NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

PhD THESIS

Byzantine fault-tolerant vote collection for D-DEMOS, a distributed e-voting system

Nikos A. Chondros

ATHENS

NOVEMBER 2016

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Συλλογή ψήφων με ανοχή λαθών Βυζαντινού τύπου για το κατανεμημένο σύστημα εκλογών D-DEMOS

Νίκος Α. Χονδρός

ΑΘΗΝΑ

ΝΟΕΜΒΡΙΟΣ 2016

PhD THESIS

Byzantine fault-tolerant vote collection for D-DEMOS, a distributed e-voting system

Nikos A. Chondros

Examination Date: November 29, 2016

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Συλλογή ψήφων με ανοχή λαθών Βυζαντινού τύπου για το κατανεμημένο σύστημα εκλογών D-DEMOS

Νίκος Α. Χονδρός

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Μέμα Ρουσσοπούλου, Αναπληρώτρια Καθηγήτρια ΕΚΠΑ

ΤΡΙΜΕΛΗΣ ΕΠΙΤΡΟΠΗ ΠΑΡΑΚΟΛΟΥΘΗΣΗΣ:

Μέμα Ρουσσοπούλου, Αναπληρώτρια Καθηγήτρια ΕΚΠΑ

Αλέξης Δελής, Καθηγητής ΕΚΠΑ

Άγγελος Κιαγιάς, Αναπληρωτής Καθηγητής ΕΚΠΑ

ΕΠΤΑΜΕΛΗΣ ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

Μέμα Ρουσσοπούλου,
Αναπληρώτρια Καθηγήτρια ΕΚΠΑ

Αλέξης Δελής,
Καθηγητής ΕΚΠΑ

Άγγελος Κιαγιάς,
Αναπληρωτής Καθηγητής ΕΚΠΑ

Γιάννης Σμαραγδάκης,
Καθηγητής ΕΚΠΑ

Ευστάθιος Χατζηευθυμιάδης,

Αναπληρωτής Καθηγητής ΕΚΠΑ

Παναγιώτα Φατούρου,
Αναπληρώτρια Καθηγήτρια
Πανεπιστήμιο Κρήτης

Βασίλης Ζήκας,
Επίκουρος Καθηγητής Rensselaer PI

Ημερομηνία Εξέτασης: 29 Νοεμβρίου 2016

ABSTRACT

E-voting systems are a powerful technology for improving democracy by reducing election cost, increasing voter participation, and even allowing voters to directly verify the entire election procedure. Unfortunately, prior internet voting systems have single points of failure, which may result in the compromise of availability, voter secrecy, or integrity of the election results.

In this thesis, we consider increasing the fault-tolerance of voting systems by introducing distributed components. This is non-trivial as, besides integrity and availability, voting requires safeguarding confidentiality as well, against a malicious adversary. We focus on the vote collection phase of the voting system, which is a crucial part of the election process.

We use the DEMOS state-of-the-art but centralized voting system as the basis for our study. This system uses vote codes to represent voters' choices, an Election Authority to setup the election and handle vote collection and result production, and a Bulletin Board for storing the election transcript for the long-term. We extract the vote collection mechanism from the centralized Election Authority component of the original DEMOS system, and replace it with a distributed system that handles vote collection in a Byzantine fault-tolerant manner. In this thesis, we present the design, security analysis, prototype implementation and experimental evaluation of this vote collection component.

We present two versions of this component: one completely asynchronous and one with minimal timing assumptions but better performance. Both versions provide immediate assurance to the voter her vote was recorded as cast, without requiring cryptographic operations on behalf of the voter. This way, a voter may cast her vote using an untrusted computer or network, and still be assured her vote was recorded as cast. For example, she may vote via a public web terminal, or by sending an SMS from a mobile phone. Even in these cases, voter's privacy is still preserved.

We provide a model and security analysis of the systems we present. We implement prototypes of the complete systems, we measure their performance experimentally, and we demonstrate their ability to handle large-scale elections. Finally, we demonstrate the performance trade-offs between the two versions of the system.

We consider the vote collection components we introduce are applicable to any voting system that uses the code-voting technique.

SUBJECT AREA: Distributed Systems

KEYWORDS: Byzantine Fault-tolerance, voting system, vote collection

ΠΕΡΙΛΗΨΗ

Τα συστήματα διαχείρισης εκλογών είναι μια δυναμική τεχνολογία που επιτρέπει την βελτίωση της δημοκρατικής διαδικασίας μέσω της μείωσης του κόστους υλοποίησης εκλογών, της αύξησης της συμμετοχής των ψηφοφόρων και της αμεσότητας παραγωγής αποτελεσμάτων. Επίσης, δίνουν την δυνατότητα στους ψηφοφόρους να επιβεβαιώσουν άμεσα την ορθή λειτουργία ολόκληρης της εκλογικής διαδικασίας. Δυστυχώς, τα υπάρχοντα τέτοια συστήματα είναι σχεδιασμένα με κεντρικά συστατικά, τα οποία και αποτελούν μοναδικά σημεία αποτυχίας. Αυτό μπορεί να οδηγήσει στην απώλεια διαθεσιμότητας, εμπιστευτικότητας, καθώς και της ακεραιότητας του εκλογικού αποτελέσματος.

Σε αυτή τη διατριβή εξετάζουμε την εισαγωγή ανοχής λαθών στα εκλογικά συστήματα, μέσω της εισαγωγής κατανεμημένων συστατικών. Αυτό είναι περίπλοκο γιατί, εκτός από την ακεραιότητα και διαθεσιμότητα, σε ένα εκλογικό σύστημα είναι σημαντικό να διαφυλαχθεί και η εμπιστευτικότητα, απέναντι σε έναν κακόβουλο αντίπαλο. Εστιάζουμε στην φάση συλλογής ψήφων του εκλογικού συστήματος, η οποία είναι ένα κρίσιμο τμήμα της εκλογικής διαδικασίας.

Χρησιμοποιούμε το σύγχρονο αλλά κεντρικοποιημένο σύστημα διαχείρισης εκλογών DEMOS σαν βάση για την μελέτη μας. Αυτό το σύστημα χρησιμοποιεί κωδικούς που αντιστοιχούν στις δυνατές επιλογές των ψηφοφόρων, μια Αρχή Εκλογών η οποία αρχικοποιεί τις εκλογές, συλλέγει τις ψήφους και παράγει το αποτέλεσμα, και έναν Πίνακα Ανακοινώσεων για την διατήρηση των στοιχείων των εκλογών μακροπρόθεσμα. Εξάγουμε τον μηχανισμό συλλογής ψήφων από την κεντρικοποιημένη Αρχή Εκλογών του αρχικού συστήματος DEMOS, και τον αντικαθιστούμε με ένα κατανεμημένο σύστημα που χειρίζεται την συλλογή ψήφων με ανοχή σε λάθη Βυζαντινού τύπου. Σε αυτή τη διατριβή, παρουσιάζουμε τον σχεδιασμό, ανάλυση ασφάλειας, την ανάπτυξη και αξιολόγηση της πρωτότυπης υλοποίησης αυτού του κατανεμημένου συστατικού συλλογής ψήφων.

Παρουσιάζουμε δύο εκδόσεις αυτού του συστατικού: μία πλήρως ασύγχρονη και μία με ελάχιστες υποθέσεις συγχρονισμού αλλά καλύτερη απόδοση. Και οι δύο εκδόσεις παρέχουν άμεση επιβεβαίωση στην ψηφοφόρο ότι η ψήφος της καταχωρήθηκε όπως υποβλήθηκε, χωρίς να απαιτούνται κρυπτογραφικές λειτουργίες από την πλευρά της ψηφοφόρου. Με αυτόν τον τρόπο, η ψηφοφόρος μπορεί να στείλει την ψήφο της χρησιμοποιώντας έναν μη ασφαλή υπολογιστή ή δίκτυο, και να συνεχίσει να είναι εξασφαλισμένη ότι η ψήφος της καταχωρήθηκε σωστά. Για παράδειγμα, μπορεί να ψηφίσει χρησιμοποιώντας έναν δημόσιο υπολογιστή, ή στέλνοντας ένα σύντομο μήνυμα μέσω κινητού τηλεφώνου. Ακόμη και σε αυτές τις περιπτώσεις, η εμπιστευτικότητα της ψήφου διατηρείται στο ακέραιο.

Δίνουμε ένα μοντέλο και μια ανάλυση ασφάλειας για τα συστήματα που παρουσιάζουμε. Υλοποιούμε πρωτότυπα από τα πλήρη συστήματα, μετράμε την απόδοσή τους πειραματικά, και επιδεικνύουμε την ικανότητά τους να χειρίζονται εκλογές μεγάλου μεγέθους. Τέλος, παρουσιάζουμε τις διαφορές απόδοσης ανάμεσα στις δύο εκδόσεις του συστήματος.

Θεωρούμε ότι τα συστατικά συλλογής ψήφων που παρουσιάζουμε σε αυτή τη διατριβή μπορούν να βρουν εφαρμογή σε οποιοδήποτε σύστημα διαχείρισης εκλογών που στηρίζεται στην τεχνική της εκπροσώπησης των επιλογών στα ψηφοδέλτια με κωδικούς.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Κατανεμημένα Συστήματα

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Ανοχή Βυζαντινών λαθών, συστήματα διαχείρισης εκλογών, συλλογή ψήφων

ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΗΣ ΔΙΔΑΚΤΟΡΙΚΗΣ ΔΙΑΤΡΙΒΗΣ

1.1. Εισαγωγή στη διατριβή

Τα συστήματα διαχείρισης εκλογών είναι μια δυναμική τεχνολογία που επιτρέπει την βελτίωση της δημοκρατικής διαδικασίας μέσω της μείωσης του κόστους υλοποίησης εκλογών, της αύξησης της συμμετοχής των ψηφοφόρων και της αμεσότητας παραγωγής αποτελεσμάτων. Επίσης, δίνουν την δυνατότητα στους ψηφοφόρους να επιβεβαιώσουν άμεσα την ορθή λειτουργία ολόκληρης της εκλογικής διαδικασίας. Δυστυχώς, τα υπάρχοντα τέτοια συστήματα είναι σχεδιασμένα με κεντρικά συστατικά, τα οποία και αποτελούν μοναδικά σημεία αποτυχίας. Αυτό μπορεί να οδηγήσει στην απώλεια διαθεσιμότητας, εμπιστευτικότητας, καθώς και της ακεραιότητας του εκλογικού αποτελέσματος.

Σε αυτή τη διατριβή εξετάζουμε την εισαγωγή ανοχής λαθών στα εκλογικά συστήματα, μέσω της εισαγωγής κατανεμημένων συστατικών. Αυτό είναι περίπλοκο γιατί, εκτός από την ακεραιότητα και διαθεσιμότητα, σε ένα εκλογικό σύστημα είναι σημαντικό να διαφυλαχθεί και η εμπιστευτικότητα, απέναντι σε έναν κακόβουλο αντίπαλο. Εστιάζουμε στην φάση συλλογής ψήφων του εκλογικού συστήματος, η οποία είναι ένα κρίσιμο τμήμα της εκλογικής διαδικασίας.

Χρησιμοποιούμε το σύγχρονο αλλά κεντρικοποιημένο σύστημα διαχείρισης εκλογών DEMOS σαν βάση για την μελέτη μας. Αυτό το σύστημα χρησιμοποιεί κωδικούς που αντιστοιχούν στις δυνατές επιλογές των ψηφοφόρων, μια Αρχή Εκλογών η οποία αρχικοποιεί τις εκλογές, συλλέγει τις ψήφους και παράγει το αποτέλεσμα, και έναν Πίνακα Ανακοινώσεων για την διατήρηση των στοιχείων των εκλογών μακροπρόθεσμα. Εξάγουμε τον μηχανισμό συλλογής ψήφων από την κεντρικοποιημένη Αρχή Εκλογών του αρχικού συστήματος DEMOS, και τον αντικαθιστούμε με ένα κατανεμημένο σύστημα που χειρίζεται την συλλογή ψήφων με ανοχή σε λάθη Βυζαντινού τύπου.

Σε αυτή τη διατριβή, παρουσιάζουμε τον σχεδιασμό, ανάλυση ασφάλειας, την ανάπτυξη και αξιολόγηση της πρωτότυπης υλοποίησης αυτού του κατανεμημένου συστατικού συλλογής ψήφων.

Παρουσιάζουμε δύο εκδόσεις αυτού του συστατικού: μία πλήρως ασύγχρονη και μία με ελάχιστες υποθέσεις συγχρονισμού αλλά καλύτερη απόδοση. Και οι δύο εκδόσεις παρέχουν άμεση επιβεβαίωση στον ψηφοφόρο ότι η ψήφος του καταχωρήθηκε όπως υποβλήθηκε, χωρίς να απαιτούνται κρυπτογραφικές λειτουργίες από την πλευρά του ψηφοφόρου. Με αυτόν τον τρόπο, ο ψηφοφόρος μπορεί να στείλει την ψήφο του χρησιμοποιώντας έναν μη ασφαλή υπολογιστή ή δίκτυο, και να συνεχίσει να είναι εξασφαλισμένος ότι η ψήφος του καταχωρήθηκε σωστά. Για παράδειγμα, μπορεί να ψηφίσει χρησιμοποιώντας έναν δημόσιο υπολογιστή, ή στέλνοντας ένα σύντομο μήνυμα μέσω κινητού τηλεφώνου. Ακόμη και σε αυτές τις περιπτώσεις, η εμπιστευτικότητα της ψήφου διατηρείται στο ακέραιο.

Συνολικά, σε αυτή τη διατριβή γίνονται οι εξής συνεισφορές:

- Εισάγεται ένα κατανεμημένο πρωτόκολλο συλλογής ψήφων, που απαιτεί ένα και μόνο σημείο συγχρονισμού για να καταλήξει σε συμφωνία.
- Βελτιώνουμε αυτό το πρωτόκολλο κάνοντάς το πλήρως ασύγχρονο. Και στις δύο περιπτώσεις πάντως, οι ψηφοφόροι μπορούν να είναι βέβαιοι ότι η ψήφος τους θα καταμετρηθεί σωστά χωρίς να απαιτείται να χρησιμοποιήσουν έμπιστες συσκευές ή δίκτυα.
- Παρουσιάζουμε μια ανάλυση ασφάλειας των δύο πρωτοκόλλων, αποδεικνύοντας τις ιδιότητες ακεραιότητας (safety) και προόδου (liveness) τους.
- Υλοποιούμε πρωτότυπες εκδόσεις των συστημάτων που παρουσιάζουμε, μετράμε εργαστηριακά την απόδοσή τους, και επιβεβαιώνουμε την δυνατότητά τους να διαχειριστούν εκλογές μεγάλης κλίμακας. Τέλος, παρουσιάζουμε τις διαφορές απόδοσης ανάμεσα στις δύο εκδόσεις του συστήματος.

Θεωρούμε ότι τα συστατικά συλλογής ψήφων που παρουσιάζουμε σε αυτή τη διατριβή μπορούν να βρουν εφαρμογή σε οποιοδήποτε σύστημα διαχείρισης εκλογών που στηρίζεται στην τεχνική της εκπροσώπησης των επιλογών στα ψηφοδέλτια με κωδικούς.

1.2. Σύντομη περιγραφή του υπάρχοντος συστήματος DEMOS

Το σύστημα DEMOS είναι ένα σύγχρονο σύστημα διαχείρισης εκλογών, το οποίο παρέχει επιβεβαιωσιμότητα από άκρη σε άκρη (end-to-end verifiability), επιτρέποντας σε οποιονδήποτε να επιβεβαιώσει την σωστή του λειτουργία προς την παραγωγή του αποτελέσματος. Στο σύστημα αυτό ο ψηφοφόρος μπορεί να επιλέξει 1 από m επιλογές, και να ψηφίσει υποβάλλοντας τον κωδικό που αντιστοιχεί στην επιλογή του. Κάθε ψηφοδέλτιο έχει δύο λειτουργικά ισοδύναμα τμήματα, και ο ψηφοφόρος επιλέγει και χρησιμοποιεί τυχαία ένα από τα δύο. Κάθε ένα τμήμα περιλαμβάνει ολόκληρη την λίστα με τις ίδιες m επιλογές και διαφορετικούς αντίστοιχους κωδικούς. Το σύστημα DEMOS περιλαμβάνει μια Αρχή Εκλογών (ΑΕ), η οποία προετοιμάζει τα ψηφοδέλτια, συλλέγει τις ψήφους, και παράγει το αποτέλεσμα. Επίσης περιλαμβάνει ένα Πίνακα Ανακοινώσεων στον οποίο αναρτώνται όλα τα στοιχεία των εκλογών, φυσικά με τρόπο που δεν παραβιάζει την ιδιωτικότητα της ψήφου.

Το σύστημα DEMOS έχει τα εξής μειονεκτήματα:

- Η ΑΕ και ο Πίνακας Ανακοινώσεων είναι μοναδικά σημεία αποτυχίας (πρόβλημα διαθεσιμότητας).
- Κατά την συλλογή ψήφων, η ΑΕ περιέχει κρυφό περιεχόμενο το οποίο, αν αποκαλυφθεί, μπορεί να επιτρέψει στον επιτιθέμενο είτε να ψηφίσει αντί για τους ψηφοφόρους, είτε να μάθει πως αυτοί ψήφισαν.
- Κατά την διάρκεια της συλλογής ψήφων, οι ψηφοφόροι δεν εξασφαλίζονται ότι η ψήφος τους καταχωρήθηκε σωστά.

Σε αυτή την διατριβή εστιάζουμε στην ανοχή οποιουδήποτε λάθους στην συλλογή ψήφων, διατηρώντας παράλληλα τις υπόλοιπες ιδιότητες του συστήματος εκλογών.

2. Επιστημονικό υπόβαθρο

Στο κεφάλαιο αυτό παρουσιάζονται μια σειρά ορισμών και γνώσης απαραίτητης για την κατανόηση των υπόλοιπων κεφαλαίων. Συγκεκριμένα, παρουσιάζονται οι επιθυμητές ιδιότητες επιβεβαιωσιμότητας, ιδιωτικότητας και ανοχής λαθών των συστημάτων εκλογών . Επίσης, ορίζονται αυστηρά τα σύγχρονα και ασύγχρονα συστήματα, και τα προβλήματα της Βυζαντινού τύπου συμφωνίας (Byzantine Agrement), Βυζαντινού τύπου συναίνεσης (Byzantine Consensus), και της διαδραστικής συνέπειας (Interactive Consistency), από την βιβλιογραφία των Κατανεμημένων Συστημάτων. Στη συνέχεια παρουσιάζονται, από την βιβλιογραφία της Κρυπτογραφίας, οι προσθετικά ομομορφικές δεσμεύσεις (additively homomorphic commitments), και οι αποδείξεις μηδενικής γνώσης (Zero-knowledge Proofs). Τέλος, παρουσιάζεται στην πληρότητά του το πλήρως κατανεμημένο σύστημα εκλογών D-DEMOS, έτσι ώστε ο αναγνώστης να δει το επιθυμητό αποτέλεσμα στο σύνολό του. Να σημειώσουμε εδώ ότι τα υποσυστήματα συλλογής ψήφων που παρουσιάζονται σε αυτή τη διατριβή αποτελούν τμήματα του D-DEMOS.

3. Περιγραφή Συστήματος

Στο κεφάλαιο αυτό, αρχικά ορίζεται το μοντέλο του συστήματος. Υποθέτουμε ένα πλήρως συνδεδεμένο δίκτυο, όπου μηνύματα μπορούν να χαθούν, να επαναληφθούν, ή να παραδοθούν εκτός σειράς. Όμως υποθέτουμε ότι το δίκτυο τελικά παραδίδει κάθε μήνυμα, εφόσον ο αποστολέας συνεχώς ξαναπροσπαθεί να το αποστείλει. Δεν υποθέτουμε τίποτα σχετικά με την ταχύτητα των επεξεργαστών των κόμβων. Υποθέτουμε όμως ότι τα ρολόγια των κόμβων είναι συγχρονισμένα αρκετά κοντά στην πραγματική ώρα, έτσι ώστε να μπορούμε να τηρήσουμε τα όρια αρχής και τέλους ψηφοφορίας που θέτουν οι εκλογικές αρχές. Στοχεύουμε στην ανοχή λαθών Βυζαντινού τύπου γιατί αναμένουμε το σύστημα να εγκατασταθεί σε κόμβους με διαφορετικούς διαχειριστές, και θέλουμε το σύστημα να ανεχθεί ανθρώπινα λάθη, όπως η κλοπή στοιχείων πρόσβασης. Τέλος, υποθέτουμε ότι ο αντίπαλος δεν μπορεί να παραβιάσει την ασφάλεια των κρυπτογραφικών εργαλείων που χρησιμοποιούμε.

Στη συνέχεια εισάγουμε το υποσύστημα Συλλογής Ψήφων (ΣΨ), το οποίο απαλλάσσει την ΑΕ από την ομώνυμη διαδικασία και την αναλαμβάνει αυτό. Βελτιώνουμε λοιπόν την ΑΕ έτσι ώστε να παράγει, εκτός από τον κωδικό που αντιστοιχεί σε κάθε επιλογή (ΚΕ), και ένα τυχαίο αριθμό που θα αποτελέσει την ”απόδειξη” καταγραφής της ψήφου. Έτσι, η ΑΕ, αρχικοποιεί τους κόμβους ΣΨ με κρυπτογραφημένους τους ΚΕ (σε δεσμευτική μορφή - committed form), και με τμήματα της απόδειξης, χρησιμοποιώντας μια προσέγγιση επιβεβαιώσιμου μοιράσματος μυστικών (Verifiable Secret Sharing). Με αυτόν τον τρόπο, κανένας κόμβος ΣΨ δεν μπορεί να αποκαλύψει τους ΚΕ, ούτε και μεμονωμένα να παρέχει την απόδειξη καταγραφής ψήφου.

Το σύστημα ΣΨ είναι κατανεμημένο, αποτελείται από N_v κόμβους, και ανέχεται μέχρι και f_v σφάλματα, όπου $f_v < N_v/3$. Οι κόμβοι έχουν ένα μυστικό κανάλι για την μεταξύ τους επικοινωνία, και ένα δημόσιο για την επικοινωνία με τους ψηφοφόρους. Η χρήση του υποσυστήματος αυτού έχει ως εξής. Ο ψηφοφόρος επιλέγει έναν κόμβο ΣΨ τυχαία και υποβάλλει σε αυτόν την ψήφο του μέσω του ΚΕ. Ο κόμβος ΣΨ επιβεβαιώνει την εγκυρότητα του ΚΕ για το συγκεκριμένο ψηφοδέλτιο, και ξεκινάει μια αλληλεπίδραση με τους άλλους κόμβους ΣΨ όπου αποκαλύπτει το δικό του τμήμα της απόδειξης και περιμένει αντίστοιχα μηνύματα από τους άλλους κόμβους ΣΨ, έτσι ώστε να αποκτήσει την απόδειξη στην ανοικτή της μορφή. Μόλις την αποκτήσει, την αποστέλλει στον ψηφοφόρο, ο οποίος και την συγκρίνει με αυτήν στο ψηφοδέλτιό του για επιβεβαίωση. Σε περίπτωση λάθους, ο ψηφοφόρος επιλέγει έναν άλλο κόμβο και ξαναπροσπαθεί. Ονομάζουμε αυτήν την αλληλεπίδραση μεταξύ των κόμβων ΣΨ Πρωτόκολλο Συλλογής Ψήφων.

Όταν παρέλθει η ώρα τέλους ψηφοφορίας, οι κόμβοι ΣΨ σταματούν να δέχονται ψήφους και ξεκινούν το Πρωτόκολλο Συμφωνίας Συνόλου Ψήφων. Με αυτό το πρωτόκολλο, όλοι οι έντιμοι κόμβοι ΣΨ θα συμφωνήσουν σε ένα και μόνο σύνολο από καταγεγραμμένες ψήφους, φροντίζοντας να συμπεριλάβουν όλες αυτές για τις οποίες αποστάλθηκε από το σύστημα απόδειξη στον ψηφοφόρο.

Οι δύο παραλλαγές του συστήματος διαφέρουν στα δύο αυτά πρωτόκολλα. Η πρώτη, D-DEMOS/IC, είναι σύγχρονη και χρησιμοποιεί ένα πρωτόκολλο Διαδραστικής Συνέπειας για την συμφωνία του συνόλου ψήφων. Η δεύτερη, D-DEMOS/Async, είναι ασύγχρονη. Χρησιμοποιεί πρωτόκολλο συναίνεσης για κάθε ψηφοδέλτιο, και προσθέτει ένα ακόμη βήμα κατά το πρωτόκολλα Συλλογής Ψήφων, όπου φροντίζει με την δημιουργία ενός Πιστοποιητικού Μοναδικότητας για την μοναδικότητα της ψήφου, χειριζόμενο με αυτόν τον τρόπο κακόβουλους ψηφοφόρους.

Αμέσως μετά την παρουσίαση κάθε έκδοσης του συστήματος ΣΨ, αποδεικνύουμε δύο ιδιότητες της ασφάλειάς του. Η πρώτη είναι η ακεραιότητα (safety), όπου δείχνουμε ότι δεν μπορεί το σύστημα να παράξει λάθος δεδομένα. Η δεύτερη είναι αυτή της προόδου (liveness), που αποδεικνύει ότι εφόσον ο ψηφοφόρος υποβάλλει την ψήφο του “έγκαιρα”, αυτή θα καταγραφεί. Δίνουμε μάλιστα έναν τύπο υπολογισμού του χρόνου που χρειάζεται το σύστημα για να παράξει την απόδειξη καταγραφής ψήφου, φροντίζοντας με αυτόν τον τρόπο να ορίσουμε αυστηρότερα τον όρο “έγκαιρα”.

4. Συζήτηση

Σε αυτό το κεφάλαιο αναπτύσσονται διάφορα θέματα σχετικά με το συγκεκριμένο αντικείμενο. Αρχικά, αιτιολογούμε την επιλογή μας να μην χρησιμοποιήσουμε μια από τις υπάρχουσες λύσεις Μηχανής Καταστάσεων με Αντίγραφα με ανοχή λαθών Βυζαντινού τύπου (Byzantine Fault Tolerant Replicated State Machines). Πρώτον, μια τέτοια προσέγγιση δεν θα προστάτευε την ιδιωτικότητα των στοιχείων εφόσον ο αντίπαλος κατάφερνε να πάρει τον έλεγχο ενός κόμβου ΣΨ. Δεύτερον, ένας τέτοιος αλγόριθμος θα απαιτούσε ο ψηφοφόρος να χρησιμοποιήσει μια έμπιστη συσκευή για να υποβάλλει την ψήφο του, αφού όλοι οι σχετικοί αλγόριθμοι απαιτούν, τουλάχιστον, την επαλήθευση ψηφιακών υπογραφών.

Στη συνέχεια αναπτύσσουμε μια σειρά από σενάρια επίθεσης του αντιπάλου, δείχνοντας σε απλή γλώσσα πως το σύστημά μας τις αντιμετωπίζει με επιτυχία.

Σε περίπτωση κακόβουλης Αρχής Εκλογών, το σύστημα προστατεύεται από τα διπλά ψηφοδέλτια. Είναι αδύνατον για την ΑΕ να προβλέψει ποια πλευρά θα χρησιμοποιήσει ο ψηφοφόρος, οπότε αν διαβάλλει την αντιστοιχία κωδικών και επιλογών, κάθε ψηφοφόρος έχει $1/2$ πιθανότητα να το διαπιστώσει.

Την περίπτωση κακόβουλου ψηφοφόρου, ο οποίος προσπαθεί να υποβάλλει διαφορετικές ψήφους σε διαφορετικούς κόμβους ΣΨ, οι δύο εκδοχές του συστήματος την αντιμετωπίζουν διαφορετικά. Το μεν D-DEMOS/IC το διαχειρίζεται με το Πρωτόκολλο Συμφωνίας Συνόλου Ψήφων, αφού ο κάθε έντιμος κόμβος έχει πρόσβαση στη γνώση όλων των άλλων έντιμων κόμβων. Το δε D-DEMOS/Async, το διαχειρίζεται στο Πρωτόκολλο Συλλογής Ψήφων με το Πιστοποιητικό Μοναδικότητας, καθώς το τελευταίο απαγορεύει την αποδοχή δεύτερης επιλογής για το ίδιο ψηφοδέλτιο.

Τέλος, παρουσιάζουμε τις δυνατότητες του επιτιθέμενου όταν έχει στον έλεγχό του έναν κόμβο ΣΨ, σε συνεργασία μάλιστα με κακόβουλους ψηφοφόρους. Εξηγούμε ότι από την στιγμή που ένας έντιμος ψηφοφόρος υποβάλλει τον ΚΕ, αυτός ο κωδικός είναι πλέον δημόσια πληροφορία καθώς εκφράζει την βούληση του ψηφοφόρου χωρίς να αποκαλύπτει την επιλογή του. Έτσι, πιθανή υποκλοπή του ΚΕ δεν αποτελεί παραβίαση της ιδιωτικότητας του ψηφοφόρου. Την περίπτωση απόπειρας υποβολής πολλαπλών ΚΕ για το ίδιο ψηφοδέλτιο, το σύστημα τη διαχειρίζεται όπως και όταν προέρχεται από τους ψηφοφόρους. Τέλος, αν ένας κακόβουλος κόμβος ΣΨ υποδεχθεί την ψήφο, δεν θα μπορέσει να επανυπολογίσει την αντίστοιχη απόδειξη παρά μόνο αν επικοινωνήσει με έναν ικανοποιητικό αριθμό έντιμων κόμβων ΣΨ, τέτοιο ώστε να διασφαλίζεται ότι η ψήφος αυτή θα συμπεριληφθεί στο συμφωνημένο Σύνολο Ψήφων.

5. Υλοποίηση

Υλοποιήσαμε το σύστημα κατά το μεγαλύτερο μέρος χρησιμοποιώντας την γλώσσα προγραμματισμού Java. Δημιουργήσαμε ένα υπόβαθρο ασύγχρονου προγραμματισμού σε αυτή τη γλώσσα, όπου οι εφαρμογές ανώτερου επιπέδου απλά ζητούν την αποστολή μηνυμάτων σε κόμβους, και το υπόβαθρο αναλαμβάνει να τα παραδώσει μέσω συνδέσεων TLS. Επίσης, δημιουργήσαμε έναν υποδοχέα HTTP σε κάθε κόμβο, ο οποίος και μετατρέπει τις εισερχόμενες αιτήσεις HTTP σε μηνύματα, τα προωθεί στην εφαρμογή για εκτέλεση, και μετατρέπει τα εξερχόμενα μηνύματα σε απαντήσεις HTTP.

Με το παραπάνω υπόβαθρο, ορίσαμε τα μηνύματα και τους χειριστές τους τόσο για το πρωτόκολλο Συλλογής Ψήφων, όσο και για αυτό της Συμφωνίας Συνόλου Ψήφων, υλοποιώντας τον κόμβο ΣΨ. Για το D-DEMOS/IC, χρησιμοποιήσαμε ένα υπάρχον πρωτόκολλο διαδραστικής συνέπειας, ενώ για το D-DEMOS/Async υλοποιήσαμε το πρωτόκολλο Βυζαντινού τύπου συναίνεσης του Bracha πάνω στο υπόβαθρο ασύγχρονου προγραμματισμού που περιγράψαμε παραπάνω. Προχωρήσαμε δε σε μια υλοποίηση του πρωτοκόλλου συναίνεσης η οποία επιτρέπει την εκτέλεση πολλαπλών συναινέσεων (πάνω σε διαφορετικά θέματα) παράλληλα (batching), εξασφαλίζοντας έτσι αισθητά μειωμένο χρόνο για την ολοκλήρωση της Συμφωνίας Συνόλου Ψήφων.

6. Αξιολόγηση

Στο κεφάλαιο αυτό περιγράφουμε την διαδικασία και τα αποτελέσματα της εργαστηριακής αξιολόγησης της πρωτότυπης υλοποίησης των δύο εκδοχών του συστήματος συλλογής ψήφων. Το εργαστήριο αποτελείται από 12 υπολογιστές συνδεδεμένους άμεσα μεταξύ τους.

Υλοποιήσαμε έναν οδηγό ο οποίος αναπαριστά μια σειρά ψηφοφόρων. Δεδομένου ενός συνόλου ψηφοδελτίων, ο οδηγός αυτός τα παίρνει με τυχαία σειρά, επιλέγει τυχαία έναν κόμβο ΣΨ, συνδέεται και δέχεται την φόρμα υποβολής ψήφου, την αποστέλλει συμπληρωμένη, περιμένει την απάντηση και συνεχίζει με το επόμενο ψηφοδέλτιο. Εκκινούμε πολλαπλές παρουσίες (instances) του οδηγού, έτσι ώστε να προσομοιώσουμε ελεγχόμενα πολλαπλούς ψηφοφόρους που υποβάλλουν την ψήφο τους.

Εκτελούμε πειράματα στα οποία η βάση δεδομένων βρίσκεται στο δίσκο, και δείχνουμε ότι η αύξηση των ψηφοφόρων ή των επιλογών στα ψηφοδέλτια δεν αλλάζει σημαντικά την απόδοση του συστήματος, επιβεβαιώνοντας έτσι τις δυνατότητες κλιμάκωσής του.

Στη συνέχεια, εκτελούμε πειράματα στα οποία τα δεδομένα έχουν φορτωθεί στη μνήμη των υπολογιστών, έτσι ώστε να μετρήσουμε μόνο την απόδοση των δικτυακών μας πρωτοκόλλων. Επίσης προσομοιώνουμε ένα δίκτυο ευρείας περιοχής (WAN), έτσι ώστε να δείξουμε την επίδρασή του στο σύστημα. Αρχικά δείχνουμε ότι την επιβάρυνση που προκύπτει όταν αυξάνουμε τον αριθμό των κόμβων ΣΨ, τόσο στην απόκριση (response time) όσο και στην διεκπεραιωτική ικανότητα (throughput). Τέλος δείχνουμε ότι η αύξηση των ταυτόχρονα συνδεδεμένων ψηφοφόρων δεν επηρεάζει την διεκπεραιωτική ικανότητα του συστήματος μας.

7. Σχετική Βιβλιογραφία

Στο κεφάλαιο αυτό κάνουμε μια ανασκόπηση της σχετικής βιβλιογραφίας. Ξεκινάμε παραθέτοντας τα συστήματα διαχείρισης εκλογών τα οποία στοχεύουν στην ανοχή λαθών και τα συγκρίνουμε με το δικό μας. Στη συνέχεια παραθέτουμε την σχετική βιβλιογραφία στα θέματα της συναίνεσης, της συμφωνίας, και της διαδραστικής συμφωνίας. Τέλος παραθέτουμε τα σχετικά συστήματα Μηχανής Καταστάσεων με Αντίγραφα (Replicated State Machines).

8. Συμπεράσματα

Σε αυτή τη διατριβή παρουσιάσαμε δύο εκδοχές ενός συστήματος συλλογής ψήφων, το οποίο είναι κατανεμημένο και ανέχεται λάθη Βυζαντινού τύπου. Και οι δύο, επιτρέπουν στον ψηφοφόρο να ψηφίσει με ασφάλεια χωρίς να χρειάζεται να εμπιστευθεί ούτε την συσκευή ούτε και το δίκτυο που χρησιμοποιεί. Όλα αυτά διατηρώντας την ιδιότητα για επιβεβαιωσιμότητα από άκρη σε άκρη του χρησιμοποιηθέντος συστήματος διαχείρισης εκλογών DEMOS.

Πιστεύουμε ότι αυτή η προσέγγιση μπορεί να φανεί χρήσιμη και σε άλλα συστήματα εκλογών που είναι μονολιθικά, έτσι ώστε να γίνουν κατανεμημένα. Έτσι, θα προκύψουν πιο αξιόπιστα συστήματα διαχείρισης εκλογών, τα οποία και θα μπορέσουν να αναλάβουν την ηλεκτρονική διεκπεραίωση εκλογών μεγάλης κλίμακας.

To my wife, my late father and my mother.

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, professor Mema Roussopoulos, for her guidance and support throughout my PhD. She was always available for me, and provided inspirational guidance, encouragement and unselfish help. She kept “asking the right questions”, forcing me to understand things better and driving me forward. I don’t think completing my doctoral work would have been possible without her.

I would also like to thank professors Alex Delis and Aggelos Kiayias, the other two members of my doctoral committee, for the great cooperation we had during the project we worked together.

Next, I would like to thank the members of the Distributed Systems and CRYPTO.SEC groups, for the very fruitful cooperation we had during all these years.

I would also like to thank the European Research Council for the financing they provided for my PhD studies, via ERC Starting Grant # 279237.

Special thanks go to my wife, for her patience and support throughout my studies. I certainly could not have made it without her standing by my side!

Finally, I need to express my gratitude to my parents, for instilling the right principles in me and setting an example by adhering to them consistently throughout their lives. Their never ending moral support and encouragement have also been very valuable.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Introduction to this thesis

E-voting systems are a powerful technology to improve the election process. Kiosk-based e-voting systems, e.g., [28, 31, 58, 29, 13, 45], allow the tally to be produced faster, but require the voter's physical presence at the booth. Internet e-voting systems, e.g., [43, 4, 36, 76, 61, 115, 28, 29, 115, 72], however, allow voters to cast their votes remotely. Internet voting systems have the potential to enhance the democratic process by reducing election costs and by increasing voter participation for social groups that face considerable physical barriers and overseas voters. In addition, several internet voting systems [4, 76, 115, 72] allow voters and auditors to directly verify the integrity of the entire election process, providing *end-to-end verifiability*. This is a highly desired property that has emerged in the last decade, where voters can be assured that no entities, even the election authorities, have manipulated the election result. Despite their potential, existing internet voting systems suffer from single points of failure, which may result in the compromise of voter secrecy, service availability, or integrity of the result [28, 31, 58, 29, 13, 43, 4, 36, 76, 61, 115, 72].

In this thesis, we consider increasing the fault-tolerance of voting systems by introducing distributed components. This is non-trivial as, besides integrity and availability (or safety and liveness, as they are often called in distributed systems terminology), voting requires safeguarding confidentiality as well, against a malicious adversary.

We use the DEMOS [72] state-of-the-art but centralized voting system as the basis for our study. This system is the first to provide end-to-end verifiability in the *standard model* (i.e., without the *random oracle* assumption). It also introduces the novel idea of using the voters choices as a source of randomness, to challenge the zero-knowledge proof (ZKP) protocols [55] which the system uses to prove its setup is correct without disclosing private voters' information. This, in turn, is the means to provide end-to-end verifiability.

In its current form, the DEMOS voting system is centralized, having an Election Authority component that handles everything from setup, to vote collection, to result production. This presents a risk to availability, as a failure of this component would prohibit voting. However, it also presents a risk to voters' privacy, as an attacker that takes control of this component can obtain each voter's ballot contents, which directly violates the voter's privacy. Finally, the original centralized DEMOS system had no need to provide feedback to the voter, besides a simple acknowledgment. In a distributed world though, the voter needs to obtain feedback to be assured the vote was actually recorded as cast in enough nodes of the system, something we tackle in this thesis.

One specific attribute of DEMOS is its use of code-voting. In this scheme, there is a setup component which generates vote codes representing the possible voter's choices, and includes them in the voters' ballots. A voter votes by submitting the vote code corresponding to her choice. Because of this technique, the voter does not need to perform cryptographic operations on the device she uses to vote. Expanding on this, we set out to introduce a distributed voting system that uses no client-side cryptography at all. This allows votes to be cast with a greater variety of client devices over public networks, such as feature phones using SMS, or (untrusted) public web terminals, while still preserving voter's privacy.

In this thesis, we present the design, security analysis, prototype implementation and experimental evaluation of the vote collection components of the *D-DEMOS* suite of distributed, end-to-end verifiable internet voting systems, with no single point of failure during the election process (that is, besides setup).

We design a distributed *Vote Collection* (*VC*) subsystem that is Byzantine fault-tolerant and able to collect votes from voters and assure them their vote was recorded as cast, without requiring any cryptographic operation from the client device. This allows voters to vote via SMS, a simple console client over a telnet session, or a public web terminal, while preserving their privacy. At election end time, *VC* nodes agree on a single set of votes. We introduce two versions of the voting components of D-DEMOS that differ in how they achieve agreement on the set of cast votes. The D-DEMOS/Async version is completely asynchronous, while D-DEMOS/IC makes minimal synchrony assumptions but is more efficient. Once agreement has been achieved, *VC* nodes upload the set of cast votes to a second distributed component, the *Bulletin Board* (*BB*). This, in turn, is a replicated service that publishes its data immediately and makes it available to the public forever.

The resulting voting systems are end-to-end verifiable, by the voters themselves and third-party auditors, while preserving voter privacy. To delegate auditing, a voter provides an auditor specific information from her ballot. The auditor, in turn, reads from the distributed *BB* and verifies the complete election process, including the correctness of the election setup by election authorities. Additionally, as the number of auditors increases, the probability of election fraud going undetected diminishes exponentially.

We prove the security attributes of both versions of the vote collection components. We show the components are both *live* and *safe*, under minimal assumptions.

Finally, we implement prototypes of the vote collection components for both D-DEMOS voting system versions. We measure their performance experimentally, under a variety of election settings, demonstrating their ability to handle thousands of concurrent connections, and thus manage large-scale elections. We also compare the two systems and emphasize the trade-offs between them, regarding security and performance.

To summarize, we make the following contributions:

- We introduce a distributed voting protocol, with a single synchronous round for achieving consensus.
- We enhance this voting protocol and introduce an asynchronous consensus phase. Note that, both systems allow voters to verify their vote was tallied-as-intended without the assistance of special software or trusted devices.
- We provide a security analysis proving the safety and liveness properties of both

protocols.

- We implement prototypes of the systems, measure their performance and demonstrate their ability to handle large-scale elections. Finally, we demonstrate the performance trade-offs between the two versions of the system.

1.2 Description of DEMOS

1.2.1 Overview

In this section, we provide a high level description of DEMOS [72]. This description will become clearer when the distributed version (D-DEMOS) is introduced later in Section 2.4. DEMOS is a state-of-the-art e-voting system that achieves *end-to-end verifiability in the standard model* (i.e., without the *random oracle* assumption) for the first time. In DEMOS, each voter may select 1 out of m options and cast her vote using vote-codes listed in her ballot. Each ballot has two functionally equivalent parts (with a complete list of the m options in each part), instructing the voter to pick one of the two parts at random.

DEMOS in its original centralized form, involves the following entities:

- The voters that use devices assumed to possess limited computational resources; i.e., the devices are not required to be capable of cryptographic operations.
- An *Election Authority* (EA), that administers the entire election procedure. Namely, the EA is responsible for (i) setting up the election, (ii) generating the ballots and delivering them to the voters, (iii) collecting the cast votes, and (iv) computing the tally and publishing the election result.
- A publicly accessible and *consistent Bulletin Board* (BB).

In the **Setup** phase, the EA generates n *double ballots* that consist of two functionally equivalent parts. Each ballot part contains a complete list of the m options that are randomly associated with m randomly generated vote-codes. The EA commits to correct ballot formation by tabulating the ballots in *committed* (encrypted) form, using commitments [105] and zero-knowledge proofs [103]. Finally, the EA distributes the ballots to the voters.

In the **Voting** phase, the voter randomly chooses one of the two parts of the ballot to vote by providing the EA with the vote-code corresponding to her intended option. The voter keeps the unused part and the submitted vote-code for auditing after the election ends.

In the **Tally** phase, the EA *homomorphically* computes the election tally (in committed form). Then, it posts the election result along with the necessary audit information, i.e. the zero-knowledge proofs, the opening of the homomorphic tally commitment, and the openings of all committed values in the unused parts of the voter's ballots.

After the election ends, any party can compute the result and verify the validity of the zero-knowledge proofs. In addition, every voter can verify the correct posting of her ballot in the BB, by comparing the information in the unused ballot part with the respective commitment openings.

1.2.2 Properties of centralized DEMOS

Under the security framework introduced in [72], centralized DEMOS achieves the following properties:

1. **End-to-end verifiability in the standard model** [72, Theorem 4] against adversaries that (a) control the entire election procedure (i.e. the EA is malicious) and (b) adaptively corrupt up to a fixed number of voters, assuming only a consistent BB.
2. **Voter privacy under the Decisional Diffie-Hellman (DDH) assumption** [72, Theorem 5] against adversaries that (a) statically corrupt up to a fixed number of voters, (b) schedule and observe the network trace of all **Cast** protocols, and (c) obtain the personal audit data (the submitted vote-code and the unused ballot part) of every voter. The EA and the BB are assumed to be honest.

1.2.3 Drawbacks in centralized DEMOS design

Despite being a state-of-the-art e-voting system, DEMOS has the following weaknesses in its design:

1. The EA and the BB are *single points of failure* throughout the election procedure.
2. After **Setup**, the EA maintains secret state and must remain live until the election ends. This makes EA a high-profile target whose compromise would severely offset DEMOS's privacy.
3. In the **Voting** phase, the voters do not obtain assurance (i.e., in the form of a receipt message) to verify the correct recording of their cast votes.

2. BACKGROUND

In this section, we provide basic background knowledge required to comprehend the system description in the next section. This includes some voting systems terminology, a quick overview of Interactive Consistency, and a series of cryptographic tools used in the design of D-DEMOS. Finally, we give an overview of the complete D-DEMOS suite of systems, so that the reader can put our vote collection algorithms in context.

2.1 Voting Systems requirements

An ideal electronic voting system would address a specific list of requirements (see [94, 70, 32] for an extensive description). Our system addresses the following requirements:

- **End-to-end verifiability:** the voters can verify that their votes were counted as they intended and any party can verify that the election procedure was executed correctly.
- **Privacy**: a party that does not monitor voters during the voting phase of the election, cannot extract information about the voters' ballots. In addition, a voter cannot prove how she voted to any party that did not monitor her during the voting phase of the election[1].
- **Fault tolerance:** the voting system should be resilient to the faulty behavior of up to a threshold number of components or parts, and be both live and safe.

2.2 Consensus, Agreement, Interactive Consistency

Our study has unveiled a large incoherence in the literature, regarding the terms "Byzantine Agreement" and "Byzantine Consensus". These are often used to refer to the same problem (e.g., [57] and [109]), while others, e.g., [99], use the terms interchangeably, even though these are two distinct problems. There is also inconsistent use of the term "Interactive Consistency", e.g., [90, 102]. To alleviate any confusion and for clarity, we start with some basic definitions.

Synchronous System. There are a priori known bounds regarding message delivery, processing speed, and node clock drifts. Algorithms designed for these systems progress in a series of lock-step *rounds*. The number of rounds, as well as the time required for each round to complete, are built-into the system. A synchronous algorithm can detect crash faults by simply observing missing messages. Typically, the number of rounds required by synchronous algorithms is directly dependent on the number of faults the system can tolerate.

[1]In [72], this property is referred as *receipt-freeness*.

Asynchronous System. There are no bounds regarding message delays and node speeds. Algorithms for these systems also operate in *phases*, however, they tend to be implemented as a collection of message handlers that can be concurrently active for different phases. Thus, it is possible to receive and process an incoming message that refers to a *phase* that is different from the one that the receiving node is currently in. In contrast to synchronous algorithms, these algorithms do not have a given completion time, meaning that they might require more, or even less, than their synchronous counterpart to terminate. Also contrary to a synchronous algorithm, an asynchronous one cannot detect crash faults by observing missing messages, because there is no built-in bound to the time required for a message to be delivered after its transmission.

Byzantine Agreement. Assume a system of n nodes, where a single source n_i has a private value v_i, and the following must be achieved:

- *Agreement:* All non-faulty nodes must agree on the same value.
- *Validity:* If n_i is non-faulty, then the agreed upon value by all non-faulty nodes is v_i.
- *Termination:* All non-faulty nodes must decide on a value.

This problem, also known as the “Byzantine Generals Problem”, was introduced by Lamport et al. [79]. Earlier work has proved there is no solution for the asynchronous case [15], when the source is faulty. Agreement algorithms that tolerate Byzantine failures of (non-source) nodes in asynchronous systems are presented in [17] and [23].

Byzantine Consensus. Assume a system of n nodes, where each node n_i has a private value $v_i \in \{0, 1\}$, and the following must be achieved:

- *Agreement:* All non-faulty nodes must agree on the same value v.
- *Validity:* If all non-faulty nodes have the same initial value v, then the agreed upon value by all non-faulty nodes is v.
- *Termination:* All non-faulty nodes must decide on a value.

One last distinction, regarding consensus protocols, revolves around the agreed upon value. All of the aforementioned protocols are binary consensus protocols, i.e., the agreed upon value is $v \in \{0, 1\}$. In the multi-valued consensus protocol of Correia et al. [41], the set of values V is of arbitrary size (while in [40], the output value is further restricted with *allowed* and *disallowed* sets).

Multi-Valued Consensus is formally defined by the following properties:

- *MVC1 Validity 1:* If all correct processes propose the same value v, then any correct process that decides, decides v.

- *MVC2 Validity 2:* If a correct process decides v, then v was proposed by some process or $v = \perp$.
- *MVC3 Validity 3:* If a value v is proposed only by corrupt processes, then no correct process that decides,decides v.
- *MVC4 Agreement:* No two correct processes decide differently.
- *MVC5 Termination:* Every correct process eventually decides.

Broadcast Primitives. All asynchronous consensus algorithms employ some form of reliable broadcast protocol, where a source *broadcasts* a message m, and every correct node eventually *delivers* m (e.g., via an up-call to the application). Such a broadcast satisfies the following properties ([66]):

- *Validity*: If a non-faulty node broadcasts a message m, then it eventually delivers m.
- *Agreement*: If a non-faulty node delivers a message m, then all non-faulty nodes eventually deliver m.
- *Integrity*: For any message m, every non-faulty node delivers m at most once *iff* m was previously broadcast by $sender(m)$.

Interactive Consistency. Assume a system of n nodes, where each node n_i has a private value v_i, and the following must be achieved:

- *Agreement:* All non-faulty nodes must agree on the same vector of values V
- *Validity:* If the private value of the non-faulty node n_i is v_i, then all non-faulty nodes agree on $V[i] = v_i$.
- *Termination:* All non-faulty nodes must decide on a vector V.

In our D-DEMOS/IC system, we use the *IC,BC-RBB* algorithm from [46], which achieves IC using a single synchronous round. This algorithm uses two phases to complete. The synchronous *Value Dissemination Phase* comes first, aiming to disperse the values across nodes. Consequently, an asynchronous *Result Consensus Phase* starts, which results in each honest node holding a vector with every honest node's slot filled with the corresponding value. We include the full definition of this algorithm and the rationale behind its design in Appendix A.

2.3 Cryptographic tools

2.3.1 Additively homomorphic commitments

To achieve integrity against a malicious election authority, our D-DEMOS utilizes lifted ElGamal [54] over elliptic curves as a *non-interactive commitment scheme* that achieves the following properties:

1. *Perfectly binding*: no adversary can open a commitment Com(m) of m to a value other than m.
2. *Hiding*: there exists a constant $c < 1$ so that the probability that a commitment Com(m) to m leaks information about m to an adversary running in $O(2^{\lambda^c})$ steps is no more than negl(λ) (i.e., it is negligible).
3. *Additively homomorphic*: $\forall m_1, m_2$, we have that Com$(m_1) \cdot$ Com$(m_2) =$ Com$(m_1 + m_2)$.

2.3.2 Zero-knowledge Proofs

D-DEMOS's security requires the election authority to show the correctness of the election setup to the public without compromising privacy. We enable this kind of verification with the use of zero-knowledge proofs. In a zero-knowledge proof, the prover is trying to convince the verifier that a statement is true, without revealing any information about the statement apart from the fact that it is true [103]. More specifically, we say an interactive proof system has the *honest-verifier zero-knowledge (HVZK)* property if there exists a probabilistic polynomial time simulator $\mathcal{S}$ that, for any given challenge, can output an accepting proof transcript that is distributed indistinguishable from the real transcript between an honest prover and an honest verifier. Here, we adopt Chaum-Pedersen zero-knowledge proofs [30], which belong in the special class of Σ protocols (i.e., 3-move public-coin special HVZK proofs), allowing the Election Authority to show that the content inside each commitment is a valid option encoding.

2.4 D-DEMOS Overview

The work described in this thesis has been performed in the context of a project aiming to produce a completely distributed, state-of-the-art, end-to-end verifiable e-voting system, namely the D-DEMOS system. Specifically, this thesis focuses on distributed vote collection for the D-DEMOS system. To help the reader understand better how exactly vote collection fits in the complete system, we provide here an overview of the full D-DEMOS's design goals and operation.

2.4.1 Problem Definition and Goals

D-DEMOS supports an *election* with a single *question* and m *options*, for a voter population of size n, where voting takes place between a certain *begin* and *end* time (the *voting hours*), and each voter may select a single *option*.

The major goals in designing D-DEMOS are the following three:

1) It has to be end-to-end verifiable, so that anyone can verify the complete election process. Additionally, voters should be able to outsource auditing to third parties, without revealing their voting choice.

2) It has to be fault-tolerant, so that an attack on system availability and correctness is hard to mount.

3) Voters should not have to trust the terminals they use to vote, as such devices may be malicious. Instead, voters should be assured their vote was recorded, without disclosing any information on how they voted to the malicious entity controlling their device.

2.4.2 System overview

D-DEMOS employs an election setup component, which is called the Election Authority (*EA*), to alleviate the voter from employing any cryptographic operations. The *EA* initializes all other system components, and then gets immediately destroyed to preserve privacy. The *Vote Collection* (*VC*) subsystem collects the votes from the voters during election hours, and assures them their vote was *recorded-as-cast*. The *Bulletin Board* (*BB*) subsystem, which is a public repository of all election-related information, is used to hold all ballots, votes, and the result, either in encrypted or plain form, allowing any party to read from the *BB* and verify the complete election process. The *VC* subsystem uploads all votes to the *BB* at election end time. Finally, D-DEMOS's design includes *trustees*, who are persons entrusted with managing all actions needed until result tabulation and publication, including all actions supporting end-to-end verifiability. *Trustees* hold the keys to uncover any information hidden in the *BB*, and D-DEMOS uses threshold cryptography to make sure a malicious minority cannot uncover any secrets or corrupt the process.

We outline the interactions between these subsystems and the actors in Figure 2.1. In the following paragraphs, we explain these interactions in more detail.

D-DEMOS starts with the *EA* generating initialization data for every component of the system. The *EA* encodes each election option, and *commits* to it using a commitment scheme, as described below. It encodes the i-th option as $\vec{e_i}$, a unit vector where the i-th element is 1 and the remaining elements are 0. The commitment of an option encoding is a vector of (lifted) ElGamal ciphertexts [53] over elliptic curve, that element-wise encrypts a unit vector. Note that this commitment scheme is also additively homomorphic, i.e., the commitment of $e_a + e_b$ can be computed by component-wise multiplying the corresponding commitments of e_a and e_b. The *EA* then creates a votecode and a receipt for each option.

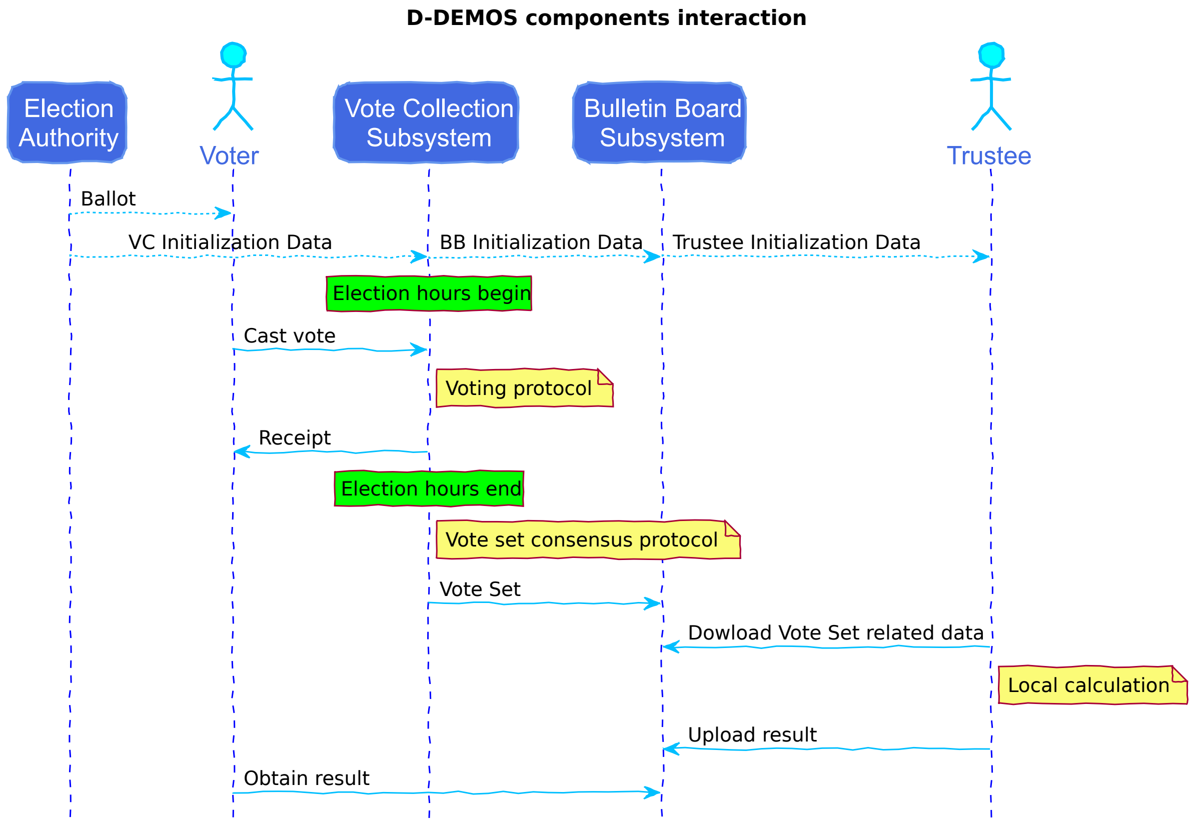


Figure 2.1: High-level diagram of interactions between subsystems and actors. Subsystems are distributed systems of their own, but are depicted as a unified entity in this diagram. Time is depicted flowing downwards.

Subsequently, the *EA* prepares one ballot for each voter, with two functionally equivalent parts. Each part contains a list of options, along with their corresponding vote codes and receipts. We consider ballot distribution to be outside the scope of this paper, but we do assume ballots, after being produced by the *EA*, are distributed in a secure manner to each voter; thus only each voter knows the vote codes listed in her ballot. In the D-DEMOS system, vote codes are not stored in clear form anywhere besides the voter's ballot.

D-DEMOS's *VC* subsystem collects the votes from the voters during election hours, by accepting up to one vote code from each voter. The *EA* initializes each *VC* node with the vote codes and the receipts of the voters' ballots. However, it hides the vote codes, using a simple commitment scheme based on symmetric encryption of the plaintext along with a random salt value. This way, each *VC* node can verify if a vote code is indeed part of a specific ballot, but cannot recover any vote code until the voter actually chooses to disclose it. Additionally, we secret-share each receipt across all *VC* nodes using an $(N-f,N)$-VSS (verifiable secret-sharing) scheme with trusted dealer [105], making sure that a receipt can be recovered and posted back to the voter only when a strong majority of *VC* nodes participates successfully in the voting protocol. With this design, the system adheres to the following contract with the voters: *Any honest voter who receives a valid* *receipt from a Vote Collector node, is assured her vote will be published on the BB, and thus it will be included in the election tally*.

The voter selects one part of her ballot at random, and posts her selected vote code to one of the *VC* nodes. When she receives a receipt, she compares it with the one on her ballot corresponding to the selected vote code. If it matches, she is assured her vote was correctly recorded and will be included in the election tally. The other part of her ballot, the one not used for voting, will be used for auditing purposes. This design is essential for verifiability, in the sense that the *EA* cannot predict which part a voter may use, and the unused part will betray a malicious *EA* with $\frac{1}{2}$ probability per audited ballot.

The second distributed subsystem is the *BB*, which is a replicated service of isolated nodes. Each *BB* node is initialized from the *EA* with vote codes and associated option encodings in committed form (again, for vote code secrecy), and each *BB* node provides public access to its stored information. At election end time, *VC* nodes run the Vote Set Consensus protocol (sections 3.6.1 and 3.7.1, which guarantees all *VC* nodes agree on a single set of voted vote codes. After agreement, each *VC* node uploads this set to every *BB* node, which in turn publishes this set once it receives the same copy from enough *VC* nodes.

The third distributed subsystem is a set of *trustees*, who are persons entrusted with managing all actions needed after vote collection, until result tabulation and publication; this includes all actions supporting end-to-end verifiability. Secrets that may uncover information in the *BB* are shared across *trustees*, making sure malicious *trustees* under a certain threshold cannot uncover and disclose sensitive information. We use Pedersen's Verifiable linear Secret Sharing (VSS) [101] to split the election data among the *trustees*. In a (k, n)-VSS, at least k shares are required to reconstruct the original data, and any collection of less than k shares leaks no information about the original data. Moreover, Pedersen's VSS is additively homomorphic, i.e., one can compute the share of $a + b$ by adding the share of a and the share of b respectively. This approach allows *trustees* to perform homomorphic "addition" on the option-encodings of cast vote codes, and contribute back a share of the opening of the homomorphic "total". Once enough *trustees* upload their shares of the "total", the election tally is uncovered and published at each *BB* node.

To ensure voter privacy, the system cannot reveal the content inside an option-encoding commitment at any point. However, a malicious *EA* might put an arbitrary value (say 9000 votes for option 1) inside such a commitment, causing an incorrect tally result. To prevent this, D-DEMOS utilizes the Chaum-Pedersen zero-knowledge proof [30], allowing the *EA* to show that the content inside each commitment is a valid option encoding, without revealing its actual content. Namely, the prover uses a Sigma OR proof to show that each ElGamal ciphertext encrypts either 0 or 1, and the sum of all elements in a vector is 1. The zero knowledge proof is organized as follows. First, the *EA* posts the initial part of the proofs on the *BB*. Second, during the election, each voter's A/B part choice is viewed as a source of randomness, $0/1$, and all the voters' choices are collected and used as the challenge of the zero knowledge proof. Finally, the *trustees* will jointly produce the final part of the proofs and post it on the *BB* before the opening of the tally. Hence, everyone can verify those proofs on the *BB*. We omit the zero-knowledge proof components in this thesis and refer the interested reader to [30] for details.

This design allows any voter to read information from the *BB*, combine it with her private ballot, and verify her ballot was included in the tally. Additionally, any third-party auditor can read the *BB* and verify the complete election process. As the number of auditors increases, the probability of election fraud going undetected diminishes exponentially. For example, even if only 10 people audit, with each one having $\frac{1}{2}$ probability of detecting ballot fraud, the probability of ballot fraud going undetected is only $\frac{1}{2}^{10} = 0.00097$. Thus, even if the *EA* is malicious and, e.g., tries to point all vote codes to a specific option, this faulty setup will be detected because of the end-to-end verifiability of the complete system.

The D-DEMOS suite comprises two different versions of the voting system, with different performance and security trade-offs. In the first version, called *D-DEMOS/IC*, Vote Set Consensus is realized by an algorithm achieving Interactive Consistency, and thus requiring synchronization. The second version, *D-DEMOS/Async*, uses an asynchronous binary consensus algorithm for Vote Set Consensus, and thus is completely asynchronous. The remainder of this thesis focuses specifically on vote collection. The performance trade-offs between the two vote collection approaches are analyzed in Section 6.

3. SYSTEM DESCRIPTION

3.1 System model

We assume a fully connected network, where each node can reach any other node with which it needs to communicate. The network can drop, delay, duplicate, or deliver messages out of order. However, we assume messages are eventually delivered, provided the sender keeps retransmitting them. For all nodes, we make no assumptions regarding processor speeds. We also assume communication between *VC* nodes happens via private and authenticated channels.

For both versions of our system, we assume the clocks of *VC* nodes are synchronized with real world time; this is needed to prohibit voters from casting votes outside election hours. For the safety of *D-DEMOS/Async* version, we make no further timing assumptions. To ensure liveness, we assume the adversary cannot delay communication between honest nodes above a certain threshold.

We consider arbitrary (Byzantine) failures, because we expect our system to be deployed across separate administrative domains and we wish to tolerate human-factor faults (e.g., passwords obtained via social engineering). Finally, we assume the adversary cannot violate the security of the underlying cryptographic primitives.

3.2 Extracting vote collection from the EA

We observe that vote collection is not an intrinsic function of the Election Authority (*EA*) component of DEMOS. In fact, the *EA* should be limited to setup functionality only; that is, to generate the initialization data for all other system components and participants. After that, and before the election starts, the *EA* should be destroyed, decreasing the system's attack surface regarding privacy.

Thus, we introduce the Vote Collection (*VC*) subsystem and define its functionality and interaction with the remaining system components as follows:

1. *VC* is initialized from the *EA*.
2. *VC* receives votes from voters.
3. *VC* accepts at most one vote from each voter.
4. *VC* provides a receipt back to the voter when the vote is valid.
5. On election end time, *VC* uploads the set of cast vote codes to the *BB*.

3.3 Election Authority (*EA*)

With the *VC* subsystem extracted, the *EA* is reduced to only produce the initialization data for each election entity in the setup phase. To enhance the system robustness, we let the *EA* generate all the public/private key pairs for all the system components (except voters) without relying on external PKI support. We use zero knowledge proofs to ensure the correctness of all the initialization data produced by the *EA*.

3.3.1 Voter Ballots

The *EA* generates one ballot ballot_ℓ for each voter ℓ, and assigns a unique 64-bit serial-no_ℓ to it. As shown below, each ballot consists of two parts: Part A and Part B. Each part contains a list of m $\langle\text{vote-code}, \text{option}, \text{receipt}\rangle$ tuples, one tuple for each election option. The *EA* generates the vote-code as a 128-bit random number, unique within the ballot, and the receipt as 64-bit random number.

serial-no_ℓ		
	Part A	
$\text{vote-code}_{\ell,1}$	$\text{option}_{\ell,1}$	$\text{receipt}_{\ell,1}$
...	...	...
$\text{vote-code}_{\ell,m}$	$\text{option}_{\ell,m}$	$\text{receipt}_{\ell,m}$
	Part B	
$\text{vote-code}_{\ell,1}$	$\text{option}_{\ell,1}$	$\text{receipt}_{\ell,1}$
...	...	...
$\text{vote-code}_{\ell,m}$	$\text{option}_{\ell,m}$	$\text{receipt}_{\ell,m}$

3.3.2 *BB* initialization data

The initialization data for all *BB* nodes is identical, and each *BB* node publishes its initialization data immediately. The *BB*'s data is used to show the correspondence between the vote codes and their associated cryptographic payload. This payload comprises the committed option encodings, and their respective zero knowledge proofs of valid encoding (first move of the prover), as described in section 2.4.2. However, the vote codes must be kept secret during the election, to prevent the adversary from “stealing” the voters' ballots and using the stolen vote codes to vote. To achieve this, the *EA* first randomly picks a 128-bit key, msk, and encrypts each vote-code using AES-128-CBC with random initialization vector (AES-128-CBC$) encryption, denoted as $[\text{vote-code}]_{\text{msk}}$. Each *BB* node is given $H_{\text{msk}} \leftarrow SHA256(\text{msk}, \text{salt}_{\text{msk}})$ and salt_{msk}, where salt_{msk} is a fresh 64-bit random salt. Hence, each *BB* node can be assured the key it reconstructs from *VC* key-shares (see below) is indeed the key that was used to encrypt these vote-codes.

The rest of the *BB* initialization data is as follows: for each serial-no_ℓ, and for each ballot part, there is a *shuffled* list of $\left\langle [\text{vote-code}_{\ell,\pi_\ell^X(j)}]_{\text{msk}}, \text{payload}_{\ell,\pi_\ell^X(j)} \right\rangle$ tuples, where $\pi_\ell^X \in S_m$ is a random permutation (X is A or B).

$(H_{\text{msk}}, \text{salt}_{\text{msk}})$	
serial-no$_\ell$	
	Part A
$[\text{vote-code}_{\ell,\pi_\ell^A(1)}]_{\text{msk}}$	$\text{payload}_{\ell,\pi_\ell^A(1)}$
$\vdots$	$\vdots$
$[\text{vote-code}_{\ell,\pi_\ell^A(m)}]_{\text{msk}}$	$\text{payload}_{\ell,\pi_\ell^A(m)}$
	Part B
$[\text{vote-code}_{\ell,\pi_\ell^B(1)}]_{\text{msk}}$	$\text{payload}_{\ell,\pi_\ell^B(1)}$
$\vdots$	$\vdots$
$[\text{vote-code}_{\ell,\pi_\ell^B(m)}]_{\text{msk}}$	$\text{payload}_{\ell,\pi_\ell^B(m)}$

We shuffle the list of tuples of each part to ensure voter's privacy. This way, nobody can guess the voter's choice from the position of the cast vote-code in this list.

3.3.3 *VC* initialization data

The *EA* uses an $(N_v - f_v, N_v)$-VSS (Verifiable Secret-Sharing) scheme to split msk and every receipt$_{\ell,j}$ into N_v shares, denoted as $(\|\text{msk}\|_1, \ldots, \|\text{msk}\|_{N_v})$ and $(\|\text{receipt}_{\ell,j}\|_1, \ldots,$ $\|\text{receipt}_{\ell,j}\|_{N_v})$ respectively. For each vote-code$_{\ell,j}$ in each ballot, the *EA* also computes $H_{\ell,j} \leftarrow SHA256(\text{vote-code}_{\ell,j}, \text{salt}_{\ell,j})$, where salt$_{\ell,j}$ is a 64-bit random number. $H_{\ell,j}$ allows each *VC* node to validate a vote-code$_{\ell,j}$ individually (without network communication), while still keeping the vote-code$_{\ell,j}$ secret. To preserve voter privacy, these tuples are also shuffled using π_ℓ^X.

The initialization data for VC_i is structured as below:

$\|\text{msk}\|_i$	
serial-no$_\ell$	
	Part A
$(H_{\ell,\pi_\ell^A(1)}, \text{salt}_{\ell,\pi_\ell^A(1)})$	$\|\text{receipt}_{\ell,\pi_\ell^A(1)}\|_i$
$\ldots$	$\ldots$
$(H_{\ell,\pi_\ell^A(m)}, \text{salt}_{\ell,\pi_\ell^A(m)})$	$\|\text{receipt}_{\ell,\pi_\ell^A(m)}\|_i$
	Part B
$(H_{\ell,\pi_\ell^B(1)}, \text{salt}_{\ell,\pi_\ell^B(1)})$	$\|\text{receipt}_{\ell,\pi_\ell^B(1)}\|_i$
$\ldots$	$\ldots$
$(H_{\ell,\pi_\ell^B(m)}, \text{salt}_{\ell,\pi_\ell^B(m)})$	$\|\text{receipt}_{\ell,\pi_\ell^B(m)}\|_i$

3.3.4 *Trustee* initialization data

The *EA* uses (h_t, N_t)-VSS to split the opening of encoded option commitments Com$(\vec{e_i})$ into N_t shares, denoted as $(\|\underline{\vec{e_i}}\|_1, \ldots, \|\underline{\vec{e_i}}\|_{N_t})$.

The initialization data for Trustee$_i$ is structured as below:

serial-no$_\ell$	
Part A	
Com$(\vec{e}_{\pi_\ell^A(i)})$	$\|\underline{\vec{e}_{\pi_\ell^A(i)}}\|_\ell$
$\ldots$	$\ldots$
Part B	
Com$(\vec{e}_{\pi_\ell^B(i)})$	$\|\underline{\vec{e}_{\pi_\ell^B(i)}}\|_\ell$
$\ldots$	$\ldots$

Similarly, the state of zero knowledge proofs for ballot correctness is shared among the *trustees* using (h_t, N_t)-VSS. For further details, we refer the interested reader to [30].

3.4 Vote Collection Subsystem

We design the *VC* subsystem as a distributed system of N_v cooperating nodes, tolerating up to f_v Byzantine faults, where $f_v < N_v/3$. Note that, we also tolerate the collusion of an arbitrary number of malicious voters with the malicious *VC* nodes. *VC* nodes have private communication channels to each other, and a public (unsecured) channel for the voters.

We modify the data generation process of DEMOS's *EA*, by adding the following two steps while generating voter's ballots:

1. The (random) vote-code corresponding to each election option is provided in committed form to each *VC* node.
2. A receipt is generated for each vote code, which is itself a random number. The receipt is secret shared across *VC* nodes with a Verifiable Secret Sharing (VSS) scheme. Each *VC* node receives one of these shares.

At step 1, the commitment scheme used hashes the plain text message along with a salt. The salt is provided along with the committed form to each *VC* node, while the opening of the commitment is the vote-code itself.

Before going into detail in the design of the Vote Collection subsystem, we give an overview of its use. *VC* nodes are initialized from the *EA* (as above). The voter receives her ballot also from the *EA*, along with the addresses of the *VC* nodes. During the election hours, *VC* nodes run the *voting protocol*, as depicted in Figure 3.1.

For this protocol to start, the voter selects one part of her ballot at random, and posts her selected vote code to one of the *VC* nodes. The *VC* node that receives her vote validates it,

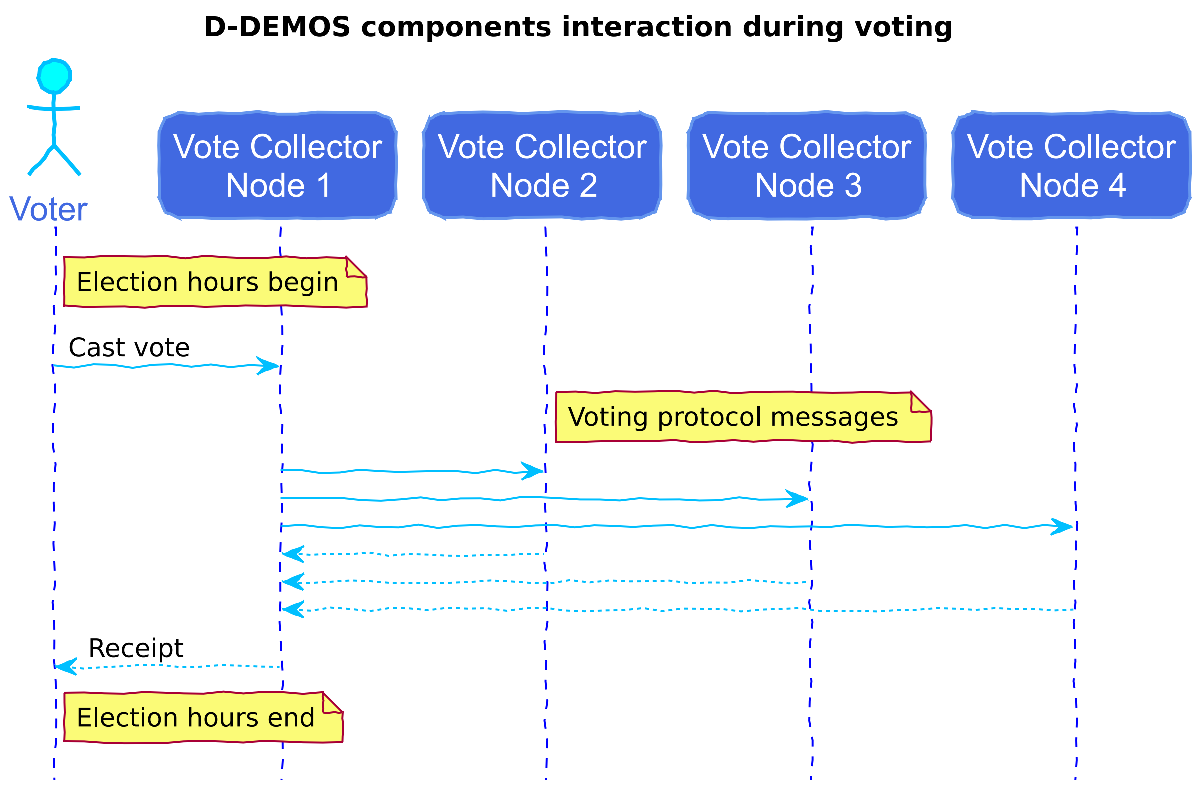


Figure 3.1: High-level diagram of component interactions during the voting phase. Message exchanges between *VC* nodes are simplified for this diagram. In this diagram, there are four *VC* nodes, tolerating up to one fault.

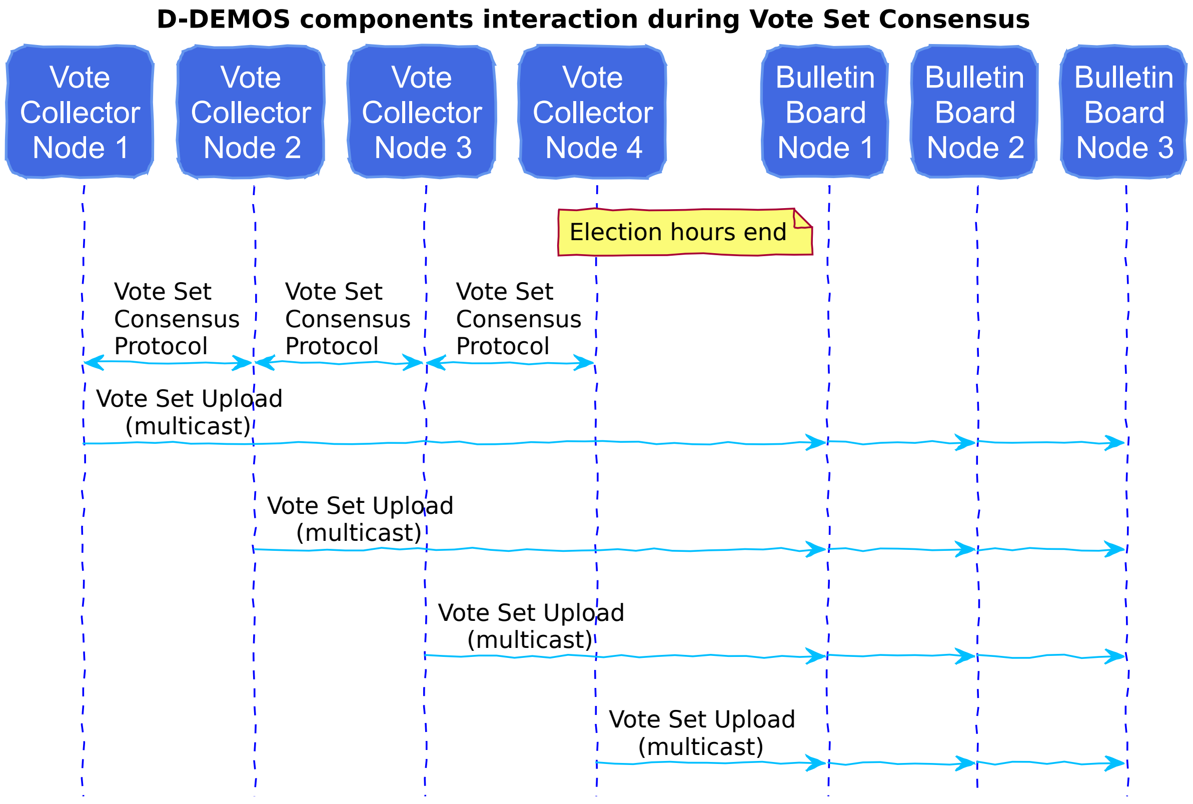

D-DEMOS components interaction during Vote Set Consensus
Vote Collector Node 1
Vote Collector Node 2
Vote Collector Node 3
Vote Collector Node 4
Bulletin Board Node 1
Bulletin Board Node 2
Bulletin Board Node 3
Election hours end
Vote Set Consensus Protocol
Vote Set Consensus Protocol
Vote Set Consensus Protocol
Vote Set Upload (multicast)
Vote Set Upload (multicast)
Vote Set Upload (multicast)
Vote Set Upload (multicast)

Figure 3.2: High-level diagram of component interactions during the vote set consensus phase. Four *VC* nodes and three *BB* nodes are shown, where each subsystem tolerates one fault. After agreeing on a single Vote Set S, each *VC* node uploads S to every *BB* node. Messages are simplified for this diagram.

interacts with the other *VC* nodes to reconstruct the receipt from the shares spread across the *VC* nodes, and posts it back to the voter. When she receives a receipt, she compares it with the one on her ballot corresponding to the selected vote code. If it matches, she is assured her vote was correctly recorded and will be included in the election tally. The other part of her ballot, the one not used for voting, will be used for auditing purposes. This design is essential for verifiability, in the sense that the *EA* cannot predict which part a voter may use, and the unused part will betray a malicious *EA* with $\frac{1}{2}$ probability per audited ballot.

At election end time, *VC* nodes run our Vote Set Consensus protocol, illustrated in Figure 3.2, which guarantees all *VC* nodes agree on a single set of voted vote codes. After agreement, each *VC* node uploads this set to every *BB* node, which in turn publishes this set once it receives the same copy from enough $(f_v + 1)$ *VC* nodes.

3.5 Voter algorithm

We expect the voter, who has received a ballot from the *EA*, to know the URLs of all *VC* nodes. To vote, she picks one part of the ballot at random, selects the vote code representing her chosen option, and enters the vote casting loop. In this loop, she selects a *VC* node at random and posts the vote code. She then waits up to a configuration-specific *timeout*. If there is a response within the *timeout*, and it is a valid receipt, she exits the loop. Otherwise, she selects the next *VC* node and repeats the vote-casting process.

Algorithm 1 Voter algorithm

```
1: procedure Vote(ballot, selection, VC-List):
2:     vote-code := ballot.lines[selection].vote-code
3:     VC-List := random-shuffle(VC-List)
4:     repeat
5:         VC-Node := VC-List.popFirst()
6:         send VC-Node VOTE⟨ballot.serial-no, vote-code⟩
7:         wait up to timeout for receipt
8:         VC-List.pushBack(VC-Node)
9:     until receipt == ballot.lines[selection].receipt
```

We depict this process in Algorithm 1, where the voter shuffles the list of *VC* nodes, pops the first, tries to vote, and adds the popped *VC* node to the back of the list again. This way, the list of *VC* nodes may be traversed multiple times. Our *VC* subsystem voting protocol allows for this as, when the receipt is already generated, the *responder VC* node simply sends it back to the voter immediately.

3.6 Synchronous *VC* subsystem

In this section, we describe the synchronous version of our vote collection subsystem. We first present the voting protocol, which is used by the voter to cast her vote. This protocol takes the equivalent of a single round-trip between two *VC* nodes to produce the receipt, and uses no explicit signatures. Then, we present the vote set consensus protocol, which runs on election-end time and guarantees that all honest nodes agree on a single set of votes. This protocol uses an Interactive Consistency algorithm to achieve agreement between *VC* nodes. Finally, we provide proofs of liveness and safety for both protocols.

3.6.1 System description

The algorithms implementing our *D-DEMOS/IC voting* protocol are presented in Algorithm 2. For simplicity, we present our algorithms operating for a single election.

The *voting* protocol starts when a voter submits a `VOTE`⟨serial-no, vote-code⟩ message to a *VC* node. We call this node the *responder*, as it is responsible for delivering the receipt to the voter. The *VC* node confirms the current system time is within the defined election

Algorithm 2 Vote Collector voting protocol algorithms for D-DEMOS/IC

```
procedure on VOTE(serial-no, vote-code) from $source$:
    if $SysTime()$ between $start$ and $end$
        if $b$ :=locateBallot(serial-no)
            if $b$.status = NotVoted
                if $l$ := $b$.VerifyVoteCode(vote-code)
                    $b$.voter := $source$              ▷ Mark the current node as the responder for source
                    $b$.status := Pending
                    $b$.used-vc := vote-code
                    sendAll(VOTE_P⟨serial-no, vote-code, $l$.share⟩)
            else if $b$.status = Voted ∧ $b$.used-vc = vote-code
                send ($source$, b.receipt)
procedure on VOTE_P(serial-no, vote-code, share) from $source$:
    if $SysTime()$ between $start$ and $end$)
        $b$ :=locateBallot(serial-no)
        if $b$.status = NotVoted
            if $l$ := $b$.VerifyVoteCode(vote-code)
                if $validShare$(share, serial-no, vote-code)
                    $b$.status := Pending
                    $b$.used-vc := vote-code
                    $b$.lrs.Append(share)              ▷ Update list of receipt shares
                    sendAll(VOTE_P⟨serial-no, vote-code, $l$.share⟩ )
        else if $b$.status = Pending ∧ $b$.used-vc = vote-code
            if $validShare$(share, serial-no, vote-code)
                $b$.lrs.Append(share)                  ▷ Update list of receipt shares
                if size($b$.lrs) = $N_v - f_v$
                    $b$.receipt := Rec($b$.lrs)
                    $b$.status := Voted
                    if $b$.voter                       ▷ If this is the responder node
                        $send$($b$.voter, $b$.receipt)
function Ballot::VerifyVoteCode(vote-code)
    for $l := 1$ to ballot_lines do
        if lines[$l$].hash = $h$(vote-code||lines[$l$].salt)
            return lines[$l$]
    return ⊥
```

hours, and locates the ballot with the specified serial-no. It also verifies this ballot has not been used for this election, either with the same or a different vote code. Then, it compares the vote-code against every hashed vote code in each ballot line, until it locates the correct entry. Subsequently, it obtains from its local database the receipt-share corresponding to the specific vote-code. Next, it marks the ballot as pending for the specific vote-code. Finally, it multicasts a `VOTE_P`$\langle$serial-no, vote-code, receipt-share$\rangle$ message to all *VC* nodes, disclosing its share of the receipt. In case the located ballot is marked as voted for the specific vote-code, the *VC* node sends the stored receipt to the voter without any further interaction with other *VC* nodes.

Each *VC* node that receives a `VOTE_P` message, first validates the received receipt-share according to the verifiable secret sharing scheme used. Then, it performs the same validations as the responder, and multicasts another `VOTE_P` message (only once), disclosing its share of the receipt. When a node collects $h_v = N_v - f_v$ valid shares, it uses the verifiable secret sharing reconstruction algorithm to reconstruct the receipt (the secret) and marks the ballot as voted for the specific vote-code. Additionally, the *responder* node sends this receipt back to the voter.

A message flow diagram of our *voting* protocol is depicted in Figure 3.3. As is evident from the diagram, the time from the multicast of the first `VOTE_P` message until collecting all receipt shares, is only slightly longer than a single round-trip between two *VC* nodes.

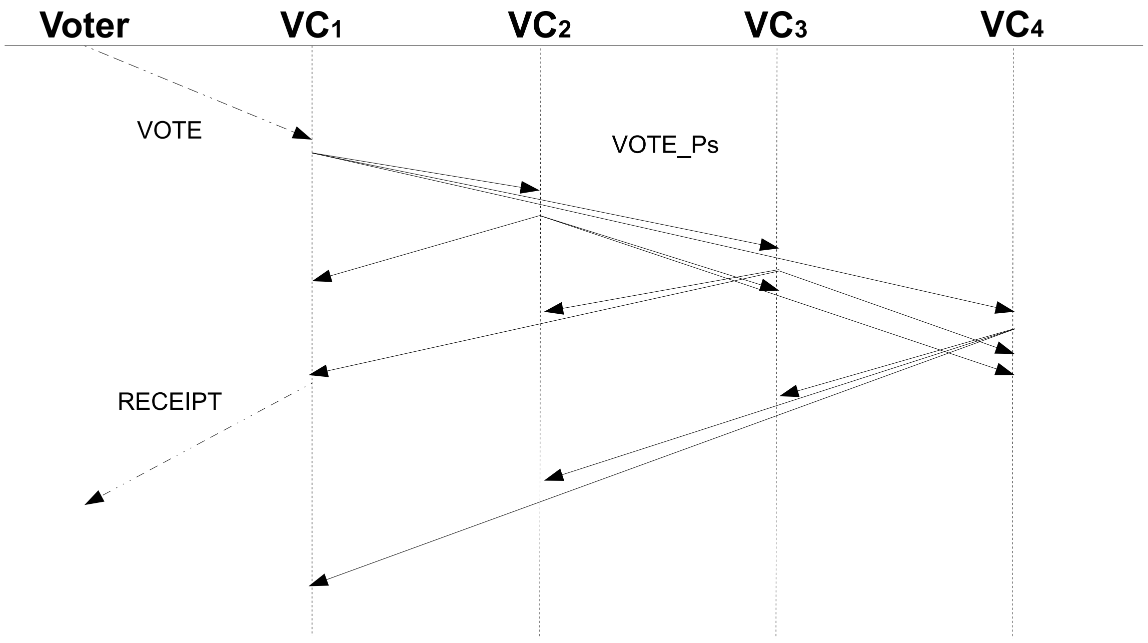


Figure 3.3: Diagram of message exchanges for a single vote during the D-DEMOS/IC vote collection phase.

At election end time, each *VC* node stops processing `VOTE` and `VOTE_P` messages, and initiates the *vote-set consensus* protocol. It creates a set VS_i of $\langle$serial-no, vote-code$\rangle$ tuples, including all *voted* and *pending* ballots. Then, it participates in the Interactive Consistency (IC) protocol of [46], with this set. At the end of IC, each node contains a vector $\langle VS_1, \ldots, VS_n \rangle$ with the Vote Set of each node, and follows the algorithm of Figure 3.4. Step 1 makes sure any ballot with multiple submitted vote codes is discarded. Since vote

Cross-tabulate $\langle VS_1, \ldots, VS_n \rangle$ per ballot, creating a list of vote codes for each ballot. Perform the following actions for each ballot:

1. If the list contains two or more distinct vote codes, mark the ballot as NotVoted and exit.
2. If a vote code vc_a appears at least $N_v - 2f_v$ times in the list, mark the ballot as Voted for vc_a and exit.
3. Otherwise, mark the ballot as NotVoted and exit.

Figure 3.4: High level description of algorithm after IC.

codes are private, and cannot be guessed by malicious vote collectors, the only way for multiple vote codes to appear is if malicious voters are involved, against whom our system is not obliged to respect our *contract*.
With a single vote code remaining, step 2 considers the threshold above which to consider a ballot as voted for a specific vote code. We select the $N_v - 2f_v$ threshold for which we are certain that even the following extreme scenario is handled. If the *responder* is malicious, submits a receipt to an honest voter, but denies it during *vote-set consensus*, the remaining $N_v - 2f_v$ honest *VC* nodes that revealed their receipt shares for the generation of the receipt, are enough for the system to accept the vote code (receipt generation requires $N_v - f_v$ nodes, of which f_v may be malicious, thus $N_v - 2f_v$ are necessarily honest).

Finally, step 3 makes sure vote codes that occur less than $N_v - 2f_v$ times are discarded. Under this threshold, there is no way a receipt was ever generated.

At the end of this algorithm, each node submits the resulting set of *voted* $\langle$serial-no, vote-code$\rangle$ tuples to each *BB* node, which concludes its operation for the specific election.

3.6.2 Proofs

We now prove the *liveness* and *safety* of both phases of our synchronous vote collection subsystem. *Liveness* is the property which assures us that an algorithm eventually terminates, producing its output. *Safety*, on the other hand, assures us that an algorithm does not exhibit erroneous behavior, i.e., it does not produce wrong output.

We consider a D-DEMOS/IC system, where voters run Algorithm 1 and *VC*nodes run Algorithm 2 for voting, and the algorithm of section 3.6.1 for Vote Set Consensus, under the system model of Section 3.1, where fault tolerance thresholds of Section 3.4 hold, and the adversary cannot guess any vote code inside honest voters' ballots. We assume the adversary can control and coordinate all faulty components, i.e., all corrupt voters and all corrupt *VC* nodes.

In all remaining text of this section, we refer to an *honest voter*, against whom we prove liveness and safety. We define the *honest voter* as one that follows Algorithm 1 of Section 3.5 to the letter, and above all selects and discloses only one vote code from her ballot, keeping all other vote codes confidential until the election ends.

We also refer to a *correct VC* node, which we define as one that is not under the control of the adversary and which does not exhibit faulty behavior.

To further clarify our wording, we explain the way a *VC* node maintains a status for each ballot. This status may be `NotVoted`, which is set when the *VC* node has not started any instance of the voting protocol for this ballot, for any vote code. The status may be `Pending` for vote code `vcode`, which means the voting protocol has started for `vcode`, but the receipt cannot be reconstructed yet. Finally, the status can be `Voted` for vote code `vcode`, which means the receipt for `vcode` has been reconstructed. Note that statuses `Pending` and `Voted` are always accompanied by a specific vote code, which we denote by writing "the status is `Voted` for vote code `vcode`".

3.6.2.1 Liveness

3.6.2.1.1 Liveness of the voting algorithm.

We first prove a few useful lemmas, that will lead to the proof of the main theorem.

Lemma 1. *If an honest voter's ballot with tag* `serno`*, at a correct VC node VC_i, has its status changed to* `Pending` *for vote code* `vcode`*, and all voting protocol messages for this instance are delivered before election end-time, then this ballot's status will eventually change to* `Voted` *for* `vcode` *at VC_i.*

Proof. The ballot's status is changed to `Pending` while processing either the `VOTE` or `VOTE_P` messages, at node VC_i. In both cases, this change is accompanied with the multicast of a `VOTE_P` message to all other *VC* nodes, disclosing VC_i's receipt share. Because we assume a lossless network, this `VOTE_P` message will be delivered to, and be processed by all correct nodes, as per Algorithm 2.

If a receiving correct node VC_j, has the ballot marked as `NotVoted`, processing of VC_i's `VOTE_P` message results in marking the ballot as `Pending` for `vcode` and disclosing VC_j's receipt share via a new `VOTE_P` message. This happens because the submitted $\langle \texttt{serno}, \texttt{vcode} \rangle$ tuple was already found valid by VC_i, and thus will also be found valid by VC_j.

If VC_j has ballot with tag `serno` marked as anything other than `NotVoted`, it has to be so for vote code `vcode`, since the voter is honest and the adversary cannot guess and submit any other vote code for this ballot. As such, VC_j must have already sent its `VOTE_P` message, and VC_i will receive it eventually.

In both of the above cases, VC_j's `VOTE_P` message will be delivered at node VC_i, since the network is lossless. As the above happens for every correct node VC_j, VC_i eventually receives all receipt shares of all correct nodes, via the corresponding `VOTE_P` messages. Because the VSS scheme's secret recovery threshold $(N_v - f_v)$ is equal to the minimum correct node count, it is guaranteed that the receipt shares are enough to reconstruct the receipt and mark the ballot as `Voted` at node VC_i (passing the threshold check at line 25 of Algorithm 2).

Thus, the ballot will be marked as `Voted` at node VC_i. □

Lemma 1 shows that a status of `Pending` is temporary for honest voters' ballots, as the protocol ensures that such a ballot's status will eventually switch to `Voted`.

Lemma 2. *If an honest voter posts $\langle \texttt{serno}, \texttt{vcode} \rangle$ to a correct VC node VC_i after election start-time, and the status of ballot with tag `serno` is `NotVoted` at VC_i, and all voting protocol messages for this instance are delivered before election end-time, then the status of this ballot will become `Voted` for `vcode` across all correct VC nodes.*

Proof. At VC_i, the status changes immediately to `Pending` and the `VOTE_P` multicast is performed. Because of Lemma 1, it will eventually change to `Voted` for `vcode`. `VOTE_P` messages are delivered at all their destinations, because we have assumed a lossless network. At each correct *VC* node that VC_i's `VOTE_P` is delivered, the status of the ballot may be:

- `NotVoted`, in which case it will change to `Pending` for `vcode`.
- `Pending`, but for the same vote code `vcode`, as the voter is honest and the adversary cannot guess the remaining vote codes in the voter's ballot. In this case, because of Lemma 1, the status will eventually change to `Voted` for `vcode`.
- `Voted`, but for the same vote code `vcode`, as the voter is honest and the adversary cannot guess the remaining vote codes in the voter's ballot.

In all above cases, the status of the ballot either immediately is, or eventually becomes `Voted` for vote code `vcode`. □

Lemma 3. *If an honest voter posts $\langle \texttt{serno}, \texttt{vcode} \rangle$ to a correct VC node VC_i before election end-time, and the status of ballot with tag `serno` is `Voted` at VC_i, she will get a receipt.*

Proof. The vote code that caused this ballot to become `Voted` can only be the same as the one submitted (`vcode`), because the voter is honest and the adversary cannot guess vote codes. As such, Algorithm 2 at line 10 replies immediately to the voter with the stored receipt. □

Theorem 1 (**Liveness of D-Demos/IC's voting algorithm**).
If an honest voter starts the voter's algorithm 1 for $\langle \texttt{serno}, \texttt{vcode} \rangle$, after the election start-time and before the election end-time, and the network delivers all protocol messages for this instance of the voting algorithm 2 before election end-time, then she will get back a receipt.

Proof. The voter's algorithm loops over VC nodes and the voter will eventually stumble upon a correct VC node VC_i, because of the fault tolerance thresholds. If the ballot's status at VC_i is `Voted`, it has to be for `vcode` as the voter is honest and the adversary cannot guess the other vote codes in her ballot. In this case, because of Lemma 3, the voter will immediately get back a receipt.

If the ballot's status at VC_i is `Pending`, it has to be for `vcode` as the voter is honest and the adversary cannot guess the other vote codes in her ballot. In this case, the voter will not get a reply back immediately. However, because of Lemma 1, this status is temporary, and it will be switched to `Voted` for `vcode` eventually.

The only remaining case is when VC_i's status for the specific ballot is `NotVoted`. Because of Lemma 2, the status for this ballot will eventually change to `Voted` for `vcode` across all correct VC nodes.

Thus, as a worst case, the voter will timeout and keep looping contacting the next *VC* node, until she reaches a correct one where the status has become `Voted`, and she will get the receipt as a reply because of Lemma 3. □

Note that the above liveness proof requires the voter to contact at least two correct *VC* nodes before getting the reply: one to start the instance of the protocol, and another to reach a node with status `Voted` and obtain the receipt. However, our protocol actually delivers the receipt back to the voter in one step (Algorithm 2, lines 28-29), as long as the voter does not timeout when casting the vote to the first correct *VC* node, and the voter stumbles upon a correct *VC* node initially.

In the above proof, we assume all messages for the specific instance of the voting protocol are delivered before election end-time. To give an estimate of the time needed for the voting protocol to complete, we provide a formula for a safety threshold in section 3.6.2.3.

3.6.2.1.2 Liveness of the vote set consensus algorithm.

We are interested in the liveness of the vote set consensus algorithm, that is, when this algorithm is started across all correct *VC* nodes at election end-time, it terminates outputting a set of voted $\langle \texttt{serno}, \texttt{vcode} \rangle$ tuples at each node.

The liveness of the vote set consensus algorithm derives directly from the (IC,BC-RBB) algorithm's liveness property. This is because the algorithm of Figure 3.4, which runs immediately after the IC one, has no external communication and also has no condition under which it does not terminate.

3.6.2.2 Safety

In this section, we prove the safety of the vote collection subsystem of D-DEMOS/IC, in two steps. First, we show that the voting algorithm produces a valid receipt, or none at all. Then, we show that if a valid receipt is indeed generated, the corresponding vote code will be included in the set of voted vote codes that is uploaded to the *BB* from each node.

3.6.2.2.1 Safety of the voting algorithm.

Theorem 2 (**Safety of D-Demos/IC voting algorithm**).
If an honest voter submits a $\langle \texttt{serno}, \texttt{vcode} \rangle$ *tuple, then she will receive a valid receipt or none at all.*

Proof. In the proof of Theorem 1 above, we showed that a receipt will be generated for a valid tuple submission to the *VC* system, if the network delivers all messages for this particular voting protocol instance before election end-time. If the network does not deliver enough messages for the protocol to complete before election-end time, the system does not output a receipt, but safety is not violated.

If a receipt is indeed generated, this receipt will be valid because at least $N_v - f_v$ correct *VC* nodes (including the *responder*) will submit their receipt's shares, and these are enough to reconstruct the correct receipt (this is an attribute of the VSS scheme). Additionally, because the secret-sharing scheme we use is verifiable, each *VC* node that reconstructs the receipt can do so by discarding corrupt shares from malicious *VC* nodes, and thus, there is no possibility of incorrect reconstruction. $\square$

3.6.2.2.2 Safety of the vote set consensus algorithm.

Beyond our standard assumptions, we additionally assume messages from correct *VC* nodes submitted for the Value Dissemination Phase of the used (IC,BC-RBB) algorithm, are delivered within the defined $barrier$.

Theorem 3 (**Safety of D-Demos/IC vote set consensus**).
If an honest voter who submits a $\langle \texttt{serno}, \texttt{vcode} \rangle$ *tuple obtains a valid receipt, then this tuple will be included in the set of voted tuples of each correct VC node.*

Proof. By the safety property of the IC algorithm, we have that each correct *VC* node will possess every other correct *VC* node's VS_i set in the result vector of IC.

Given the above, a $\langle \texttt{serno}, \texttt{vcode} \rangle$ tuple for which a valid receipt has been received by the voter, is not included in the set of *voted* tuples *only* in the following cases:

1. There is another tuple, with a different vote code, for the same ballot (see Step 1 of Figure 3.4).
   This however contradicts either our honest voter assumption, or the assumption that the adversary is unable to guess vote codes.

2. The specific tuple appears in the VS_i sets of less than $N_v - 2f_v$ *VC* nodes (see Step 2 of Figure 3.4).
   This is not possible because of the secret-sharing threshold of the VSS scheme used to share the receipt shares across *VC* nodes. Specifically, for a receipt to be generated, $N_v - f_v$ shares are required, of which at least $N_v - 2f_v$ are correct. Thus, at least $N_v - 2f_v$ nodes will include this tuple in their corresponding VS_i sets.

The above are the only cases where a $\langle \texttt{serno}, \texttt{vcode} \rangle$ tuple for which a valid receipt has been received by the voter is not included in the set of voted ballots, and we have shown that all above cases are invalid under our assumptions. $\square$

3.6.2.3 Safety threshold

We remind the reader that *election hours*, that is, the hours where voters are allowed to vote, are defined by the election administrator. Our system has to obey these time constraints, by servicing vote requests only within these *election hours*. We express our liveness property without such timing constraints, but we assume the network delivers all protocol messages within the election hours.

To help the system administrator provide an estimate to the voters as to how long before the election end-time it is safe to cast a vote, we define a *safety threshold*. To define this safety threshold, we need to analyze the time it takes for the *responder* node from receiving the vote to producing and responding with the corresponding receipt. We use worst-case assumptions throughout this definition, preferring to err on the safe side. Such assumptions, evident in our formulas below, are the following:

- There are f_v malicious and fast *VC* nodes active. A correct node receives erroneous messages from all malicious nodes first, filters them out, and waits until it receives incoming messages from all the correct nodes to obtain the needed $N_v - f_v$ valid messages and make progress.
- Propagation of messages from different nodes to a single target happen serially, because they share the majority of the communication links. Thus, when sending N_v messages to different hosts, for simplicity we sum the individual propagation delays, assuming no parallelism at all on the communication links. We apply the same logic for incoming messages from multiple hosts to a single receiver.

3.6.2.3.1 Receipt generation time formula.

In general, the delay for delivering a message between two nodes is given by the following formula:

$$d_{end-to-end} = d_{transmission} + d_{propagation} + d_{queuing} + d_{processing} \tag{3.1}$$

In the above formula, we have the following terms:

- $d_{transmission}$ is the transmission delay, i.e., the time required for the sender to transmit the complete message over the communication link; the transmission delay is dependent on the message size.
- $d_{propagation}$ is the time it takes for the first bit of the message to travel the communication link between the two nodes, and is dependent on the length of the communication link (or, the distance between the two nodes).

- $d_{queuing}$ is the time the input message waits in the input queue of the receiving node, and is dependent on the concurrency (load) of the receiving node.
- $d_{processing}$ is the time it takes to process the message and, either forward it to the next node, or produce a reply and enqueue it in the output queue.

In our case, we know the vast majority of the messages nodes exchange are short, and each message fits in a single Ethernet packet. Because of this, we consider $d_{transmission}$ negligible and integrate it into $d_{propagation}$ in all the following formulas.

Thus, the end-to-end delay $d_{receipt\_ic}$ for the voter obtaining a receipt is given by the following formula:

$$d_{receipt\_ic} = 2d_{c\_propagation} + d_{r\_processing\_ic} + d_{queuing} \tag{3.2}$$

In the above formula we assume there is no processing and queuing delay when obtaining the receipt on the voter's device. $d_{c\_propagation}$ is the propagation delay between the client (voter) and the *responder VC* node, and its factor of 2 expresses the delay until both the request (vote cast) and the reply (receipt) reach their targets. $d_{queuing}$ is the queuing delay of the `VOTE` message in the *responder* node.

Finally, $d_{r\_processing\_ic}$ is the time it takes the *responder* to generate the receipt. This includes validating the input, sending out one `VOTE_P` message to each node in the *VC* subsystem, waiting for N_v `VOTE_P` replies (we assume malicious nodes are active and faster than correct nodes, which is the worst case), and recreating the receipt from the shares it receives. We observe a *VC* node performs a computation, sends a network message, and either finishes its operation or blocks, waiting for input. To simplify things, we make the assumption that all such computations take an equal amount of time, which we denote as $d_{computation}$.

Thus, the formula to calculate $d_{r\_processing\_ic}$ is:

$$d_{r\_processing\_ic} = 2d_{computation} + 2N_v d_{v\_propagation} + N_v d_{queuing} + (d_{v\_processing\_ic} + d_{queuing}) \tag{3.3}$$

In formula 3.3 above, the terms are as follows:

- $2d_{computation}$ denotes the two computation steps performed at the *responder* node. The first is for validating the input and the second is for recreating the receipt from the shares.
- $2N_v d_{v\_propagation}$ is the propagation delay between *VC* nodes. The factor $2N_v$ includes sending the message to N_v nodes and receiving their replies.
- $N_v d_{queuing}$ expresses the time the N_v replies wait in the queue of the *responder* node.
- $(d_{v\_processing\_ic} + d_{queuing})$ is the time required for a non-responder *VC* node to submit its receipt share, which includes the processing and queuing delay. There is no N_v factor here, because processing and queuing happen in parallel across *VC* nodes. Thus, the *responder* node actually notices a delay equal to the processing and queuing time at a single *VC* node.

A non-responder *VC* node simply validates its input and discloses its receipt share. Thus, we have that:

$$d_{v\_processing\_ic} = d_{computation} \tag{3.4}$$

Combining equations 3.2, 3.3, and 3.4, we derive the following formula:

$$d_{receipt\_ic} = 2d_{c\_propagation} + 2N_v d_{v\_propagation} + 3d_{computation} + (N_v + 2)d_{queuing} \tag{3.5}$$

3.6.2.3.2 Safety threshold formula.

If the voter sets the timeout, for Algorithm 1 when waiting for a receipt, to $d_{receipt\_ic}$ (defined in formula 3.5 above), we estimate that a safe amount of time the voter can vote and expect her vote to be registered, to be:

$$safety\_threshold\_ic = (f_v + 2)d_{receipt\_ic} \tag{3.6}$$

In the above calculation, we assume the voter contacts all malicious nodes first, then the first correct node times out (potentially due to high load), and a second correct one responds properly. Of course, this threshold can be set in a more conservative fashion by adjusting the factor to multiples of N_V, instead of f_v, as the voter cycles though all nodes while voting. For example, a factor of $2Nv$ instead of $f_v + 2$ will guarantee enough time for the voter to cycle through all *VC* nodes twice, with an even higher probability of obtaining the receipt.

3.7 Asynchronous *VC* subsystem

In this section, we describe the asynchronous version of our vote collection subsystem. We first present the voting protocol, which is used by the voter to cast her vote. This protocol requires approximately two round-trips between two *VC* nodes to produce the receipt, and also requires signatures. Then, we present the vote set consensus protocol, which runs on election-end time and guarantees that all honest nodes agree on a single set of votes. This protocol requires one multicast and one Binary Consensus Instance for each defined ballot. Finally, we provide proofs of liveness and safety for both protocols.

3.7.1 System description

We make the following enhancements to the Vote Collection subsystem, to achieve the completely asynchronous version *D-DEMOS/Async*. During voting we introduce another step, which guarantees only a single vote code can be accepted (towards producing a receipt) for a given ballot. We also employ an asynchronous binary consensus primitive to achieve Vote Set Consensus.

More specifically, during voting, the *responder VC* node validates the submitted vote code, but before disclosing its receipt share, it multicasts an $\mathtt{ENDORSE}\langle$serial-no, vote-code$\rangle$ message to all *VC* nodes. Each *VC* node, after making sure it has not endorsed another vote code for this ballot, responds with an $\mathtt{ENDORSEMENT}\langle$serial-no, vote-code, $\text{sig}_{\text{VC}_\text{i}}\rangle$ message, where $\text{sig}_{\text{VC}_\text{i}}$ is a digital signature of the specific serial-no and vote-code, with VC_i's private key. The responder collects $N_v - f_v$ valid signatures and forms a uniqueness certificate UCERT for this ballot. It then discloses its receipt share via the `VOTE_P` message, but also includes the formed UCERT in the message.

Each *VC* node that receives a VOTE_P message, first verifies the validity of UCERT and discards the message on error. On success, it proceeds as per the *D-DEMOS/IC* protocol (validating the receipt share it receives and then disclosing its own receipt share).

The algorithms implementing our *D-DEMOS/Async voting* protocol are presented in Algorithm 3.

The voting process is outlined in the diagram of Figure 3.5, where we now see two round-trips are needed before the receipt is reconstructed and posted to the voter.

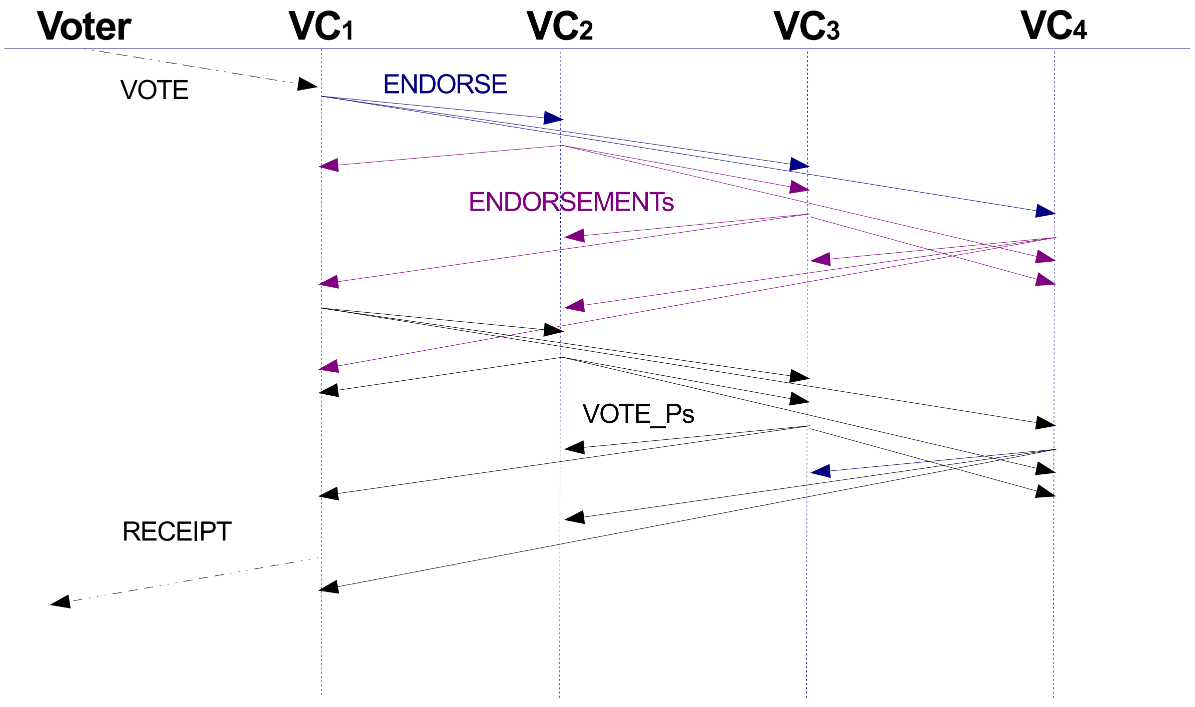


Figure 3.5: Diagram of message exchanges for a single vote during the D-DEMOS/Async vote collection phase.

The formation of a valid UCERT gives our algorithms the following guarantees:

a) No matter how many responders and vote codes are active at the same time for the same ballot, if a UCERT is formed for vote code vc_a, no other uniqueness certificate for any vote code different than vc_a can be formed.

Algorithm 3 Vote Collector voting protocol algorithms for D-DEMOS/Async

```
1: procedure on VOTE(serial-no, vote-code) from $source$:
2:     if $SysTime()$ between $start$ and $end$
3:         if $b$ :=locateBallot(serial-no)
4:             if $b$.status = NotVoted
5:                 if $l := b$.VerifyVoteCode(vote-code)
6:                     $b$.voter := $source$          ▷ Mark the current node as the responder for source
7:                     sendAll(ENDORSE⟨serial-no, vote-code⟩)
8:             else if $b$.status = Voted $\wedge$ $b$.used-vc = vote-code
9:                 send ($source$, b.receipt)
10: procedure on ENDORSEMENT(serial-no, vote-code, signature) from $source$:
11:     if $validEndorsement$(signature, serial-no, vote-code, $source$)
12:         if $SysTime()$ between $start$ and $end$
13:             if $b$ :=locateBallot(serial-no)
14:                 if $b$.status = NotVoted
15:                     if $l := b$.VerifyVoteCode(vote-code)
16:                         $b$.UCERT.Append(signature)          ▷ Fill in Uniqueness Certificate
17:                         if size($b$.UCERT) = $N_v - f_v$
18:                             $b$.status := Pending
19:                             $b$.used-vc := vote-code
20:                             sendAll(VOTE_P⟨serial-no, vote-code, $l$.share, $b$.UCERT⟩)
21: procedure on VOTE_P(serial-no, vote-code, share, UCERT) from $source$:
22:     if $validUCert$(UCERT, serial-no, vote-code)
23:         if $SysTime()$ between $start$ and $end$
24:             if $b$ :=locateBallot(serial-no)
25:                 if $b$.status = NotVoted
26:                     if $l := b$.VerifyVoteCode(vote-code)
27:                         if $validShare$(share, serial-no, vote-code)
28:                             $b$.status := Pending
29:                             $b$.used-vc := vote-code
30:                             $b$.lrs.Append(share)          ▷ Update list of receipt shares
31:                             sendAll(VOTE_P⟨serial-no, vote-code, $l$.share⟩ )
32:                 else if $b$.status = Pending AND $b$.used-vc = vote-code
33:                     if $validShare$(share, serial-no, vote-code)
34:                         $b$.lrs.Append(share)          ▷ Update list of receipt shares
35:                         if size($b$.lrs) = $N_v - f_v$
36:                             $b$.receipt := Rec($b$.lrs)
37:                             $b$.status := Voted
38:                             if $b$.voter          ▷ If this is the responder node
39:                                 $send(b$.voter, $b$.receipt)
40: procedure on ENDORSE(serial-no, vote-code) from $source$:
41:     if $SysTime()$ between $start$ and $end$
42:         if $b$ :=locateBallot(serial-no)
43:             if $b$.status = NotVoted
44:                 if $l := b$.VerifyVoteCode(vote-code)
45:                     if $b$.endorsed $\in \{\perp$, vote-code$\}$
46:                         $b$.endorsed := vote-code
47:                         $s := sign$(serial-no, vote-code)
48:                         send($source$, ENDORSEMENT⟨serial-no, vote-code, $s$⟩)
49: function Ballot::VerifyVoteCode(vote-code)
50:     for $l := 1$ to ballot_lines do
51:         if lines[$l$].hash = $h$(vote-code||lines[$l$].salt)
52:             return lines[$l$]
53:     return $\perp$
```

b) By verifying the UCERT before disclosing a *VC* node's receipt share, we guarantee the voter's receipt cannot be reconstructed unless a valid UCERT is present.

At election end time, each *VC* node stops processing `ENDORSE`, `ENDORSEMENT`, `VOTE` and `VOTE_P` messages, and follows the *vote-set consensus* algorithm in Figure 3.6, for each registered ballot.

1. Send `ANNOUNCE`$\langle$serial-no, vote-code, UCERT$\rangle$ to all nodes. The vote-code will be $\perp$ if the node knows of no vote code for this ballot.
2. Wait for $N_v - f_v$ such messages. If any of these messages contains a valid vote code vc_a, accompanied by a valid UCERT, change the local state immediately, by setting vc_a as the vote code used for this ballot.
3. Participate in a Binary Consensus protocol, with the subject "Is there a valid vote code for this ballot?". Enter with an opinion of 1, if a valid vote code is locally known, or a 0 otherwise.
4. If the result of Binary Consensus is 0, consider the ballot not voted.
5. Else, if the result of Binary Consensus is 1, consider the ballot voted. There are two sub-cases here:
   a) If vote code vc_a, accompanied by a valid UCERT is locally known, consider the ballot voted for vc_a.
   b) If, however, vc_a is not known, send a `RECOVER-REQUEST`$\langle$serial-no$\rangle$ message to all *VC* nodes, wait for the first valid `RECOVER-RESPONSE`$\langle$serial-no, vc_a, UCERT$\rangle$ response, and update the local state accordingly.

Figure 3.6: High level description of algorithm for asynchronous vote set consensus. This algorithm runs for each registered ballot.

Steps 1-2 ensure used vote codes are dispersed across nodes. Recall our receipt generation requires $N_v - f_v$ shares to be revealed by distinct *VC* nodes, of which at least $N_v - 2f_v$ are honest. Note that any two $N_v - f_v$ subsets of N_v contain at least $f_v + 1$ honest nodes (because $f_v > N_v/3$), and at least one of the $f_v + 1$ honest nodes has participated in receipt generation. Because of this, if a receipt was generated, at least one honest node's `ANNOUNCE` will be processed by every honest node, and all honest *VC* nodes will obtain the corresponding vote code in these two steps. Consequently, all honest nodes enter step 3 with an opinion of 1 and binary consensus is guaranteed to deliver 1 as the resulting value, thus safeguarding our contract against the voters. In any case, step 3 guarantees all *VC* nodes arrive at the same conclusion, on whether this ballot is voted or not.

In the algorithm outlined above, the result from binary consensus is translated from $0/1$ to a status of "not-voted" or a unique valid vote code, in steps 4-5. Step 5b requires additional explanation. Assume, for example, that a voter submitted a valid vote code vc_a, but a receipt was not generated before election end time. In this case, an honest vote collector node VC_i may not be aware of vc_a at step 3, as steps 1-2 do not make any guarantees in this case. Thus, VC_i may rightfully enter consensus with a value of 0. However, when honest nodes' opinions are mixed, the consensus algorithm may produce either 0 or 1. In case the result is 1, VC_i will not possess the correct vote code vc_a, and thus will not be able to properly translate the result. Thus we introduce a recovery protocol with which VC_i will issue a `RECOVER-REQUEST` multicast. We claim that another honest node, VC_h, exists that *possesses* vc_a and *replies* with vc_a and the correct UCERT. The reason for the existence of an honest VC_h is straightforward and stems from the properties of the binary consensus problem definition. If all honest nodes enter binary consensus with the same opinion a, the result of any consensus algorithm is guaranteed to be a. Since we have an honest node VC_i, that entered consensus with a value of 0, but a result of 1 was produced, there has to exist another honest node VC_h that entered consensus with an opinion of 1. Since VC_h is honest, it must *possess* vc_a, along with the corresponding UCERT (as no other vote code vc_b can be active at the same time for this ballot). Again, because VC_h is honest, it will follow the protocol and *reply* with a well formed RECOVER-REPLY. Additionally, the existence of UCERT guarantees that any malicious replies can be safely identified and discarded by VC_i.

As per *D-DEMOS/IC*, at the end of this algorithm, each node submits the resulting set of *voted* $\langle$serial-no, vote-code$\rangle$ tuples to each *BB* node, which concludes its operation for the specific election.

3.7.2 Proofs

We now prove the *liveness* and *safety* properties of both phases of our asynchronous vote collection subsystem for D-DEMOS/Async.

We consider a D-DEMOS/Async system, where voters run Algorithm 1 and *VC* nodes run Algorithm 3 for voting, and the algorithm of section 3.7.1 for Vote Set Consensus, under the system model of Section 3.1, where the fault tolerance thresholds of Section 3.4 hold, and the adversary cannot guess any vote code inside honest voters' ballots. We assume the adversary can control and coordinate all faulty components, i.e., all corrupt voters and all corrupt *VC* nodes.

We use the same wording as explained in section 3.6.2 for the *honest voter*, the *correct VC* node, and the status of a ballot maintained by a *VC* node. Note that we have not defined a separate status for the ballot while the Uniqueness Certificate is formed, so the possible statuses remain three (`NotVoted`,`Pending`,`Voted`).

The following lemmas will prove useful for both liveness and safety proofs.

Lemma 4. *The intersection of every two ($N_v - f_v$)-sized subsets of the set of VC nodes, contains at least one correct node.*

Proof. Because the two subsets have cardinality $N_v - f_v$, their intersection I has cardinality at least $N_v - 2f_v$. Lets assume that all f_v corrupt nodes are present in I, which is the worst case. This leaves the subset of I with only correct nodes (say CI) with at least $N_v - 3f_v$ nodes. Because $N_v > 3f$, CI has at least one member, which is correct because all corrupt nodes have already been subtracted from I. □

Lemma 5. *If a* UCERT U_a *is formed for a voter's ballot with tag* serno*, for vote code* $\mathtt{vcode_a}$*, then no other* UCERT U_b *can be assembled for a vote code* $\mathtt{vcode_b}$ *with* $\mathtt{vcode_a} \neq \mathtt{vcode_b}$*, for the same ballot with tag* serno*.*

Proof. We prove this by contradiction. Assume there is such a U_b for $\mathtt{vcode_b}$, with $\mathtt{vcode_a} \neq \mathtt{vcode_b}$. This means that $N_v - f_v$ *VC* nodes endorsed $\mathtt{vcode_b}$, and these nodes form the S_b subset of the set of *VC* nodes.

Because of U_a, $N_v - f_v$ *VC* nodes endorsed $\mathtt{vcode_a}$, and these nodes form the S_a subset of the set of *VC* nodes.

However, because of Lemma 4, the intersection of S_a and S_b contains at least one correct *VC* node.

This means that this correct *VC* node endorsed two different vote codes for the same ballot. But this is impossible because the "on ENDORSE" procedure of Algorithm 3 (line 45) explicitly endorses a single vote code for each ballot. Thus, there cannot be two UCERTs for two different vote codes for the same ballot. □

Note that Lemma 5 is not restricted to honest voters, but applies equally to malicious ones as well.

3.7.2.1 Liveness

3.7.2.1.1 Liveness of the voting algorithm.

We first prove a few useful lemmas, that will lead to the proof of the main theorem.

Lemma 6. *If an honest voter's ballot with tag* serno*, at a correct VC node* VC_i*, has its status changed to* *Pending* *for vote code* vcode*, and all voting protocol messages for this instance are delivered before election end-time, then this ballot's status will eventually change to* *Voted* *for* vcode *at* VC_i*.*

Proof. The ballot's status is changed to Pending while processing either the ENDORSEMENT or VOTE_P messages, at node VC_i. In both cases, this change is accompanied with the multicast of a VOTE_P message to all other *VC* nodes, disclosing VC_i's receipt share, attaching a valid UCERT. Because we assume a lossless network, this VOTE_P message will be delivered to, and be processed by all correct nodes, as per Algorithm 3.

If a receiving correct node VC_j, has the ballot marked as NotVoted, processing of VC_i's VOTE_P message results in marking the ballot as Pending for vcode and disclosing VC_j's receipt share via a new VOTE_P message. This happens because the submitted $\langle \mathtt{serno}, \mathtt{vcode} \rangle$ tuple was already found valid by VC_i, and thus will also be found valid by VC_j, while the message contains a valid receipt share and a valid UCERT (since the sender is correct).

If VC_j has ballot with tag `serno` marked as anything other than `NotVoted`, it has to be so for vote code `vcode`, since the voter is honest and the adversary cannot guess and submit any other vote code for this ballot. As such, VC_j must have already sent its `VOTE_P` message, and VC_i will receive it eventually.

In both of the above cases, VC_j's `VOTE_P` message will be delivered at node VC_i, since the network is lossless. As this is true for every correct node VC_j, VC_i eventually receives all receipt shares of all correct nodes, via the corresponding `VOTE_P` messages. Because the VSS scheme's secret recovery threshold ($N_v - f_v$) is equal to the minimum correct node count, it is guaranteed that the receipt shares are enough to reconstruct the receipt and mark the ballot as `Voted` at node VC_i (passing the threshold check at line 35 of Algorithm 3).

Thus, the ballot will be marked as `Voted` at node VC_i. □

Lemma 6 shows that a status of `Pending` is temporary for honest voters' ballots, as the protocol ensures that such a ballot's status will eventually switch to `Voted`.

Lemma 7. *If an honest voter's ballot with tag* `serno`*, at a correct VC node* VC_i*, has its status changed to* `Pending` *for vote code* `vcode`*, and all voting protocol messages for this instance are delivered before election end-time, then this ballot's status will eventually change to* `Voted` *for* `vcode` *at all correct VC nodes.*

Proof. At VC_i, the status will eventually change to `Voted` because of Lemma 6. However, node VC_i also multicasts its `VOTE_P` message immediately upon switching its status to `Pending` for `vcode`. This multicast will be delivered, because of our lossless network assumption, at all correct *VC* nodes.

If a receiving correct node VC_j, has the ballot marked as `NotVoted`, processing of VC_i's `VOTE_P` message results in marking the ballot as `Pending` for vote code for `vcode`. Due to Lemma 6, the `Pending` status will eventually change to `Voted` for `vcode`.

If VC_j has ballot with tag `serno` marked as anything other than `NotVoted`, it has to be so for vote code `vcode`, since the voter is honest and the adversary cannot guess and submit any other vote code for this ballot. Thus, if it is `Pending` for `vcode`, because of Lemma 6 this status will eventually change to `Voted` for `vcode`. If it already `Voted` for `vcode` then our goal has already been reached. □

Lemma 7 shows that our protocol is "unstoppable" by the adversary, since once a correct node makes the `VOTE_P` multicast for a ballot and a vote code, all correct nodes will mark this ballot as `Voted` for the specific vote code.

Lemma 8. *If an honest voter posts* $\langle \texttt{serno}, \texttt{vcode} \rangle$ *to a correct VC node* VC_i *after election start-time, and the status of ballot with tag* `serno` *is* `NotVoted` *at* VC_i*, and all voting protocol messages for this instance are delivered before election end-time, then the status of this ballot will become* `Voted` *for* `vcode` *across all correct VC nodes.*

Proof. Node VC_i begins the UCERT forming process immediately, by multicasting an `ENDORSE` message. This message will be delivered across all correct nodes, because of our lossless network assumption. A correct *VC* node that processes the `ENDORSE` message cannot ignore it because it has already endorsed another vote code, because our voter is honest and the adversary cannot guess other vote codes in the voter's ballot.

However, it may ignore it because its status has progressed from `NotVoted`. In this case, its status of either `Pending` or `Voted` has to be for the same vote code `vcode` (for the same reasons a node cannot have endorsed a different vote code, explained above).

If a single *VC* node, however, has a status other than `NotVoted` for `vcode`, this means that at some point the status was `Pending` for `vcode`, as no node can reach the `Voted` status for a ballot without going through the `Pending` status for this ballot first. But once that single *VC* node's status of ballot with tag `serno` became `Pending` for `vcode`, because of Lemma 7, all correct *VC* nodes will reach the goal of marking the ballot as `Voted` for `vcode`.

The last remaining case is if all correct *VC* nodes have a status of `NotVoted` for ballot with tag `serno`. In this case, all correct nodes will reply with the corresponding `ENDORSEMENT` message. All correct nodes are enough to form the UCERT for `vcode`, and thus node VC_i will mark the ballot as `Pending` for `vcode`. Again because of Lemma 7, this results in all correct *VC* nodes reaching the goal of marking the ballot as `Voted` for `vcode`. □

Lemma 9. *If an honest voter posts $\langle \texttt{serno}, \texttt{vcode} \rangle$ to a correct VC node VC_i before election end-time, and the status of ballot with tag `serno` is `Voted` at VC_i, she will get a receipt.*

Proof. The vote code that caused this ballot to become `Voted` can only be the same as the one submitted (`vcode`), because the voter is honest and the adversary cannot guess vote codes. As such, Algorithm 3 at line 8 replies immediately to the voter with the stored receipt. □

Theorem 4 (**Liveness of D-Demos/Async's voting algorithm**).
If an honest voter starts the voter's algorithm 1 for $\langle \texttt{serno}, \texttt{vcode} \rangle$, after the election start-time and before the election end-time, and the network delivers all protocol messages for this instance of the voting algorithm 3 before election end-time, then she will get back a receipt.

Proof. The voter's algorithm loops over VC nodes and the voter will eventually stumble upon a correct VC node VC_i, because of the fault tolerance thresholds.

If the ballot's status at VC_i is `Voted`, it has to be for `vcode` as the voter is honest and the adversary cannot guess the other vote codes in her ballot. In this case, because of Lemma 9, the voter will immediately get back a receipt.

If the ballot's status at VC_i is `Pending`, it has to be for `vcode` as the voter is honest and the adversary cannot guess the other vote codes in her ballot. In this case, the voter will not get a reply back immediately. However, because of Lemma 6, this status is temporary, and it will be switched to `Voted` for `vcode` eventually.

The only remaining case is when VC_i's status for the specific ballot is `NotVoted`. Because of Lemma 8, the status for this ballot will eventually change to `Voted` for `vcode` across all correct VC nodes.

Thus, as a worst case, the voter will timeout and keep looping contacting the next *VC* node, until she reaches a correct one where the status has become `Voted`, and she will get the receipt as a reply because of Lemma 9. □

Note that the above liveness proof requires the voter to contact at least two correct *VC* nodes before getting the reply: one to start the instance of the protocol, and another to reach a node with status `Voted` and obtain the receipt. However, our protocol actually delivers the receipt back to the voter in one step (Algorithm 3, lines 38-39), as long as the voter does not timeout when casting the vote to the first correct *VC* node, and the voter stumbles upon a correct *VC* node initially.

In the above proof, we assume all messages for the specific instance of the voting protocol are delivered before election end-time. To give an estimate of the time needed for the voting protocol to complete, we provide a formula for a safety threshold in section 3.7.2.3.

3.7.2.1.2 Liveness of the vote set consensus algorithm.

We are interested in the liveness of the vote set consensus algorithm, that is, that when this algorithm is started across all correct *VC* nodes at election end-time, it terminates outputting a set of voted $\langle \mathtt{serno}, \mathtt{vcode} \rangle$ tuples at each node.

The liveness of the vote set consensus algorithm is achieved because of the following:

1. The wait for `ANNOUNCE` messages completes because of the fault-tolerance threshold of the *VC* subsystem; by assumption, there will always be $N_v - f_v$ correct nodes to send `ANNOUNCE` messages and complete this phase at each node.

2. The Binary Consensus algorithm completes in accordance with its corresponding liveness property.

3. The recovery protocol, if needed, terminates because of the existence of a correct *VC* node with a valid UCERT for the recovery case (Step 5b) (for the full justification, see the description of this algorithm in Section 3.7.1).

3.7.2.2 Safety

In this section, we prove the safety of the voting subsystem of D-DEMOS/Async, in two steps. First, we show that the voting algorithm produces a valid receipt, or none at all. Then, we show that if a valid receipt is indeed generated, the corresponding vote code is included in the set of voted vote codes.

3.7.2.2.1 Safety of the voting algorithm.

Theorem 5 (**Safety of D-Demos/Async voting algorithm**).
If an honest voter submits a $\langle \texttt{serno}, \texttt{vcode} \rangle$ *tuple, then she will receive a valid receipt or none at all.*

Proof. In the proof of Theorem 4 above, we showed that a receipt will be generated for a valid tuple submission to the *VC* system, if the network delivers all messages for this particular voting protocol instance before election end-time. If the network does not deliver enough messages for the protocol to complete before election-end time, the system does not output a receipt, but safety is not violated.

If a receipt is indeed generated, this receipt will be valid because at least $N_v - f_v$ correct *VC* nodes (including the responder) will submit their receipt's shares, and these are enough to reconstruct the correct receipt (this is an attribute of the VSS scheme). Additionally, because the secret-sharing scheme we use is verifiable, each *VC* node that reconstructs the receipt can do so by discarding corrupt shares from malicious *VC* nodes, and thus, there is no possibility of incorrect reconstruction. □

3.7.2.2.2 Safety of the vote set consensus algorithm.

Theorem 6 (**Safety of D-Demos/Async vote set consensus**).
If an honest voter who submits a valid $\langle \texttt{serno}, \texttt{vcode} \rangle$ *tuple obtains a valid receipt, then this tuple will be included in the set of voted tuples of each correct VC node.*

Proof. A $\langle \texttt{serno}, \texttt{vcode} \rangle$ tuple for which a valid receipt has been received by the voter, is not included in the set of *voted* tuples *only* because of one of the following reasons:

1. Not all correct *VC* nodes possess the vote code and corresponding UCERT, in which case binary consensus is not guaranteed to produce 1.
   This is not possible because of the `ANNOUNCE` phase. Suppose a correct node VC_i does not possess the vote code and UCERT. For a receipt to be generated at any *VC* node, $N_v - f_v$ shares from different *VC* nodes are required (let's call this subset of *VC* nodes S_r). The `ANNOUNCE` phase at *VC* node VC_i requires $N_v - f_v$ replies to progress, from a subset S_a of the set of *VC* nodes.
   However, because of Lemma 4, S_r and S_a have at least one correct *VC* node in common. This means that the common correct *VC* node possesses the `vcode` and the corresponding UCERT and will disclose it via the `ANNOUNCE` message, that is processed by VC_i. When VC_i processes this `ANNOUNCE` message, it modifies its state immediately to include `vcode`.
   Thus, it is impossible for any correct *VC* node to not possess in its set of votes a `vcode` for which a reply was generated.

2. A correct node obtains a 1 as the result of binary consensus but has not received `vcode` and the UCERT.
   This is not possible, because this is what the recovery protocol of the algorithm covers (Step 5b). In this recovery step, the correct node that misses `vcode` and the UCERT broadcasts a `RECOVER-REQUEST` and at least one correct node will respond with `RECOVER-RESPONSE` disclosing `vcode` and the UCERT. The reason why one such correct node exists stems from the properties of the binary consensus problem definition. If all correct nodes enter binary consensus with the same opinion a, the result of any consensus algorithm is guaranteed to be a. Since we have a correct node VC_i, that entered consensus with a value of 0, but a result of 1 was produced, there has to exist another correct node VC_c that entered consensus with an opinion of 1 (otherwise, Binary Consensus's agreement property assures us a 0 would have been produced). Since VC_c is correct, it must *possess* `vcode`, along with the corresponding UCERT (as no other vote code $\texttt{vcode}_\texttt{b}$ can be registered at any correct *VC* node for the ballot with tag `serno`, because of Lemma 5). Again, because VC_c is correct, it will follow the protocol and *reply* with a well formed `RECOVER-RESPONSE`. Additionally, the existence of UCERT guarantees that any malicious replies can be safely identified and discarded by VC_i. Thus, it is impossible for any correct *VC* node to not possess in its set of votes a `vcode` for which binary consensus resulted in 1.

The above are the only cases where a $\langle \texttt{serno}, \texttt{vcode} \rangle$ tuple for which a valid receipt has been received by the voter is not included in the set of voted ballots, and we have shown that all above cases are invalid under our assumptions.

□

3.7.2.3 Safety threshold

Because of the election end-time hard threshold, which we explained in detail in Section 3.6.2.3, we again define a *safety threshold* for the voter to obtain a receipt with high probability, based on the receipt generation time formula.

3.7.2.3.1 Receipt generation time formula.

Following the same logic as the one used in formula 3.1, the end-to-end delay for the voter obtaining a receipt in D-DEMOS/Async is given by the following formula:

$$d_{receipt\_async} = 2d_{c\_propagation} + d_{r\_processing\_async} + d_{queuing} \tag{3.7}$$

In the above formula we assume there is no processing and queuing delay when obtaining the receipt on the voter's device. $d_{c\_propagation}$ is the propagation delay between the client (voter) and the *responder VC* node, and the factor 2 covers both casting the vote and receiving the receipt.

Finally, $d_{r\_processing\_async}$ is the time it takes the *responder* to generate the receipt. This includes validating the input, generating the UCERT, sending out one message to each node in the *VC* subsystem, waiting for N_v replies (we assume malicious nodes are active and faster than correct ones, which is the worst case), and recreating the receipt from the shares it received. Generating a UCERT requires an extra round-trip to the *VC* nodes, where each one produces an endorsement by signing the input data, and the *responder* verifies each received `ENDORSEMENT`, which contains a signature. We observe a *VC* node performs a computation, sends a network message and either finishes its operation or blocks waiting for input. To simplify things, we make the assumption that all such computations take an equal amount of time, which we denote as $d_{computation}$.

Thus, the formula to calculate $d_{r\_processing\_async}$ is:

$$d_{r\_processing\_async} = 2d_{computation} + d_{ucert\_gen} + 2N_v d_{v\_propagation} + N_v d_{queuing} + (d_{v\_processing\_async} + d_{queuing}) \quad (3.8)$$

In formula 3.8 above, the terms are as follows:

- $2d_{computation}$ denotes the two computation steps performed at the *responder* node. The first is for validating the input and the second is for recreating the receipt from the shares.
- d_{ucert_gen} is the time required to generate the UCERT, and it is defined below.
- $2N_v d_{v\_propagation}$ is the propagation delay between *VC* nodes. The factor $2N_v$ includes sending the message to N_v nodes and receiving their replies.
- $N_v d_{queuing}$ denotes the time the N_v replies wait in the queue of the *responder* node.
- $(d_{v\_processing\_async} + d_{queuing})$ is the time required for a non-responder *VC* node to submit its receipt share, which includes the processing and queuing delay. There is no N_v factor here, because processing and queuing happen in parallel across *VC* nodes. Thus, the *responder* node actually notices a delay equal to the processing and queuing time at a single *VC* node.

We now define the formula for d_{ucert_gen}, as follows:

$$d_{ucert\_gen} = 2N_v d_{v\_propagation} + (d_{e\_processing} + d_{queuing}) + N_v d_{queuing} + d_{sig\_prod} + N_v d_{sig\_ver} \quad (3.9)$$

In formula 3.9, the terms are as follows:

- $2N_v d_{v\_propagation}$ expresses the time required for the *responder* to send the `ENDORSE` message to all *VC* nodes, and for each *VC* node to respond with the `ENDORSEMENT` message.

- $(d_{e\_processing} + d_{queuing})$ is the time required for a non-*responder VC* node to produce its response (defined below), plus the delay of the generated `ENDORSE` message in the queue of the *VC* node. As already noted, there is no N_v factor here, because processing and queuing happen in parallel across *VC* nodes, so the time *responder* actually waits equals the processing and queuing time at a single *VC* node.

- $N_v d_{queuing}$ expresses the time all `ENDORSEMENT` messages wait in the queue of the *responder*.

- d_{sig_prod} is the time required for the *responder* to sign the `ENDORSEMENT` request (one signature production).

- $N_v d_{sig\_ver}$ expresses the time required for the *responder* to verify each received `ENDORSEMENT` message (one signature verification for each message).

$d_{e\_processing}$, the time required to produce an `ENDORSEMENT` response at a *VC* node, is defined as follows:

$$d_{e\_processing} = d_{sig\_ver} + d_{computation} + d_{sig\_prod} \tag{3.10}$$

In the above formula, $d_{computation}$ approximates the time required for the *VC* node to ensure there is no other vote code already endorsed for the specific ballot. Besides that, the *VC* node also verifies the signature of the *responder* in the `ENDORSE` message (d_{sig_ver}), and generates a signature of its own for the `ENDORSEMENT` reply (d_{sig_prod}).

Besides verifying its input (as in D-DEMOS/IC), each *VC* node receiving a `VOTE_P` message, also needs to verify the attached UCERT first. As the UCERT comprises $N_v - f_v$ distinct signatures, its processing time is given by the following formula:

$$d_{ucert\_ver} = (N_v - f_v) d_{sig\_ver} \tag{3.11}$$

The formula for $d_{v\_processing\_async}$, the time a *VC* node takes to produce its receipt share, is the following:

$$d_{v\_processing\_async} = d_{computation} + d_{ucert\_ver} = d_{computation} + (N_v - f_v) d_{sig\_ver} \tag{3.12}$$

Combining equations 3.7, 3.8, and 3.12, we derive the following formula for receipt generation time in D-DEMOS/Async:

$$\begin{aligned}
&d_{receipt\_async} = 2d_{c\_propagation} + d_{r\_processing\_async} + d_{queuing} \\
&= 2d_{c\_propagation} + 2d_{computation} + d_{ucert\_gen} + 2N_v d_{v\_propagation} + N_v d_{queuing} + d_{v\_processing\_async} \\
&\quad + d_{queuing} + d_{queuing} \\
&= 2d_{c\_propagation} + 2d_{computation} + 2N_v d_{v\_propagation} + d_{e\_processing} + d_{queuing} + N_v d_{queuing} + d_{sig\_prod} \\
&\quad + N_v d_{sig\_ver} + 2N_v d_{v\_propagation} + N_v d_{queuing} + d_{v\_processing\_async} + d_{queuing} + d_{queuing} \\
&= 2d_{c\_propagation} + 2d_{computation} + 2N_v d_{v\_propagation} + d_{sig\_ver} + d_{computation} + d_{sig\_prod} + d_{queuing} \\
&\quad + N_v d_{queuing} + d_{sig\_prod} + N_v d_{sig\_ver} + 2N_v d_{v\_propagation} + N_v d_{queuing} + d_{v\_processing\_async} \\
&\quad + d_{queuing} + d_{queuing} \\
&= 2d_{c\_propagation} + 2d_{computation} + 2N_v d_{v\_propagation} + d_{sig\_ver} + d_{computation} + d_{sig\_prod} + d_{queuing} \\
&\quad + N_v d_{queuing} + d_{sig\_prod} + N_v d_{sig\_ver} + 2N_v d_{v\_propagation} + N_v d_{queuing} + d_{computation} \\
&\quad + (N_v - f_v) d_{sig\_ver} + d_{queuing} + d_{queuing} \\
&= 2d_{c\_propagation} + 4N_v d_{v\_propagation} + (2N_v + 3) d_{queuing} + 4d_{computation} + 2d_{sig\_prod} \\
&\quad + (2N_v - f_v + 1) d_{sig\_ver}
\end{aligned} \tag{3.13}$$

3.7.2.3.2 Safety threshold formula.

If the voter sets the timeout, for Algorithm 1 when waiting for a receipt, to $d_{receipt\_async}$ (defined in formula 3.13 above), we estimate that a safe amount of time the voter can vote and expect her vote to be registered, to be:

$$safety\_threshold\_async = (f_v + 2) d_{receipt\_async} \tag{3.14}$$

In the above calculation, we assume the voter stumbles upon all malicious nodes first, then the first correct node times out (potentially due to high load), and a second correct one responds properly. Of course, this threshold can be set in a more conservative fashion by adjusting the factor to multiples of N_V, instead of f_v, as the voter cycles though all nodes while voting.

3.8 Remaining D-DEMOS system components

For completeness, we provide a small overview of the *BB* and the *trustees* subsystems, and a short description of the auditor role. Please note that these components are beyond the scope of this thesis. For details for these components, as well as the proofs for the privacy and end-to-end verifiability properties of D-DEMOS, we refer the interested reader to [34] and [35].

3.8.1 Bulletin Board

A *BB* node functions as a public repository of election-specific information. By definition, it can be read via a public and anonymous channel. Writes, on the other hand, happen over an authenticated channel, implemented with PKI originating from the voting system. *BB* nodes are independent from each other, as a *BB* node never directly contacts another *BB* node. Readers are expected to issue a read request to all *BB* nodes, and trust the reply that comes from the majority. Writers are also expected to write to all *BB* nodes; their submissions are always verified, and explained in more detail below. The *BB* subsystem consists of N_b nodes and tolerates $f_b < N_b/2$ faults.

After the setup phase, each *BB* node publishes its initialization data. During election hours, *BB* nodes remain inert. After the voting phase, each *BB* node receives from each *VC* node, the final vote-code set and the shares of msk. Once it receives $f_v + 1$ identical final vote code sets, it accepts and publishes the final vote code set. Once it receives $N_v - f_v$ valid key shares (again from *VC* nodes), it reconstructs the msk, decrypts all the encrypted vote codes in its initialization data, and publishes them.

At this point, the cryptographic payloads corresponding to the cast vote codes are made available to the *trustees*. *Trustees*, in turn, read from the *BB* subsystem, perform their individual calculations and then write to the *BB* nodes; these writes are verified by the *trustees*' keys, generated by the *EA*. Once enough *trustees* have posted valid data, the *BB* node combines them and publishes the final election result.

We intentionally designed the *BB* nodes to be as simple as possible for the reader, refraining from using a *Replicated State Machine*, which would require readers to run algorithm-specific software. The robustness of *BB* nodes comes from controlling all write accesses to them. Writes from *VC* nodes are verified against their honest majority threshold. Further writes are allowed only from *trustees*, verified by their keys.

Finally, a reader of the *BB* nodes should post her read request to all nodes, and accept what the majority responds with ($f_b + 1$ is enough). We acknowledge there might be temporary state divergence (among *BB* nodes), from the time a writer updates the first *BB* node, until the same writer updates the last *BB* node. However, given the fault-tolerance threshold, this should be only momentary, alleviated with simple retries. Thus, if there is no reply backed by a clear majority, the reader should retry until there is one.

3.8.2 *Trustees*

After the end of election hours, each *trustee* fetches all the election data from the *BB* subsystem and verifies its validity. For each ballot, there are two possible valid outcomes: i) one of the A/B parts are voted, ii) none of the A/B parts are voted. If both A/B parts of a ballot are marked as voted, then the ballot is considered as invalid and is discarded. Similarly, *trustees* also discard those ballots where more than one commitments in an A/B part are marked as voted.

In case (i), for each encoded option commitment in the unused part, Trustee$_\ell$ submits its corresponding share of the opening of the commitment to the *BB*. For each encoded option commitment in the voted part, Trustee$_\ell$ computes and posts the share of the final message of the corresponding zero knowledge proof, showing the validity of those commitments. Meanwhile, those commitments marked as voted are collected to a tally set $\mathbf{E}_{\text{tally}}$. In case (ii), for each encoded option commitment in both parts, Trustee$_\ell$ submits its corresponding share of the opening of the commitment to the *BB*. Finally, denote $\mathbf{D}^{(\ell)}_{\text{tally}}$ as Trustee$_\ell$'s set of shares of option encoding commitment openings, corresponding to the commitments in $\mathbf{E}_{\text{tally}}$. Trustee$_\ell$ computes the opening share for E_{sum} as $T_\ell = \sum_{D \in \mathbf{D}^{(\ell)}_{\text{tally}}}$ and then submits T_ℓ to each *BB* node.

3.8.3 Auditors

Auditors are participants of the D-DEMOS system who can verify the election process. The role of the auditor can be assumed by voters or any other party. After election end time, auditors read information from the *BB* and verify the correct execution of the election, by verifying the following:

1. within each opened ballot, no two vote codes are the same;
2. there are no two submitted vote codes associated with any single ballot part;
3. within each ballot, no more than one part has been used;
4. all the openings of the commitments are valid;
5. all the zero-knowledge proofs associated with the used ballot parts are completed and valid.

The auditors receive audit information (an unused ballot part and a cast vote code) from voters who wish to delegate verification, and they can also verify:

6. the submitted vote codes are consistent with the ones received from the voters;
7. the openings of the unused ballot parts are consistent with the ones received from the voters.

4. DISCUSSION

4.1 Why not State Machine Replication for *VC*

It can be argued that a far simpler design for the *VC* subsystem would be one using State Machine Replication (SMR). In such a design, *VC* nodes would run a Byzantine Fault Tolerate state machine replication protocol such as [25]. In fact, we analyzed this protocol early on in our research and provided solutions to some of its shortcomings in [33]. More specifically, we added management of clients that join and leave the system dynamically, and we integrated an SQL engine to manage the application level state in an application-friendly fashion. However, a BFT SMR protocol approach to vote collection poses two major problems.

The first problem is that of privacy, as such algorithms tolerate faults to liveness and safety, but not to privacy. That is, if the adversary takes over one of the nodes (still below the fault tolerance threshold), he obtains knowledge of all the data stored in the node. In our case, the adversary would gain access to all voters' ballots and would be able to vote on behalf of them. There has been work from Yin et al. [114], which proposed distinct roles for nodes of the system. The define *agreement* nodes that simply run the agreement protocol and have no access to application-specific data, and *execution* nodes that execute the requests and have access to application data. They also introduce a *privacy firewall* between agreement nodes and execution nodes. This approach however, makes the underlying assumption that execution nodes are physically restricted to communicate only with other execution nodes and firewall nodes. This is impossible to implement when targeting separate administrative domains, as we do. Additionally, it increases considerably the cost of a solution, because it requires a multitude of physical machines to implement a single logical replicated state machine (at least $5f + 1$ plus the firewall layer nodes).

The second problem is that such a solution would violate our goal of relieving the end-user from relying on cryptographic computations on the client device. This is because all SMR protocols require the client device to verify signatures of received messages when it decides if the reply is correct or not.

For these reasons, we refrained from using SMR and designed our custom voting protocol.

4.2 Potential attacks

In this section, we outline some of the possible attacks against the D-DEMOS systems, and the way our systems thwart them. This is a high level discussion, aiming to help the reader understand *why* our systems work reliably. In Sections 3.6.2 and 3.7.2 we provide the proofs of liveness and safety, which are the foundation of this discussion.

In this high-level description, we intentionally do not focus on Denial-of-Service attacks, as these kind of attacks attempt to stop the system from producing a result, or stop voters from casting their votes. Although these attacks are important, they cannot be hidden, as voters will notice immediately the system not responding (either because of our receipt mechanism and our liveness property, or because of lack of information in the *BB*). Instead, we focus on attacks on the correctness of the election result, as these have consequences simple voters cannot identify easily. In this discussion, we assume the fault thresholds of section 3.4 are not violated, and the attacker cannot violate the security of the underlying cryptographic primitives.

In this section, we focus on correctness, noting that our systems' privacy is achieved by the security of the cryptographic schemes used , and the partial initialization data that each node of the distributed subsystems receives during the setup phase.

4.2.1 Malicious Election Authority Component

At a high level, the *EA* produces vote codes and corresponding receipts. Vote codes are pointers to the associated cryptographic payload, which includes *option encodings*. Option encodings are used to produce the tally using homomorphic addition. If the *EA* misencodes any option, it will be identified by the Zero-Knowledge proof validation performed by the Auditors.

The *EA* may instead try to "point" a vote code to a valid but different option encoding (than the one described in the voter's ballot), in an attempt to manipulate the result. In this case, the *EA* cannot predict which one of the two ballot parts the voter will use. Recall that the unused part of the ballot will be opened in the *BB* by the *trustees*, and thus the voters can read and verify the correctness of their unused ballot parts.

If none of the above attacks take place, there is perfect consistency between each voter's ballot and its corresponding information on the *BB*. Because of this, as well as the correctness and the perfect hiding property of our commitment scheme, the homomorphic tally will be opened to the actual election result.

4.2.2 Malicious Voter

A malicious voter can try to submit multiple vote codes to the *VC* subsystem, attempting to cause disagreement between its nodes. In this case, a receipt *may* be generated, depending on the order of delivery of network messages. Note that, our safety *contract* allows our system to either accept only one vote code for this ballot, or discard the ballot altogether, as the voter is malicious and our contract holds only for honest voters.

In the D-DEMOS/IC case, this is resolved at the *Vote Set Consensus* phase. During the *voting* phase, each *VC* node accepts only the first vote code it receives (via either a `VOTE` or a `VOTE_P` message), and attempts to follow our *voting* protocol. This results in the generation of at most one receipt, for one of the posted vote codes. However, during *Vote Set Consensus*, honest *VC* nodes will typically identify the multiple posted vote codes and discard the ballot altogether, even if a receipt was indeed generated. If the ballot is not discarded (e.g., because malicious vote collector nodes hid the extra vote codes and honest nodes knew only of one), our $N_v - 2f_v$ threshold guarantees that no vote codes with generated receipts are discarded.

In the D-DEMOS/Async case, this is resolved completely at the *voting* phase. Each *VC* node still accepts only the first vote code it receives, but additionally attempts to build a UCERT for it. As the generation of a UCERT is guaranteed to be successful only for a single vote code, the outcome of the *voting* protocol will be either no UCERT being built, resulting in considering the ballot as not-voted, or a single UCERT generated.

Thus, the two systems behave differently in the case of multiple posted vote codes, as D-DEMOS/IC typically discards such ballots, while D-DEMOS/Async may process some of them, when a UCERT is successfully built.

4.2.3 Malicious Vote Collector

A malicious *VC* node cannot easily guess the vote codes in the voters' ballots, as they are randomly generated. Additionally, because vote codes are encrypted in the local state of each *VC* node, *VC* nodes cannot decode and use them. Note that, a vote code in a voter's ballot is considered private until the voter decides to use it and transmits it over the network. From this point on, the vote code can be intercepted by the attacker, as the only power it gives him is to cast it.

A malicious *VC* node can obtain vote codes from colluding malicious voters. In this case, the only possible attack on correctness is exactly the same as if it originated from the malicious voter herself, and we already described our counter-measures in Section 4.2.2.

A malicious *VC* node may become a *responder*. In this case, this *VC* node may *selectively* transmit the cast vote code to a subset of the remaining *VC* nodes, potentially including all the other malicious and colluding nodes, and deliver the receipt to an honest voter. Consequently, the attacker controlling the malicious entities, may try to "confuse" the honest *VC* nodes and have them disagree on whether the ballot is voted or not, by having all malicious *VC* nodes lie at *vote set consensus* time, reporting the ballot as not voted.

Recall that, for the receipt to be generated, $N_v - f_v$ *VC* nodes need to cooperate, of which up to f_v may be malicious. This leaves $N_v - 2f_v$ honest nodes always present.

In the case of D-DEMOS/IC, these $N_v - 2f_v$ honest nodes will show up in the per ballot cross-tabulation, and will drive the decision to mark the ballot as voted (note that, in the algorithm of Figure 3.4, $N_v - 2f_v$ is the lower threshold for a ballot to be marked as voted). In the case of D-DEMOS/Async, we include the `ANNOUNCE`-exchanging phase before the consensus algorithm, to guarantee at least one of the $N_v - 2f_v$ honest nodes' `ANNOUNCE` message will be processed by every honest node. In this case, on entering consensus, all honest nodes will agree that the ballot is voted, which guarantees the outcome of consensus to be in accordance.

5. IMPLEMENTATION

5.1 Infrastructure

The majority of the software for the prototype implementation of D-DEMOS is developed in Java. We use an asynchronous, event-based approach to software architecture in this work. To support this architecture, we build an *asynchronous communications stack* (ACS) on top of Java, using Netty [38] and the asynchronous PostgreSQL driver from [77].

We build a *Communicator* class that handles message passing for the upper layers of the application, while it uses the *Node* and *Message* concepts (see Figure 5.1 for a class diagram). A *Node* represents a network node that can be the recipient and source of a *Message*. A *Node* has an identifier, an IP Address and port number, and a public key, that are common across all instances of *ACS*. Additionally, nodes are grouped into node groups, allowing nodes to send messages to all nodes of a group. We use TLS authenticated channels for inter-node communication, and provide a public HTTP channel for client (voter) access. A *Message* is the object that is exchanged across nodes. It is specialized in descended classes, which are identified by a unique message identifier (an integer), and which provide type-specific serialization and deserialization operations. We create child classes of our *AbstractCommunicator* for TCP/TLS connections (*NettyCommunicator*) and for debugging purposes (*DummyCommunicator*).

Communicator works as follows. It is initialized with the *Node* instances of all nodes in the system (we use a static membership model). The application registers handlers for all messages it can process, via *registerMessage*. It then gives control to the *Communicator* instance via *start*. The *Communicator* initializes its structures and all network channels. With Netty, this means it starts secondary threads to handle network input, which decode incoming messages and place them in a shared input queue. We use Google Protocol Buffers [69] to encode and decode messages efficiently. In the main thread, *Communicator* enters the dispatch loop, where it calls the registered handler for each message in the input queue, passing along the sending *Node* as the *source*. The application uses one of *send*, *sendAll*, or *sendGroup* to send a message a single, all, or a group of nodes. Note that *Communicator* creates the connections (TLS channels) to remote nodes on demand, when the first message to a specific node is enqueued.

Our *NettyCommunicator* implementation supports a public HTTP channel with the *start(..., httpPort, httpParser)* operation. *NettyCommunicator* creates an HTTP listening socket on the specified port, and listens to connections. Once a connection is established, a new *AnonymousNode* instance is created to represent the client of this specific channel. Once an HTTP request is received, it is given to the specified httpParser, which transforms the request into a *Message* instance. This *Message* is then enqueued, along with *AnonymousNode* as the source. This way, the corresponding message handler can respond by sending a message to the source node, which in turn transforms it in plain text and puts it in the HTTP response.

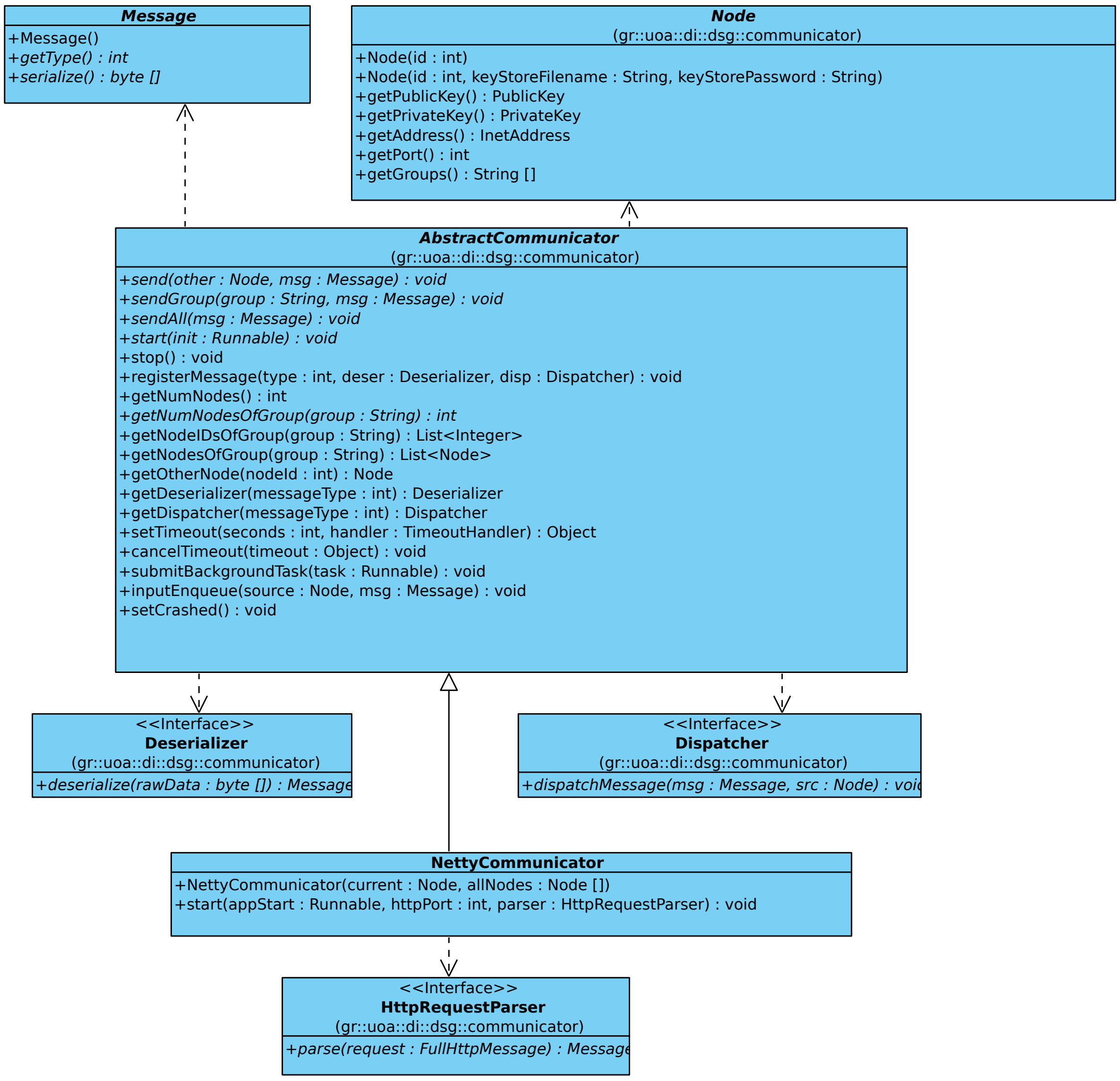

Message
+Message()
+getType() : int
+serialize() : byte []
Node
(gr::uoa::di::dsg::communicator)
+Node(id : int)
+Node(id : int, keyStoreFilename : String, keyStorePassword : String)
+getPublicKey() : PublicKey
+getPrivateKey() : PrivateKey
+getAddress() : InetAddress
+getPort() : int
+getGroups() : String []
AbstractCommunicator
(gr::uoa::di::dsg::communicator)
+send(other : Node, msg : Message) : void
+sendGroup(group : String, msg : Message) : void
+sendAll(msg : Message) : void
+start(init : Runnable) : void
+stop() : void
+registerMessage(type : int, deser : Deserializer, disp : Dispatcher) : void
+getNumNodes() : int
+getNumNodesOfGroup(group : String) : int
+getNodeIDsOfGroup(group : String) : List<Integer>
+getNodesOfGroup(group : String) : List<Node>
+getOtherNode(nodeId : int) : Node
+getDeserializer(messageType : int) : Deserializer
+getDispatcher(messageType : int) : Dispatcher
+setTimeout(seconds : int, handler : TimeoutHandler) : Object
+cancelTimeout(timeout : Object) : void
+submitBackgroundTask(task : Runnable) : void
+inputEnqueue(source : Node, msg : Message) : void
+setCrashed() : void
<<Interface>>
Deserializer
(gr::uoa::di::dsg::communicator)
+deserialize(rawData : byte []) : Message
<<Interface>>
Dispatcher
(gr::uoa::di::dsg::communicator)
+dispatchMessage(msg : Message, src : Node) : voi
NettyCommunicator
+NettyCommunicator(current : Node, allNodes : Node [])
+start(appStart : Runnable, httpPort : int, parser : HttpRequestParser) : void
<<Interface>>
HttpRequestParser
(gr::uoa::di::dsg::communicator)
+parse(request : FullHttpMessage) : Message

Figure 5.1: A UML Class Diagram of the Asynchronous Communications Stack.

Besides *Messages*, *Communicator* also supports setting and getting notifications of *Timeouts*. This allows nodes to progress in case of lack of complete input (normally due to node crashes). Our *Communicator* implementation also allows for simulating crashes, by marking a specific node as crashed (see *setCrashed*), which simulates a crashed node by throwing away all application-level messages.

Note that *Communicator* does not handle multiplexing multiple instances of a specific protocol. We delegate this to the initial message handler, which maintains a map of instances and forwards it to the handler of the appropriate instance. For example, each binary consensus instance is identified by a consensus id, and a map stores the mapping from a consensus id to a ConsensusInstance class. The initial message handler for any consensus message creates a new instance of the ConsensusInstance class to handle incoming messages for consensus ids not in the map of instances.

Communicator allows offloading long-running tasks to secondary threads. We use this interface to offload signature operations (signing, verification) to secondary threads. An event handler calls *submitBackgroundTask*, which schedules the operation a thread pool. Once the result is ready, it is enqueued in the input queue. Postgresql [39] database operations are scheduled via the asynchronous PostgreSQL driver from [77], which uses its own pool of database connections, and results are again enqueued in the input queue.

5.2 D-DEMOS Election Authority

For this work, we obtained the Election Authority code from the original (centralized) DEMOS voting system, implemented in C++, and extended it to generate the receipts and split the *VC* initialization data across *VC* nodes. We use the MIRACL library [91] for elliptic-curve cryptographic operations.

The *EA* is a stand-alone executable than takes as input the number of voters, the number of options and the list of addresses of *VC* nodes. It proceeds to generate the ballots for the voters, and the initialization data for *VC* nodes, *BB* nodes and *trustees*.

5.3 *VC* node

We implement the *VC* node in Java, on top of *ACS*.

5.3.1 Voting

We define a `VOTE` and `VOTE_P` messages as described in Section 3.4. We also add an election id parameter, to allow handling of multiple elections. We then define a *VoteCollector* class which handles processing of above messages. All data access for a specific election is abstracted behind an *Election* object, with an *ElectionManager* class handling the mapping between an election identifier and an *Election* object. We provide different

Election implementations, one which accesses the database for every operation, and another witch loads all ballots in memory on initialization and does not access the database during voting.

We depict these classes in the UML diagram of Figure 5.2.

The public HTTP channel of the vote collector node offers an HTML form that allows the voter to input the serial number and vote code of her choice. Once this form is received, the input is transformed to a `VOTE` message, and the *responder VC* node starts the voting protocol, communicates with the other *VC* nodes, recreates the receipt, and responds with it to the voter using the original HTTP channel.

We implement "verifiable secret sharing with honest dealer", by utilizing Shamir's Secret Share library implementation [108], and having the *EA* sign each share. This way, each node can verify a share it receives.

5.3.2 Vote Set Consensus

For vote set consensus in D-DEMOS/IC, we use the implementation of *IC,BC-RBB* (Interactive Consistency algorithm, using asynchronous binary consensus and reliable broadcast without signatures) from [46]. We use the election end time as a synchronization point to start the algorithm, and configure the timeout of the first phase of the algorithm according to the number of *VC* nodes and the number of ballots in the election. Note that *IC,BC-RBB* is implemented on top of the same ACS described above and thus is smoothly integrated in our system.

For D-DEMOS/Async, we implement Bracha's Binary Consensus directly on top of the ACS (see Figure 5.3), and we use that to implement our Vote Set Consensus algorithm (depicted in Figure 3.6).

Bracha's consensus includes a Reliable Broadcast Mechanism, which we implement in the *Broadcast* class. The class is passed a handler for accepted broadcast on construction. The node begins a new broadcast using the *broadcast* operation, specifying the consensus and broadcast ids, and the value as an array of integers.

Using the broadcast mechanism, we implement a *ConsensusManager* which manages multiple instances of a *ConsensusInstance*. Again, the handler for finished broadcasts is passed at the constructor, and de-multiplexing is handled using the Consensus Id of each instance. Our implementation of Bracha's consensus, handles all corner cases of a malicious node submitting multiple messages from the same round. We handle this by keeping separate structures for messages from each origin, and either looking into any one of them when trying to validate remote input, or selecting only the first one when we decide for the local node's next round value.

Finally, we introduce a version of Binary Consensus that operates in batches of arbitrary size; this way, we achieve greater network efficiency while running one instance of Binary Consensus for each ballot. Capitalizing on the broadcast primitive's ability to handle

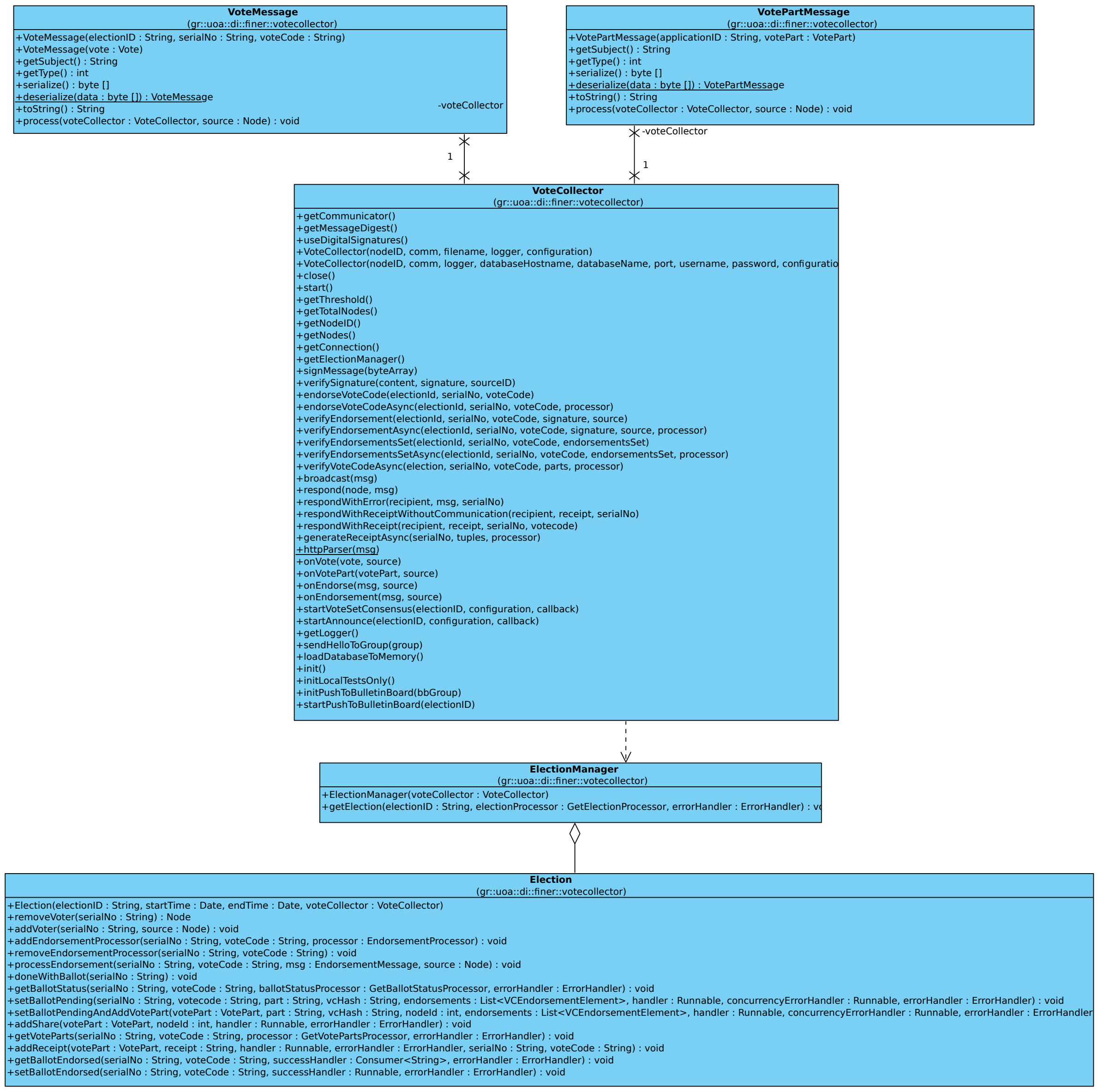

VoteMessage
(gr::uoa::di::finer::votecollector)
+VoteMessage(electionID : String, serialNo : String, voteCode : String)
+VoteMessage(vote : Vote)
+getSubject() : String
+getType() : int
+serialize() : byte []
+deserialize(data : byte []) : VoteMessage
+toString() : String
+process(voteCollector : VoteCollector, source : Node) : void
-voteCollector
1
VotePartMessage
(gr::uoa::di::finer::votecollector)
+VotePartMessage(applicationID : String, votePart : VotePart)
+getSubject() : String
+getType() : int
+serialize() : byte []
+deserialize(data : byte []) : VotePartMessage
+toString() : String
+process(voteCollector : VoteCollector, source : Node) : void
-voteCollector
1
VoteCollector
(gr::uoa::di::finer::votecollector)
+getCommunicator()
+getMessageDigest()
+useDigitalSignatures()
+VoteCollector(nodeID, comm, filename, logger, configuration)
+VoteCollector(nodeID, comm, logger, databaseHostname, databaseName, port, username, password, configuratio
+close()
+start()
+getThreshold()
+getTotalNodes()
+getNodeID()
+getNodes()
+getConnection()
+getElectionManager()
+signMessage(byteArray)
+verifySignature(content, signature, sourceID)
+endorseVoteCode(electionId, serialNo, voteCode)
+endorseVoteCodeAsync(electionId, serialNo, voteCode, processor)
+verifyEndorsement(electionId, serialNo, voteCode, signature, source)
+verifyEndorsementAsync(electionId, serialNo, voteCode, signature, source, processor)
+verifyEndorsementsSet(electionId, serialNo, voteCode, endorsementsSet)
+verifyEndorsementsSetAsync(electionId, serialNo, voteCode, endorsementsSet, processor)
+verifyVoteCodeAsync(election, serialNo, voteCode, parts, processor)
+broadcast(msg)
+respond(node, msg)
+respondWithError(recipient, msg, serialNo)
+respondWithReceiptWithoutCommunication(recipient, receipt, serialNo)
+respondWithReceipt(recipient, receipt, serialNo, votecode)
+generateReceiptAsync(serialNo, tuples, processor)
+httpParser(msg)
+onVote(vote, source)
+onVotePart(votePart, source)
+onEndorse(msg, source)
+onEndorsement(msg, source)
+startVoteSetConsensus(electionID, configuration, callback)
+startAnnounce(electionID, configuration, callback)
+getLogger()
+sendHelloToGroup(group)
+loadDatabaseToMemory()
+init()
+initLocalTestsOnly()
+initPushToBulletinBoard(bbGroup)
+startPushToBulletinBoard(electionID)
ElectionManager
(gr::uoa::di::finer::votecollector)
+ElectionManager(voteCollector : VoteCollector)
+getElection(electionID : String, electionProcessor : GetElectionProcessor, errorHandler : ErrorHandler) : vo
Election
(gr::uoa::di::finer::votecollector)
+Election(electionID : String, startTime : Date, endTime : Date, voteCollector : VoteCollector)
+removeVoter(serialNo : String) : Node
+addVoter(serialNo : String, source : Node) : void
+addEndorsementProcessor(serialNo : String, voteCode : String, processor : EndorsementProcessor) : void
+removeEndorsementProcessor(serialNo : String, voteCode : String) : void
+processEndorsement(serialNo : String, voteCode : String, msg : EndorsementMessage, source : Node) : void
+doneWithBallot(serialNo : String) : void
+getBallotStatus(serialNo : String, voteCode : String, ballotStatusProcessor : GetBallotStatusProcessor, errorHandler : ErrorHandler) : void
+setBallotPending(serialNo : String, votecode : String, part : String, vcHash : String, endorsements : List<VCEndorsementElement>, handler : Runnable, concurrencyErrorHandler : Runnable, errorHandler : ErrorHandler) : void
+setBallotPendingAndAddVotePart(votePart : VotePart, part : String, vcHash : String, nodeId : int, endorsements : List<VCEndorsementElement>, handler : Runnable, concurrencyErrorHandler : Runnable, errorHandler : ErrorHandler
+addShare(votePart : VotePart, nodeId : int, handler : Runnable, errorHandler : ErrorHandler) : void
+getVoteParts(serialNo : String, voteCode : String, processor : GetVotePartsProcessor, errorHandler : ErrorHandler) : void
+addReceipt(votePart : VotePart, receipt : String, handler : Runnable, errorHandler : ErrorHandler, serialNo : String, voteCode : String) : void
+getBallotEndorsed(serialNo : String, voteCode : String, successHandler : Consumer<String>, errorHandler : ErrorHandler) : void
+setBallotEndorsed(serialNo : String, voteCode : String, successHandler : Runnable, errorHandler : ErrorHandler) : void

Figure 5.2: A UML Class Diagram of the voting protocol related classes of the Vote Collector implementation.

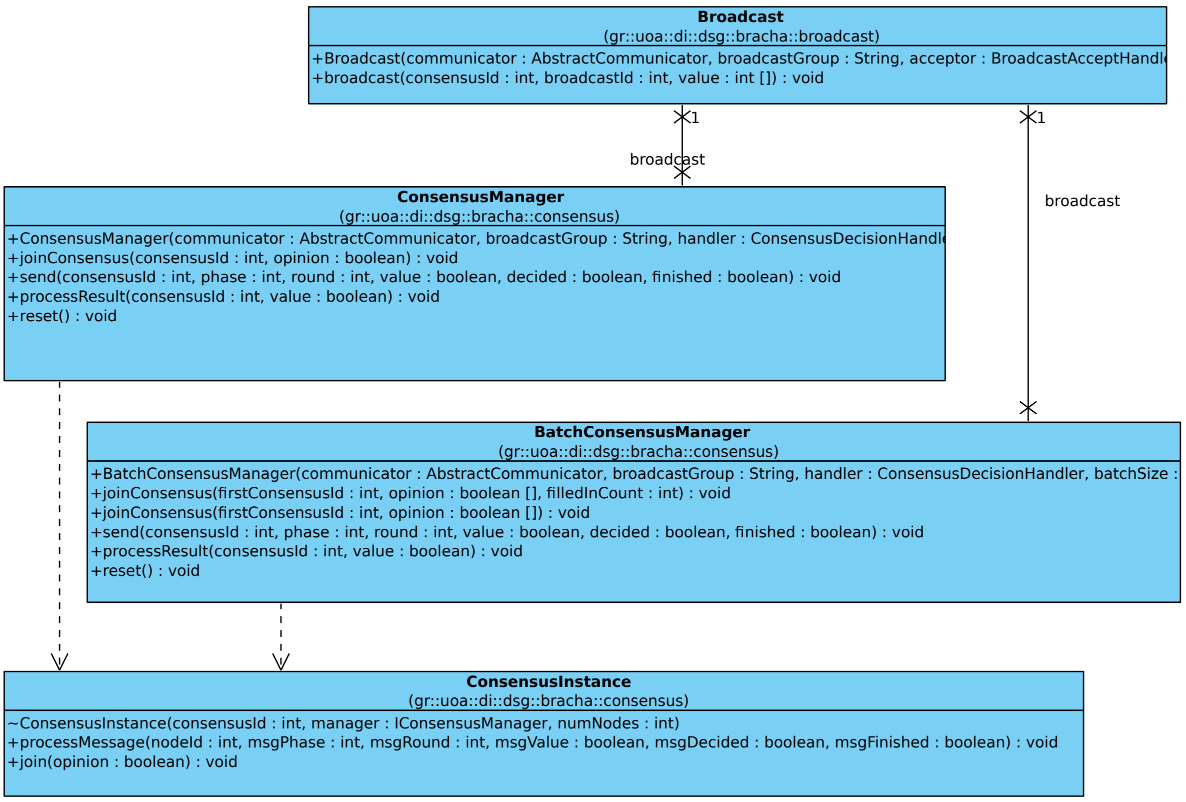

Broadcast
(gr::uoa::di::dsg::bracha::broadcast)
+Broadcast(communicator : AbstractCommunicator, broadcastGroup : String, acceptor : BroadcastAcceptHandl
+broadcast(consensusId : int, broadcastId : int, value : int []) : void
1
broadcast
1
broadcast
ConsensusManager
(gr::uoa::di::dsg::bracha::consensus)
+ConsensusManager(communicator : AbstractCommunicator, broadcastGroup : String, handler : ConsensusDecisionHandl
+joinConsensus(consensusId : int, opinion : boolean) : void
+send(consensusId : int, phase : int, round : int, value : boolean, decided : boolean, finished : boolean) : void
+processResult(consensusId : int, value : boolean) : void
+reset() : void
BatchConsensusManager
(gr::uoa::di::dsg::bracha::consensus)
+BatchConsensusManager(communicator : AbstractCommunicator, broadcastGroup : String, handler : ConsensusDecisionHandler, batchSize :
+joinConsensus(firstConsensusId : int, opinion : boolean [], filledInCount : int) : void
+joinConsensus(firstConsensusId : int, opinion : boolean []) : void
+send(consensusId : int, phase : int, round : int, value : boolean, decided : boolean, finished : boolean) : void
+processResult(consensusId : int, value : boolean) : void
+reset() : void
ConsensusInstance
(gr::uoa::di::dsg::bracha::consensus)
~ConsensusInstance(consensusId : int, manager : IConsensusManager, numNodes : int)
+processMessage(nodeId : int, msgPhase : int, msgRound : int, msgValue : boolean, msgDecided : boolean, msgFinished : boolean) : void
+join(opinion : boolean) : void

Figure 5.3: A UML Class Diagram of the Bracha Consensus implementation.

arrays of integers, we encode different broadcasts into such arrays. The class that implements this is *BatchConsensusManager*, which is initialized with a batch size. The batch version of the consensus manager expects the local node to join a series of consensus instances at once, by providing an array of bit values. It uses the same implementation of *ConsensusInstance* but waits for all instances to output their messages, batches them and starts a single consensus instance for all of them. This reduces greatly the number of messages required for the batch.

Besides binary consensus batching, we batch most of the asynchronous vote set consensus "announce" phase's messages. If this phase was implemented without optimization, it would result in a message complexity of $n * N_v$ (individual ANNOUNCE messages), imposing a significant network load. This is because each node has to multicast an `ANNOUNCE` message for each ballot, and wait for $n(N_v - f_v)$ replies to progress.

To optimize this, we have each node consult its local database and detect cases where another node already knows the correct vote code and UCERT for a specific ballot. This is feasible because when a node VC_b discloses its share using the VOTE_P message, it also includes the UCERT, and this fact is recorded in the recipient's node (VC_a) database along with the sender node's share. For these cases, we produce ANNOUNCE_RANGE messages addressed to individual nodes, having the source node VC_a announce a range of ballot serial numbers as voted, a fact that is already known to the recipient node VC_b (because VC_a located the recorded VOTE_P messages from VC_b). We use the same mechanism to announce ranges of not-voted ballots.

6. EVALUATION

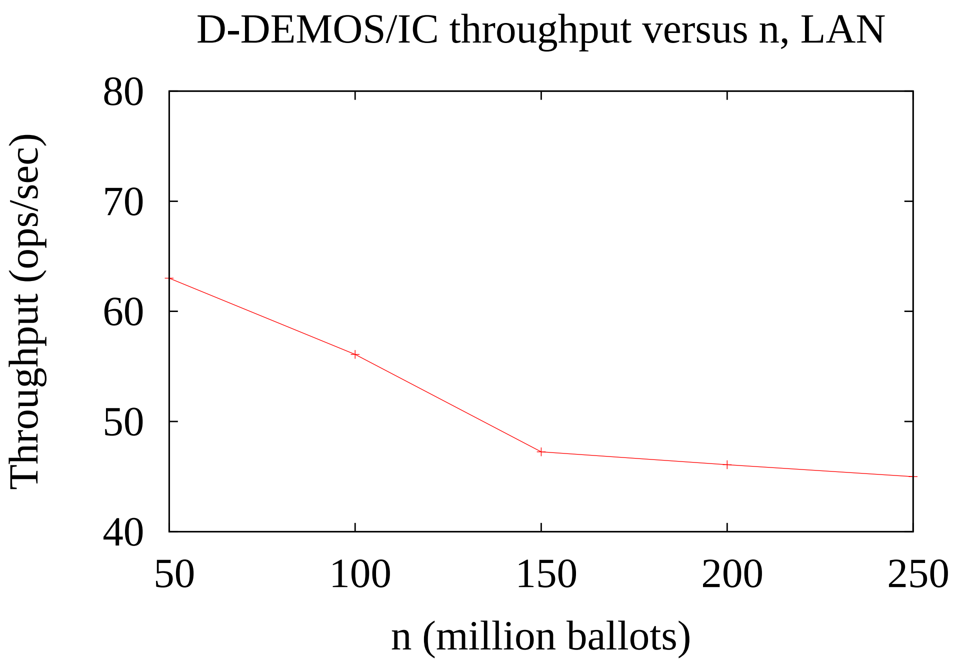
We experimentally evaluate the performance of our voting system, focusing mostly on our vote collection algorithm, which is the most performance critical part. We conduct our experiments using a cluster of 12 machines, connected over a Gigabit Ethernet switch. The first 4 are equipped with Hexa-core Intel Xeon E5-2420 @ 1.90GHz, 16GB RAM, and one 1TB SATA disk, running CentOS 7 Linux, and we use them to run our VC nodes. The remaining 8 comprise dual Intel(R) Xeon(TM) CPUs @ 2.80GHz, with 4GB of main memory, and two 50GB disks, running CentOS 6 Linux, and we use them as clients.

We implement a multi-threaded voting client to simulate concurrency. This client starts the requested number of threads, each of which loads its corresponding ballots from disk and waits for a signal to start. From then on, the thread enters a loop where it picks one VC node and vote code at random, requests the voting page from the selected VC (HTTP GET), submits its vote (HTTP POST), and waits for the reply (receipt). This simulates multiple concurrent voters casting their votes in parallel, and gives an understanding of the behavior of the system under the corresponding load. We employ the PostgreSQL RDBMS [39] to store all VC initialization data from the *EA*.

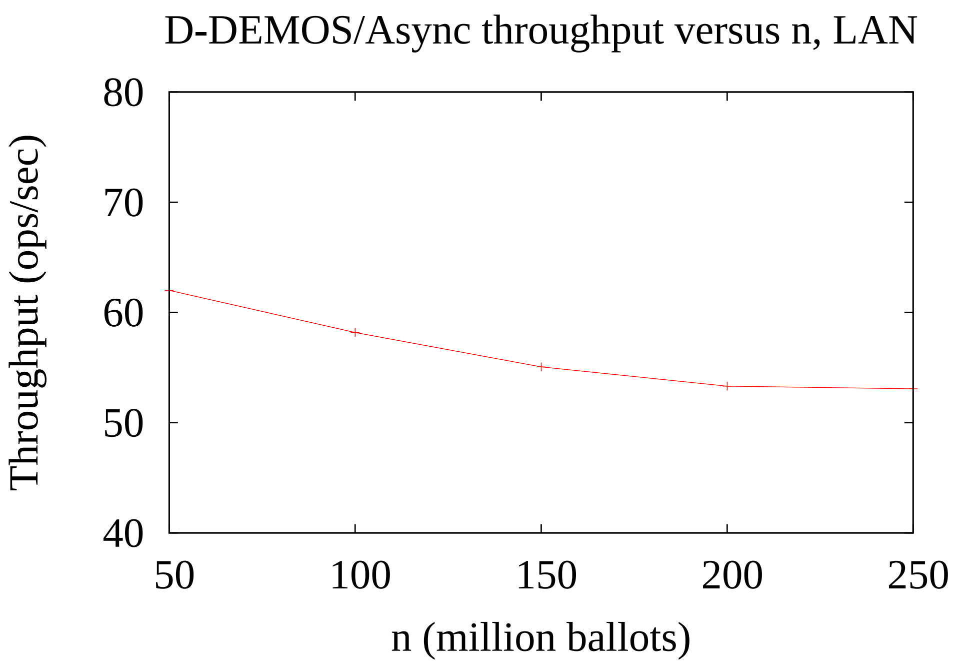
We start off by demonstrating our system's capability of handling large-scale elections. To this end, we generate election data for referendums, i.e., $m = 2$, and vary the total number of ballots n from 50 million to 250 million (note the 2012 US voting population size was 235 million). This causes the database size to increase accordingly and impact queries. We fix the number of concurrent clients to 400 and cast a total of 200,000 ballots, which are enough for our system to reach its steady-state operation (larger experiments result in the same throughput). Figure 6.1 shows the throughput of both D-DEMOS/IC and D-DEMOS/Async declines slowly, even with a five-fold increase in the number of eligible voters. The cause of the decline is the increase of the database size.

In our second experiment, we explore the effect of m, i.e., the number of election options, on system performance. We vary the number of options from $m = 2$ to $m = 10$. Each election has a total of $n = 200,000$ ballots which we spread evenly across 400 concurrent clients. As illustrated in Figure 6.2, our vote collection protocol manages to deliver approximately the same throughput regardless of the value of m, for both D-DEMOS/IC and D-DEMOS/Async. Notice that the major extra overhead m induces during vote collection, is the increase in the number of hash verifications during vote code validation, as there are more vote codes per ballot. The increase in number of options has a minor impact on the database size as well (as each ballots has $2m$ options).

Next, we evaluate the scalability of our vote collection protocol by varying the number of vote collectors and concurrent clients. We eliminate the database, by caching the election data in memory and servicing voters from the cache, to measure the net communication and processing costs of our voting protocol. We vary the number of VC nodes from 4 to 16, and distribute them across the 4 physical machines. Note that, co-located nodes are unable to produce vote receipts via local messages only, since the $N_v - f_v$ threshold cannot be satisfied, i.e., cross-machine communication is still the dominant factor in receipt

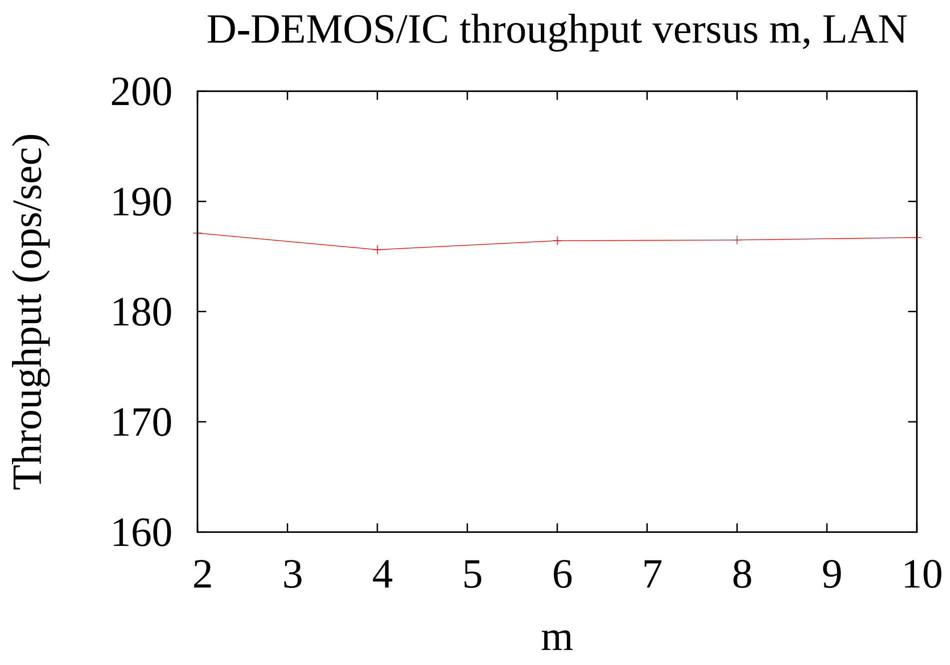


(a)

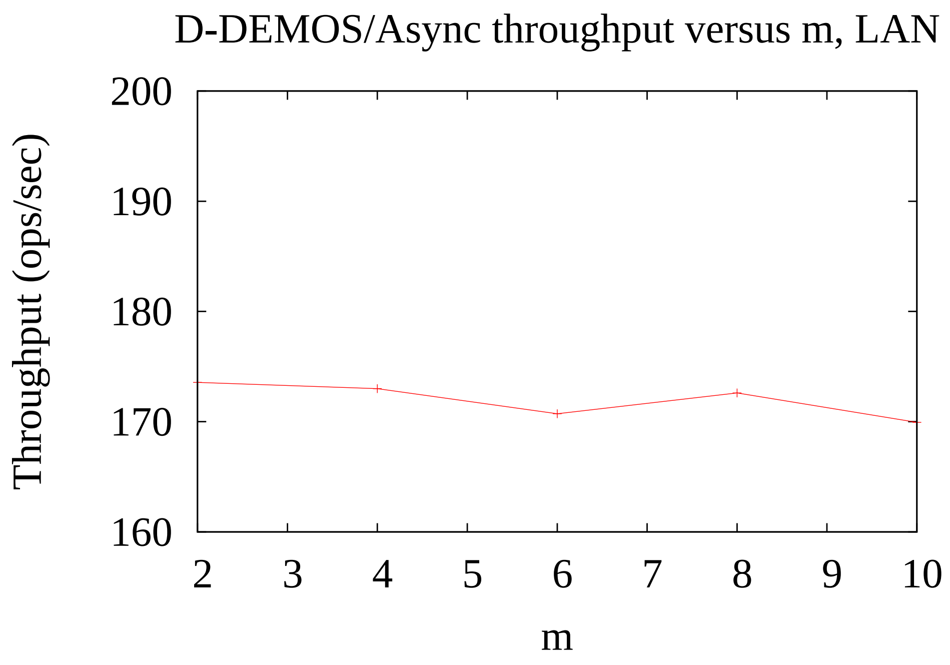


(b)

Figure 6.1: Vote collection throughput graphs for D-DEMOS/IC (6.1a) and D-DEMOS/Async(6.1b), versus the number of total election ballots n.



(a)



(b)

Figure 6.2: Vote collection throughput graphs for D-DEMOS/IC (6.2a) and D-DEMOS/Async(6.2b), versus the number of election options m.

generation. For election data, we use the dataset with $n = 200,000$ ballots and $m = 4$ options, which is enough for our system to reach its steady state.

In Figure 6.3, we plot the average response time of both our vote collection protocols, versus the number of vote collectors, under different concurrency levels, ranging from 500 to 2000 concurrent clients. Results for both systems illustrate an almost linear increase in the client-perceived latency, for all concurrency scenarios, up to 13 *VC* nodes. From this point on, when four logical *VC* nodes are placed on a single physical machine, we notice a non-linear increase in latency. We attribute this to the overloading of the memory bus, a resource shared among all processors of the system, which services all (in-memory) database operations. D-DEMOS/IC has a slower response time with its single round intra-*VC* node communication, while D-DEMOS/Async is slightly slower due to the extra Uniqueness Certificate round.

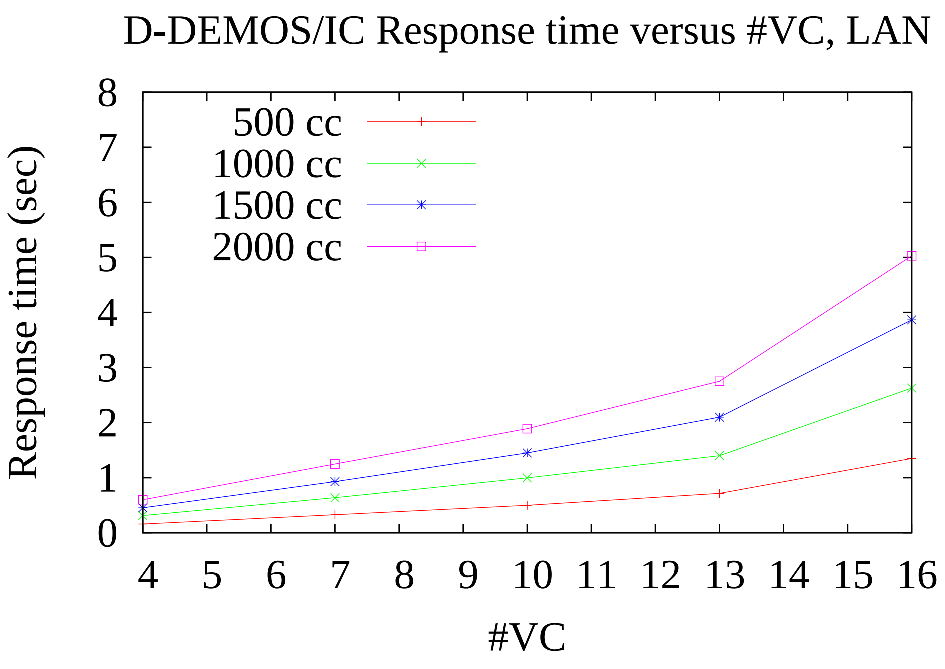
Figure 6.4 shows the throughput of both our vote collection protocols, versus the number of vote collectors, under different concurrency levels. We observe that, in terms of overall system throughput, the penalty of tolerating extra failures (increasing the number of vote collectors) manifests early on. We notice an almost 50% decline in system throughput from 4 to 7 *VC* nodes for D-DEMOS/IC, and a bigger one for D-DEMOS/Async. However, further increases in the number of vote collectors lead to a much smoother, linear decrease. Overall, D-DEMOS/IC achieves better throughput than D-DEMOS/Async, due to exchanging fewer messages and lacking signature operations.

In Figure 6.5, we plot a different view of both our systems' throughput, this time versus the concurrency level (ranging from 100 to 2000). Plots represent number of *VC* node settings (4 to 16), thus different fault tolerance levels. Results show both our systems have the nice property of delivering nearly constant throughput, regardless of the incoming request load, for a given number of *VC* nodes.

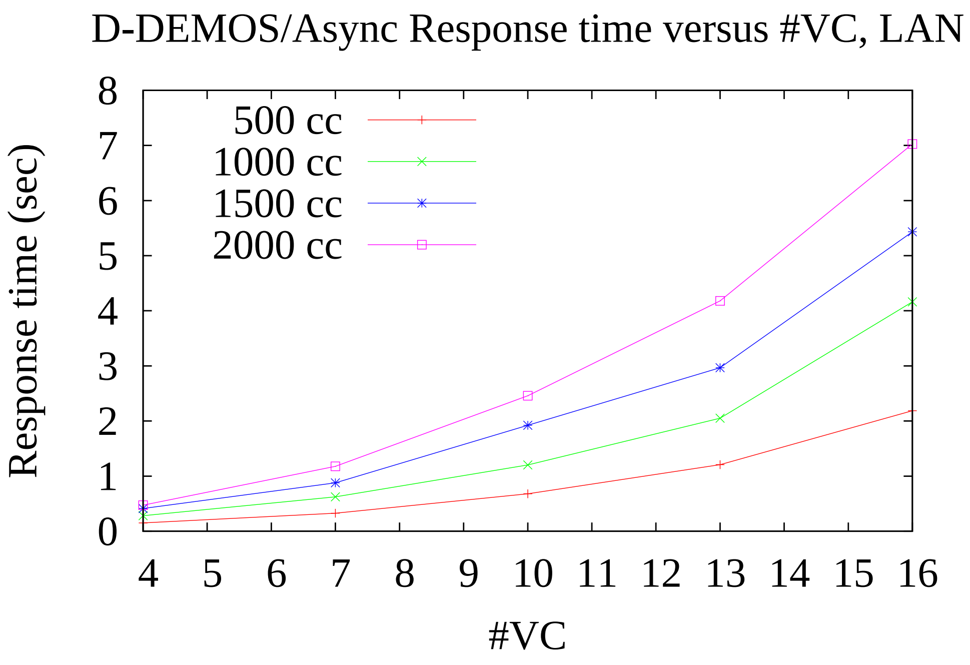
We repeat the same experiment by emulating a WAN environment using *netem* [68], a network emulator for Linux. We inject a uniform latency of 25ms (typical for US coast-to-coast communication [62]) for each network packet exchanged between vote collector nodes, and present our results in Figures 6.6, 6.7, and 6.8. A simple comparison between LAN and WAN plots illustrates our system manages to deliver the same level of throughput and average response time, regardless of the increased intra-*VC* communication latency.

The benefits of the in-memory approach, expressed both in terms of sub-second client (voter) response time and increased system throughput, make it an attractive alternative to the more standard database setup. For instance, in cases where high-end server machines are available, it would be possible to service mid to large scale elections completely from memory. We estimate the size of the in-memory representation of a $n = 200K$ ballot election, with $m = 4$ options, at approximately 322MB (see [87] for derivation details). In this size, we include 64-bit Java pointers overhead, as we are using simple hash-maps of plain old Java classes. This size can be decreased considerably in a more elaborate implementation, where data is serialized by Google Protocol Buffers, for example.

Finally, in Figure 6.9, we illustrate a breakdown of the duration of the two phases for Vote Collection, voting and vote set consensus, for D-DEMOS/IC and D-DEMOS/Async, versus

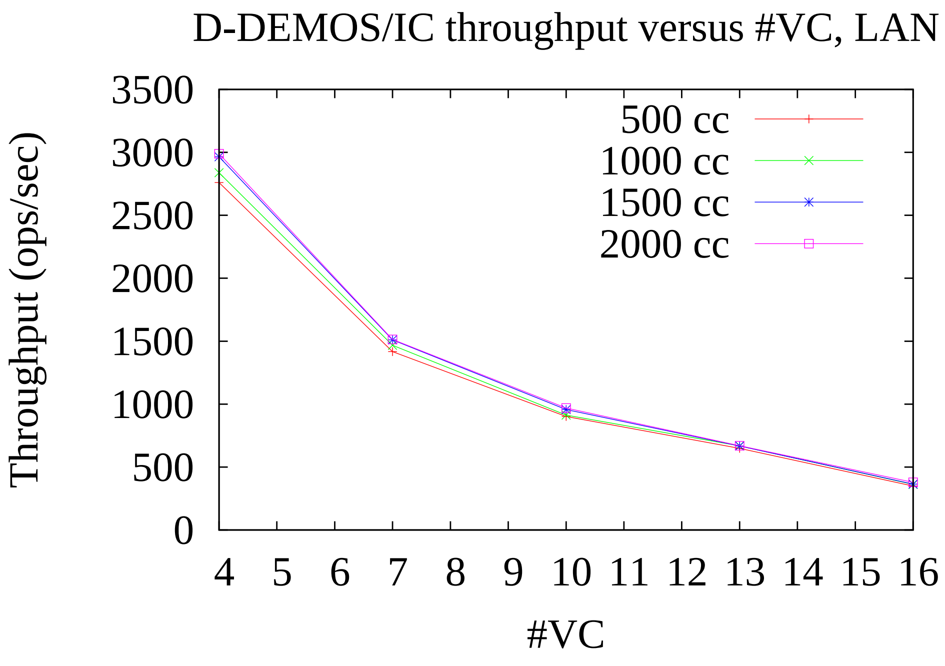


(a)

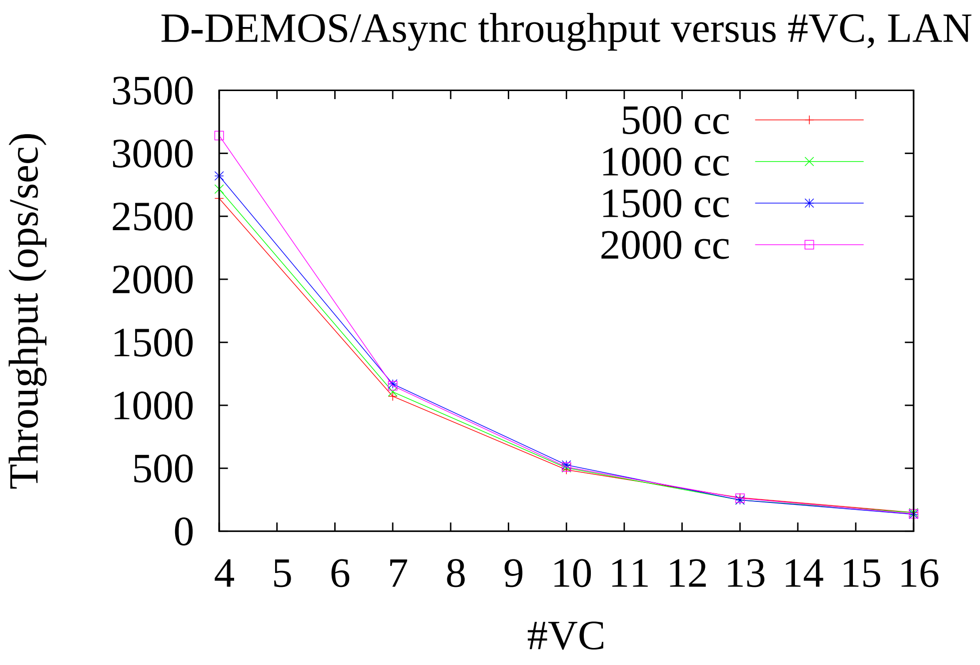


(b)

Figure 6.3: Vote Collection response time of D-DEMOS/IC (6.3a) and D-DEMOS/Async (6.3b), versus the number of *VC* nodes, under a LAN setting. Election parameters are n = 200,000 and m = 4.

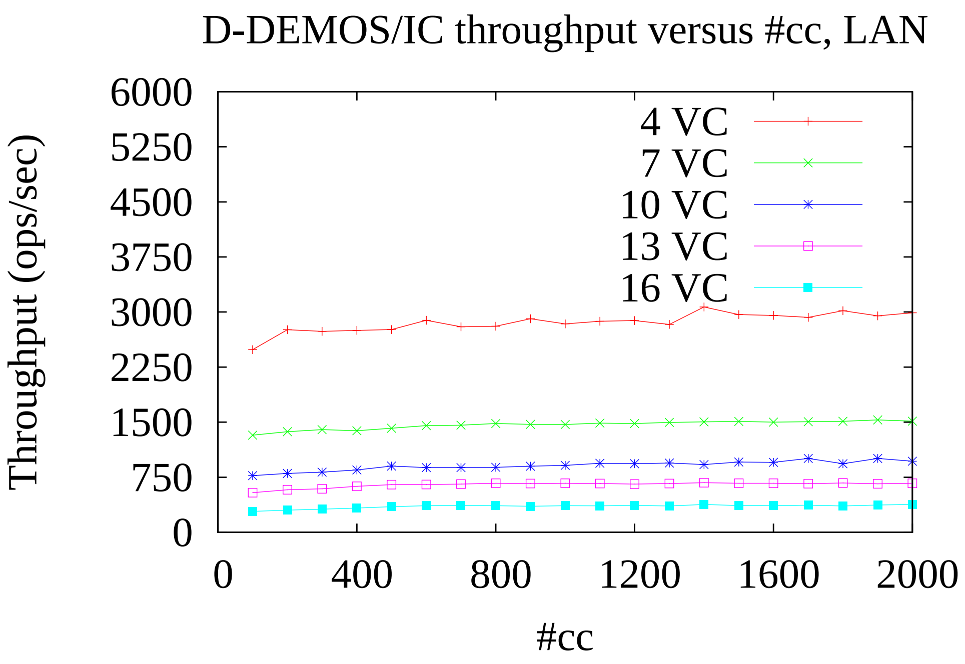


(a)

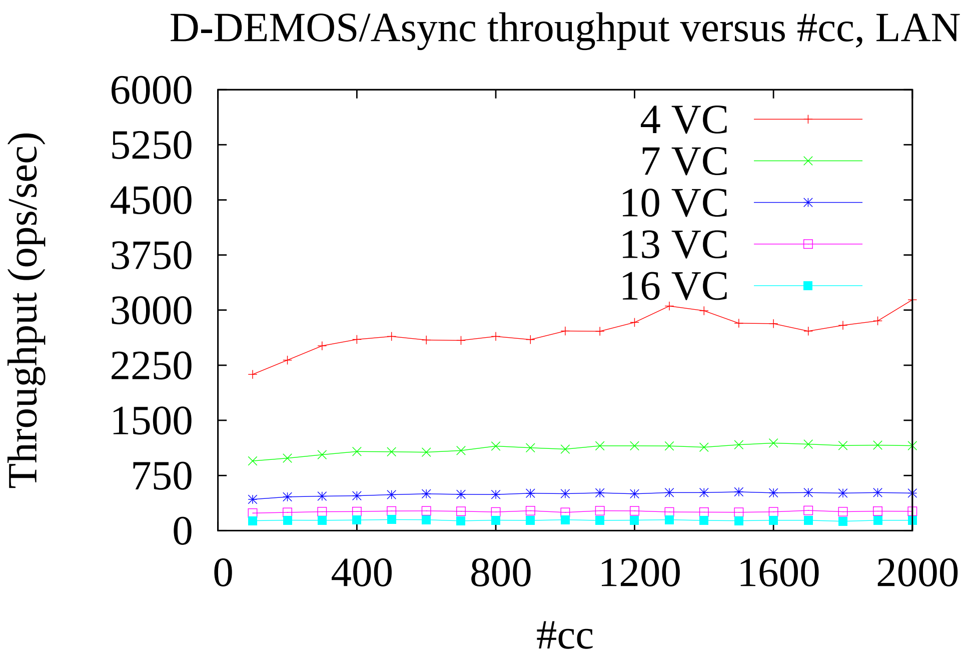


(b)

Figure 6.4: Vote Collection throughput of D-DEMOS/IC (6.4a) and D-DEMOS/Async (6.4b), versus the number of *VC* nodes, under a LAN setting. Election parameters are n = 200,000 and m = 4.

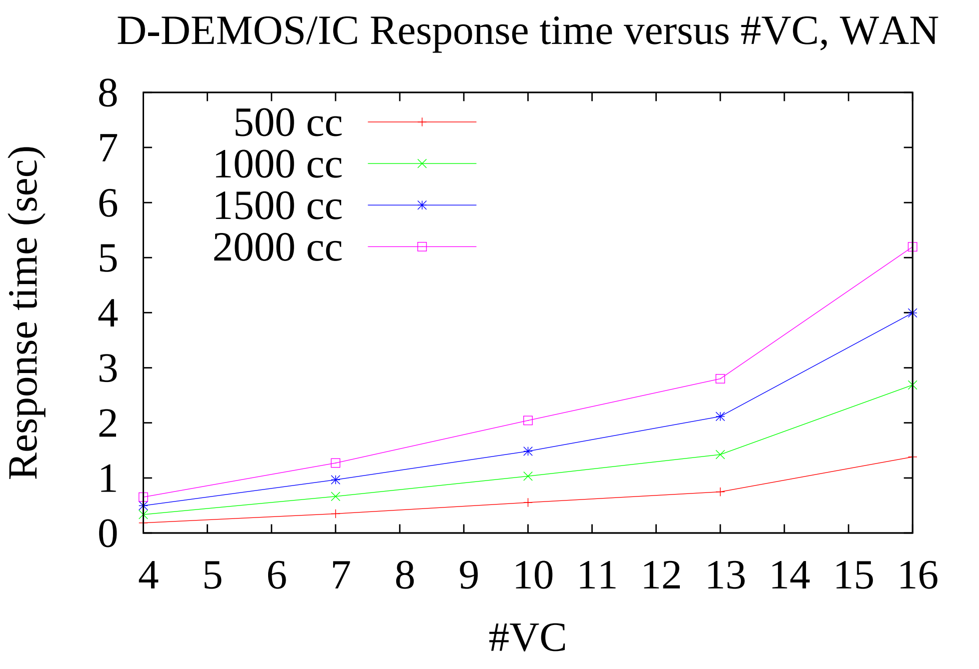


(a)

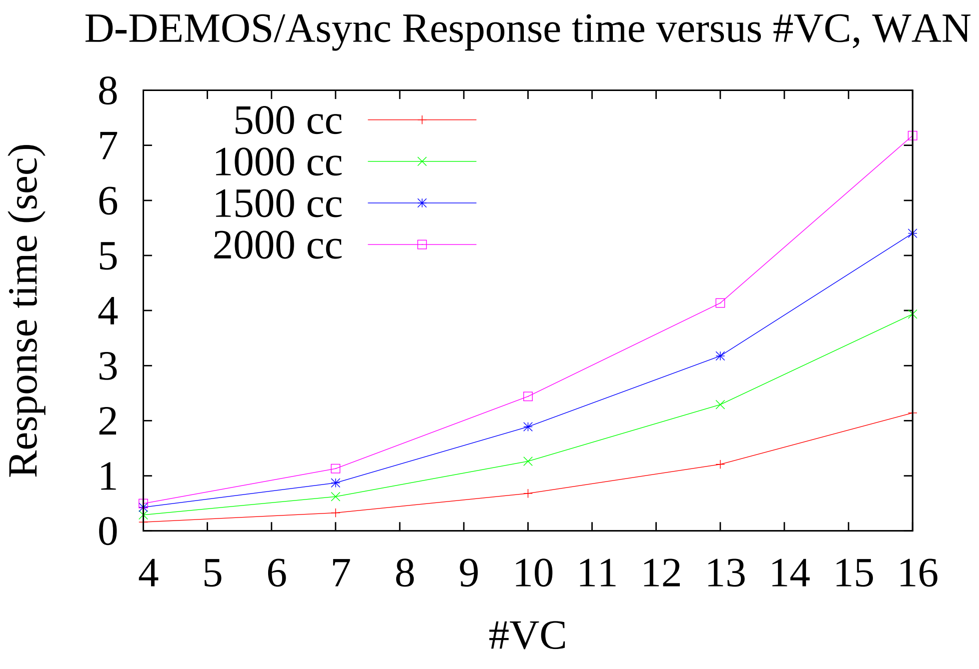


(b)

Figure 6.5: Vote Collection throughput of D-DEMOS/IC (6.5a) and D-DEMOS/Async (6.5b), versus the number of concurrent clients, under a LAN setting. Plots illustrate performance for different cardinalities of *VC* nodes, thus different fault tolerance settings. Election parameters are n = 200,000 and m = 4.

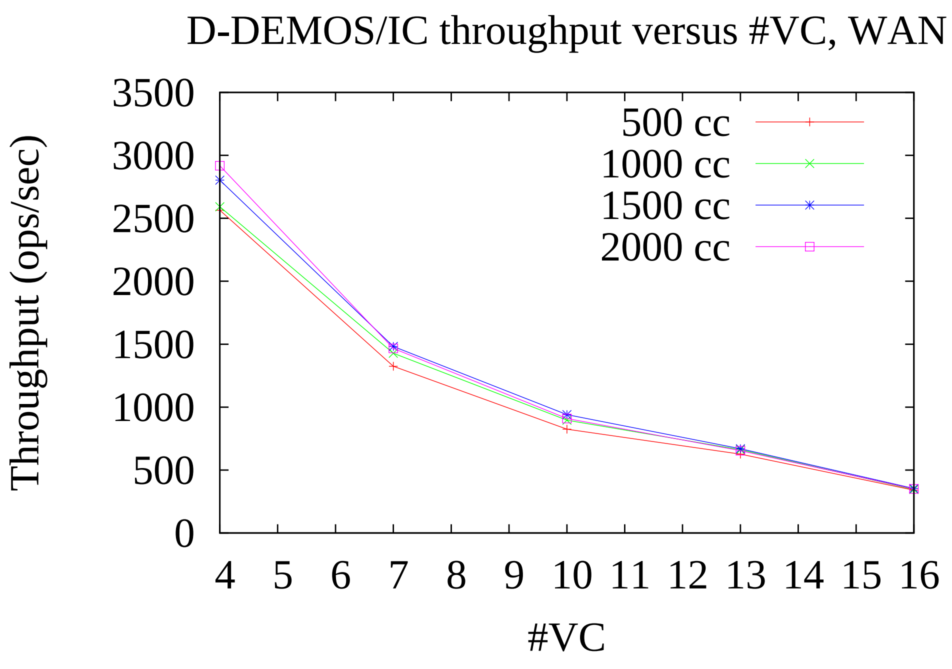


(a)

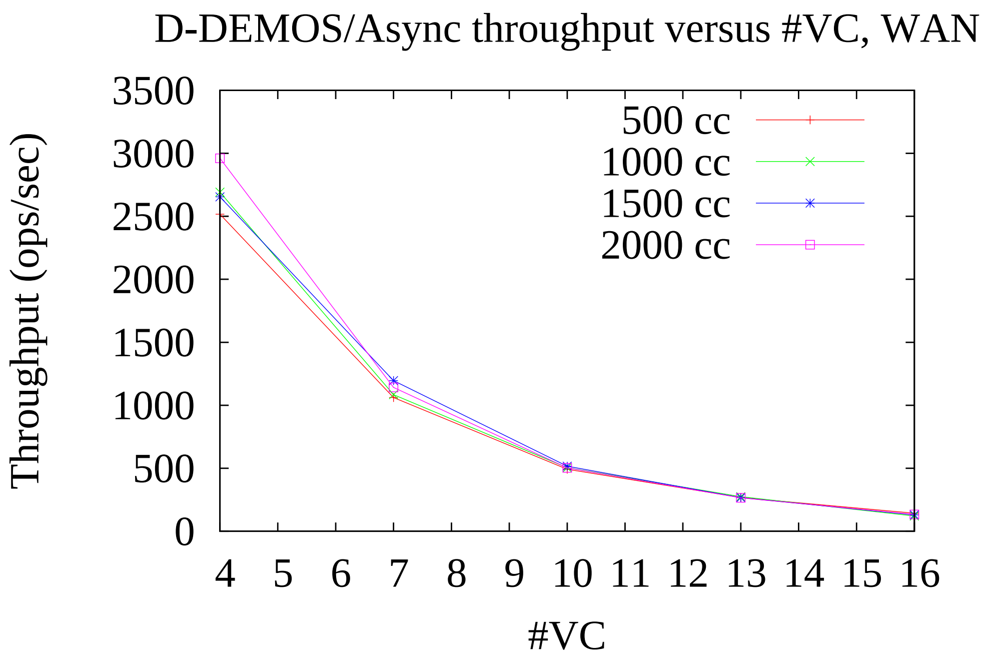


(b)

Figure 6.6: Vote Collection response time of D-DEMOS/IC (6.6a) and D-DEMOS/Async (6.6b), versus the number of *VC* nodes, under a WAN setting. Election parameters are n = 200,000 and m = 4.

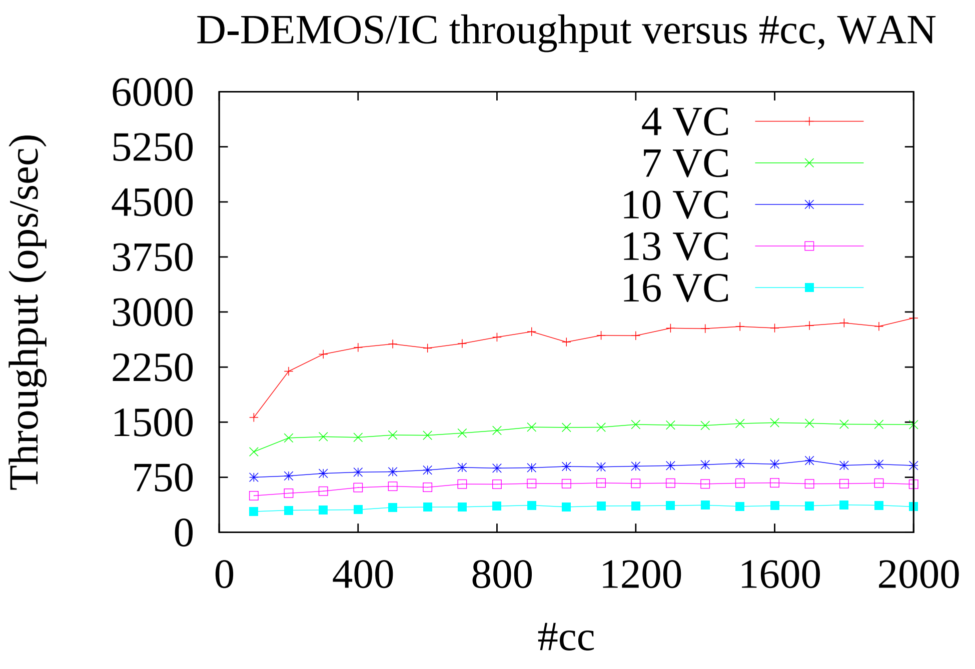


(a)

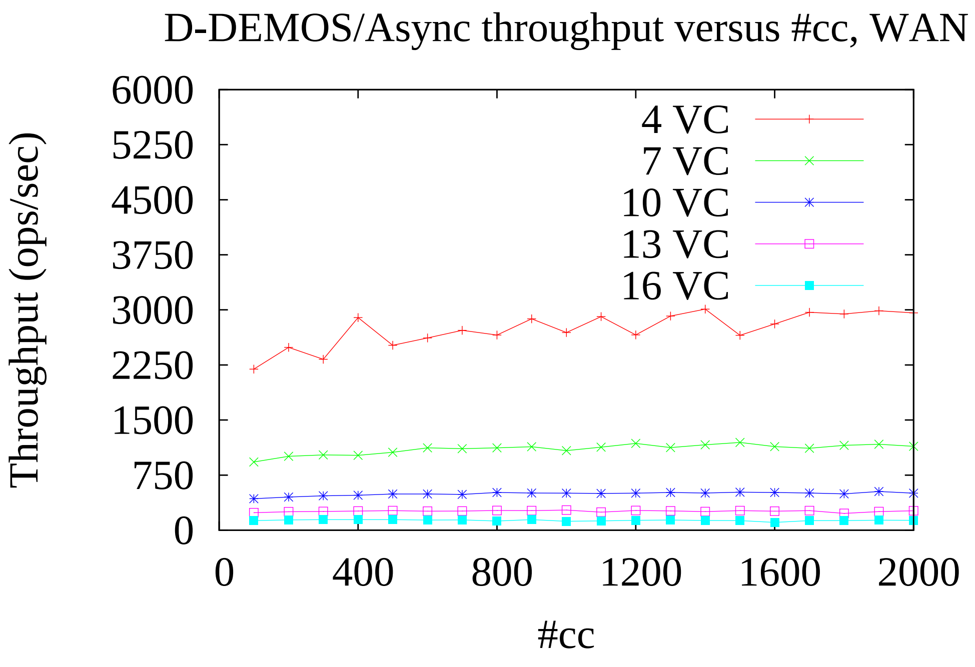


(b)

Figure 6.7: Vote Collection throughput of D-DEMOS/IC (6.7a) and D-DEMOS/Async (6.7b), versus the number of *VC* nodes, under a WAN setting. Election parameters are n = 200,000 and m = 4.

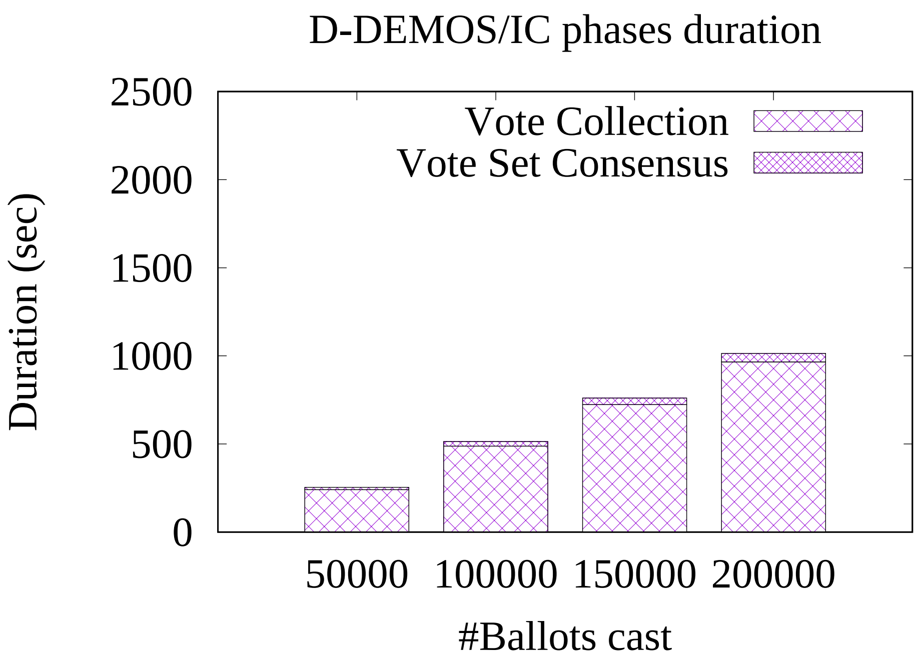


(a)

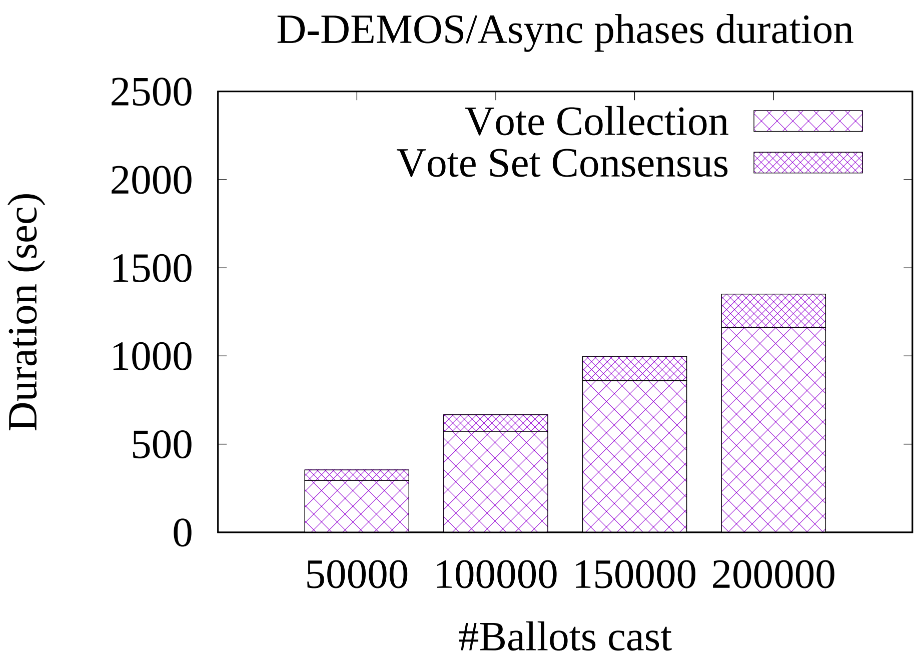


(b)

Figure 6.8: Vote Collection throughput of D-DEMOS/IC (6.8a) and D-DEMOS/Async (6.8b), versus the number of concurrent clients, under a WAN setting. Plots illustrate performance for different cardinalities of *VC* nodes, thus different fault tolerance settings. Election parameters are n = 200,000 and m = 4.



(a)



(b)

Figure 6.9: This figure illustrates the duration of the voting and vote set consensus, for a range of number of ballots. Results depicted are for 4 VCs, n = 200,000 and m = 4. All phases are disk based.

the total number of ballots cast. We have the voting driver use all ballots to vote, then start vote set consensus immediately, and we record the time taken to complete each phase. Comparing the two versions of D-DEMOS, we observe D-DEMOS/IC is faster during both Vote Collection and Vote Set Consensus phases. This is expected, because of the extra communication round of D-DEMOS/Async during voting, as well as the more complex consensus-per-ballot approach to achieving Vote Set Consensus. In fact, D-DEMOS/IC completes voting 15% faster than D-DEMOS/Async, while it completes vote set consensus 75% faster. However, note that D-DEMOS/Async is more robust than D-DEMOS/IC, as it does not require any kind of synchronization between nodes.

Overall, although we introduced Byzantine Fault Tolerance during the vote collection phase of a state-of-the-art voting system, we demonstrate it achieves high performance, enough to run real-life elections of large electorate bodies.

7. RELATED WORK

7.1 Voting systems

Several end-to-end verifiable e-voting systems have been introduced, e.g. the kiosk-based systems [31, 58, 29, 13, 92] and the internet voting systems [4, 76, 115, 72]. In all these works, the Bulletin Board (*BB*) is a single point of failure and has to be trusted.

Dini presents a distributed e-voting system, which however is not end-to-end verifiable [47]. In [45], there is a distributed *BB* implementation, also handling vote collection, according to the design of the vVote end-to-end verifiable e-voting system [44], which in turn is an adaptation of the Prêt à Voter e-voting system [31]. In [45], the proper operation of the *BB* during ballot casting requires a trusted device for signature verification. In contrast, our vote collection subsystem is done so that correct execution of ballot casting can be “human verifiable”, i.e., by simply checking the validity of the obtained receipt. Additionally, our vote collection subsystem in D-DEMOS/Async is fully asynchronous, always deciding with exactly $n - f$ inputs, while in [45], the system uses a synchronous approach based on the FloodSet algorithm from [86] to agree on a single version of the state.

DEMOS [72] is an end-to-end verifiable e-voting system, which introduces the novel idea of extracting the challenge of the zero-knowledge proof protocols from the voters’ random choices; we leverage this idea in our system too. However, DEMOS uses a centralized Election Authority (EA), which maintains all secrets throughout the entire election procedure, collects votes, produces the result and commits to verification data in the *BB*. Hence, the EA is a single point of failure, and because it knows the voters’ votes, it is also a critical privacy vulnerability. In this thesis, we address these issues by introducing distributed components for vote collection and result tabulation, and we do not assume any trusted component during election. Additionally, DEMOS does not provide any recorded-as-cast feedback to the voter, whereas our system includes such a mechanism.

Furthermore, none of the above works provide any performance evaluation results. Finally, [8] outlines the difficulties in managing seals for kiosks and ballot boxes, supporting our position towards the use of internet voting.

7.2 Consensus, Agreement, Interactive Consistency

Consensus. The Byzantine consensus problem is one of the most studied topics in distributed systems and the main topic of the well-known FLP impossibility result ([57]). There are several types of consensus protocols. The first distinction revolves around determinism (or non-determinism). In a deterministic consensus protocol, given the set of input values on all nodes, the message schedule and the failures that occur (if any), the result will always be the same. Deterministic consensus protocols require a synchronous system ([51]). In a purely asynchronous system, consensus can be achieved by randomization. FLP is circumvented by having nodes locally toss a coin to decide on their input values, in round $r+1$, in cases where consensus cannot be achieved in round r. Thus, the result may be different across executions with the same inputs. Examples of randomized protocols that employ the local coin construct are introduced by Bracha [15], Bracha and Toueg [16] and Ben-Or [11]. These algorithms guarantee eventual termination after a probabilistic number of rounds. In [109], a trusted, non-faulty dealer is additionally employed to bound the number of rounds required to achieve consensus. In our work, we leverage the randomized approach by Bracha to ensure termination because we believe it is controversial to assume a trusted entity in an otherwise Byzantine environment.

Other works ([23], [98]) leverage verifiable secret sharing techniques to implement a shared, or, common coin. These consensus algorithms are polynomially efficient and terminate in a constant number of rounds. Canetti et al. [23] present one of most well-established and signature-free common coin protocols. However, this protocol, although polynomial, is complex to implement and has very high bit complexity [93]. Mostéfaoui et al. [93] employ the common coin protocol that is presented in [21] which has guaranteed termination but requires a trusted dealer. We did not consider these algorithms as they are either inefficient or require a trusted dealer.

Failure Detectors. In [27], Chandra and Toueg proposed a solution for the consensus problem, in an asynchronous crash-fault environment, introducing a module called *unreliable failure detector* (FD). It is shown that even if the FD erroneously considers some processes faulty, it can be used to solve consensus.

There is extensive literature that further expands the family of FDs to a number of applications ([52, 64, 67, 84, 9, 73]). The muteness failure detector presented in [52] is a failure detector can detect nodes that, after a time, either crash, or behave arbitrarily. In [64], FDs are used to solve the transaction commit problem in distributed databases. Mostefaoui et al. [67], use perfect failure detectors to compute global data, used in the distributed termination detection problem. Larrea et al. [84] introduces the notion of Eventually Consistent FD and utilizes it to solve the consensus problem in a crash-fault tolerant setting. Arevalo er al. [9], also solve the problem of consensus in a crash fault environment but they consider a special case of a “homonymous” network, where processes share the same identifiers. Finally, Kihlstrom et al. [73], extend the work of Chandra and Toueg and propose a failure detector that can effectively detect Byzantine behaviour. However, FDs detecting Byzantine behaviour are no longer autonomous, as they require input from the higher level algorithm [19].

FDs are an implicit part of leader-based consensus algorithms, hidden behind the leader election primitives. Such algorithms are Lamport’s Paxos in the crash-fault tolerant setting [80, 82] or the Byzantine-fault tolerant setting [88, 81], and PBFT [25] which is based on view-stamped replication [96]. The similarities between Paxos and PBFT are analyzed in [83, 18]. Further performance enhancements to leader-based consensus protocols using FDs are described in [113].

Although these all provide solutions to consensus, they also solve *Atomic Broadcast*, i.e., enforcing total ordering across multiple events (consensus instances). Additionally, in their

Byzantine fault tolerant incarnations, they explicitly require digital signatures. In this work, we utilize the approach of Bracha [15] for solving consensus, because it is more focused on the specific problem, as it does not deal with ordering multiple consensus instances, and does not require digital signatures for its operation.

Finally, Cason et al. [24] study total order broadcast in a type-hybrid model, but only for LANs and only for crash faults; we target WAN deployments and Byzantine fault tolerance.

Broadcast protocols. In [15], Bracha introduced a $\frac{n}{3}$-resilient reliable broadcast primitive (RBB, for Reliable Broadcast of Bracha) to solve the consensus problem. Another type of broadcast primitive, with lower message complexity, is *consistent broadcast* (CB). CB is designed to relax the *agreement* property of reliable broadcast, by allowing *some* non-faulty nodes to deliver m, while others may deliver nothing. The standard implementation of consistent broadcast is *Reiter's echo multicast* [104].

Interactive Consistency. Interactive consistency was first introduced and studied by Pease et al. [100], and has been the topic of several research papers ([111, 107, 85, 60, 78, 12]), focusing on synchronous systems. While these approaches might be feasible in environments such as shared memory multi-processors or digital flight control systems, we believe they are ill-suited for practical, real-world distributed systems. In [90] and [102], the authors provide solutions to various forms of consensus, despite their title references to IC.

A closely related problem to IC is vector consensus [52, 12, 95, 22] (in [22] it is called "agreement on a core set"). Vector consensus differs from IC only in terms of the *Validity* condition, as the former delivers a vector with at least $2t + 1$ values, where at least $t + 1$ values were proposed by non-faulty nodes. That is, in vector consensus, a valid output vector may freely miss some values from correct nodes. The reason for this difference is that in asynchronous systems, it is impossible to simultaneously ensure *input completeness* and *liveness* [71]. The focus of this work on IC is achieving input completeness in the fault-free case, while also ensuring liveness when faults occur.

7.3 State Machine Replication

Achieving consensus in the presence of arbitrary faults is a well studied field in distributed systems. It was first introduced by Pease et al. in a synchronous setting [100], while the term "Byzantine" was introduced by Lamport et al. in [79]. Consensus is also the basis, used by Fisher et al. [57], to express the FLP impossibility result in an asynchronous setting, which states the impossibility of achieving *deterministic consensus* even with one faulty process. There are several proposed solutions to circumvent the FLP impossibility result in asynchronous systems. Many of these approaches utilize randomization techniques that ensure termination after a probabilistic number of rounds ([11, 15, 16, 41]), while others assume trusted modules that can effectively detect the misbehaviour of the systems' components ([5, 27, 52, 64, 67]).

Castro et al. [25, 26] introduce a practical Byzantine Fault Tolerant Replicated State Machine (RSM) protocol, which could potentially be used for the implementation of the Vote Collection subsystem. Since this seminal work, there has been a flurry of research activity focused on improving the BFT middleware performance [114, 75, 3, 42, 59, 74, 110, 37, 48, 112], replication cost [114, 49, 112, 10], and robustness under both faulty servers and faulty clients [6, 37]. A large majority of these systems [114, 75, 74, 6, 37, 59, 112] are direct descendants of the Castro and Liskov PBFT system and reuse and build upon the Castro codebase.

Separating agreement from execution [114] introduced the concept of a separate agreement and execution cluster. It also introduced the privacy firewall, for avoiding leakage of sensitive information from a faulty execution node. The latter, though, requires $(h+1)^2$ nodes for tolerating h faults in the privacy firewall cluster.

Several attempts have been made to address the inability of replicated BFT services to mesh with the rest of the infrastructure in today's multi-tier world. Merideth et al. [89] introduced *Thema*, which aims to mask BFT complexity from the application developer of web services based applications. An agent, visible to the unaffected outside world, plays the role of the client of a BFT system. Additionally, a proxy collects the multiple out-call requests from the replicas of a BFT system, and issues the actual out-call on their behalf, returning the reply when available. Unfortunately, both the agent and the proxy are centralized components which are inappropriate for applications such as ours which require completely distributed design.

Pallemulle et al. [97] focus on interoperability between BFT systems, while enforcing fault isolation and introduce a new protocol, named *Perpetual* to achieve this. Sen et al. [106] in a system called Prophecy, designed to increase BFT performance, introduce a *Sketcher* component, that tries to trade space for performance, by storing a historical log of request/reply pairs and allowing the application to differentiate its requests, asking for possible log-based replies. In its distributed incarnation, *D-Prophecy* is simply an attempt to avoid re-execution of duplicate requests. In the centralized version, *Prophecy*, the *Sketcher* completely avoids BFT access but now becomes a single point of failure.

Amir et al. [7] introduce *Steward*, a hierarchical BFT architecture, that tries to scale BFT to a wide-area network, by introducing an abstraction layer above PBFT using a Paxos-based protocol. It uses a threshold signature scheme to ensure the recipient of a cross-domain message that enough replicas at the originating site agreed with the request. Both these features are welcome to security-conscious Internet application services. Unfortunately, no source code is available.

Finally, Guerraoui et al. [63] introduce a new abstraction allowing for the construction of new BFT protocols with a fraction of the code currently necessary, thus vastly simplifying the BFT researcher's task. Having waded through the 20,000 lines of PBFT code while investigating its use for this project, we applaud this effort and emphasize here the need to simplify the end application developer's task as well.

Our system does not use the state machine replication approach to handle vote collection, as it would be inevitably more costly. Each of our vote collection nodes can validate a voter's requests on its own. In addition, we are able to process multiple different voters' requests concurrently, without enforcing the total ordering inherent in replicated state machines. Finally, we do not wish voters to use special client-side software to access our system.

8. CONCLUSIONS AND FUTURE WORK

8.1 Conclusion and future work

E-voting systems are a powerful tool for improving society; they allow more frequent elections, thus providing for direct democracy. While nation-scale kiosk-based e-voting systems speed up production of the election result considerably, they pose many undesired necessities. They require setting up booths throughout a country, which means allocating physical space and incurring setup costs. They also require voters to physically come to the booths, instead of voting from the comfort of their home or office. These facts make them impractical for frequent elections. Thus, in our work we have chosen to focus on internet (remote) voting systems, and this decision has been the key driver determining our requirements, each of which we outline below.

We recognize that malware has spread all over the world, making the public unable to trust their personal computers for such a critical function as selecting a nation's next government. For this reason, we set the goal of producing a vote collection mechanism that does not require voters to perform advanced functions, such as storing private keys and performing cryptographic calculations, on their personal devices. Additionally, we wish to accommodate voters who do not own computers. Our systems allow such voters to vote using a public terminal or a friend's device without compromising their privacy.

Robustness, in the sense of tolerating faulty components, has not received much thought from the voting community. The only related work we found explicitly focusing on fault tolerance, produced a robust Web Bulletin Board ([45]) that was used to produce a kiosk-based voting system ([44]) and not an internet voting one. Most designs from scientific papers assume centralized components that pose a risk to both availability and privacy ([28, 31, 58, 29, 13, 43, 4, 36, 76, 61, 115, 72]). Moreover, most real-world implementations, even kiosk-based ones, use centralized components, making them vulnerable to single points of failure and/or attack.

The importance of end-to-end verifiability cannot be overstated. As a testimony to this statement, the results of the US presidential elections of 2016 were recently challenged in three states [1, 2]. In particular, a number of academics and activists are calling for US authorities to fully audit or recount the 2016 presidential election vote in the states of Michigan, Pennsylvania and Wisconsin, in case "the results could have been skewed by foreign hackers". In urging for a recount, experts, in essence want to ”verify the votes” of citizens with ”post-election ballot audits”. This open doubt about the correctness of the election tally results in the public losing trust in the election system as a whole. All this could have been avoided if end-to-end verifiable voting systems had been deployed. The experts could perform their own audits themselves, without requiring cooperation from the election authorities. If during an audit, the experts found something wrong, they could *prove* that the election result was not correct, without unnecessary and prolonged public discussion or court sessions.

We chose the DEMOS voting system as a model for our study because it allowed us to achieve our objectives. Being vote code based, it allowed us to offload cryptography to the setup component (the Election Authority), allowing voters to use the output of this setup in a simple manner. We only require basic string-matching skills to verify a receipt, to obtain recorded-as-cast assurance. Moreover, both our vote collection systems do not require the use of a public key infrastructure for voters.

We also designed our voter's algorithm with a human in mind, and not a machine. This is why the voter is not required to perform multicasts. We only ask the voter to wait for a while, for a receipt to be sent back, and verify it against the expected one. For example, the voter can simply access one *VC* node's URL, cast her vote by filling an HTML form, wait for her browser to time out, and in such a case (or in the case of a faulty reply), she simply accesses another *VC* node's URL and casts her vote again.

This thesis work was part of a larger project called FINER from the GRNET ARISTEIA program. The goal of the FINER project was to produce a *complete* and *fault-tolerant* internet voting system that is at the same time *privacy-preserving* and *end-to-end verifiable*. Both of our vote collection systems have been integrated into the voting system D-DEMOS, which is the outcome of project FINER and fulfills all of the above requirements.

Our first approach to vote collection, which resulted in D-DEMOS/IC is simpler and faster. However, it makes the strong assumption that at a very specific period of time, right after voting is over, the system is able to deliver all messages from non-faulty nodes. This was the result of our choice to select an algorithm achieving Interactive Consistency, which is impossible in asynchronous systems, to obtain Vote Set Consensus. We then focused on exploring the fundamental reasons behind the inability to achieve consensus in D-DEMOS/IC. We identified the attack from a malicious voter casting multiple vote codes across different *VC* nodes as the major obstacle and decided to prohibit it during voting instead of during vote set consensus. This new approach resulted in D-DEMOS/Async, which is the first internet voting system that is, at the same time, Byzantine fault-tolerant, asynchronous, privacy-preserving and end-to-end verifiable, without requiring the voters to trust the devices they use to vote.

We made performance comparisons between the two vote collection systems. During voting, D-DEMOS/IC is faster but only by a small margin of 15%. We thus consider both systems equally able to handle large-scale elections. However, during vote set consensus, D-DEMOS/IC is four times faster than D-DEMOS/Async. Considering this is a phase that runs only once at election end-time, and is, in any case, far faster than manually counting votes from a ballot box, we claim D-DEMOS/Async is able to handle this phase too for a real-world election.

While D-DEMOS/Async has advanced the state-of-the-art significantly, a number of challenges remain before internet voting systems become ubiquitous in election procedures.

First, we have focused our efforts on supporting *1-out-of-m* elections, in which voters choose only one out of m options from their ballots. Real world elections often require more complex schemes. The next logical step is to handle *k-out-of-m* elections, where the voter can select more than one options. Then, there is the more complex variation which adds a weight to each voter choice; e.g., the voter's first choice is more important than the second and that needs to be recorded and tallied accordingly. Finally, there are hierarchical schemes as well, where the voters first select a party to vote for, and then select the candidates of their choice from the selected party only. We leave as future work the handling of these more complex voting scenarios.

Second, we have used a setup component (the *EA*) to prepare the election and initialize all other components. We have assumed secure out-of-bands communication between the *EA* and all other components. Although this may be feasible using a manual process for initializing the *VC* nodes, it is troublesome when we consider delivering the ballots to the voters in a secure and anonymous way. At present, the most secure way to distribute the ballots to voters is to have the ballots printed and have each voter pick one at random. All other mechanisms require trusting additional system components (the email server, the authentication server, etc). Thus, *secure ballot distribution* is a problem that needs immediate attention, to allow for really painless frequent elections.

Third, we have relied on the end-to-end verifiability that we inherited from DEMOS and carefully preserved, to detect a malicious setup of the election. This is feasible as long as the fault-tolerance thresholds, for all system components are not violated. But what if they are? What if more that f_v *VC* nodes fall under the control of the adversary? More importantly, what if other distributed D-DEMOS components, such as the Bulletin Board nodes or *trustees* have their fault-tolerance thresholds exceeded? In this case, we cannot *attribute* an error to a specific system component. Although this may not seem important while modeling the system and considering a single adversary coordinating all malicious components, in the real world, it is important to be able to detect the faulty component that causes trouble. Thus, *fault attribution* is an interesting research area for voting systems.

Finally, *coercion-resistance* is a very important topic for voting systems, and one that is difficult to define precisely for an internet voting system. In our approach, the voter needs to preserve the privacy of her ballot and find a moment alone to cast her vote, but that is not enough to achieve coercion-resistance. We have considered the use of fake ballots. For example, the voter can obtain one real and one fake ballot from the system, use the real one to vote while in private, and present her vote via the fake one to the adversary. The system, during tallying, must count all fake ballots as 0 votes, effectively discarding them. This does not change vote collection at all, but does affect the end-to-end verifiability of the voting system as a whole. What if the Election Authority produces and distributes fake ballots to a targeted subset of the electorate body?

Concluding, the vote collection systems presented in this thesis are applicable to any voting system that uses the code-voting technique. Thus, we believe our work is a required step towards producing higher quality voting systems that can handle large-scale elections efficiently and reliably.

ABBREVIATIONS - ACRONYMS

ACS	Asynchronous Communications Stack
AES	Advanced Encryption Standard
BB	Bulletin Board
BFT	Byzantine Fault Tolerant
CBC	Cipher Block Chaining
EA	Election Authority
FD	Failure Detector
HTTP	HyperText Transfer Protocol
IC	Interactive Consistency
LAN	Local Area Network
RSM	Replicated State Machine
SMR	State-Machine Replication
TLS	Transport Layer Security
UCERT	Uniqueness Certificate
VC	Vote Collection
VSS	Verifiable Secret Sharing
WAN	Wide Area Network
ZKP	Zero-Knowledge Proof

appendix

APPENDIX A. ACHIEVING INTERACTIVE CONSISTENCY IN MOSTLY-ASYNCHRONOUS SYSTEMS

In this Appendix, we present the details of the Interactive Consistency (IC) algorithm we used for D-DEMOS/IC. We first present the system model we consider in section A.1, which is a *mostly-asynchronous* one because of the impossibility of achieving Interactive Consistency in asynchronous systems.

After that, in Section A.2 we present our work on developing a series of IC algorithms. This section provides the rationale behind our approach and system design.

A.1 System Model

We assume a distributed system consisting of n nodes that are fully connected over a network. The network is mostly asynchronous, i.e., it exhibits one (or more, depending on the algorithm) period of synchrony, during which message delivery is timely. The network can drop, delay, duplicate, or deliver messages out of order. However, we assume that messages are eventually delivered, provided that the corresponding senders keep on retransmitting them. We assume authenticated channels, where the receiver of a message can always identify its sender. Each node has a public/private key pair and all nodes know the others' public keys. We use these keys to implement authenticated channels, and sign messages where needed.

We assume a Byzantine failure model where nodes may deviate arbitrarily from the protocol. We allow for a strong adversary that can coordinate faulty nodes. However, we assume he cannot delay the delivery of messages, or processing on correct nodes beyond the system's synchrony assumptions. The adversary is also assumed to be computationally bounded, meaning he cannot subvert common cryptographic techniques such as signatures and message authentication codes (MACs).

A.2 Practical Interactive Consistency

A.2.1 Adapting approaches from synchronous systems

The original algorithm of Pease et al. [100] requires a total of $t+1$ rounds to achieve IC in a synchronous system, tolerating up to t faults, with a total message complexity of $(t+1)n^2$. Our first approach is to adapt the same algorithm by simulating synchronous rounds with timeouts. Messages delivered after the time frame of each round, will be disregarded and counted towards the t system faults, according to the model presented in [50].

Two issues arise from the use of timeouts, as highlighted in [65]. The first one is efficiency. Assuming a timeout value of T_r for each round, the system will always require a constant amount of time, i.e., $(t+1)T_r$, to execute a request even in the presence of a single failure. The second is choosing a correct value for T_r. If we choose a conservative approach and set a large value for T_r, we could increase the execution time of the algorithm dramatically, thus, making it less practical. On the contrary, a small value might cause some slow nodes, who are otherwise correct, to be considered faulty. If this occurs multiple times, as is the case when one relies on multiple timeouts, it is possible that we will exceed the upper bound t of total failures in the system.

To avoid the issues associated with multiple timeouts, one might attempt to reduce IC to Byzantine Agreement (BA), by running n parallel instances of BA, as it was suggested for synchronous systems ([56]). In each instance, a node n_i would spread its private value v_i to the rest of the system. In a synchronous setting, this would result in all non-faulty nodes having the same vector of values. However, in a completely asynchronous environment, BA is impossible ([15]), as a crashed node may never even start its instance of BA, and nodes cannot distinguish between crashed nodes and slow nodes. Therefore, the non-faulty nodes need to decide, at a certain point, to exclude the suspected crashed nodes from IC and store a default (e.g., *null*) value at the slot corresponding to each crashed node. Thus, they need a synchronization point, where they decide on the result vector; we call this point a *barrier*. This synchrony assumption allows for the circumvention of the impossibility of simultaneously achieving input completeness and guaranteed termination in an asynchronous system ([71]).

The introduction of the barrier introduces a new challenge as, at that point, a BA instance may have *delivered* the result in some nodes but not yet in others. This, for example, may be triggered by an adversary starting his own BA instance near the barrier. Thus, honest nodes will need to achieve consensus, for each individual slot of the result vector, on the value to be placed in that slot. We observe that the barrier splits the procedure in two phases. We call the first phase the *value dissemination phase*, where we assume the network delivers all messages of non-faulty nodes by the end of the phase. Recall that, as we stated in the first paragraph of this section, messages delivered after the time frame of the first phase will be counted towards the t system faults. We call the second phase, the *result consensus phase*, which can be completely asynchronous. Note that we have employed the costly BA approach for the first phase, but have shown that a consensus phase is still required.

A.2.2 Solution using Multi-Valued Consensus

With these observations, we seek less costly alternatives for the first phase, i.e., avoiding BA. Our first approach is to use a simple point-to-point message exchange, where each node announces its own private value to the rest of the system. As this exchange is unrestricted, it may result in each honest node receiving a different value from a malicious node. Thus, during the result consensus phase, nodes need to agree on the value to be placed in each slot of the result vector. We employ the multi-valued consensus (MC) algorithm from [41]; recall that this algorithm utilizes a binary consensus and a reliable broadcast primitive. We want to refrain from making any further synchrony assumptions, thus, mak-

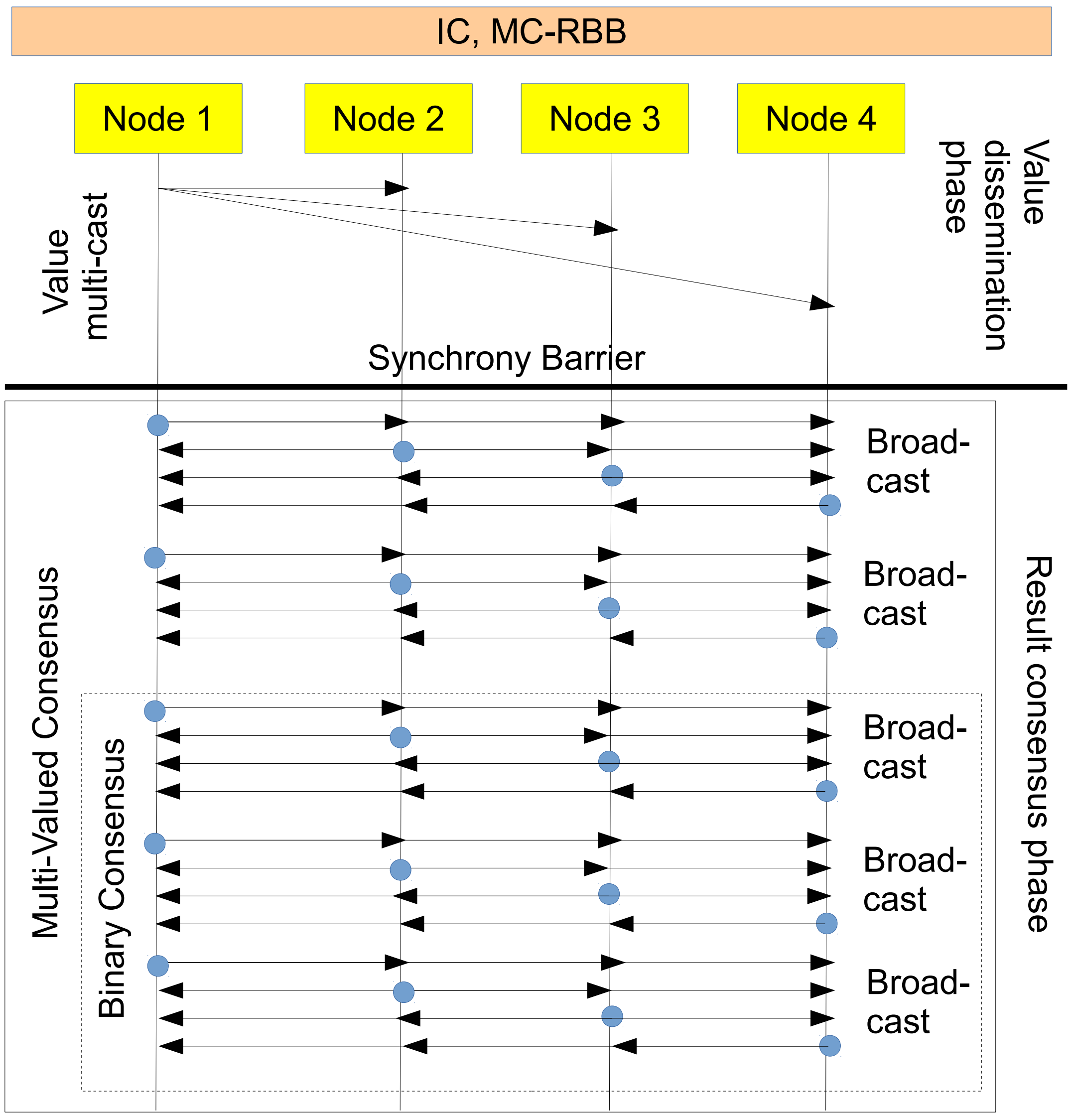


Figure A.1: Diagram of message exchange for *IC,MC-RBB*, for a single value of the result vector (repeated n times to achieve IC).

ing the result consensus phase completely asynchronous. In order to circumvent FLP, which states that achieving deterministic consensus is impossible in purely asynchronous systems, we employ a randomized consensus protocol. We use Bracha's binary consensus (BC) and reliable broadcast (RBB) primitives from [14], and we run n parallel instances of MC, one for each value of the vector.

This algorithm (*IC,MC-RBB*) achieves IC because, regardless of the unrestricted value dissemination phase, each instance of MC ensures that nodes agree on a single value for each slot of the result vector respectively. *(IC,MC-RBB)* uses only one synchrony barrier, as opposed to the adaptation of Pease's algorithm which needs $t+1$. Its overall message complexity is $10n^4 + 5n^3 + n^2$. Figure A.1 demonstrates the message exchanges for this protocol.

A.2.3 Solution using Binary Consensus

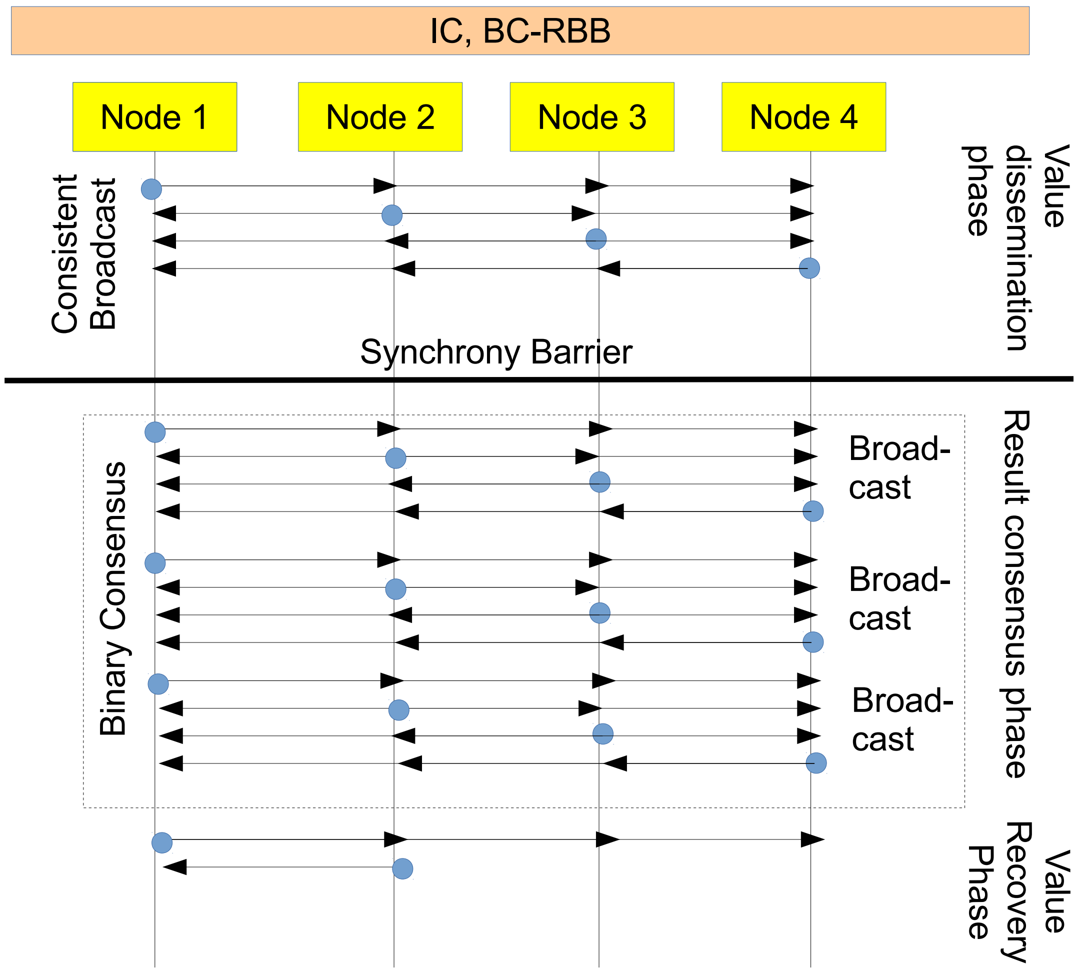


Figure A.2: Diagram of message exchanges for *(IC,BC-RBB)*, for a single value of the result vector (repeated n times to achieve IC).

Our next approach reduces the aforementioned message complexity. We observe multi-valued consensus uses one binary consensus and two reliable broadcast instances. We avoid the use of MC by changing the subject on which consensus is required. In the previous algorithm, the consensus question is “what is the actual value to be placed in the corresponding slot of the result vector?”, because the first (value dissemination) phase is insecure. We make the first phase secure by using Consistent Broadcast (CB, [20]). Here, the source n_i first sends its value v_i to each node; then it collects signed endorsement responses. A recipient node endorses only the first value for each broadcast. Once $n - t$ such responses are accumulated, the sender forms a uniqueness certificate c_i that includes these endorsements, and sends <n_i, c_i> to the rest of the nodes. CB *delivers* v_i *iff* c_i has at least $n - t$ valid signatures. Assuming signatures are unforgeable, it is impossible for a malicious node to construct two valid certificates for two different values. Thus, this protocol bounds the sender to either send a single value, or not send a value at all. As this value is guaranteed to be unique, we change the question of the result consensus phase to “is there a value to be placed in the corresponding slot of the result vector?”. This question can now be answered by a binary consensus protocol, and we utilize Bracha’s protocol ([14]) in our approach. Figure A.2 depicts this protocol’s message exchanges.

An outcome of 0 from BC causes each node to place the *null* value in the corresponding slot of the result vector. Accordingly, a result of 1 from BC instructs each node to place the (unique) value v_i in the result vector. There are cases, however, where a consensus instance may produce a result different than the opinion with which an honest node entered BC. This can happen when the corresponding instance of CB *delivered* the value v_i at some nodes, but not at others (e.g., when a malicious CB source sends the value, along with the uniqueness certificate, only to some nodes). Thus, a node may possess a value for this slot, and the result of consensus may be 0, in which case it simply replaces the value with *null*. However, the contrary may also happen, where a node did not possess a value when it entered BC, but consensus resulted in 1. For this case, we add a final *recovery* phase, where a node asks all other nodes for the correct value of the i^{th} position of the result vector. Any node that receives such a message replies with the <v_i, c_i> tuple it possesses. At least one honest node is guaranteed to exist and submit such a reply, as, by definition of BC, if all honest nodes entered consensus with 0, the result would have been 0. As the result is 1, at least one honest node exists which has entered consensus with 1, thus possessing the correct value and uniqueness certificate for it.

To summarize, this IC algorithm *(IC,BC-RBB)* achieves IC because: a) during the value dissemination phase, an honest node either obtains a value guaranteed to be unique, or no value at all, b) during the result consensus phase, all nodes agree, for each slot of the result vector, whether to place a (guaranteed unique) value, or the *null* value, and c) during the recovery phase, any honest node is guaranteed to obtain missing values. The overall complexity of *(IC,BC-RBB)* is $6n^4 + 3n^3 + 3n^2$ messages and $n^3 + 2n^2$ signature operations.

Please note that we have only provided an overview of this subject here. For more information, including performance evaluation and message complexity derivations, we refer the interested reader to [46].

BIBLIOGRAPHY

[1] Hillary clinton urged to call for election vote recount in battleground states. https://www.theguardian.com/us-news/2016/nov/23/hillary-clinton-election-vote-recount-michigan-pennsylvania-wisconsin, November 2016.

[2] Hillary clinton's team to join wisconsin recount pushed by jill stein. http://www.nytimes.com/2016/11/26/us/politics/clinton-camp-will-join-push-for-wisconsin-ballot-recount.html?_r=0, November 2016.

[3] M. Abd-El-Malek, G.R. Ganger, G.R. Goodson, M.K. Reiter, and J.J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles, Brighton, UK*, pages 59–74. ACM, October 2005.

[4] Ben Adida. Helios: Web-based open-audit voting. In *Proceedings of the 17th USENIX Security Symposium, San Jose, CA, USA*, pages 335–348. USENIX Association, July 2008.

[5] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, 1999.

[6] Y. Amir, B.A. Coan, J. Kirsch, and J. Lane. Byzantine replication under attack. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2008), Anchorage, Alaska, USA*, pages 197–206. IEEE Computer Society, Jun 2008.

[7] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-rotaru, J. Olsen, and D. Zage. Scaling byzantine fault-tolerant replication to wide area networks. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2006), Philadelphia, Pennsylvania, USA*, pages 105–114. IEEE Computer Society, June 2006.

[8] Andrew W. Appel. Security seals on voting machines: A case study. *ACM Transactions on Information and System Security*, 14(2):18:1–18:29, September 2011.

[9] Sergio Arévalo, Antonio Fernández Anta, Damien Imbs, Ernesto Jiménez, and Michel Raynal. Failure detectors in homonymous distributed systems (with an application to consensus). *Journal of Parallel and Distributed Computing*, 83:83–95, 2015.

[10] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. *ACM Transactions on Computer Systems (TOCS)*, 32(4):12, 2015.

[11] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada*, PODC 1983, pages 27–30. ACM.

[12] Michael Ben-Or and Ran El-Yaniv. Resilient-optimal interactive consistency in constant time. *Distributed Computing*, 16(4):249–262, 2003.

[13] Josh Benaloh, Michael D. Byrne, Bryce Eakin, Philip T. Kortum, Neal McBurnett, Olivier Pereira, Philip B. Stark, Dan S. Wallach, Gail Fisher, Julian Montoya, Michelle Parker, and Michael Winn. STAR-vote: A secure, transparent, auditable, and reliable voting system. In *Proceedings of the Electronic Voting Technology Workshop / Workshop on Trustworthy Elections, EVT/WOTE '13, Washington, D.C., USA*. USENIX Association, August 2013.

[14] Gabriel Bracha. An asynchronous $[(n-1)/3]$-resilient consensus protocol. In *PODC*, 1984.

[15] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, November 1987.

[16] Gabriel Bracha and Sam Toueg. Resilient consensus protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC 1983, pages 12–26, New York, NY, USA, August 1983. ACM.

[17] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[18] Christian Cachin. Yet another visit to paxos. *IBM Research, Zurich, Switzerland, Tech. Rep. RZ3754*, 2009.

[19] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[20] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology, Crypto 2001*. Springer, 2001.

[21] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing, Portland, Oregon*, pages 123–132. ACM, July 2000.

[22] Ran Canetti. *Studies in Secure Multiparty Computation and Applications*. PhD thesis, 1996.

[23] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, San Diego, CA, USA*. ACM, May 1993.

[24] Daniel Cason and Luiz E. Buzato. Time hybrid total order broadcast: Exploiting the inherent synchrony of broadcast networks. *Journal of Parallel and Distributed Computing*, 77:26 – 40, 2015.

[25] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA*, pages 173–186. USENIX Association, February 1999.

[26] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[27] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[28] David Chaum. Surevote: Technical overview. In *Proceedings of the Workshop on Trustworthy Elections*, WOTE, Aug. 2001.

[29] David Chaum, Aleks Essex, Richard Carback, Jeremy Clark, Stefan Popoveniuc, Alan Sherman, and Poorvi Vora. Scantegrity: End-to-end voter-verifiable optical-scan voting. *Security & Privacy, IEEE*, 6(3):40–46, 2008.

[30] David Chaum and Torben P. Pedersen. Wallet databases with observers. In *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology - CRYPTO '92, Santa Barbara, California, USA*, pages 89–105. Springer-Verlag, August 1993.

[31] David Chaum, Peter Y. A. Ryan, and Steve A. Schneider. A practical voter-verifiable election scheme. In *Proceedings of the 10th European Symposium on Research in Computer Security - ESORICS 2005, Milan, Italy*, pages 118–139. Springer, September 2005.

[32] Nikos Chondros, Alex Delis, Dina Gavatha, Aggelos Kiayias, Charalampos Koutalakis, Ilias Nicolacopoulos, Lampros Paschos, Mema Roussopoulos, Giorge Sotirelis, Panos Stathopoulos, Pavlos Vasilopoulos, Thomas Zacharias, Bingsheng Zhang, and Fotis Zygoulis. Electronic voting systems - from theory to implementation. In *E-Democracy, Security, Privacy and Trust in a Digital World*, pages 113–122, Dec. 2013.

[33] Nikos Chondros, Konstantinos Kokordelis, and Mema Roussopoulos. On the practicality of practical byzantine fault tolerance. In *Proceedings of the ACM/IFIP/USENIX 13th International Middleware Conference (Middleware 2012), Montreal, QC, Canada*, pages 436–455. Springer, December 2012.

[34] Nikos Chondros, Bingsheng Zhang, Thomas Zacharias, Panos Diamantopoulos, Stathis Maneas, Christos Patsonakis, Alex Delis, Aggelos Kiayias, and Mema Roussopoulos. D-demos: A distributed, end-to-end verifiable, internet voting system. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, Jun 2016.

[35] Nikos Chondros, Bingsheng Zhang, Thomas Zacharias, Panos Diamantopoulos, Stathis Maneas, Christos Patsonakis, Alex Delis, Aggelos Kiayias, and Mema Roussopoulos. Distributed, end-to-end verifiable, and privacy-preserving internet voting systems. *CoRR*, abs/1608.00849, 2016.

[36] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 2008), Oakland, California, USA*, pages 354–368. IEEE Computer Society, May 2008.

[37] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, Boston, MA, USA*, volume 9, pages 153–168. USENIX Association, April 2009.

[38] Netty community. Netty, an asynchronous event-driven network application framework. `http://netty.io/`, 2015.

[39] PostgreSQL community. Postgresql rdbms. `http://www.postgresql.org/`, 2015.

[40] Miguel Correia, Alysson Neves Bessani, and Paulo Veríssimo. On byzantine generals with alternative plans. *Journal of Parallel and Distributed Computing*, 68(9):1291–1296, 2008.

[41] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82–96, January 2006.

[42] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 177–190. USENIX Association, Nov 2006.

[43] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. In *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques, Advances in Cryptology - EUROCRYPT '97, Konstanz, Germany*, pages 103–118. Springer, May 1997.

[44] Chris Culnane, Peter Y. A. Ryan, Steve Schneider, and Vanessa Teague. vvote: A verifiable voting system. *ACM Transactions on Information and System Security*, 18(1):3:1–3:30, June 2015.

[45] Chris Culnane and Steve Schneider. A peered bulletin board for robust use in verifiable voting systems. In *Proceedings of the IEEE 27th Computer Security Foundations Symposium (CSF 2014), Vienna, Austria*, pages 169–183. IEEE, July 2014.

[46] P. Diamantopoulos, S. Maneas, C. Patsonakis, N. Chondros, and M. Roussopoulos. Interactive consistency in practical, mostly-asynchronous systems. In *Proceedings of the IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS 2015)*, pages 752–759. IEEE, Dec 2015.

[47] Gianluca Dini. A secure and available electronic voting service for a large-scale distributed system. *Future Generation Computer Systems*, 19(1):69–85, 2003.

[48] T. Distler and R. Kapitza. Increasing performance in byzantine fault-tolerant systems with on-demand replica consistency. In *Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, Salzburg, Austria*, pages 91–106. ACM, April 2011.

[49] T. Distler, R. Kapitza, I. Popov, H.P. Reiser, and W. Schroder-Preikschat. Spare: Replicas on hold. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA*. The Internet Society, February 2011.

[50] D. Dolev and R. Strong. *Distributed Commit with Bounded Waiting*. 1982.

[51] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)*, 34(1):77–97, January 1987.

[52] Assia Doudou and André Schiper. Muteness detectors for consensus with byzantine processes. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC ’98, Puerto Vallarta, Mexico*. ACM, June 1998.

[53] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of Advances in Cryptology (CRYPTO ’84), Santa Barbara, California, USA*, pages 10–18. Springer, August 1984.

[54] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[55] Uriel Feige, Amos Fiat, and Adi Shamir. Zero-knowledge proofs of identity. *Journal of Cryptology*, 1(2):77–94, 1988.

[56] Michael J Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *FCT*, 1983.

[57] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, April 1985.

[58] K. Fisher, R. Carback, and A. Sherman. Punchscan: introduction and system definition of a high-integrity election system. In *IAVoSS Workshop On Trustworthy Elections (WOTE 2006), Cambridge, United Kingdom*, June 2006.

[59] R. Garcia, R. Rodrigues, and N. Preguica. Efficient middleware for byzantine fault tolerant database replication. In *Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, Salzburg, Austria*, pages 107–122. ACM, April 2011.

[60] Adria Gascón and Ashish Tiwari. A synthesized algorithm for interactive consistency. In *Proceedings of the 6th NASA International Symposium on Formal Methods, NFM 2014, Houston, TX, USA*, pages 270–284. Springer, April 2014.

[61] Kristian Gjøsteen. The norwegian internet voting protocol. *IACR Cryptology ePrint Archive*, 2013:473, 2013.

[62] Ilya Grigorik. High performance browser networking: What every web developer should know about networking and web performance. `http://chimera.labs.oreilly.com/books/1230000000545/ch01.html#PROPAGATION_LATENCY`, 2013.

[63] R. Guerraoui, N. Knezevic, V. Quema, and M. Vukolic. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France*, pages 363–376. ACM, April 2010.

[64] Rachid Guerraoui and Petr Kouznetsov. On the weakest failure detector for non-blocking atomic commit. In *Foundations of Information Technology in the Era of Networking and Mobile Computing, IFIP 17th World Computer Congress - TC1 Stream / 2nd IFIP International Conference on Theoretical Computer Science (TCS 2002), August 25-30, 2002, Montréal, Québec, Canada*, pages 461–473. Springer, August 2002.

[65] Rachid Guerraoui and Andre Schiper. Consensus: the big misunderstanding. In *FTDCS*, 1997.

[66] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. *Technical Report*, 1994.

[67] J.-M. Hélary, M. Hurfin, A. Mostefaoui, M. Raynal, and F. Tronel. Computing global functions in asynchronous distributed systems with perfect failure detectors. *IEEE Transactions on Parallel and Distributed Systems*, 11(9):897–909, September 2000.

[68] Stephen Hemminger et al. Network emulation with netem. In *Linux Conference Australia (linux.conf.au lca2005), Canberra, Australia*, pages 18–23, April 2005.

[69] Google Inc. Google protocol buffers. `https://code.google.com/p/protobuf/`, 2015.

[70] Internet Policy Institute, University of Maryland, and College Park. *Report of the National Workshop on Internet Voting: issues and research agenda*. Internet Policy Institute, March 2001.

[71] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In *Theory of cryptography (TCC)*, pages 477–498. Springer, 2013.

[72] Aggelos Kiayias, Thomas Zacharias, and Bingsheng Zhang. End-to-end verifiable elections in the standard model. In *Proceedings of the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Advances in Cryptology - EUROCRYPT 2015, Sofia, Bulgaria*, pages 468–498. Springer, April 2015.

[73] Kim Potter Kihlstrom, Louise E Moser, and P Michael Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, 2003.

[74] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA*, pages 45–58. ACM, Oct 2007.

[75] R. Kotla and M. Dahlin. High throughput byzantine fault tolerance. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2004), Florence, Italy*, pages 575–584. IEEE Computer Society, June 2004.

[76] Miroslaw Kutylowski and Filip Zagórski. Scratch, click & vote: E2E voting over the internet. In *Towards Trustworthy Elections, New Directions in Electronic Voting*, volume 6000 of *Lecture Notes in Computer Science*, pages 343–356. Springer, 2010.

[77] Antti Laisi. Asynchronous postgresql java driver. `https://github.com/alaisi/postgres-async-driver/`, 2015.

[78] Jaynarayan H Lala. A byzantine resilient fault tolerant computer for nuclear power plant applications. In *FTCS 16th annual international symposium on fault-tolerant computing systems (Digest of papers)*. IEEE, July 1986.

[79] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, July 1982.

[80] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[81] Leslie Lamport. Byzantizing paxos by refinement. In *Proceedings of the 25th International Symposium on Distributed Computing (DISC 2011), Rome, Italy*, pages 211–224. Springer, September 2011.

[82] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[83] Butler Lampson. The ABCD's of paxos. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing, PODC 2001, Newport, Rhode Island, USA*, page 13. ACM, August 2001.

[84] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. Eventually consistent failure detectors. *Journal of Parallel and Distributed Computing*, 65:361–373, 2005.

[85] Patrick Lincoln and John Rushby. Formal verification of an interactive consistency algorithm for the draper FTP architecture under a hybrid fault model. In *Proceedings of the Ninth Annual Conference on Computer Assurance, 1994. COMPASS'94 Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security, Gaithersburg, MD, USA*, pages 107–120. IEEE, June 1994.

[86] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[87] Stathis Maneas. *Implementation and evaluation of a distributed, end-to-end verifiable, internet voting system*. Msc. thesis, University of Athens, May 2015.

[88] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine paxos. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2005), Yokohama, Japan*, pages 402–411. IEEE Computer Society, June 2004.

[89] M.G. Merideth, A. Iyengar, T.A. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan. Thema: Byzantine-fault-tolerant middleware for web-service applications. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 131–140. IEEE, October 2005.

[90] Zarko Milosevic, Martin Hutle, and André Schiper. Unifying byzantine consensus algorithms with weak interactive consistency. In *Principles of Distributed Systems*, pages 300–314. Springer, 2009.

[91] MIRACL. Miracl multi-precision integer and rational arithmetic c/c++ library. `http://www.certivox.com/miracl/`, 2015.

[92] Tal Moran and Moni Naor. Split-ballot voting: Everlasting privacy with distributed trust. *ACM Transactions on Information and System Security*, 13(2):16:1–16:43, March 2010.

[93] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous byzantine consensus with $t < n/3$ and o(n^2) messages. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France*, pages 2–9. ACM, July 2014.

[94] P.G. Neumann. Security criteria for electronic voting. In *Proceedings of the 16th National Computer Security Conference, Baltimore, Maryland, USA*, pages 478–481. National Institute of Standards and Technology (NIST), September 1993.

[95] Nuno Ferreira Neves, Miguel Correia, and Paulo Veríssimo. Solving vector consensus with a wormhole. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 16(12):1120–1131, 2005.

[96] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing, PODC'88, Toronoto, ON, Canada*, pages 8–17. ACM, August 1988.

[97] Sajeeva L. Pallemulle, Haraldur D. Thorvaldsson, and Kenneth J. Goldman. Byzantine fault-tolerant web services for n-tier and service oriented architectures. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), Beijing, China*, pages 260–268. IEEE Computer Society, June 2008.

[98] Arpita Patra, Ashish Choudhury, and C. Pandu Rangan. Asynchronous byzantine agreement with optimal resilience. *Distributed Computing*, 27(2):111–146, 2014.

[99] Arpita Patra and C. Pandu Rangan. Communication optimal multi-valued asynchronous byzantine agreement with optimal resilience. In *Proceedings of the 5th International Conference on Information Theoretic Security - ICITS 2011, Amsterdam, The Netherlands*, pages 206–226. Springer, May 2011.

[100] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, April 1980.

[101] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proceedings of the 11th Annual International Cryptology Conference, Advances in Cryptology - CRYPTO '91, Santa Barbara, California, USA*, pages 129–140. Springer, August 1991.

[102] André Postma and Thijs Krol. Interactive consistency in quasi-asynchronous systems. In *Proceedings of the Second IEEE International Conference on Engineering of Complex Computer Systems, Montrealm Quebec, Canada*, pages 2–9. IEEE, October 1996.

[103] Jean-Jacques Quisquater, Myriam Quisquater, Muriel Quisquater, Michaël Quisquater, Louis Guillou, Marie Guillou, Gaïd Guillou, Anna Guillou, Gwenolé Guillou, and Soazig Guillou. How to explain zero-knowledge protocols to your children. In *Proceedings of Advances in Cryptology-CRYPTO'89, Santa Barbara, California, USA*, pages 628–631. Springer, August 1990.

[104] Michael K Reiter. Secure agreement protocols: Reliable and atomic group multicast in rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security, Fairfax, VA, USA*, pages 68–80. ACM, November 1994.

[105] Bruce Schneier. *Applied cryptography*. John Wiley & Sons, 1996.

[106] S. Sen, W. Lloyed, and M. Freedman. Prophecy: Using history for high-throughput fault tolerance. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, San Jose, California, USA*, pages 345–360. USENIX Association, April 2010.

[107] Philip M. Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *Proceedings of the Seventh Symposium on Reliable Distributed Systems - SRDS 1988, Columbus, OH, USA*, pages 93–100. IEEE, October 1988.

[108] Tim Tiemens. Shamir's secret share in java. `https://github.com/timtiemens/secretshare`, 2015.

[109] Sam Toueg. Randomized byzantine agreements. In *Proceedings of the third annual ACM symposium on Principles of distributed computing, Vancouver, Canada*, pages 163–178. ACM, 1984.

[110] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07), Stevenson, WA, USA*, pages 59–72. ACM, October 2007.

[111] Shun-Sheng Wang, Kuo-Qin Yan, and Shu-Ching Wang. Achieving efficient agreement within a dual-failure cloud-computing environment. *Expert Systems with Applications*, 38(1):906–915, 2011.

[112] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. ZZ and the art of practical BFT execution. In *Proceedings of the Sixth European Conference on Computer Systems, EuroSys 2011, Salzburg, Austria*, pages 123–138. ACM, April 2011.

[113] Weigang Wu, Jiannong Cao, Jin Yang, and Michel Raynal. Using asynchrony and zero degradation to speed up indulgent consensus protocols. *Journal of Parallel and Distributed Computing*, 68(7):984 – 996, 2008.

[114] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Michael Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA*, pages 253–267. ACM, October 2003.

[115] Filip Zagórski, Richard T Carback, David Chaum, Jeremy Clark, Aleksander Essex, and Poorvi L Vora. Remotegrity: Design and use of an end-to-end verifiable remote voting system. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security, ACNS 2013, Banff, AB, Canada*, pages 441–457. Springer, June 2013.