USA, August 2001, pp. 284–292.
[23] J.K. Jan, C.C. Tai, A secure electronic voting protocol with IC cards, J. Syst. Softw. 39 (2) (1997) 93–101.
[24] A. Jerichow, J. Müller, A. Pfitzmann, B. Pfitzmann, M. Waidner, Real-time mixes: a bandwidth-efficient anonymity protocol, IEEE J. Selected Areas Commun. 16 (4) (1998) 495–509.
[25] J. Karro, J. Wang, Towards a practical, secure, and very large-scale online election, in: Proceedings of the 15th IEEE Annual Computer Security Applications Conference, Phoenix, AZ, USA, December 6–10, 1999, pp. 161–169.
[26] B.W. Lampson, Atomic transactions, in: B.W. Lampson, M. Paul, H. Siegert (Eds.), Distributed Systems—Architecture and Implementation, Vol. 105, Lecture Notes in Computer Science, Springer, Berlin, 1981, pp. 246–265.
[27] K.R.T. Larsen, Voting technology implementation, Commun. ACM 42 (12) (1999) 55–57.
[28] D. Malkhi, M.K. Reiter, Byzantine quorum systems, Distr. Comput. 11 (4) (1998) 203–213.
[29] A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, Handbook of Applied Cryptography, CRC Press, Boca Raton, October 1996.
[30] Y. Mu, V. Varadharajan, Anonymous secure e-voting over a network, in: Proceedings of the 14th IEEE Annual Computer Security Applications Conference, Phoenix, AZ, USA, December 7–11, 1998, pp. 293–299.
[31] MySQL, 1999. www.mysql.com.
[32] Data Encryption Standard, FIPS Pub. 46 Ed., National Bureau of Standards, January 1977.
[33] A.C. Neff, A verifiable secret shuffle and its application to e-voting, in: Proceedings of the Eighth ACM Conference on Computer and Communication Security, Philadelphia, PA, USA, 2001, pp. 116–125.
[34] H. Nurmi, A. Salomaa, L. Santean, Secret ballot elections in computer networks, Comput. Security 10 (6) (1991) 553–560.
[35] T. Okamoto, Receipt-free electronic voting schemes for large-scale elections, in: Proceedings of the Fifth Workshop on Security Protocols 1997, Lecture Notes in Computer Science, Paris, France, April 7–9, 1997, pp. 25–35.
[36] I. Ray, I. Ray, N. Narasimhamurthi, An anonymous electronic voting protocol for voting over the internet, in: Proceedings of the Third International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems (WECWIS 2001), San Juan, CA, USA, June 2001, pp. 188–190.
[37] M.G. Reed, P.F. Syverson, D.M. Goldschlag, Anonymous connections and onion routing, IEEE J. Selected Areas Commun. 16 (4) (1998) 482–494.
[38] A. Riera, J. Borrel, J. Rifà, An uncoercible verifiable electronic voting protocol, in: Proceedings of the 14th International Security Conference (IFIP/SEC'98), September 1998, pp. 206–215.
[39] R.L. Rivest, The MD5 message-digest algorithm, Internet Request for Comment, RFC 1321, Internet Engineering Task Force, April 1992.
[40] R.L. Rivest, A. Shamir, L.M. Adleman, A method for obtaining digital signatures and public key cryptosystems, Commun. ACM 21 (2) (1978) 120–126.

[41] A. Rubin, Security considerations for remote electronic voting over the internet, in: Proceedings of the 29th Research Conference on Communication, Information and Internet Policy (TPRC 2001), Alexandria, Virginia, October 27–29, 2001. The paper is available from the conference's Web site: http://www.tprc.org/TPRC01/2001.HTM.

[42] K. Sako, J. Kilian, Receipt-free mix-type voting scheme—a practical implementation of a voting booth, in: Advances in Cryptology—CRYPTO'95, Vol. 921, Lecture Notes in Computer Science, Springer, Berlin, 1995, pp. 393–403.

[43] B. Schoenmakers, A simple publicly verifiable secret sharing scheme and its application to electronic voting, in: Advances in Cryptology—CRYPTO'99, Santa Barbara, CA, USA, August 1999, pp. 148–164.

Gianluca Dini was born in Livorno, Italy, in 1965. He received his Dott. Ing. degree in electronics engineering from the University of Pisa, Pisa, Italy, in 1990, and his Perfezionamento (equivalent to a PhD degree) in computer engineering from Scuola Superiore di Studi Universitari e di Perfezionamento "S. Anna", Pisa, Italy, in 1995. From 1993 to 1999 he was a researcher at the Dipartimento di Ingegneria della Informazione, Elettronica, Informatica, Telecomunicazioni of the University of Pisa, where he is now a Associate Professor. His research interests are in the field of fault-tolerance and security in distributed systems. He made research in the area of distributed operating systems and distributed group programming.