Technische Universität München

Department of Informatics

Master's Thesis in Informatics

Byzantine Fault Tolerant Set Consensus with Efficient Set Reconciliation

Florian Dold

Technische Universität München

Department of Informatics

Master's Thesis in Informatics

Byzantine Fault Tolerant Set Consensus with Efficient Set Reconciliation

Byzantinischer Consensus auf Mengen mit effizientem Mengenabgleich

Author	Florian Dold
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Christian Grothoff, PhD (UCLA)
Submission Date	December 21, 2015

Informatik VIII
Chair for Network Architectures and Services

I confirm that this master's thesis is my own work and I have documented all sources and material used.

December 21, 2015

Signature

Abstract

Byzantine consensus is a fundamental and well-studied problem in the area of distributed system. It requires a group of peers to reach agreement on some value, even if a fraction of the peers is controlled by an adversary. This thesis proposes *set union consensus*, an efficient generalization of Byzantine consensus from single elements to sets. This is practically motivated by Secure Multiparty Computation protocols such as electronic voting, where a large set of elements must be collected and agreed upon. Existing practical implementations of Byzantine consensus are typically based on state machine replication and not well-suited for agreement on sets, since they must process individual agreements on all set elements in sequence. We describe and evaluate our implementation of set union consensus in GNUnet, which is based on a composition of Eppstein set reconciliation protocol with the simple gradecast consensus prococol described by Ben-Or.

Zusammenfassung

Byzantinischer Consensus ist ein fundamentales und weitläufig erforschtes Problem aus dem Bereich der verteilten Systeme. Ziel dabei ist es, dass sich eine Gruppe von Rechnerknoten auf einen einzigen Ausgabewert einigt, selbst dann wenn ein Teil der Rechnerknoten von einem Gegenspieler kontrolliert wird. In dieser Arbeit wird Byzantischer Consensus auf Mengen von Elementen als Generalisierung von Consensus auf einzelnen Werten vorgestellt. Dies ist durch die Anwendung in Protokollen für Secure Multiparty Computation motiviert, wie beispielsweise bei elektronischen Wahlen, für die eine große Menge an Eingabewerten gesammelt werden muss, und sich anschließend auf die gesamte Menge geeinigt werden muss. Bestehende, praktisch orientierte Lösungen basieren üblicherweise auf replizierten Zustandsautomaten und sind nicht für die Einigung auf Mengen geeignet, da jedes die einigung auf jedes Element der Menge sequentiell bearbeitet werden muss. Wir beschreiben und evaluieren eine Implentation für effizienten Consensus auf der Vereinigung von Mengen in GNUnet, welches auf einer Anwendung von Eppsteins Protokoll für effizienten Mengenabgleich auf Ben-Ors Consensus-Prototoll mit Gradecasts besteht.

Contents

1 Introduction 1
- 1.1 Background . . . 1
  - 1.1.1 Consensus . . . 2
  - 1.1.2 The FLP Impossibility Result . . . 2
  - 1.1.3 Interfaces to Consensus Protocols . . . 3
  - 1.1.4 Byzantine Consensus . . . 4
  - 1.1.5 Other consensus models . . . 5
- 1.2 Our contribution . . . 5
- 1.3 Roadmap . . . 6

2 Set Reconciliation 7
- 2.1 Background and Related Work . . . 7
- 2.2 High-Level Overview . . . 8
  - 2.2.1 Invertible Bloom Filter . . . 8
- 2.3 Difference Estimation . . . 9
- 2.4 Detailed Description . . . 9
- 2.5 Implementation in GNUnet . . . 11
  - 2.5.1 GNUnet's Architecture and Set . . . 11
  - 2.5.2 Optimizations . . . 12
- 2.6 Evaluation . . . 13

3 Set Union Byzantine Consensus 17
- 3.1 System Model . . . 17
  - 3.1.1 An argument against full asynchrony . . . 17
- 3.2 Simple Gradecast Consensus . . . 18
  - 3.2.1 Gradecast . . . 18
  - 3.2.2 Consensus . . . 19
- 3.3 Set Union Consensus . . . 20
  - 3.3.1 Definition . . . 20
  - 3.3.2 Set-Valued Gradecast . . . 21
  - 3.3.3 Correctness Sketch . . . 22
  - 3.3.4 Set-valued simple gradecast consensus . . . 22

3.4 Implementation in GNUnet 23
3.4.1 API 23
3.4.2 Space optimizations 23
3.4.3 Evaluation 24
3.5 Optimizations and Future Work 28
3.5.1 Fast Dissemination 28
3.5.2 Extension to Partial Synchrony 29
3.5.3 Set Canonicalization 29
3.5.4 Persistent Data Structures 30

4 Application to Secure Multiparty Computation 31
4.1 Bulletin Board for Electronic Voting 32
4.2 Distributed Threshold Key Generation and Cooperative Decryption . . 33
4.3 Electronic Voting with Homomorphic Encryption 34

5 Conclusion and Future Work 35

A Appendix 37
A.1 Set API Reference 37
A.2 Consensus API Reference 48

Chapter 1

Introduction

This thesis presents a new Byzantine fault-tolerant set union consensus protocol, and demonstrates the utility of the construction. The protocol allows a set of peers to compute and agree upon the union of sets. Each member begins with a set of values. At the end of the protocol, all honest members end up with the same superset, which includes at least all of the values from the subsets of the non-faulty peers, even in the presence of an adversary that can control a fraction of the peers.

We assume a partially synchronous communication model, where non-faulty peers are guaranteed to successfully receive values transmitted by other non-faulty peers within an existing but unknown finite period of time [22]. The protocol then allows for $\lceil n/3 \rceil - 1$ Byzantine faults among $n > 3$ participating peers.

1.1 Background

Distributed systems often implement services that look to a client as if they were implemented by a single computing facility. In order to achieve this, the components of the distributed system (henceforth called peers, since this work is primarily concerned with peer-to-peer networks) must be able to reach agreement amongst each other about input data, state transitions and outputs during a distributed computation.

While this problem is conceptually quite simple, it has proven to be very challenging when considering the following two aspects:

1. Peers can fail. In the simplest case, peers exhibit a *crash-fault* and simply stop doing any work. With *Byzantine*[1] faults, faulty peers can exhibit arbitrary and potentially coordinated behavior, making these faults much more challenging to

[1] The term was popularized by Lamport [39] and refers to a story about the Byzantine army, whose generals must agree on the choice to attack a city or retreat in the presence of traitorous generals.

handle. Practical systems have to consider Byzantine faults not only as a result of the presence of a attackers, but also as a result of software bugs and hardware malfunction.

2. The messaging system that peers use to communicate is often unreliable and *asynchronous*, meaning that there is no upper bound on the time it takes to deliver a message. Intuitively, asynchrony makes agreement harder since it is not possible anymore to distinguish between a faulty peer and a delayed message.

3. The clocks of peers run at different speeds (a type of asynchrony that is called processor asynchrony), preventing the reliable use of timeouts.

Note that the term "asynchronous" is is used somewhat inconsistently in the literature and sometimes refers to a model with an asynchronous messaging system but synchronous clocks.

1.1.1 Consensus

Many specific variants of the agreement problem (such as the interactive consistency [26], k-set consensus [16], or leader election [41] and many others [27]) exist. We will focus on the consensus problem, wherein each peer in a set of peers $\{P_1, \ldots, P_n\}$ starts with an initial value $v_i' \in M$ for an arbitrary fixed set M. At some point during the execution of the consensus protocol, each peer irrevocably decides on some output value $v_i \in M$. Informally, a protocol that solves the consensus problem must fulfill the following properties[2]:

- *Agreement:* If two peers P_i and P_j are correct then $v_i = v_j$.
- *Termination:* The protocol terminates in a finite number of steps.
- *Validity:* If all correct peers have the same input value $\tilde{v}$, then all correct peers decide on $\tilde{v}$.

Some definitions also include *strong validity*, which requires that the value that is agreed upon must have been the initial value of some correct peer [47].

1.1.2 The FLP Impossibility Result

A fundamental theoretical result (often called FLP impossibility for the initials of the authors) states, informally, that no deterministic protocol can solve the consensus problem in the asynchronous communication model, even in the presence of only one crash-fault [28].

[2]Different variations and names can be found in the literature. We have chosen a definition that naturally extends to our generalization to sets later on.

While this result initially seems discouraging, the conditions in which FLP impossibility holds are quite specific and subtle [3]. They have been challenged in a number of ways, including the following:

- *Common coin:* Some protocols introduce a shared source of randomness that the adversary cannot predict or bias. This breaks the assumption that the protocol must be deterministic. In practice, these protocols are very complex and often use variants of secret-sharing and weaker forms of Byzantine agreement to implement the common coin [24, 25, 45].
- *Failure oracles:* Approaches based on unreliable failure detectors [34] augment the model with oracles for the detection of faulty nodes. Much care has to be taken not to violate correctness of the protocol by classifying too many correct peers as faulty; this is problem is present in early systems such as Rampart [53] and SecureRing [35] as noted by Castro and Liskov [13, 12]. While the theory of failure detectors is quite established for the non-Byzantine case, it is not clear whether they are still useful in the presence of Byzantine faults.
- *Partial synchrony:* A model where a bound on the message delay or clock shift exists but is unknown or is known but only holds from an unknown future point in time is called partial synchrony. The FLP result does not hold in this model [22].
- *Minimal synchrony:* The definition of synchrony used by the FLP impossibility result can be split into three types of synchrony: Processor synchrony, communication synchrony and and message ordering synchrony. Dolev et al. [20] show that consensus is still possible if only certain subsets of these three synchrony assumptions are fulfilled.

1.1.3 Interfaces to Consensus Protocols

Consensus protocols described in theory-oriented work typically are designed to solve the problem of agreeing on a binary flag or a value from a set that is usually small. However, this primitive is not what typical applications need. While there are constructions that transform consensus on a small sets into efficient consensus on larger messages [29], we are not aware of any well-known practical implementation of the simple value-agreement protocols.

Instead, virtually all practical implementations of consensus use the state machine replication (SMR) approach [56], which by now is a well-established sub-field of distributed computing. In this framework, peers agree on a sequence of transitions of a state machine that are triggered by requests from clients. This approach makes it easy to "port" existing, non-fault tolerant services to a Byzantine fault tolerant implementation [13].

For non-Byzantine consensus, the most well-studied practical protocol is Lamport's Paxos [38, 37]. Various other solutions with subtle differences have been discovered [61]. Paxos uses a leader to coordinate peers; committing updates requires agreement from a quorum of the participating peers. Paxos is infamous for its complexity and difficulty, and some recent efforts [49] seek to provide a more understandable alternative, with Raft being the most prominent one.

SMR is, however, not well-suited for set union consensus, since the most direct implementation of set union agreement would reach agreement element-by-element. Considering that during non-civil periods, that is periods where Byzantine behavior is present, agreement on a single transition requires all-to-all communication, the communication complexity for agreement on k set elements would require $O(kn^2)$ bits communication. Since for real-world applications of set union agreement such as electronic voting, k greatly exceeds n, implementing this with SMR is not efficient.

Rather than seeing SMR as an alternative, our set union consensus construction should be understood as a complementary approach, that can be combined with SMR in two ways:

- The set union consensus protocol could be driven by state transitions of the replicated state machine.
- The set union consensus protocol could be invoked as a sub-protocol by each of the replicas.

In both alternatives, the state machines caches requests for insertion of elements from clients and agrees on the whole set at once.

Since the implementation of SMR is difficult and subtle [6], we leave the integration of our approach with existing SMR protocols as future work.

1.1.4 Byzantine Consensus

The Byzantine consensus problem [39] is a generalization of the consensus problem where peers might also exhibit Byzantine faults. A fundamental result is that no Byzantine consensus protocol with n peers can support $\lceil n/3 \rceil$ or more Byzantine faults in the asynchronous model [22].

Early attempts at implementing Byzantine consensus with the state machine approach are SecureRing [35] and Rampart [53]. These two approaches suffered from sacrificing correctness for progress guarantees in the presence of asynchrony [13].

Castro and Liskov's Practical Byzantine Fault Tolerance (PBFT) [13, 12] does not suffer from this problem. PBFT guarantees progress as long as the message delay does not

grow indefinitely for some fixed growth function[3]. Similar to Paxos, PBFT uses a leader to coordinate peers (called *replicas* in BPFT terminology). When replicas detect that the leader is faulty, they run a leader-election protocol to appoint a new leader.

In practice the approach taken by BPFT (and several derived protocols) has several problems [14]: Malicious clients can reduce the throughput of system to zero, and malicious leaders can slow down the system significantly. Correctness proofs for the respective protocols and the implementation of state machine replication are notoriously difficult [6].

Some more recent Byzantine state machine replication protocols such as Q/U [1] or Zyzzyva [36] have less overhead per request since they optimize for the non-Byzantine case. This is, however, often comes at the expense of robustness in the presence of Byzantine faults [14].

1.1.5 Other consensus models

Blockchain technology such as the cryptocurrency Bitcoin [46] has gained immense popularity over the past few years. The blockchain solves a slight variation of Byzantine consensus without strong validity [42, 31].

Ripple [57] purports to implement a variation of Byzantine consensus over sets of financial transactions. The set of peers that participate in the Ripple consensus is not globally defined, but each peer its own fixed list of peers, called the Unique Node List (UNL). Each UNL is individually assumed to hold a 80% majority of correct peers. In Ripple there is no consensus on the entirety of the set, but only a majority vote on elements that are accepted and applied as valid transactions. This is different from our model, where peers have to agree on the entirety of a set at once, instead of incrementally outputting accepted elements.

1.2 Our contribution

The work in this thesis proposes a new interface that lies between one-shot consensus and state machine replication, namely Byzantine set union consensus, where the value domain M contains the subsets of some potentially infinite universe U, that is $M = 2^U$.

While from a theoretical perspective an existing protocol could be used for set-valued consensus, it would not be efficient since elements of $M = 2^U$ are typically very large. If the initial values of the correct peers have a large intersection, redundant communication would take places when naively transmitting these values. Even if the initial overlap is

[3] In practice, exponential back-off is used

small, later stages of multi-valued consensus would incur large communication costs due to the redundancy in the set elements that are being transmitted.

Our protocol combines an existing protocol for Byzantine consensus [8] with Eppstein's protocol for efficient set reconciliation [23].

We demonstrate the practical applicability of our resulting abstraction by using Byzantine fault-tolerant set union consensus to implement distributed key generation, ballot collection and cooperative decryption from the Cramer-Gennaro-Schoenmakers remote electronic voting scheme [15] in GNUnet[4], a framework for secure peer-to-peer networking.

1.3 Roadmap

Chapter 2 is a self-contained description of set reconciliation based Eppstein's protocol [23] and our implementation of it in GNUnet. Chapter 3 applies set reconciliation to a simple consensus protocol based on graded broadcast [8] to yield an efficient protocol for set union consensus. We describe and benchmark our implementation of this protocol in GNUnet. Chapter 4 discusses the applications of set union consensus in secure multiparty computation protocols. Chapter 5 concludes and gives an overview of possible future work.

[4] `https://gnunet.org/`

Chapter 2

Set Reconciliation

The goal of set reconciliation is to identify the differences between two large sets, say S_a and S_b, that are stored on two different machines in a network. A simple but inefficient solution would be to transmit the smaller of the two sets, and let then receiver compute and announce the difference. In this chapter, we are concerned with protocols that are more efficient than this naive approach with respect to the amount of data that needs to be communicated when the sets S_a and S_b are large, but their symmetric difference $S_a \oplus S_b$ is small.

We will first discuss the theoretical aspects of communication-efficient set reconciliation, and then discuss our implementation of an existing approach, namely set reconciliation with invertible Bloom filters [23] it in GNUnet.

2.1 Background and Related Work

An early attempt to efficiently reconcile sets [43] was based on representing sets by their characteristic polynomial over a finite field. Conceptually, dividing the characteristic polynomials of two sets cancels out the common set elements, leaving only the set difference. The characteristic polynomials are transmitted as a sequence of sampling points, where the number of sampling points is proportional to the size of the symmetric difference of the sets S_a and S_b. The number of sampling points can be approximated with an upper bound, or increased on the fly should a peer be unable to interpolate a polynomial.

While theoretically elegant, the protocol is not efficient in practice. The computational complexity of the polynomial interpolation grows as $O(|S_a \oplus S_b|^3)$ and uses rather expensive arithmetic operations over large finite fields.

A more practical protocol was proposed by Eppstein et al. in 2011 [23]. It is based on Invertible Bloom Filters (IBFs), a data structure that is related to Bloom filters [10].

An attractive property of this approach is that IBFs are used both to construct an estimator for the size of the symmetric difference between two sets, as well as for the the reconciliation itself, which requires this estimate. We are not aware of any publicly available implementations of efficient set reconciliation, other than ours.

There is a generalization of IBFs to multi-party set reconciliation [44]. Since the approach is based on network coding and requires trusted intermediaries, it is not applicable to networks with Byzantine faults.

2.2 High-Level Overview

In this section, we will informally describe the set reconciliation protocol based on invertible Bloom filters. A formal derivation of the expected overhead and failure probabilities can be found in the work of Eppstein et al. [23]. Other work derives tighter bounds by applying the theory of random hypergraphs to invertible Bloom filters [54, 33].

2.2.1 Invertible Bloom Filter

The invertible Bloom filter (IBF) is a probabilistic data structure that encodes updates (insertions and deletions) to a set in constant space. The set elements affected by updates are represented by a constant-size key, derived from the element via a hash function. Under certain conditions, it is possible to "invert" the data structure and thereby extract some or all updates (consisting of the key and the direction that is either insert or delete) recorded in it.

Extracting an update is generally only possible if the update was only recorded once in the IBF. Recording a deletion and an insertion of the same element causes the two updates to cancel each other out, but storing a deletion or insertion of the same element twice or more makes this update impossible to decode. Updating an IBF is a commutative operation.

Since the data structure uses constant space, encoding cannot always succeed. Extracting updates (also called *decoding* an IBF) is a probabilistic operation that is more likely to succeed when the IBF is sparse, that is the number of encoded operations (*excluding* the operations that canceled each other out) is small. The decoding process can also be partially successful, if some elements could be extracted but the remaining IBF is non-empty.

In addition to recording single insertions or deletions, IBFs of the same size can also be combined with each other. When *subtracting* IBF_b from IBF_a, the resulting structure $\mathrm{IBF}_c = \mathrm{IBF}_a - \mathrm{IBF}_b$ contains all insertions and deletions from IBF_a, and an insertion

operation from IBF_b is recorded as a deletion and vice versa. Effectively the IBF subtraction allows to compute the difference between two sets when each set was encoded as an IBF containing only insertion operations.

When the symmetric difference between the sets is small enough compared to the size of the IBFs, the result IBF_c of the subtraction can be decoded, since the common elements encoded in IBF_a and IBF_b cancel each other out. This makes it possible to obtain the elements of the symmetric difference, even when the IBFs that represent the full sets can not be decoded.

As long as the symmetric difference between the original sets S_a and S_b can be approximated well enough, IBFs can be used for set reconciliation by encoding S_a in IBF_a and S_b in IBF_b. One of the IBFs is sent over the network, the $\text{IBF}_c = \text{IBF}_a - \text{IBF}_b$ is computed and decoded. Should the decoding (partially) fail, the same procedure is repeated with larger IBFs.

2.3 Difference Estimation

In order to select the size of the IBF appropriately for the set reconciliation protocol, one needs to estimate the symmetric difference between the sets that are being reconciled. Eppstein et al. [23] describe a very simple technique, called strata estimation, that is accurate for small differences. While Eppstein et al. suggest combining the strata estimator, with a min-wise estimator, which is more accurate for large differences, we only discuss strata estimators here for simplicity.

The set difference is estimated by having both peers encode their set in a strata estimator. One of the strata estimators is then sent over to the other peer, which uses the two strata estimators to estimate the size of the symmetric difference between the sets they encode.

A strata estimator is an array of fixed-size IBFs. These fixed-size IBFs are called *strata* since each of them contains a sample of the whole set, with increased sampling probability towards inner strata. Similar to how two IBFs can be subtracted, strata estimators are subtracted by pairwise subtraction of the IBFs they consist of.

With every IBF of the strata estimator that results from the subtraction, a decoding attempt is made. The number of successfully decoded elements in each stratum allows an estimate to be made on the set difference.

2.4 Detailed Description

Under the hood, an IBF of size n is an array of n buckets. Each bucket holds three values:

- A signed counter that handles overflow via wrap-around. Recording an insertion or deletion adds -1 or $+1$ to the counter respectively. In our implementation, we use an 8-bit counter.
- An $\oplus$-sum[1], called the `keySum`, over the keys that identify set the elements that were recorded in the bucket. We write $H(e)$ for the key derived from the set element e. In our implementation, we use 64-bit keys.
- An $\oplus$-sum, called the `keyHashSum`, over a the hash $h(\cdot)$ of each key that was recorded in the bucket. In our implementation, we use 32-bit key hashes.

Encoding an update in an IBF records the update in k different buckets of the IBF. The indices of buckets that record the update are derived via a k independent hash functions from the 64-bit key of the element that is subject of the update. We write $\text{Pos}(x)$ for the set of array positions that correspond to the element key x.

Before we describe the decoding process, we introduce some terminology. A bucket is called a *candidate bucket* if its counter is -1 or $+1$, which might indicate that the `keySum` field contains the key of an element that was the subject of an update. Candidate buckets that contain the key of an element that was previously updated are called *pure buckets*.

Candidate buckets are not necessarily pure buckets, since a candidate bucket could also result from, for example, first inserting an element key e_1 and then deleting e_2 when $\text{Pos}(e_1) \cap \text{Pos}(e_1) \neq \emptyset$ and $\text{Pos}(e_1) \neq \text{Pos}(e_2)$.

The `keyHashSum` provides a way to detect if a candidate bucket is not a pure bucket, namely when $h(\texttt{keySum}) \neq \texttt{keyHashSum}$. The probability of classifying an impure bucket as pure with this method is dependent on the probability of a hash collision. Another method to check for an impure candidate buckets with index i is to check whether $i \notin \text{Pos}(\texttt{keySum})$.

The decoding process then simply searches for buckets that are, with high probability, pure. When the `count` field of the bucket is 1, the key decoding procedure reports the key as "inserted" and exececutes a deletion operation with that key. When the `count` field is -1, the key is reported as "deleted" and subsequently an insertion operation is executed.

With a probability that increases with sparser IBFs, decoding one element will cause one or more other buckets to become pure, and the decoding can be repeated to yield more encoded updates. This iteration either ends with an empty, undecodable or looping IBF. In case of an undecodable IBF, a larger IBF must be used or the reconciliation must fall back to the naive approach of sending the whole set. A looping IBF is an IBF (not considered by Eppstein et al. [23]) where the iterated decoding does not terminate. This

[1]The $\oplus$ denotes bit-wise exclusive or.

case can be handled by stopping the iteration and reporting the IBF as undecodable when the number of decoded elements exceeds a threshold.

2.5 Implementation in GNUnet

Set reconciliation in GNUnet is implemented in the *Set* service. The *Set* service provides a generic interface for set operations between two peers; the set operations currently implemented are the IBF-based set reconciliation described in this chapter and set intersection based on Bloom filters [59].

In addition to the operation-specific protocols, the following aspects are handled generically (i.e. independent of the specific remote set operation) in the SET service:

Local set operations Applications need to create sets and perform actions (including iteration, insertions, deletions) on them locally.

Concurrent modifications While a local set is used in a set operation, the application may still mutate that set. To allow this without interfering with the set operation protocols, changes to sets are versioned. A set operation only sees the state of a set at the time the operation was started.

Lazy copying Some applications building on the SET service, especially the CONSENSUS service described in the next chapter, manage many local sets that are large but only differ in a few elements. We optimize for this case by providing a lazy copy operation, that returns a logical copy of the set without physically duplicating the whole sets. The same mechanism that handles versioning of sets for concurrent modification also allows "forks" of sets to be created efficiently. A more efficient implementation of this data structure could probably be achieved with functional/persistent data structures for sets [48].

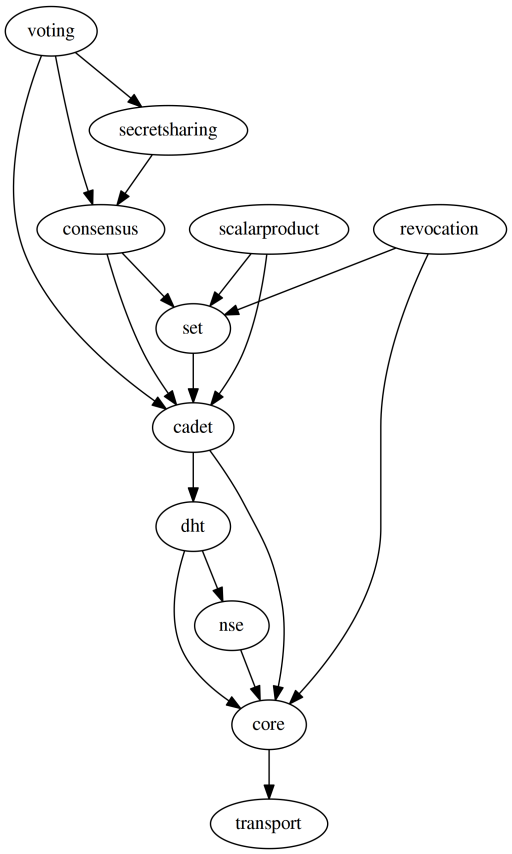
Negotiating remote operations In a remote set operation, the involved peers have one of two roles: The acceptor, which waits for remote operation requests and accepts or rejects them, as well as the initiator, which sends the operation request.

2.5.1 GNUnet's Architecture and SET

GNUnet is composed of various components that run in separate operating system processes and communicate via message passing. Components that expose an interface to other components are called *services* in GNUnet. A subset of the service dependencies is shown in Figure 2.1.

The complete interface to the SET service is given in the Appendix.

Figure 2.1: Dependencies and dependents of the SET service in GNUnet



The SET service uses the CADET service to establish an encrypted and authenticated communication channel between two peers [52].

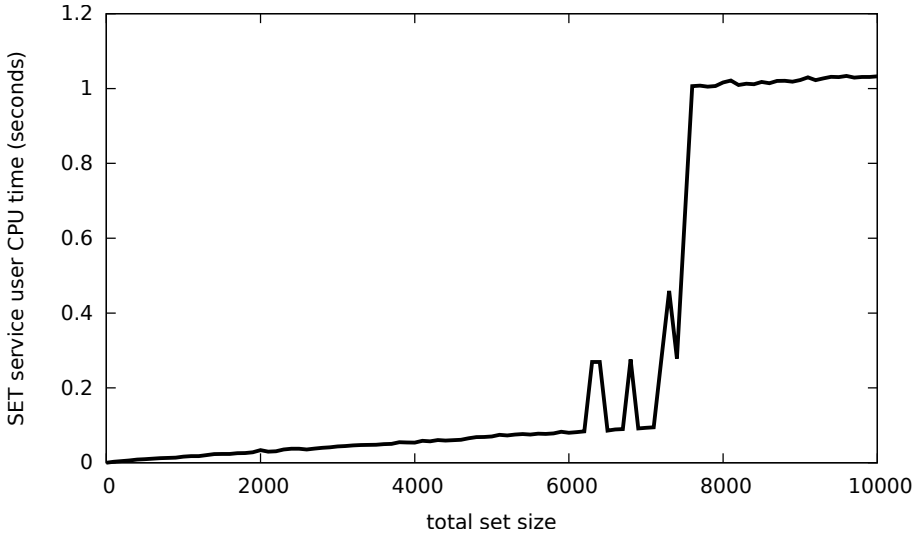
Figure 2.1 shows the dependencies between services in GNUnet that involve SET. Currently the GNU Name System [62] uses the reconciliation protocol in SET for key revocation. The SECRETSHARING service uses SET as the communication primitive for distributed threshold key generation and cooperative encryption without a trusted third party [19, 30].

The main application for SET in this thesis is the CONSENSUS service discussed in the next chapter.

2.5.2 Optimizations

Our implementation estimates the initial difference between sets only using strata estimators. We gzip-compress [18] the strata estimator, which is 60KB uncompressed. The compression is particularly effective for very small and very large sets, due to the high probability of long runs of zeros or ones in the most sparse or most dense strata

Figure 2.2: CPU system time for the SET service in relation to total set size. No difference between sets. Average over five executions.



respectively.

We also use a *salt* when deriving the bucket indices from the element keys. When the decoding of an IBF fails, the IBF size is doubled and the salt is changed. This prevents decoding failures in scenarios where keys map to the same bucket indices even modulo a power of two, where doubling the size of the IBF does not remove the collision. A further possible optimization that we did not implement would be to only change the salt and keep the size of the IBF when the decoding process failed after some elements were already decoded and the remaining IBF is very sparse.

2.6 Evaluation

The results in this section were generated with the `gnunet-consensus-profiler` tool, which uses GNUnet's Testing library to execute a set union operation via loopback on one GNUnet peer. Set elements are randomly generated and always 64 bytes large. All benchmarks were run on a machine with a 24-core 2.30GHz Intel Xeon E5-2630 CPU, and GNUnet SVN revision 36765.

The sudden jump in processing time that is visible in Figure 2.2 can most likely be explained by effects of the processor cache. The effect could not be observed when we ran the experiment under profiling tools.

The logarithmic increase of the traffic with larger sets (see Figure 2.3) can be explained by the compression of strata estimators. Since the k-th strata samples the set with

Figure 2.3: CADET traffic the SET service in relation to total set size. No difference between sets. Average over five executions.

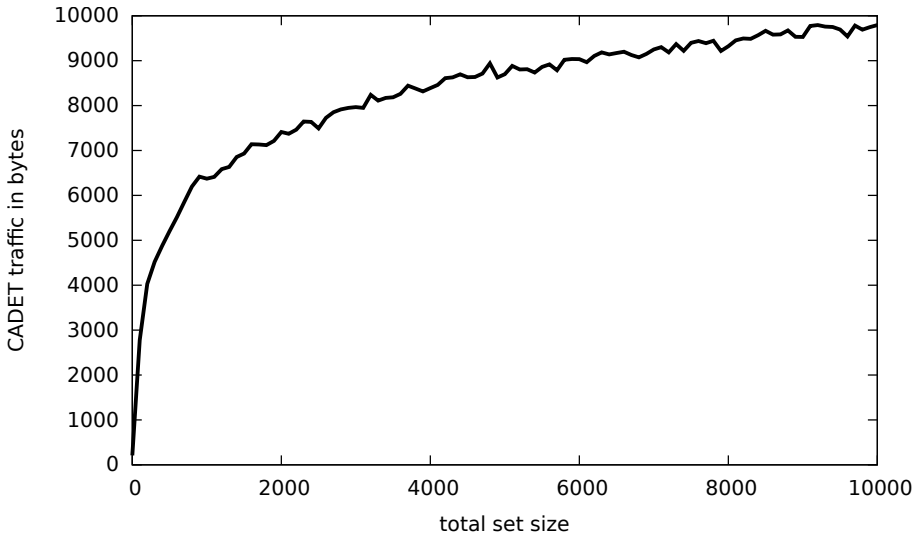


Figure 2.4: CPU system time for the SET service in relation to symmetric set difference. No common elements. Average over five executions.

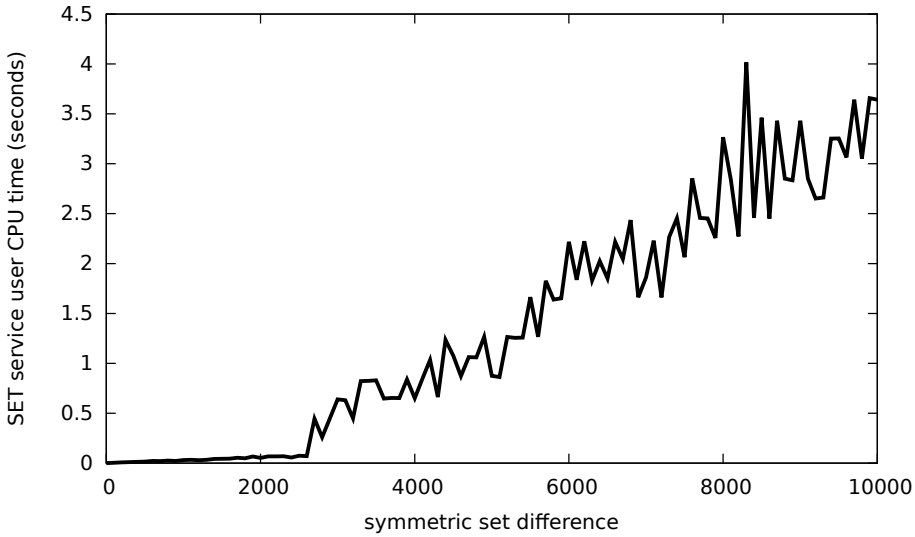


Figure 2.5: CADET traffic the SET service in relation to symmetric difference. No common elements. Average over five executions.

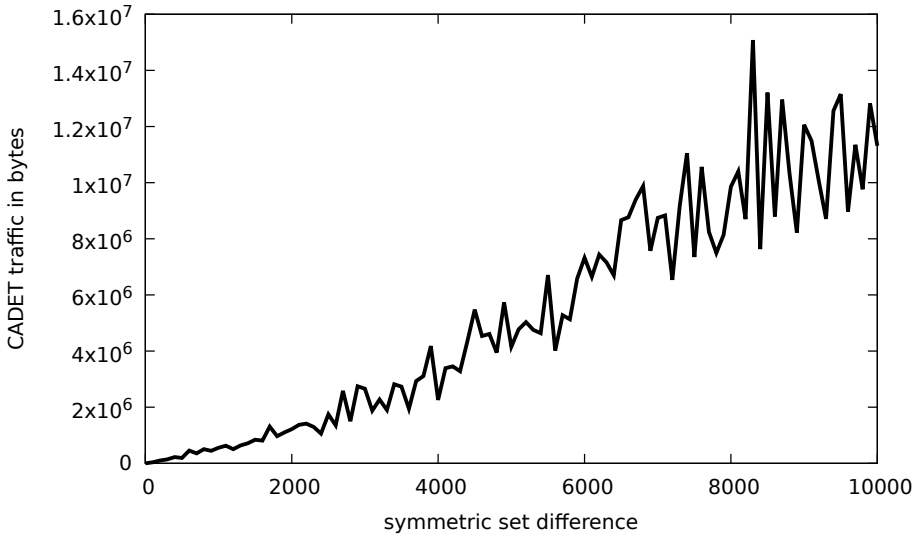


Figure 2.6: Number of times that IBF decoding failed and a larger IBF had to be sent, average over five executions.

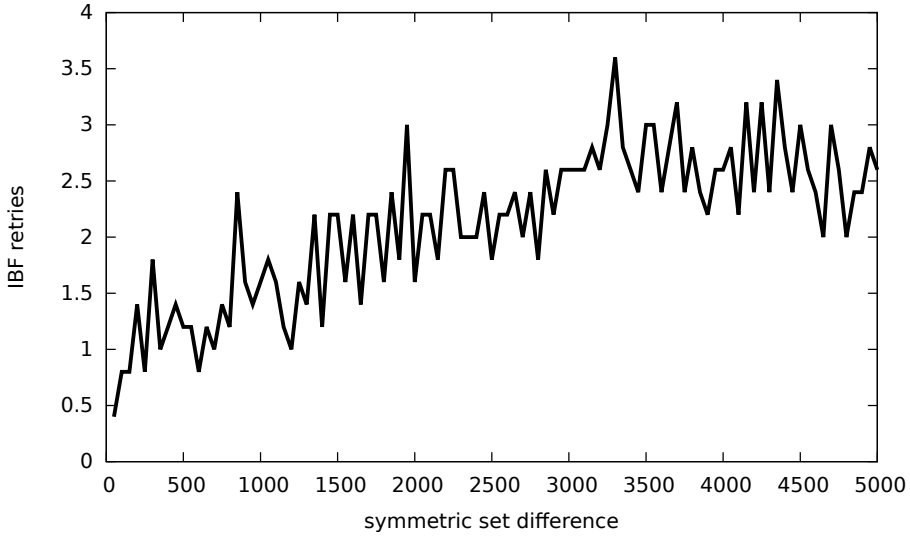
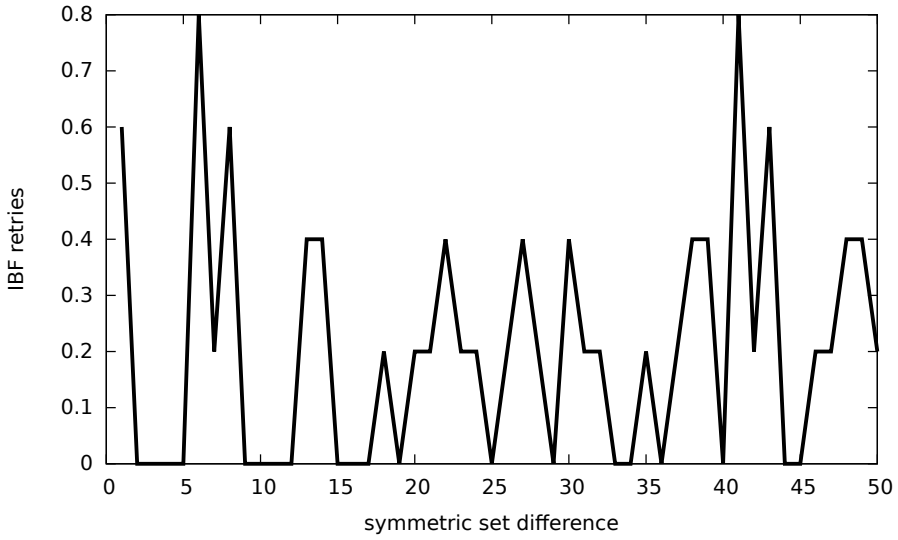


Figure 2.7: Number of times that IBF decoding failed and a larger IBF had to be sent, average over five executions.



probability 2^{-k}, the sparser strata tend to contain long runs of zeros that are easily compressed.

The data shown in Figure 2.6 suggests that our difference estimator tends to underestimate the difference for larger symmetric differences. Figure 2.7 shows that for smaller differences, the number of retries is lower. The estimation could be improved by introducing a correction factor or following the suggestion by Eppstein et al. and combine strata estimation with MinWise difference estimators [40], which are more accurate for larger differences but less accurate for smaller ones.

Chapter 3

Set Union Byzantine Consensus

In this chapter we combine the efficient set reconciliation described in the previous chapter with an existing protocol for Byzantine consensus, namely simple gradecast consensus [8][1]. We first explain the assumptions about our communication and adversary model, give a definition of set union consensus, describe the simple gradecast consensus and extend it to sets. We will furthermore describe our implementation of this protocol in GNUnet and evaluate its performance.

3.1 System Model

We assume a computationally unbounded adversary that can corrupt at most $t = \lceil n/3 \rceil + 1$ peers. The adversary is static, i.e. the set of corrupted peers is fixed before the protocol starts, but this set is not available to the correct peers. The actual number of faulty peers is denoted by f, with $f \leq t$.

Peers communicate over pairwise channels that are authenticated. Message delivery is reliable (i.e. messages arrive uncorrupted and in the right order) but the receipt of messages may be delayed. We make the same assumption as Castro and Liskov [13, 12] about this delay, namely that it does not grow faster than some function (usually $f(t) = 2^t$) of wall clock time.

3.1.1 An argument against full asynchrony

In the literature, the asynchronous model is often assumed to be the only one that matches real distributed systems, sometimes "distributed" is even used as a synonym

[1] Ben-Or called the algorithm ByzConsensus in "Simple Gradecast Based Algorithms". Since Gradecast was originally discovered in the context of a more complex protocol for fully asynchronous consensus [25], the name "simple gradecast consensus" seems appropriate.

for asynchronous [34].

We argue that this view is rather restrictive. On the one hand, protocols that are used in practice (such as Paxos and BPFT) are often described as working in the asynchronous model, but they make synchrony assumptions for lifeness, and are strictly speaking partially synchronous. On the other hand, we are not aware of any truly asynchronous consensus protocol that can claim to be practical. Some protocols provide rather strong guarantees on both correctness and lifeness, the expected number of rounds is usually constant [45]. Some protocols have simplicity as a goal [45], but are still not practical; they shift complexity to other building blocks (such as a common coin oracle) that are assumed to be available, but difficult to implement in practice. Implementing a common coin oracle resilient against an active adversary is non-trivial and usually required extra assumptions such as a trusted dealer in the startup phase [11] or shared memory [4].

3.2 Simple Gradecast Consensus

The simple gradecast consensus [8] builds a consensus protocol by composing executions of a weakly broadcast protocol, namely gradecast [8, 25]. The resulting deterministic protocol has message complexity $O(f \cdot n^2)$. The asymptotic message complexity is not optimal, but the algorithm is simple and does not make any unrealistic assumptions about the system model.

3.2.1 Gradecast

A *graded broadcast* is a communication protocol where a leader P_L broadcasts a message m among a fixed set $\mathcal{P} = \{P_1, \ldots, P_n\}$ of peers. For notational convenience, we assume that $P_L \in \mathcal{P}$. In contrast to an unreliable broadcast, the graded broadcast provides certain correctness properties to the receivers, even if the leader is exhibiting Byzantine faults.

Specifically, receiver P_i obtains not only a message m_i but also a confidence value $c_i \in \{0, 1, 2\}$ that "grades" the correctness of the broadcast.

The following correctness properties must hold:

1. If $c_i \geq 1$ then $m_i = m_j$ for correct P_i and P_j
2. If P_L is correct, then $c_i = 2$ and $m_i = m$ for correct P_i.
3. $|c_i - c_j| \leq 1$ for correct P_i and P_j.

Informally speaking, when a correct peer P_i receives a graded broadcast with confidence 2, it can deduct that all other peers received the same message, but some other peers might have only received it with a confidence of 1.

Receiving a graded broadcast with confidence 1 also guarantees that all other correct peers received the same message. However it indicates that the leader behaved incorrectly. No assumption can be made about the confidence of other peers.

Receiving a graded broadcast with confidence 0 indicates that the leader behaved incorrectly and, crucially, that all other correct peers *know* that the leader behaved incorrectly.

These are the communication steps for peer P_i:

1. LEAD: If $i = L$ then send the input value v to $P_1, \ldots, P_n$.
2. ECHO: Send the value $\bar{v}$ received in LEAD to $P_1, \ldots, P_n$.
3. CONFIRM: If a common value Z was received at least $n - t$ times in round ECHO, send Z to $P_1, \ldots, P_n$. Otherwise, send nothing.

The grading is done with the following rules. With $\perp$ we denote as special value that indicates the absence of a meaningful value. The output of the grading is a tuple containing the output value and the confidence.

- If some X was received at least $n - t$ times in CONFIRM, output $(X, 2)$.
- Otherwise, if some X was received in CONFIRM at least $t + 1$ times, output $(X, 1)$.
- Otherwise, output $(\perp, 0)$

A proof that the above protocol satisfies the three gradecast properties based on a simple counting argument is given by Feldman et al. [25]. We also give a proof sketch for the set-valued gradecast protocol in section 3.3.3.

3.2.2 Consensus

The gradecast can be used to implement a consensus protocol [8].

Each peer stores a list of blacklisted peers. Blacklisted peers are excluded from the protocol; their messages are ignored. In the simple gradecast consensus protocol, peers corrupted by the adversary are forced to either expose themselves as faulty (and consequently be excluded) by gradecasting a value with low confidence, or follow the protocol and allow all peers to reach agreement.

The protocol consists of $t + 1$ super-rounds. Each peer stores in addition to the blacklist a candidate value that is set to the peer's initial value before the first round. In each super-round, each peer gradecasts their candidate value. Peers that gradecast with a confidence less than 2 are put in the blacklist. Recall that different correct peers might receive a gradecast with different confidence, but the difference between confidences is at most 1; thus peers do not necessarily agree on the blacklist. At the end of each super-round, peers change their candidate value to the value that was received most

often from gradecasts with a confidence of as least 1. After the last super-round, the peers commit to their current candidate value.

A full proof of the correctness of simple gradecast consensus was given by Ben-Or [8]. The protocol described in that paper additionally uses early stopping [21]. We do not include early stopping because it would make the implementation more complex.

If the candidate value that was determined after the last super-round did not receive a majority of at least $2t + 1$ or the blacklist has more than t entries, either more than t faults happened or, in the partially synchronous model, correct peers did not receive a message within the designated round due to the delayed delivery.

3.3 Set Union Consensus

We now show how the simple gradecast consensus can be combined with set reconciliation to yield an efficient protocol for exact agreement on sets.

The basic idea is to replace sending single values with a set reconciliation. The grading must be lifted from single values to sets.

3.3.1 Definition

We now give a definition of set union consensus that is motivated by practical applications to secure multiparty computation protocols such as electronic voting, which are discussed in more detail in the next chapter.

Consider a set of n peers $\mathcal{P} = \{P_1, \ldots, P_n\}$. Fix some (possibly infinite) universe M of set elements that can be represented by a bit string. Each peer P_i has an initial set $S_i^{(0)} \subseteq M$.

Let $R(s) : 2^M \to 2^M$ be a function that canonicalizes subsets of M by replacing multiple conflicting elements with the lexically smallest element in the conflict set and removes invalid elements. What is considered conflicting or invalid is application-specific.

During the execution of the set union consensus protocol, after finite time each peer P_i irrevocably commits to a set S_i such that

1. For any pair of correct peers P_i, P_j it holds that $S_i = S_j$.
2. If P_i is correct and $e \in S_i^0$ then $e \in S_i$.
3. The set S_i is canonical, that is $S_i = R(S_i)$.

For certain applications, the canonicalization function enables to set an upper bound on the number of elements that can simultaneously be in a set. For example in electronic

voting, canonicalization would remove malformed votes and only keep one vote for two different (encrypted) votes submitted by the same voter identity.

3.3.2 Set-Valued Gradecast

The decision of the CONFIRM step and the grading can be lifted to sets by considering elements separately and then choosing the lowest grading. A problem with this direct lifting is that "send nothing" in the CONFIRM step cannot be represented directly anymore. We resolve that by marking the whole set that contains a minority element as "contested".

We introduce some notation first. Let $\chi^+(i, R, x)$ and $\chi^-(i, R, x)$ be the number of peers that included / excluded the element x in the set reconciled with P_i in round R.

The gradecast protocol then proceeds as follows:

1. LEAD: If $i = L$ then send the input *set* to $P_1, \ldots, P_n$.
2. ECHO: Send the *set* received in LEAD to $P_1, \ldots, P_n$.
3. CONFIRM:
   - Let $x \in Z_{maj}$ if $\chi^+(i, ECHO, x) \geq n - t$.
   - Let $x \in Z_{min}$ if $\chi^+(i, ECHO, x) < n - t$ or $\chi^-(i, ECHO, x) < n - t$
   - If $Z_{min} = \emptyset$, send (NonContested, Z_{maj}) to $P_1, \ldots, P_n$.
   - Otherwise send (Contested, Z_{maj}) to $P_1, \ldots, P_n$.

Note that sending a set can be done efficiently with set reconciliation, where the receiving party reconciles with a set that, in an execution without Byzantine faults, would match the sender's set.

The following rules are applied for grading:

- Let g_{nc} be the number of (NonContested, _) messages from the CONFIRM round.
- Let $x \in X$ iff $\chi^+(i, CONFIRM, x) \geq t + 1$
- If $g_{nc} \geq n - t$ then output $(X, 2)$
- Otherwise, if $g_{nc} \geq t + 1$ then output $(X, 1)$.
- Otherwise, output $(\perp, 0)$

3.3.3 Correctness Sketch

We now show that this protocol satisfies the three gradecast properties. The proof is modeled after the argument given by Feldman et al. [25, 24].

- Lemma 1: If two correct peers confirm (NonContested, A) and (NonContested, B) then $A = B$.
  - Proof via contradiction / counting argument
  - Assume w.l.o.g. $x \in A$ and $x \notin B$.
  - At least $n - t$ peers must have echoed a set that includes x to the first peer.
  - Suppose f of these peers were faulty, then $n - t - f$ good peers included x
  - That leaves $(n - f) - (n - t - f) = t$ good peers that could have excluded x
  - Thus only $2t$ could have echoed a set to the second peer that excludes x.
  - But $2t < n - t$, thus correct peers can't confirm (NonContested, B)
- Property 1: If $c_i, c_j \geq 1$ then $m_i = m_j$ for correct P_i and P_j
  - Directly follows from grading rules and Lemma 1
- Property 2: If P_L is correct, then $c_i = 2$ and $m'_i = m$ for correct P_i.
  - All good players ECHO and CONFIRM the same set
- Property 3: $|c_i - c_j| \leq 1$ for correct P_i and P_j.
  - If a correct player sets $c_i = 2$, then at least $2t + 1$ players must have sent (NonContested, _), and at least $t + 1$ of them most have been good. Thus good players decide at least $c_j \geq 1$

3.3.4 Set-valued simple gradecast consensus

The extension of the simple gradecast consensus protocol to sets is very close to the original protocol. Instead of a candidate value, peers now have a candidate set.

To guarantee the second correctness property for set union consensus, namely that an element that was in the input set of one correct peer will be in the output set of all correct peers, every peer does a set reconciliation with each other peer. New elements learned during these reconciliation are added to the peers' candidate sets.

The rest of the protocol proceeds just like the single-value case, except that the gradecasts are replaced with set gradecasts, and the new candidate set is obtained by only including elements that were in a majority of set gradecasts with confidence > 0.

3.4 Implementation in GNUnet

We implemented the set union consensus protocol in the Consensus service of GNUnet. Consensus builds on the Set service described in the previous chapter.

3.4.1 API

To request the execution of a set union consensus, peers need to specify the following parameters:

- The list of other peers that participate
- A 512-bit application ID, which allows multiple instances of the protocol to be executed concurrently without interference.
- The desired start and end time of the protocol execution.

These parameters are hashed into a global identifier that uniquely identifies the consensus session.

The API client then gives the list of set elements that should be used in the consensus protocol. The client is notified by a callback when the consensus succeeds or an error occurs.

3.4.2 Space optimizations

To keep the description of the set union consensus protocol in the previous section succinct, we assumed that peers transmit sets using the reconciliation protocol, effectively reconciling the set is being sent with every peer's current set, resulting in a new set.

Our implementation adds some optimizations to make the protocol more practical. Since the new sets usually differ in only a few elements, we do not create new sets. Instead, in the leader round we just store the set of differences with a reference to the original set. In the ECHO and CONFIRM round, we also reconcile with respect to the set we received from the leader and not a peer's current set.

In the ECHO round, we only store one set and annotate each element with that indicates which peer included or excluded that set element. This allows a rather efficient computation of the set and contestation flag that is used in the CONFIRM round.

3.4.3 Evaluation

Methodology

We implemented a profiler for the Consensus service using GNUnet's Testbed framework [60]. The profiler emulates a netwok of GNUnet peers connected in a clique.

The Consensus service can be configured to exhibit the following types of adversarial behavior:

- *SpamAlways:* With this behavior enabled, a peer adds a constant number of additional elements in every reconciliation.
- *SpamLeader:* This behavior is like *SpamAlways*, except that additional elements are only added in reconciliations where the peer acts as a leader.
- *SpamEcho:* Likewise, elements are only inserted in echo rounds.
- *Idle:* With this behavior enabled, peers do not participate actively in the protocol, which amounts to a crash fault from the start of the protocol. This type of behavior is not interesting for the evaluation, but used to test the implementation with regards to timeouts and majority counting.

For the *Spam-** behaviors, two different variations are implemented. One of them ("*-replace") always generates new elements for every reconciliation. This is not typical for real applications (since the number of stuffable elements would be limited by set canonicalization), but shows the performance impact in the worst case. The other variation ("*-noreplace") reuses the same set of additional elements for all reconciliations, which is more realistic for most cases.

We did not implement adversarial behaviour where elements are elided, since the resulting traffic is the same as for additional elements, and memory usage would only be reduced.

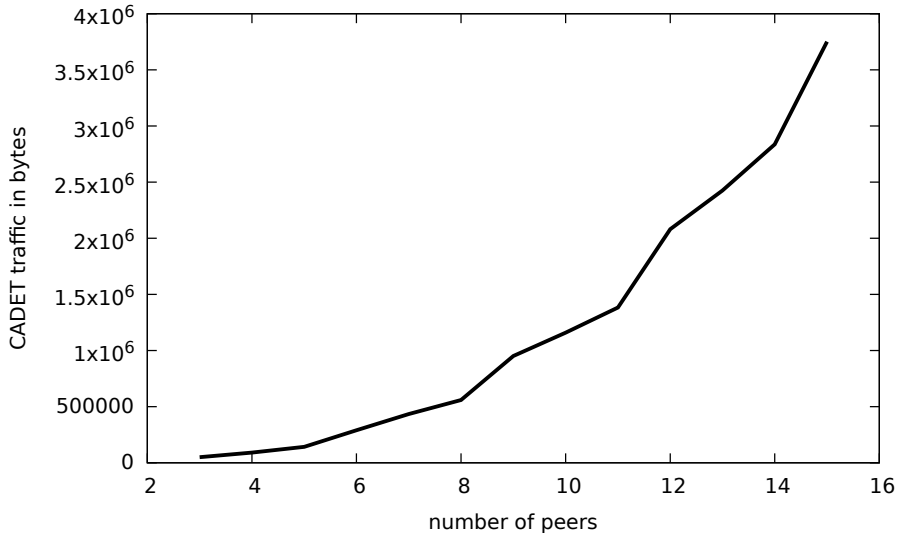
The traffic consumption was measured using the statistics that the Cadet service provides. Processor time was measured using GNUnet's resource reporting functionality, which uses the `wait3` system call for that purpose. The peak heap memory usage was measured with Valgrind's Massif tool[2].

All benchmarks were run on a machine with a 24-core 2.30GHz Intel Xeon E5-2630 CPU, and GNUnet SVN revision 36765.

All peers in our experiment start with the same set of elements; different sets would only affect the all-to-all union phase of the protocol which only does pairwise set reconciliation.

[2] `http://valgrind.org/docs/manual/ms-manual.html`

Figure 3.1: Cadet traffic per peer for 100 elements and only correct peers. Average over five executions.



Results

As expected, traffic increases cubically with the number of peers when no malicious peers are present (Figure 3.1). Most of the CPU time (Figure 3.3) is taken up by CADET, which uses expensive cryptographic operations [52]. Since we ran the experiments on a multicore machine, the total latency follows the same pattern as the traffic.

Note that even though our implementation falls back to synchronous rounds when they do not received an expected reconciliation after a timeout, in the non-malicious case the consensus can stop early and does not have to wait for round timeouts (see Figure 3.2).

The number of additional elements that occured during set reconciliations is shown in Figure 3.6. The number of stuffed elements for the “SpamEcho” behavior is larger than for “SpamLead”, since multiple ECHO rounds are executed for one LEAD round, and the number of stuffed elements is fixed per reconciliation. When malicious peers add extra elements during the LEAD round, the effect of that is amplified, since all correct receivers have to re-distribute the additional elements in the ECHO/CONFIRM round. Even though adding elements in the LEAD round requires the least bandwidth from the leader the effect on traffic and latency is the largest (see Figures 3.4 and 3.5).

Finally, when the number of stuffed elements is limited (“SpamAll-noreplace” in Figs. 3.4, 3.5, 3.6), to a fixed set (instead of stuffing fresh elements in each reconciliation, as with the “SpamEcho-noreplace” and “SpamEcho-replace” behaviors), the effect on the performance is very limited.

Figure 3.2: Runtime of consensus for 100 elements of 64 bytes and only correct peers. Average over five executions.

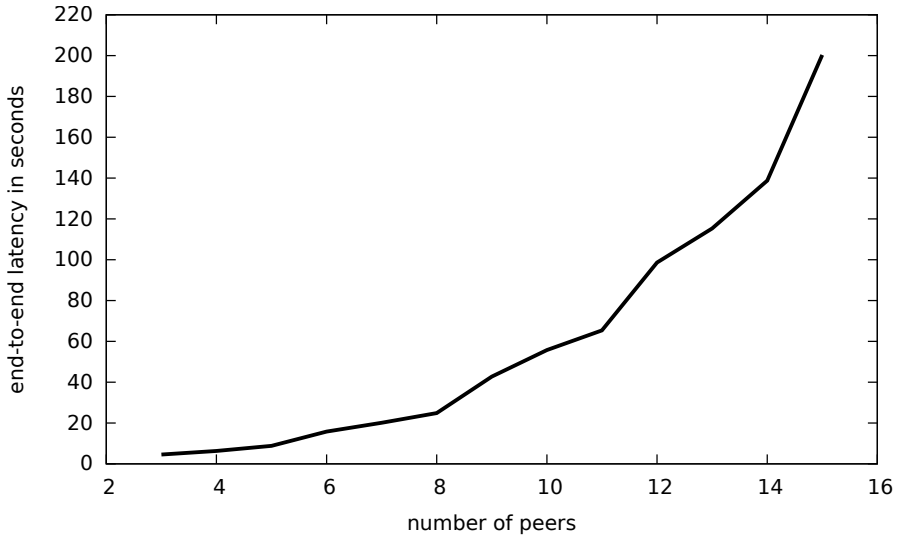


Figure 3.3: CPU of consensus for 100 elements of 64 bytes and only correct peers. Average over five executions.

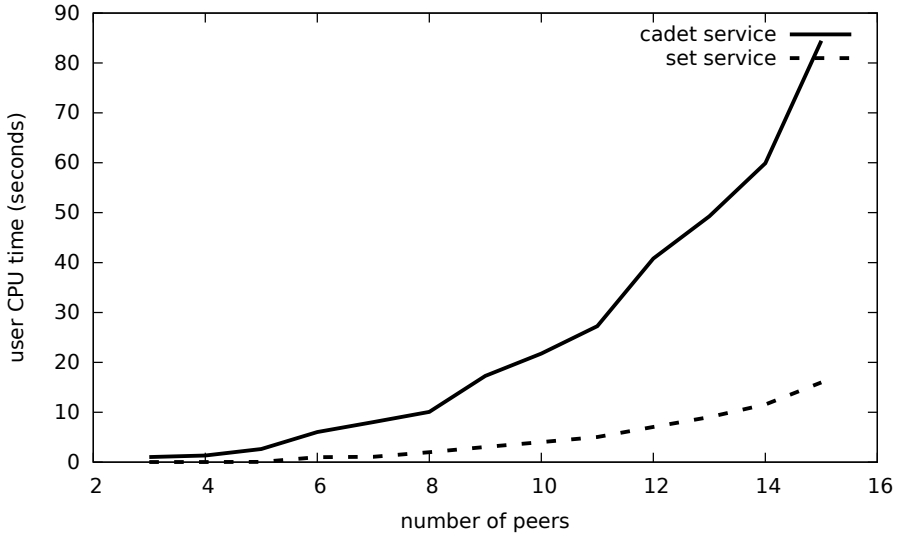


Figure 3.4: Cadet traffic for consensus on 100 elements of 64 bytes and one malicious peer with the indicated mode. Average over five executions.

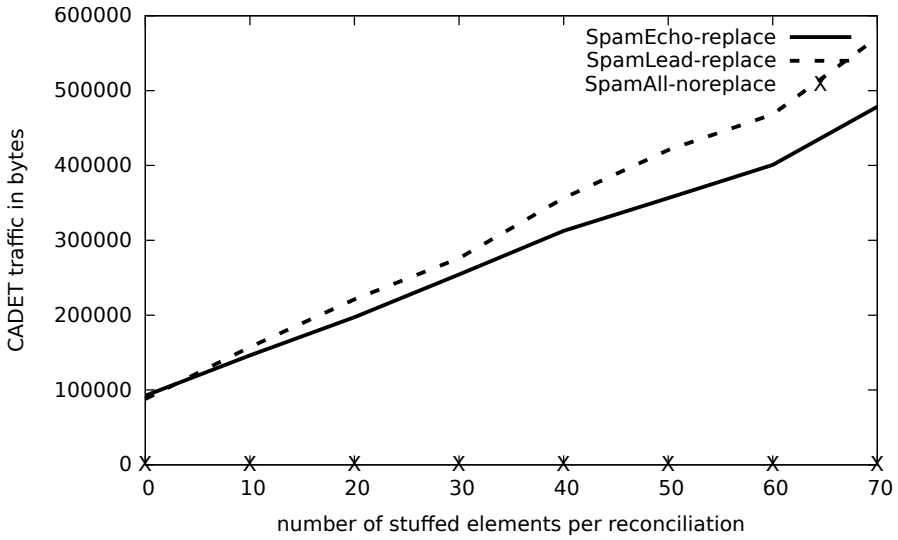


Figure 3.5: Latency for consensus on 100 elements of 64 bytes and one malicious peer with the indicated mode. Average over five executions.

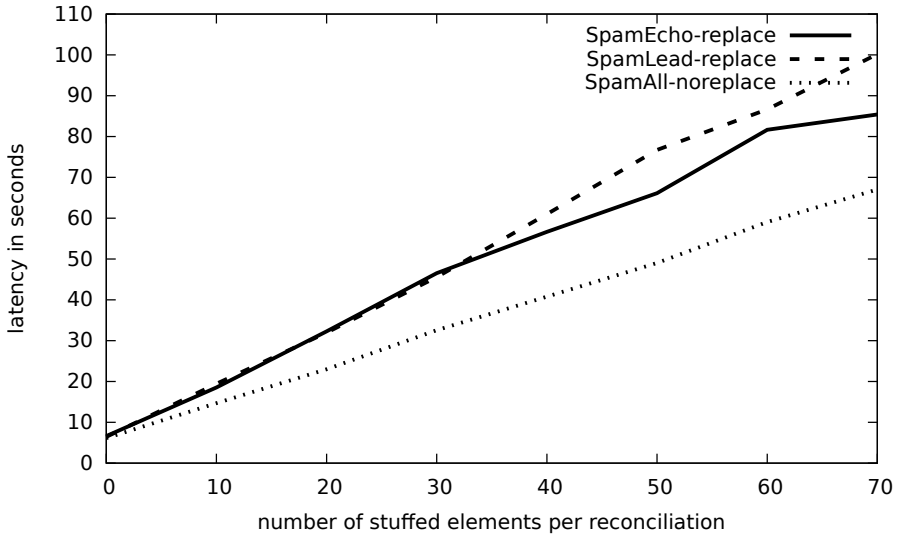
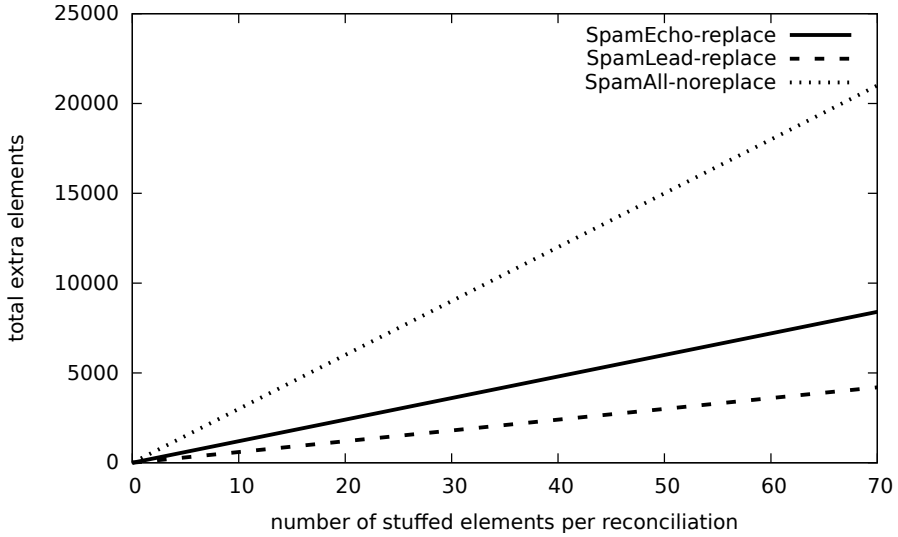


Figure 3.6: Total number of extra elements received by each peer for consensus on 100 elements of 64 bytes and one malicious peer with the indicated mode. Average over five executions.



3.5 Optimizations and Future Work

We now discuss possible future work, some of which concerns limitations of the current implementation.

3.5.1 Fast Dissemination

Recall that in order to be included in the final set, an element must be sent to at least $t+1$ peers, so that at least one correct peer will receive the element. In applications of set union consensus such as electronic voting, the effort to the client should be minimized, and it is thus in practice elements will be sent only to $t+1$ peers, which leads to large initial symmetric differences between peers.

A possible optimization would be to add another dissemination round that only requires $n \log_2 n$ reconciliations to achieve perfect element distribution when only correct peers are present. The n^2 reconciliations that follow will consequently be more efficient, since no difference has to be reconciled when all peers are correct. In the presence of faulty peers, the optimization adds more overhead through the additional dissemination round.

More concretely, in the additional dissemination round the peers reconcile with their 2^ℓ-th neighbour (for some arbitrary, fixed order on the peers) in the ℓ-th subround of the dissemination round. After $\lceil \log_2 \rceil$ of these subrounds, the elements are perfectly distributed as long as every peer passed along their current set correctly.

3.5.2 Extension to Partial Synchrony

The prototype used in the evaluation only works in the synchronous model. It would be trivial to extend it to the partially synchronous model with synchronous clocks by using the same construction as BPFT [13], namely retrying the protocol with larger round timeouts (usually doubled on each retry) when it did not succeed.

It might be worthwhile to further investigate the Byzantine round synchronization protocols discovered independently by Attya and Dolev [5] as well as Dwork, Lynch and Stockmeyer [22]. Running a Byzantine clock synchronization protocol interleaved with consensus protocol might lead to a protocol with lower latency, since the timeouts are dynamically adjusted instead of being increased for each failed protocol run.

3.5.3 Set Canonicalization

A faulty leader in can add or remove elements from the sets, thus increasing the set difference and making the reconciliation less efficient.

An upper bound on the number of added elements can be obtained by using an appropriate canonicalization function. The prototype used for the evaluation does not yet support canonicalization functions, but our evaluation showed that it is critical for performance to bound the extra elements added by the adversary particularly in the leader round.

We assume that the adversary has a fixed number k of elements that are known to the adversary, do not belong to any initial set of a correct peer and would not be removed by the canonicalization function.

It is possible to limit the number of elements that a peer corrupted by the adversary can elide when that peer is the current gradecast leader.

An additional simple gradecast consensus is added before after the all-to-all round, in which the peers agree on a vector that contains the size of every peer's candidate set.

The $\lfloor n/2 \rfloor$-smallest element of that vector is then used as a threshold for the size of the leader set, leader sets smaller than the threshold are ignored and not echoed by honest peers.

The adversary could distribute the k stuffable elements so that every excluded leader is a correct peer. The validity property is still preserved, since all non-stuffable elements were distributed in the all-to-all round, and are thus guaranteed to end up in the output set, since they will have at least a $\lfloor n/2 \rfloor + 1$ majority.

It is not possible to guarantee k as an upper bound on the set difference for a leader, This will significantly reduce wasted traffic and memory usage that is amplified by the

echoing of the leader's set.

During echo / confirm, the adversary could still cause more wasted traffic / storage, but that is without the amplification factor caused by the re-broadcasting.

3.5.4 Persistent Data Structures

Both the Set and Consensus service have to store many variations of the same set when faulty peers elide or add elements. While the Set service API already supports lazy copying, the underlying implementation is inefficient and based on a log of changes per set element with an associated version number. It might be possible to reduce memory usage and increase performance of the set element storage by using data structures that are more well suited, such as the persistent data structures described by Okasaki [48].

Chapter 4

Application to Secure Multiparty Computation

Secure Multiparty Computation (SMC) is an area of cryptography that is concerned with protocols that allow a group of peers $\mathcal{P} = P_1, \ldots, P_n$ to jointly compute a function $y = f(x_1, \ldots, x_n)$ over private input values $x_1, \ldots, x_n$ without using a trusted third party [32]. Each peer P_i contributes its own input value x_i, and during the course of the SMC protocol only learns the output y, but no information about the other peers' input values. Practical applications of SMC include electronic voting, secure auctions and privacy-preserving data mining.

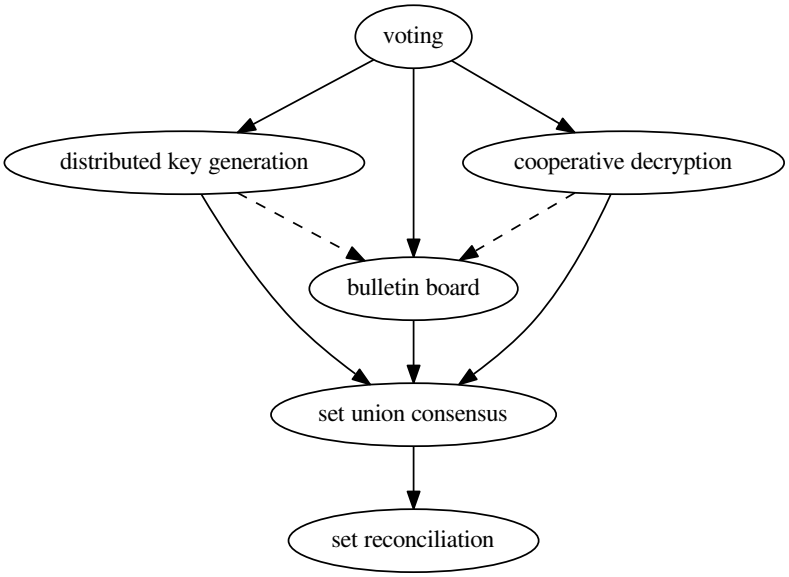
SMC protocols assume a threshold $t < n$ on the amount of peers controlled by an adversary, which is typically either *honest-but-curious* (i.e. tries to learn as much information as possible but follows the protocol) or *actively malicious*. In the actively malicious case, the common definition of SMC mandates the availability of Byzantine consensus as a building block [55].[1]

In practical applications, the inputs typically consist of sets of values that were given to the peers $\mathcal{P}$ by external clients: In electronic voting protocols the peers need to agree on the set of votes; with secure auctions, the peers need to agree on bids, and so on.

In this chapter, we focus on one practical problem, namely electronic voting. We show how set-valued Byzantine consensus is used at multiple stages of the protocol, and discuss how our approach differs from existing solutions found in the literature. The scheme was previously implemented in GNUnet without Byzantine agreement [19].

[1] An attempt has been made to relax the definition of SMC to alleviate this requirement, resulting in a weaker definition that includes non-unanimous *aborts* as a possible result [32]. This definition is mainly useful in scenarios without an honest 2/3 majority, where Byzantine consensus is not possible in the asynchronous model [22].

Figure 4.1: Relation of different SMC protocols and communication primitives in GNUnet. Dashed arrows indicate optional dependencies.



4.1 Bulletin Board for Electronic Voting

The *Bulletin Board* is communication abstraction commonly used for electronic voting [7, 51]. It is a stateful, append only channel that participants of the election can post messages to. Participants of the election identify themselves with a public signing key and must sign all messages that they post to the bulletin board. The posted messages are publicly available to facilitate independent auditing of elections.

Existing work on electronic voting either does not provide a Byzantine fault-tolerant bulletin board in the first place [2] and instead relies on trusted third parties or suggests the use of state machine replication [15].

Some of the bulletin board protocols surveyed by Peters [51] additionally use threshold signatures to certify to the voter that the vote was accepted by a sufficiently large fractions of the peers that jointly provide the bulletin board service. With a naive approach, the message that certifies acceptance by t peers is the concatenation of the peers' individual signatures and thus $O(t)$ bits large. Threshold signature schemes allow smaller signatures, but at the expense of a more complex protocol. Since the number of peers is typically not very large, a linear growth in t is acceptable makes the simple scheme sufficient for practical implementations.

It is easy to implement a variant of the bulletin board with set union consensus. In contrast to traditional bulletin boards, this variant has *phases*, where posted messages are only visible after the group of peers have agreed that a phase is concluded. The concept of phases maps well to the requirements of existing voting protocols. Every phase is implemented with one set union consensus execution. To guarantee that a message is posted to the bulletin board, it must be sent to at least one correct peer from the group of peers that jointly implements the bulletin board.

4.2 Distributed Threshold Key Generation and Cooperative Decryption

Voting schemes as well as other secure multiparty computation protocols often rely on threshold cryptography [17]. The basic intuition behind threshold cryptography is that some operations (such as signing a message or decrypting a ciphertext) should only succeed if a large enough fraction of some group of peers agrees that the operation should be executed.

Typically the public key of the threshold cryptosystem is publicly known, while the private key is not known by any entity but reconstructible from the shares that are distributed among the participants, for example with Shamir's secret sharing scheme [58].

Generating this shared secret key either requires a trusted third party, (which is undesirable for most practical applications since it creates a single point of failure) or a protocol for distributed key generation [30, 50]. Typical distributed key generation protocols require the peers to agree on a number of *pre-shares*, where every peer contributes a number of pre-shares. After the pre-shares are agreed upon, they are re-combined in different ways by each peer respectively, yielding the shares of the private threshold key.

In the key generation protocol used for the Cramer et al. voting scheme, the number of pre-shares that need to be agreed upon is quadratic in the number of peers. Every peer needs to know every pre-share, even if it is not required by the individual peer for reconstructing the share, since the pre-shares are usually accompanied by non-interactive proofs of correctness.

Thus the number of values that needs to be agreed upon is quadratic in the number of peers, which makes the use of set union consensus attractive compared to individual agreement.

Even though the pre-shares can be checked for correctness, Byzantine consensus on the set of shares is still necessary for the case when a malicious peer submits a incorrect

share to only some peers. Without Byzantine consensus, different correct recipients might exclude different peers, resulting in inconsistent shares.

Similarly, when a message that was encrypted with the threshold public key shall be decryped, every peer contributes a *partial decryption* with a proof of correctness. While the set of partial decryptions is typically linear in the number of peers, set union consensus is still a reasonable choice here, the whole system only needs one agreement primitive.

4.3 Electronic Voting with Homomorphic Encryption

While various conceptually different voting schemes use homomorphic encryption, we look as the scheme by Cramer et al. [15] as a modern and practical representative. A fundamental mechanism of the voting scheme is that a set of voting authorities $A_1, \ldots, A_n$ establish a *threshold* key pair that allows any entity that knows the public part of the key to encrypt a message that can only be decrypted when a threshold of the voting authorities cooperate. The homomorphism in the cryptosystem enables the computation of an encrypted tally with only the ciphertext of the submitted votes. Votes represent a choice of one candidate from a list of candidate options. The validity of encrypted votes is ensured by equipping them with a non-interactive zero-knowledge proof of their validity.

It is assumed that the adversary is not able to corrupt more than 1/3 of the authorities. The voting process itself is then facilitated by all voters encrypting their vote and submitting it to the authorities.

The encrypted tally is computed by every authority and then cooperatively decrypted by the authorities and published. Since correct authorities will only agree to decrypt the final tally and not individual votes, the anonymity of the voter is preserved.

For the voting scheme to work correctly, all correct peers must agree on exactly the same set of ballots before the cooperative decryption process starts, otherwise the decryption of the tally will fail.

Chapter 5

Conclusion and Future Work

We have shown that set union consensus is a versatile primitive that can be used as the sole communication primitive for different secure multiparty computation protocols. We have also given performance characteristics of our implementation.

In future work, we would like to apply the idea of set union consensus to Byzantine consensus protocols that are more efficient than the simple gradecast consensus. We also would like to give a comparison between our implementation and a concrete implementation of a bulletin board for remote electronic voting that uses replicated state machines. Promising candidates for comparison that have a publicly available and actively maintained implementation are BFT-SMaRt [9] and ARCHISTAR [1].

[1] `http://bft-smart.github.io/library/`, `https://github.com/Archistar/archistar-core`

Appendix A

Appendix

In the following, we give the documented interface for the C programming language to GNUnet's Set and Consensus service.

A.1 Set API Reference

```
/**
 * Opaque handle to a set.
 */
struct GNUNET_SET_Handle;

/**
 * Opaque handle to a set operation request from another peer.
 */
struct GNUNET_SET_Request;

/**
 * Opaque handle to a listen operation.
 */
struct GNUNET_SET_ListenHandle;

/**
 * Opaque handle to a set operation.
 */
struct GNUNET_SET_OperationHandle;


/**
```

```
 * The operation that a set set supports.
 */
enum GNUNET_SET_OperationType
{
  /**
   * A purely local set that does not support any operation.
   */
  GNUNET_SET_OPERATION_NONE,

  /**
   * Set intersection, only return elements that are in both sets.
   */
  GNUNET_SET_OPERATION_INTERSECTION,

  /**
   * Set union, return all elements that are in at least one of the sets.
   */
  GNUNET_SET_OPERATION_UNION
};


/**
 * Status for the result callback
 */
enum GNUNET_SET_Status
{
  /**
   * Everything went ok, we are transmitting an element of the
   * result (in set, or to be removed from set, depending on
   * the `enum GNUNET_SET_ResultMode').
   *
   * Only applies to
   * #GNUNET_SET_RESULT_FULL,
   * #GNUNET_SET_RESULT_ADDED,
   * #GNUNET_SET_RESULT_REMOVED,
   */
  GNUNET_SET_STATUS_OK,

  /**
   * Element should be added to the result set
   * of the local peer, i.e. the local peer is
```

```
   * missing an element.
   *
   * Only applies to #GNUNET_SET_RESULT_SYMMETRIC
   */
  GNUNET_SET_STATUS_ADD_LOCAL,

  /**
   * Element should be added to the result set
   * of the remove peer, i.e. the remote peer is
   * missing an element.
   *
   * Only applies to #GNUNET_SET_RESULT_SYMMETRIC
   */
  GNUNET_SET_STATUS_ADD_REMOTE,

  /**
   * The other peer refused to to the operation with us,
   * or something went wrong.
   */
  GNUNET_SET_STATUS_FAILURE,

  /**
   * Success, all elements have been returned (but the other peer
   * might still be receiving some from us, so we are not done).  Only
   * used during UNION operation.
   */
  GNUNET_SET_STATUS_HALF_DONE,

  /**
   * Success, all elements have been sent (and received).
   */
  GNUNET_SET_STATUS_DONE
};


/**
 * The way results are given to the client.
 */
enum GNUNET_SET_ResultMode
{
  /**
```

```
   * Client gets every element in the resulting set.
   *
   * Only supported for set intersection.
   */
  GNUNET_SET_RESULT_FULL,

  /**
   * Client gets notified of the required changes
   * for both the local and the remote set.
   *
   * Only supported for set
   */
  GNUNET_SET_RESULT_SYMMETRIC,

  /**
   * Client gets only elements that have been removed from the set.
   *
   * Only supported for set intersection.
   */
  GNUNET_SET_RESULT_REMOVED,

  /**
   * Client gets only elements that have been removed from the set.
   *
   * Only supported for set union.
   */
  GNUNET_SET_RESULT_ADDED
};


/**
 * Element stored in a set.
 */
struct GNUNET_SET_Element
{
  /**
   * Number of bytes in the buffer pointed to by data.
   */
  uint16_t size;

  /**
```

```
   * Application-specific element type.
   */
  uint16_t element_type;

  /**
   * Actual data of the element
   */
  const void *data;
};


/**
 * Continuation used for some of the set operations
 *
 * @param cls closure
 */
typedef void (*GNUNET_SET_Continuation) (void *cls);


/**
 * Callback for set operation results. Called for each element
 * in the result set.
 *
 * @param cls closure
 * @param element a result element, only valid if status is #GNUNET_SET_STATUS_OK
 * @param status see `enum GNUNET_SET_Status`
 */
typedef void (*GNUNET_SET_ResultIterator) (void *cls,
                                           const struct GNUNET_SET_Element *element,
                                           enum GNUNET_SET_Status status);

/**
 * Iterator for set elements.
 *
 * @param cls closure
 * @param element the current element, NULL if all elements have been
 *        iterated over
 * @return #GNUNET_YES to continue iterating, #GNUNET_NO to stop.
 */
typedef int (*GNUNET_SET_ElementIterator) (void *cls,
                                           const struct GNUNET_SET_Element *element);
```

```
/**
 * Called when another peer wants to do a set operation with the
 * local peer. If a listen error occurs, the @a request is NULL.
 *
 * @param cls closure
 * @param other_peer the other peer
 * @param context_msg message with application specific information from
 *        the other peer
 * @param request request from the other peer (never NULL), use GNUNET_SET_accept()
 *        to accept it, otherwise the request will be refused
 *        Note that we can't just return value from the listen callback,
 *        as it is also necessary to specify the set we want to do the
 *        operation with, whith sometimes can be derived from the context
 *        message. It's necessary to specify the timeout.
 */
typedef void
(*GNUNET_SET_ListenCallback) (void *cls,
                              const struct GNUNET_PeerIdentity *other_peer,
                              const struct GNUNET_MessageHeader *context_msg,
                              struct GNUNET_SET_Request *request);



typedef void
(*GNUNET_SET_CopyReadyCallback) (void *cls,
                                 struct GNUNET_SET_Handle *copy);


/**
 * Create an empty set, supporting the specified operation.
 *
 * @param cfg configuration to use for connecting to the
 *        set service
 * @param op operation supported by the set
 *        Note that the operation has to be specified
 *        beforehand, as certain set operations need to maintain
 *        data structures spefific to the operation
 * @return a handle to the set
 */
```

```
struct GNUNET_SET_Handle *
GNUNET_SET_create (const struct GNUNET_CONFIGURATION_Handle *cfg,
                   enum GNUNET_SET_OperationType op);


/**
 * Add an element to the given set.
 * After the element has been added (in the sense of being
 * transmitted to the set service), @a cont will be called.
 * Calls to #GNUNET_SET_add_element can be queued
 *
 * @param set set to add element to
 * @param element element to add to the set
 * @param cont continuation called after the element has been added
 * @param cont_cls closure for @a cont
 * @return #GNUNET_OK on success, #GNUNET_SYSERR if the
 *         set is invalid (e.g. the set service crashed)
 */
int
GNUNET_SET_add_element (struct GNUNET_SET_Handle *set,
                        const struct GNUNET_SET_Element *element,
                        GNUNET_SET_Continuation cont,
                        void *cont_cls);


/**
 * Remove an element to the given set.
 * After the element has been removed (in the sense of the
 * request being transmitted to the set service), cont will be called.
 * Calls to remove_element can be queued
 *
 * @param set set to remove element from
 * @param element element to remove from the set
 * @param cont continuation called after the element has been removed
 * @param cont_cls closure for @a cont
 * @return #GNUNET_OK on success, #GNUNET_SYSERR if the
 *         set is invalid (e.g. the set service crashed)
 */
int
GNUNET_SET_remove_element (struct GNUNET_SET_Handle *set,
                           const struct GNUNET_SET_Element *element,
```

```
                          GNUNET_SET_Continuation cont,
                          void *cont_cls);


void
GNUNET_SET_copy_lazy (struct GNUNET_SET_Handle *set,
                      GNUNET_SET_CopyReadyCallback cb,
                      void *cls);


/**
 * Destroy the set handle, and free all associated resources.
 * Iterations must have completed (or be explicitly canceled)
 * before destroying the corresponding set.  Operations may
 * still be pending when a set is destroyed.
 *
 * @param set set to destroy
 */
void
GNUNET_SET_destroy (struct GNUNET_SET_Handle *set);


/**
 * Prepare a set operation to be evaluated with another peer.
 * The evaluation will not start until the client provides
 * a local set with GNUNET_SET_commit().
 *
 * @param other_peer peer with the other set
 * @param app_id hash for the application using the set
 * @param context_msg additional information for the request
 * @param result_mode specified how results will be returned,
 *        see `enum GNUNET_SET_ResultMode`.
 * @param result_cb called on error or success
 * @param result_cls closure for @a result_cb
 * @return a handle to cancel the operation
 */
struct GNUNET_SET_OperationHandle *
GNUNET_SET_prepare (const struct GNUNET_PeerIdentity *other_peer,
                    const struct GNUNET_HashCode *app_id,
                    const struct GNUNET_MessageHeader *context_msg,
                    enum GNUNET_SET_ResultMode result_mode,
```

```
                    GNUNET_SET_ResultIterator result_cb,
                    void *result_cls);


/**
 * Wait for set operation requests for the given application ID.
 * If the connection to the set service is lost, the listener is
 * re-created transparently with exponential backoff.
 *
 * @param cfg configuration to use for connecting to
 *            the set service
 * @param operation operation we want to listen for
 * @param app_id id of the application that handles set operation requests
 * @param listen_cb called for each incoming request matching the operation
 *                  and application id
 * @param listen_cls handle for @a listen_cb
 * @return a handle that can be used to cancel the listen operation
 */
struct GNUNET_SET_ListenHandle *
GNUNET_SET_listen (const struct GNUNET_CONFIGURATION_Handle *cfg,
                   enum GNUNET_SET_OperationType op_type,
                   const struct GNUNET_HashCode *app_id,
                   GNUNET_SET_ListenCallback listen_cb,
                   void *listen_cls);


/**
 * Cancel the given listen operation.  After calling cancel, the
 * listen callback for this listen handle will not be called again.
 *
 * @param lh handle for the listen operation
 */
void
GNUNET_SET_listen_cancel (struct GNUNET_SET_ListenHandle *lh);


/**
 * Accept a request we got via GNUNET_SET_listen().  Must be called during
 * GNUNET_SET_listen(), as the `struct GNUNET_SET_Request` becomes invalid
 * afterwards.
 * Call GNUNET_SET_commit() to provide the local set to use for the operation,
```

```
 * and to begin the exchange with the remote peer.
 *
 * @param request request to accept
 * @param result_mode specified how results will be returned,
 *        see `enum GNUNET_SET_ResultMode`.
 * @param result_cb callback for the results
 * @param result_cls closure for @a result_cb
 * @return a handle to cancel the operation
 */
struct GNUNET_SET_OperationHandle *
GNUNET_SET_accept (struct GNUNET_SET_Request *request,
                   enum GNUNET_SET_ResultMode result_mode,
                   GNUNET_SET_ResultIterator result_cb,
                   void *result_cls);


/**
 * Commit a set to be used with a set operation.
 * This function is called once we have fully constructed
 * the set that we want to use for the operation.  At this
 * time, the P2P protocol can then begin to exchange the
 * set information and call the result callback with the
 * result information.
 *
 * @param oh handle to the set operation
 * @param set the set to use for the operation
 * @return #GNUNET_OK on success, #GNUNET_SYSERR if the
 *         set is invalid (e.g. the set service crashed)
 */
int
GNUNET_SET_commit (struct GNUNET_SET_OperationHandle *oh,
                   struct GNUNET_SET_Handle *set);


/**
 * Cancel the given set operation.  May not be called after the
 * operation's `GNUNET_SET_ResultIterator` has been called with a
 * status that indicates error, timeout or done.
 *
 * @param oh set operation to cancel
 */
```

```
void
GNUNET_SET_operation_cancel (struct GNUNET_SET_OperationHandle *oh);


/**
 * Iterate over all elements in the given set.
 * Note that this operation involves transferring every element of the set
 * from the service to the client, and is thus costly.
 * Only one iteration per set may be active at the same time.
 *
 * @param set the set to iterate over
 * @param iter the iterator to call for each element
 * @param iter_cls closure for @a iter
 * @return #GNUNET_YES if the iteration started successfuly,
 *         #GNUNET_NO if another iteration was still active,
 *         #GNUNET_SYSERR if the set is invalid (e.g. the server crashed,
           disconnected)
 */
int
GNUNET_SET_iterate (struct GNUNET_SET_Handle *set,
                    GNUNET_SET_ElementIterator iter,
                    void *iter_cls);

/**
 * Stop iteration over all elements in the given set.  Can only
 * be called before the iteration has "naturally" completed its
 * turn.
 *
 * @param set the set to stop iterating over
 */
void
GNUNET_SET_iterate_cancel (struct GNUNET_SET_Handle *set);

/**
 * Create a copy of an element.  The copy
 * must be GNUNET_free-d by the caller.
 *
 * @param element the element to copy
 * @return the copied element
 */
struct GNUNET_SET_Element *
```

```
GNUNET_SET_element_dup (const struct GNUNET_SET_Element *element);

/**
 * Hash a set element.
 *
 * @param element the element that should be hashed
 * @param ret_hash a pointer to where the hash of @a element
 *        should be stored
 */
void
GNUNET_SET_element_hash (const struct GNUNET_SET_Element *element,
                         struct GNUNET_HashCode *ret_hash);
```

A.2 Consensus API Reference

```
/**
 * Called when a new element was received from another peer, or an error occured.
 * May deliver duplicate values.
 * Elements given to a consensus operation by the local peer are NOT given
 * to this callback.
 *
 * @param cls closure
 * @param element new element, NULL on error
 */
typedef void (*GNUNET_CONSENSUS_ElementCallback) (void *cls,
                                                  const struct GNUNET_SET_Element *element);



/**
 * Opaque handle for the consensus service.
 */
struct GNUNET_CONSENSUS_Handle;


/**
 * Create a consensus session.  The set being reconciled is initially
 * empty.
 *
 * @param cfg
 * @param num_peers
```

```
 * @param peers array of peers participating in this consensus session
 *              Inclusion of the local peer is optional.
 * @param session_id session identifier
 *                  Allows a group of peers to have more than consensus session.
 * @param start start time of the consensus, conclude should be called before
 *              the start time.
 * @param deadline time when the consensus should have concluded
 * @param new_element_cb callback, called when a new element is added to the set by
 *                   another peer. Also called when an error occurs.
 * @param new_element_cls closure for new_element
 * @return handle to use, NULL on error
 */
struct GNUNET_CONSENSUS_Handle *
GNUNET_CONSENSUS_create (const struct GNUNET_CONFIGURATION_Handle *cfg,
                         unsigned int num_peers,
                         const struct GNUNET_PeerIdentity *peers,
                         const struct GNUNET_HashCode *session_id,
                         struct GNUNET_TIME_Absolute start,
                         struct GNUNET_TIME_Absolute deadline,
                         GNUNET_CONSENSUS_ElementCallback new_element_cb,
                         void *new_element_cls);


/**
 * Called when an insertion (transmission to consensus service, which
 * does not imply fully consensus on this element with all other
 * peers) was successful.  May not call GNUNET_CONSENSUS_destroy();
 * schedule a task to call GNUNET_CONSENSUS_destroy() instead (if
 * needed).
 *
 * @param cls
 * @param success #GNUNET_OK on success, #GNUNET_SYSERR if
 *        the insertion and thus the consensus failed for good
 */
typedef void (*GNUNET_CONSENSUS_InsertDoneCallback) (void *cls,
                                                     int success);


/**
 * Insert an element in the set being reconsiled.  Only transmit changes to
 * other peers if GNUNET_CONSENSUS_begin() has been called.
```

```
 * Must not be called after GNUNET_CONSENSUS_conclude().
 * May not call GNUNET_CONSENSUS_destroy(); schedule a task to call
 * GNUNET_CONSENSUS_destroy() instead (if needed).
 *
 * @param consensus handle for the consensus session
 * @param element the element to be inserted
 * @param idc function called when we are done with this element and it
 *            is thus allowed to call GNUNET_CONSENSUS_insert() again
 * @param idc_cls closure for @a idc
 */
void
GNUNET_CONSENSUS_insert (struct GNUNET_CONSENSUS_Handle *consensus,
                         const struct GNUNET_SET_Element *element,
                         GNUNET_CONSENSUS_InsertDoneCallback idc,
                         void *idc_cls);



/**
 * Called when a conclusion was successful.
 *
 * @param cls
 */
typedef void (*GNUNET_CONSENSUS_ConcludeCallback) (void *cls);


/**
 * We are finished inserting new elements into the consensus;
 * try to conclude the consensus within a given time window.
 *
 * @param consensus consensus session
 * @param conclude called when the conclusion was successful
 * @param conclude_cls closure for the conclude callback
 */
void
GNUNET_CONSENSUS_conclude (struct GNUNET_CONSENSUS_Handle *consensus,
                           GNUNET_CONSENSUS_ConcludeCallback conclude,
                           void *conclude_cls);


/**
```

```
 * Destroy a consensus handle (free all state associated with
 * it, no longer call any of the callbacks).
 *
 * @param consensus handle to destroy
 */
void
GNUNET_CONSENSUS_destroy (struct GNUNET_CONSENSUS_Handle *consensus);
```

Bibliography

[1] Michael Abd-El-Malek et al. “Fault-scalable Byzantine fault-tolerant services”. In: *ACM SIGOPS Operating Systems Review* 39.5 (2005), pp. 59–74 (cit. on p. 5).

[2] Ben Adida. “Helios: Web-based Open-Audit Voting.” In: *USENIX Security Symposium*. Vol. 17. 2008, pp. 335–348 (cit. on p. 32).

[3] Marcos K Aguilera. “Stumbling over consensus research: Misunderstandings and issues”. In: *Replication*. Springer, 2010, pp. 59–72 (cit. on p. 3).

[4] James Aspnes. “Lower bounds for distributed coin-flipping and randomized consensus”. In: *Journal of the ACM (JACM)* 45.3 (1998), pp. 415–450 (cit. on p. 18).

[5] Chagit Attiya, Danny Dolev, and Joseph Gil. “Asynchronous byzantine consensus”. In: *Proceedings of the third annual ACM symposium on Principles of distributed computing*. ACM. 1984, pp. 119–133 (cit. on p. 29).

[6] Pierre-Louis Aublin et al. “The Next 700 BFT Protocols”. In: *ACM Trans. Comput. Syst.* 32.4 (Jan. 2015), 12:1–12:45. ISSN: 0734-2071. DOI: 10.1145/2658994. URL: http://doi.acm.org/10.1145/2658994 (cit. on pp. 4, 5).

[7] Josh Daniel Cohen Benaloh. *Verifiable secret-ballot elections*. Yale University. Department of Computer Science, 1987 (cit. on p. 32).

[8] Michael Ben-Or, Danny Dolev, and Ezra N Hoch. “Simple gradecast based algorithms”. In: *arXiv preprint arXiv:1007.1049* (2010) (cit. on pp. 6, 17–20).

[9] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. “State machine replication for the masses with BFT-SMaRt”. In: *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE. 2014, pp. 355–362 (cit. on p. 35).

[10] Burton H Bloom. “Space/time trade-offs in hash coding with allowable errors”. In: *Communications of the ACM* 13.7 (1970), pp. 422–426 (cit. on p. 7).

[11] Christian Cachin, Klaus Kursawe, and Victor Shoup. “Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography”. In: *Journal of Cryptology* 18.3 (2005), pp. 219–246 (cit. on p. 18).

[12] Miguel Castro and Barbara Liskov. “Practical Byzantine fault tolerance and proactive recovery”. In: *ACM Transactions on Computer Systems (TOCS)* 20.4 (2002), pp. 398–461 (cit. on pp. 3, 4, 17).

[13] Miguel Castro, Barbara Liskov, et al. “Practical Byzantine fault tolerance”. In: *OSDI*. Vol. 99. 1999, pp. 173–186 (cit. on pp. 3, 4, 17, 29).

[14] Allen Clement et al. “Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults.” In: *NSDI*. Vol. 9. 2009, pp. 153–168 (cit. on p. 5).

[15] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. “A secure and optimally efficient multi-authority election scheme”. In: *European transactions on Telecommunications* 8.5 (1997), pp. 481–490 (cit. on pp. 6, 32, 34).

[16] Roberto De Prisco, Dahlia Malkhi, and Michael Reiter. “On k-set consensus problems in asynchronous systems”. In: *Parallel and Distributed Systems, IEEE Transactions on* 12.1 (2001), pp. 7–21 (cit. on p. 2).

[17] Yvo G Desmedt. “Threshold cryptography”. In: *European Transactions on Telecommunications* 5.4 (1994), pp. 449–458 (cit. on p. 33).

[18] L Peter Deutsch. “GZIP file format specification version 4.3”. In: (1996) (cit. on p. 12).

[19] Florian Dold. “Cryptographically Secure, Distributed Electronic Voting”. Bachelor’s Thesis. Technische Universität München, 2014 (cit. on pp. 12, 31).

[20] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. “On the minimal synchronism needed for distributed consensus”. In: *Journal of the ACM (JACM)* 34.1 (1987), pp. 77–97 (cit. on p. 3).

[21] Danny Dolev, Ruediger Reischuk, and H. Raymond Strong. “Early Stopping in Byzantine Agreement”. In: *J. ACM* 37.4 (Oct. 1990), pp. 720–741. ISSN: 0004-5411. DOI: 10.1145/96559.96565. URL: http://doi.acm.org/10.1145/96559.96565 (cit. on p. 20).

[22] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. “Consensus in the presence of partial synchrony”. In: *Journal of the ACM (JACM)* 35.2 (1988), pp. 288–323 (cit. on pp. 1, 3, 4, 29, 31).

[23] David Eppstein et al. “What’s the difference?: efficient set reconciliation without prior context”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 41. 4. ACM. 2011, pp. 218–229 (cit. on pp. 6–10).

[24] Paul Neil Feldman. “Optimal algorithms for Byzantine agreement”. PhD thesis. Massachusetts Institute of Technology, 1988 (cit. on pp. 3, 22).

[25] Paul Feldman and Silvio Micali. “Optimal algorithms for Byzantine agreement”. In: *Proceedings of the twentieth annual ACM symposium on Theory of computing*. ACM. 1988, pp. 148–161 (cit. on pp. 3, 17–19, 22).

[26] Michael J Fischer and Nancy A Lynch. *A lower bound for the time to assure interactive consistency*. Tech. rep. DTIC Document, 1981 (cit. on p. 2).

[27] Michael J Fischer, Nancy A Lynch, and Michael Merritt. “Easy impossibility proofs for distributed consensus problems”. In: *Distributed Computing* 1.1 (1986), pp. 26–39 (cit. on p. 2).

[28] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. “Impossibility of distributed consensus with one faulty process”. In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382 (cit. on p. 2).

[29] Matthias Fitzi and Martin Hirt. "Optimally efficient multi-valued byzantine agreement". In: *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*. ACM. 2006, pp. 163–168 (cit. on p. 3).

[30] Pierre-Alain Fouque and Jacques Stern. "One round threshold discrete-log key generation without private channels". In: *Public Key Cryptography*. Springer. 2001, pp. 300–316 (cit. on pp. 12, 33).

[31] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. "The bitcoin backbone protocol: Analysis and applications". In: *Advances in Cryptology-EUROCRYPT 2015*. Springer, 2015, pp. 281–310 (cit. on p. 5).

[32] Shafi Goldwasser and Yehuda Lindell. "Secure multi-party computation without agreement". In: *Journal of Cryptology* 18.3 (2005), pp. 247–287 (cit. on p. 31).

[33] Michael T Goodrich and Michael Mitzenmacher. "Invertible bloom lookup tables". In: *Communication, Control, and Computing (Allerton), 2011 49th Annual Allerton Conference on*. IEEE. 2011, pp. 792–799 (cit. on p. 8).

[34] Rachid Guerraoui et al. "Consensus in asynchronous distributed systems: A concise guided tour". In: *Advances in Distributed Systems*. Springer, 2000, pp. 33–47 (cit. on pp. 3, 18).

[35] Kim Potter Kihlstrom, Louise E Moser, and P Michael Melliar-Smith. "The SecureRing protocols for securing group communication". In: *System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on*. Vol. 3. IEEE. 1998, pp. 317–326 (cit. on pp. 3, 4).

[36] Ramakrishna Kotla et al. "Zyzzyva: speculative byzantine fault tolerance". In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 6. ACM. 2007, pp. 45–58 (cit. on p. 5).

[37] Leslie Lamport. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25 (cit. on p. 4).

[38] Leslie Lamport. "The part-time parliament". In: *ACM Transactions on Computer Systems (TOCS)* 16.2 (1998), pp. 133–169 (cit. on p. 4).

[39] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine generals problem". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 382–401 (cit. on pp. 1, 4).

[40] Ping Li and Arnd Christian König. "Theory and applications of b-bit minwise hashing". In: *Communications of the ACM* 54.8 (2011), pp. 101–109 (cit. on p. 16).

[41] Navneet Malpani, Jennifer L Welch, and Nitin Vaidya. "Leader election algorithms for mobile ad hoc networks". In: *Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications*. ACM. 2000, pp. 96–103 (cit. on p. 2).

[42] Andrew Miller and Joseph J LaViola Jr. "Anonymous byzantine consensus from moderately-hard puzzles: A model for bitcoin". In: *Retrieved from Anonymous Byzantine Consensus from Moderately-Hard Puzzles: A Model for Bitcoin* (2014) (cit. on p. 5).

[43] Yaron Minsky, Ari Trachtenberg, and Richard Zippel. "Set reconciliation with nearly optimal communication complexity". In: *Information Theory, IEEE Transactions on* 49.9 (2003), pp. 2213–2218 (cit. on p. 7).

[44] Michael Mitzenmacher and Rasmus Pagh. "Simple Multi-Party Set Reconciliation". In: *arXiv preprint arXiv:1311.2037* (2013) (cit. on p. 8).

[45] Achour Mostefaoui, Hamouma Moumen, and Michel Raynal. "Signature-free asynchronous Byzantine consensus with t< n/3 and O (n 2) messages". In: *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. ACM. 2014, pp. 2–9 (cit. on pp. 3, 18).

[46] Satoshi Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: *Consulted* 1.2012 (2008), p. 28 (cit. on p. 5).

[47] Gil Neiger. "Distributed consensus revisited". In: *Information Processing Letters* 49.4 (1994), pp. 195–201 (cit. on p. 2).

[48] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999 (cit. on pp. 11, 30).

[49] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm". In: *Proc. USENIX Annual Technical Conference*. 2014, pp. 305–320 (cit. on p. 4).

[50] Torben Pryds Pedersen. "A threshold cryptosystem without a trusted party". In: *Advances in Cryptology—EUROCRYPT'91*. Springer. 1991, pp. 522–526 (cit. on p. 33).

[51] RA Peters. "A Secure Bulletin Board". Master's Thesis. Technische Universiteit Eindhoven, 2005 (cit. on p. 32).

[52] Bartlomiej Polot and Christian Grothoff. "Cadet: Confidential ad-hoc decentralized end-to-end transport". In: *Ad Hoc Networking Workshop (MED-HOC-NET), 2014 13th Annual Mediterranean*. IEEE. 2014, pp. 71–78 (cit. on pp. 12, 25).

[53] Michael K Reiter. "The Rampart toolkit for building high-integrity services". In: *Theory and Practice in Distributed Systems*. Springer, 1995, pp. 99–110 (cit. on pp. 3, 4).

[54] Michael Rink. "Mixed Hypergraphs for Linear-Time Construction of Denser Hashing-Based Data Structures". English. In: *SOFSEM 2013: Theory and Practice of Computer Science*. Ed. by Peter van Emde Boas et al. Vol. 7741. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 356–368. ISBN: 9783642358425. DOI: 10.1007/978-3-642-35843-2_31. URL: http://dx.doi.org/10.1007/978-3-642-35843-2_31 (cit. on p. 8).

[55] Jared Saia and Mahdi Zamani. "Recent results in scalable multi-party computation". In: *SOFSEM 2015: Theory and Practice of Computer Science*. Springer, 2015, pp. 24–44 (cit. on p. 31).

[56] Fred B Schneider. "Implementing fault-tolerant services using the state machine approach: A tutorial". In: *ACM Computing Surveys (CSUR)* 22.4 (1990), pp. 299–319 (cit. on p. 3).

[57] David Schwartz, Noah Youngs, and Arthur Britto. “The Ripple protocol consensus algorithm”. In: *Ripple Labs Inc White Paper* (2014) (cit. on p. 5).

[58] Adi Shamir. “How to share a secret”. In: *Communications of the ACM* 22.11 (1979), pp. 612–613 (cit. on p. 33).

[59] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. “Theory and practice of bloom filters for distributed systems”. In: *Communications Surveys & Tutorials, IEEE* 14.1 (2012), pp. 131–155 (cit. on p. 11).

[60] Sree Harsha Totakura. “Large Scale Distributed Evaluation of Peer-to-Peer Protocols”. Masters. Garching bei Muenchen: Technische Universitaet Muenchen, June 2013, p. 76 (cit. on p. 24).

[61] Robbert Van Renesse, Nicolas Schiper, and Fred B Schneider. “Vive la différence: Paxos vs. Viewstamped Replication vs. Zab”. In: (2014) (cit. on p. 4).

[62] Matthias Wachs, Martin Schanzenbach, and Christian Grothoff. “A censorship-resistant, privacy-enhancing and fully decentralized name system”. In: *Cryptology and Network Security*. Springer, 2014, pp. 127–142 (cit. on p. 12).