Fast Asynchronous Consensus with Optimal Resilience

Ittai Abraham, Marcos K. Aguilera, and Dahlia Malkhi
Microsoft Research Silicon Valley

Abstract. We give randomized agreement algorithms with constant expected running time in asynchronous systems subject to process failures, where up to a minority of processes may fail. We consider three types of process failures: crash, omission, and Byzantine. For crash or omission failures, we solve consensus assuming private channels or a public-key infrastructure, respectively. For Byzantine failures, we solve weak Byzantine agreement assuming a public-key infrastructure and a broadcast primitive called weak sequenced broadcast. We show how to obtain weak sequenced broadcast using a minimal trusted platform module. The presented algorithms are simple, have optimal resilience, and have optimal asymptotic running time. They work against a sophisticated adversary that can adaptively schedule messages, processes, and failures based on the messages seen by faulty processes.

1 Introduction

In the consensus problem, each process starts with some initial value and must make an irrevocable decision on one of the initial values, such that all correct processes decide on the same value. The challenge lies in solving consensus in the presence of faulty processes. In this paper we consider asynchronous message-passing systems, where processes communicate by sending messages, and there are no bounds on message delays or on the relative speed of processes. In such a system, consensus cannot be solved [21], but it *can* be solved if processes need to terminate only with probability one [3].

Chor, Merritt, and Shmoys [11] gave an algorithm that solves consensus with probability one in constant expected time, in a system with crash failures and $n \geq \frac{3+\sqrt{5}}{2}f+1 \approx 2.62f+1$, where n is the number of processes and f is the maximum number of faulty processes. Subsequently, Attiya and Welch [2] gave an algorithm based on an observation of Gafni, which works with the optimal resiliency of $n \geq 2f+1$, but the algorithm requires a message-independent adversary—one that acts independently of the content of messages.

Our first result is an algorithm that works in a system with $n \geq 2f+1$ and a rather sophisticated adversary that can adaptively schedule messages, processes, and failures based on the message contents seen by faulty processes. The algorithm relies on private channels. It uses a secret sharing scheme and a simple *binding gather* primitive, explained in Section 4.

Theorem 1 Consensus can be solved by an algorithm with constant expected running time in an asynchronous system with private channels, a sophisticated adversary, and $n \geq 2f+1$ processes, where f processes may crash.

Our result for crash failures can be easily extended to (general) omission failures, assuming the presence of a public-key infrastructure (PKI). Roughly speaking, a PKI provides a public-key cryptographic system in which each process p_i has a secret decryption function D_i and secret signing function S_i, and all processes know the public encryption function E_j and the public one-way signature verification function V_j of all other processes p_j. Except with negligible probability, signatures cannot be forged and encrypted messages cannot be decrypted by a party that does not have the secret signing or decryption functions. A PKI can be implemented under suitable cryptographic assumptions. By using a PKI, we require the adversary to be computationally bounded and our algorithms have a negligible probability of not terminating due to the adversary breaking the PKI.

Theorem 2 Consensus can be securely implemented by an algorithm with constant expected running time in an asynchronous system with a computationally-bounded sophisticated adversary, a public-key infrastructure, and $n \geq 2f+1$ processes, where f processes may experience omission failures.

Our next result concerns Byzantine failures, and it builds upon our result for omission failures. A simple partitioning argument shows that Byzantine agreement (consensus) cannot be solved with $2f+1 \leq n \leq 3f$, even with a PKI. Hence, we augment the system with a primitive called *weak sequenced broadcast*. Roughly speaking, weak sequenced broadcast is a broadcast that ensures that (a) messages from a given sender are delivered by correct processes in the same order; this is an ordering *per-sender*, similar to FIFO broadcast, and (b) if the sender is correct then eventually all processes will receive all its messages. We show one way to implement weak sequenced broadcast by using a minimal trusted platform module, a component which is becoming standard.

Even with weak sequenced broadcast, Byzantine agreement cannot be solved with $2f+1 \leq n \leq 3f$, because correct processes may decide a value that is not one of their initial values, even if their initial values are all the same, violating the Validity property of Byzantine agreement. So we require the Validity property to hold only if all processes are correct, and therefore we solve *weak Byzantine agreement* [26]. The algorithm has constant expected running time in a system with weak sequenced broadcast and $n \geq 2f+1$. It borrows from the approach of Feldman and Micali [18, 19] based on verifiable secret sharing (VSS). Using weak sequenced broadcast, we can implement several building blocks like sequenced broadcast (which we define later) and VSS. We are not aware of other work for asynchronous systems with Byzantine failures and $n \geq 2f+1$.

Theorem 3 Weak Byzantine agreement can be securely implemented by an algorithm with constant expected running time in an asynchronous system with a computationally-bounded sophisticated adversary, a public-key infrastructure, weak sequenced broadcast, and $n \geq 2f+1$ processes, where f processes may be Byzantine.

Weak sequenced broadcast itself may be implemented in many ways, one of which is demonstrated here by using a minimal trusted platform module (TPM). The TPM has a register which can be updated only by invoking guarded functions of the module, and stores a private key which is used for signing the

register value upon request. This system model is useful when one can trust the hardware platform but the system is otherwise vulnerable (say, to malware). But in some other settings, the hardware of some machines may not be trusted. In this case, we can use an external service to aid in implementing weak sequenced broadcast, e.g., using a public certification authority [22] or other means.

All the algorithms we present have optimal resilience and asymptotically optimal (constant) expected running time. They are based on a suitable combination of known techniques as we explain below. Due to space limitations, the proofs of our results are omitted; they will be included in the full version.

Related work. Our work touches one of the most active areas of research in distributed computing, and it is therefore beyond our scope to cover all related work. Below, we cover the most relevant results and we refer the reader to [28, 2] for an extensive treatment.

Byzantine failures. There have been proposals to solve Byzantine agreement in a system with $n \geq 2f+1$ using strong primitives such as an *append-only log* [12] or trusted increment [27]. The append-only log is similar to weak sequenced broadcast. However, these works differ from ours because they consider partially synchronous systems, that is, the liveness of the algorithms is conditional on eventual synchrony. In contrast, we consider a fully asynchronous system and provide randomized algorithms that terminate with probability one (except for a negligible probability that the cryptosystem is broken). Our implementation of weak sequenced broadcast using TPM's is similar to the scheme by [25]. A formal specification of TPM's is given in [31]. The idea of using strong primitives to boost the resilience of Byzantine agreement algorithms has been suggested in the wormhole approach of Correia et al [13]. Our solution for the Byzantine case uses a simple asynchronous VSS protocol for $n \geq 2f+1$, which is based on zero knowledge and weak sequenced broadcast. Our asynchronous VSS protocol is similar to the PVSS protocol of [30] (but they have not considered asynchronous VSS or use weak sequenced broadcast). Other asynchronous VSS protocols work only for $n \geq 3f+1$ [9, 6].

A public-key infrastructure has often been used for reaching agreement. In *synchronous* systems with Byzantine failures, Dolev and Strong [15] show how to solve terminating reliable broadcast for $n \geq f+1$. Katz and Koo [24] show how to solve Byzantine agreement for $n \geq 2f+1$ in constant expected time [24] (also see [17]). Our asynchronous randomized probabilistic agreement primitive is similar to the leader based approach of [24]. In *asynchronous systems*, [8] shows how to solve Byzantine agreement for $n \geq 3f+1$ in constant expected time with asymptotically optimal message complexity.

Without computational assumptions, Byzantine agreement with probability one in an asynchronous system is given by Ben-Or [3] for $n \geq 5f+1$ and by Bracha [5] for $n \geq 3f+1$. However, these algorithms take expected exponential time for decision. Our solution is based on the approach of [3].

Algorithms for solving Byzantine agreement in constant expected time are given by Feldman and Micali [18, 19] for $n \geq 3f+1$ for the synchronous model and $n \geq 4f+1$ for the asynchronous model [17]. For $n \geq 3f+1$ and an asyn-

chronous model, [9, 29] solve Byzantine agreement with probability $1-\epsilon$ and, conditional on success, processes terminate in constant expected number of rounds, while [1] solves Byzantine agreement with probability one termination and a polynomial expected running time. If $n \leq 3f$, randomized Byzantine agreement is impossible in an asynchronous system even with a PKI. It is also impossible in a synchronous system without a PKI [20]. [4] solves Byzantine agreement in expected constant rounds in an asynchronous system with $n \geq 5f+1$. This protocol has $O(\log n)$ communication complexity per message; however, it assumes a trusted dealer and does not obtain optimal resilience. Our solution, like those of [18, 11, 19, 9, 29, 1], require polynomial communication complexity per message.

Benign failures. [11] addresses the problem of solving consensus in constant expected time with crash or omission failures. There are algorithms for *synchronous systems* and $n \geq 2f+1$, and for *asynchronous systems* and $n \geq \frac{3+\sqrt{5}}{2}f+1$. An algorithm for asynchronous systems and $n \geq 2f+1$ is given in [2] using a *get-core* primitive suggested by Gafni, but this algorithm requires a message-independent adversary. Our solutions use a similar structure and primitives as [11, 2]. The main differences are the following: (1) to handle stronger adversaries we need a stronger *binding gather* primitive; (2) we use probabilistic agreement to solve multi-valued consensus while [11, 2] use a common coin to just solve binary consensus; (3) to handle stronger adversaries, we use verifiable secret sharing; (4) we give an algorithm that tolerates Byzantine failures. For asynchronous systems, it is conjectured that the protocol of [1] solves consensus for $n \geq 3f+1$ in constant expected time (without a PKI) for a message-dependent adversary and omission failures. Our work differs because we give algorithms for *asynchronous systems* and $n \geq 2f+1$. Consensus can also be solved with $n \geq 2f+1$ in partially synchronous systems [14, 16], or in systems with failure detectors [10].

2 Model

We consider a system with n processes denoted $p_1, \ldots, p_n$ that can communicate with each other via point-to-point messages. The system is *asynchronous* meaning that there are no bounds on message delays or on the relative speed of processes. Processes have access to a source of uniformly random bits. The system is subject to process failures, and we consider several possibilities:

1. Crash failures. A process may fail by crashing, that is, it stops taking steps. Communication between every pair of processes is reliable. More precisely, the following properties are satisfied for every processes p_i and p_j:

- *Integrity.* If p_i receives a message m from p_j exactly k times by time t then p_j sent m to p_i at least k times before time t.
- *No Loss.* If p_j does not crash and p_i sends m to p_j exactly k times by time t then p_j eventually receives m from p_i at least k times.

2. Omission failures. A faulty process may experience omission failures, in which it fails to send or receive messages. More precisely, for every processes p_i and p_j, the Integrity property above holds, but the No Loss property is guarantees to hold only if p_i and p_j are not faulty.

3. Byzantine failures. A faulty process may behave arbitrarily, including deviating from its code. The Integrity and No Loss properties hold for every pair of *correct* processes p_i and p_j. If p_i or p_j is Byzantine, neither property may hold.

If a process never becomes faulty we say that it is *correct*.

Power of adversary. When designing fault-tolerant algorithms, we often assume that an intelligent adversary has some control of the system: it may be able to control the occurrence and the timing of process failures, the message delays, and the scheduling of processes. Adversaries may have limitations on their computing power and on the information that they can obtain from the system. Different algorithms are designed to defeat different types of adversaries. The simplest adversary is the *message-oblivious adversary*, which cannot look at the internal state of processes or the contents of messages. Our algorithms can defeat a stronger adversary, called the *sophisticated adversary*, which we now describe. At any point in the execution, the adversary may choose to make a process faulty, provided that at most $f < n/2$ processes are faulty. A faulty process may not exhibit faulty behavior immediately: with Byzantine failures, a faulty process may continue to behave well for a while; with omission failures, a faulty process may continue to handle messages without loss; with crash failures, a faulty process may continue to execute for a while. The adversary has full knowledge of the internal state of a faulty process. In particular, it knows all the messages that it sends and receives. With this information, at any time in the execution, the adversary can dynamically select which process takes the next step and which message this process receives (if any). The adversary, however, operates under the following restrictions: the final schedule must fair, meaning that all correct processes take infinitely many steps, and the messages sent and received must satisfy the Integrity and No Loss property according to the type of failure considered (crash, omission, or Byzantine) as described above. In some cases, we consider a computationally-bounded adversary, which has the additional requirement that its computation is limited to polynomially-bounded functions.

3 Problem

We are interested in solving consensus and probabilistic agreement, which we now define.

Consensus. Each process starts with some initial value and must decide on a single value. In the classical consensus problem, the following must hold:

- *Validity.* If a correct process decides on a value v then v is one of the initial values.
- *Agreement.* No two correct processes decide differently.
- *Termination.* Every correct process eventually decides.

Consensus cannot be solved in an asynchronous system subject to failures [21], so we consider the following weakening of termination:

- *Termination with probability one.* With probability one, all correct processes eventually decide.

We are interested in *fast* algorithms, which we define to be algorithms in which correct processes decide in *constant expected time*.

For a system with Byzantine failures, we consider a weakening of consensus called weak Byzantine agreement, obtained by replacing the validity property with the following:

- *No-Failure Validity.* If all processes are correct and a correct process decides on a value v then v is one of the initial values.[1]

Probabilistic agreement. Probabilistic agreement is a variant of consensus in which processes may decide different values with some probability smaller than one. More precisely, we require the Validity and Termination properties as defined above, as well as the following:

- *Uncertain agreement.* With probability at least $\rho > 0$, no two correct processes decide differently.

Here $\rho > 0$ is some constant. For practical purposes it should be large, say $1/3$.

For a system with Byzantine failures, we consider a variant called *certified probabilistic agreement*, in which the initial value of a process (whether correct or Byzantine) is certified by some computationally unforgeable means, as we later explain. Algorithms will use this certification to provide the validity property.

Cryptographic primitives. The use of cryptography in algorithms can create a negligible probability of failure of the algorithm, including non-termination, due to the adversary breaking the cryptographic primitives by randomly guessing keys. Technically, we say that an algorithm *securely implements* consensus (instead of "implements consensus") to indicate this negligible probability of failure.

4 Binding gather

Our fast algorithms for benign failures are based on a simple primitive called *binding gather*. All correct processes invoke *binding gather*(v) with some input value v, and the primitive returns as output a *set* of values, such that the following holds:

- *Validity.* Every value in every output set is the input value of some process.
- *Binding Commonality.* There exists some value v such that v is in every output set, and v can be determined by an external observer when the first process outputs its set.
- *Termination.* All correct processes eventually output some non-empty set of values.

A value that is in every output set is called a *common value*. The binding gather primitive is easy to implement in a system with crash or omission failures using three asynchronous rounds of a full-information protocol (Section 6.3 extends the primitive and implementation to Byzantine failures):

[1] This property implies the standard validity property of weak Byzantine agreement [26], which states that if all processes are correct and they have the same initial value v then no correct process decides on a value other than v.

Round 1. p_i sends its input to all, waits for $n-f$ values, and stores it in set $seen_i$.

Round 2. p_i sends set $seen_i$ to all, waits for $n-f$ sets, and stores their union in set $seenmore_i$.

Round 3. p_i sends set $seenmore_i$ to all, waits for $n-f$ sets, and returns the union of these sets.

By using another round of a full-information protocol (a total of four rounds), we can ensure that there exists a set of $n-f$ common values. A similar three round protocol called *get-core* appeared in [2]: it obtains a set of $n-f$ common values where each common value has the first and third properties of *binding gather* but does not provide Binding Commonality, a property needed against sophisticated adversaries. If we used *get-core*, a sophisticated adversary could influence which values appear in the common set, and this would break our running time guarantees.

5 Algorithms for crash failures

We now present algorithms for consensus and probabilistic agreement that tolerate crash failures and terminate in constant expected time, assuming $n \geq 2f+1$. The algorithm for consensus uses probabilistic agreement as a building block, so we start with the latter.

5.1 Fast probabilistic agreement

The fast algorithm for probabilistic agreement is similar to the algorithms in [18, 24], except that we use binding gather to make it work for $n \geq 2f+1$ with crash failures. Each process p_i starts with an initial value v_i. The rough idea of the algorithm is that each process p_i picks a random rank for itself (a rank is a number) and sends to all a message containing its rank and v_i. A process collects $n-f$ such pairs and calls binding gather to share its set of $n-f$ pairs. Binding gather will return a set of such sets, with the property that at least one set C of $n-f$ pairs is returned to every process. Each process looks at all pairs in all sets that it gets from binding gather, finds the largest rank, and decides on the value associated with that rank. If the ranks are uniformly random, the true largest rank M will be in C with probability at least $(n-f)/n \geq 1/2$, which will cause all processes to pick M and hence decide on the same value. One technicality is that the ranks need to be picked from a bounded interval, and there is a probability that two different processes pick the same rank. We must choose the interval to be large enough $(0 \ldots n^4-1)$ so that this collision happens with probability less than $1/n^2$. The algorithm is given in Figure 1.

This algorithm works with the simple message-oblivious adversary, but it does not quite work with the sophisticated adversary: after the adversary learns the largest rank M, it can coordinate the execution of binding gather to ensure that C does not include M. To solve this problem, we use secret sharing to hide the values of the ranks from the adversary until the common value of binding gather has been determined.

The full algorithm is given in Figure 2. It uses binding gather and n^2 copies of the secret sharing primitives denoted *dealer-share*^{ij}, *share*^{ij}, and *reconstruct*^{ij}

Process p_i has initial value v_i and executes the following code:

1. *rank*$_i$:= random number between 0 and $n^2 - 1$
2. send (*rank*$_i$, v_i) to all
3. wait to receive (*rank*$_j$, v_j) from $n-f$ processes j
4. R_i := set of received $(rank_j, v_j)$'s
5. $V_i := \cup$*binding-gather*(R_i) (* *binding-gather* outputs a set of sets; flatten it out to V_i *)
6. *max_rank* := $\max\{$*rank* : (*rank*, $*) \in V_i\}$
7. choose v such that (*max_rank*, $v) \in V_i$
8. *decide*(v)

Fig. 1. Fast algorithm for probabilistic agreement with crash failures, $n \geq 2f+1$, and a message-oblivious adversary.

Process p_i has initial value v_i and executes the following code:

(* *part 1: send values and shares* *)
1. $r_{i1}, \ldots, r_{in}$:= pick n random numbers between 0 and $n^4 - 1$
2. for $j = 1..n$ do *dealer-share*$^{ij}(r_{ij})$
3. for $j, k = 1..n$ do fork *share*$^{jk}()$ (* run share protocol; it may not return if dealer crashes, so fork *)
4. wait for *share*^{ji} to return for $n-f$ distinct j's, and let S_i be the set of those j's (* processes in S_i are those that contribute to p_i's rank *)
5. send (S_i, v_i, i) to all
6. wait to receive (S_j, v_j, j) from $n-f$ processes; let J_i be the set of such messages (* J_i has data of processes whose rank can be retrieved *)

(* *part 2: gather* *)
7. *bigJ* := $\cup$*binding-gather*(J_i) (* *binding-gather* outputs a set of sets; flatten it *)
8. $P := \{j : (*, *, j) \in$ *bigJ*$\}$ (* P contains processes whose rank can be retrieved *)
9. for each $(S, v, j) \in$ *bigJ* do $(S_j, v_j) := (S, v)$ (* extract S_j and v_j *)

(* *part 3: recover secrets* *)
10. for each $j, k = 1..n$ do fork { wait for *share*^{jk} to return; r_{jk} := *recover*$^{jk}()$ } (* *share*^{jk} or *recover*^{jk} may not return, so do in background *)
11. for each $j \in P$ do
12. wait for *recover*^{kj} to return for all $k \in S_j$
13. *rank*$_j$:= $\sum_{k \in S_j} r_{kj} \bmod n^4$ (* recover j's rank *)

(* *part 4: choose winner and decide* *)
14. *winner* := *argmax*$_j\{$*rank*$_j : j \in P\}$ (* winner is process with highest rank *)
15. *decide*(v_{winner})

Fig. 2. Fast algorithm for probabilistic agreement with crash failures, $n \geq 2f+1$, and a sophisticated adversary.

---

where $i, j = 1..n$ The protocol has four parts. In the first part, p_i picks n random values $r_{i1}, \ldots, r_{in}$. The idea is that we want to hide p_i's rank from p_i itself, because p_i could be a faulty process that the adversary has access to. We use the idea in Feldman and Micali's protocol [18]: each process will pick a uniform random value between 0 and n^4-1 and p_i's rank will be the sum of $n-f$ such values (modulo n^4), which is also uniformly distributed. Thus, the random value r_{ij} that p_i picks is p_i's contribution to the rank of p_j. Process p_i then uses secret sharing to distribute shares of the r_{ij}'s to all processes. It then waits to receive shares of r_{ji} for $n-f$ values of j; we let S_i denote those values of j. Intuitively, S_i are those j's that will contribute to the rank of p_i. Process p_i then sends (S_i, v_i, i) to all, where v_i is its initial input; other processes will use S_i to reconstruct the rank of p_i later, and v_i to decide in case p_i is the process with the highest rank. Finally, p_i waits for such triples (S_j, v_j, j) from $n-f$ processes, and stores them in J_i. Intuitively, J_i has the data (S_j, v_j) of processes whose rank can be

retrieved: they did not crash too early in the protocol. In the second part of the algorithm, p_i calls binding gather with its set J_i, and obtains a bunch of such sets from other processes, and puts all the triples (S_j, v_j, j) obtained in a big set *bigJ*. At this point, the common set C of binding gather has been determined, by definition of binding gather, and with probability at least $(n-f)/n - 1/n \geq 1/3$ (assuming $f \geq 1$), C includes the process with largest rank and such a process is unique. Process p_i then unravels *bigJ* to extract a set P of processes and for each $j \in P$, extracts their values of S_j and v_j. In the third part, processes recover the random numbers and add them together to produce the rank of each process in P. A process may not be able to recover r_{kj} (i.e., return from $recover^{kj}$) since process p_k may have crashed before returning from $dealer\text{-}share^{kj}$. However, this does not happen for $j \in P$ and $k \in S_j$: for those values of j and k, p_j sent a *done* message and completed $share_{kj}$ by definition of P and S_j. Therefore, for all $j \in P$, p_i can retrieve the rank of p_j by adding together the appropriate r_{*j}'s mod n^4. This is stored in variable $rank_j$. Finally, in part 4, process p_i picks the process in P with largest rank and decides on the value of that process.

5.2 Fast consensus

Our consensus algorithm is obtained by modifying Ben-Or's algorithm [3], which is a *binary* consensus algorithm in which the processes' initial values must be 0 or 1. The key idea of Ben-Or's protocol is that, if all processes start a round with the same estimate, then they all decide in that round. If there is no decision, at the end of the round some processes will set their estimate to a random bit in the hope that, if they are very lucky, processes will all end up with the same bit and therefore will decide in the next round. If all n processes pick a bit randomly, the probability that they will pick the same bit is exponentially small in n. As a result, the expected number of rounds until decision is exponentially large in n.

We modify Ben-Or's algorithm so that, at the end of each round, instead of using a random coin, processes use an instance of probabilistic agreement to set their estimate (there is an instance of probabilistic agreement per round). The rationale is that probabilistic agreement has a high (constant) probability that processes will pick the same value and hence decide in the next round. As a result, the expected number of rounds until decision is constant. There is another more subtle difference between our algorithm and Ben-Or's. To ensure agreement, in Ben-Or's algorithm a process does *not* change its estimate to a random value at the end of a round if it believes another process may have decided. In our algorithm, all processes unconditionally change their estimate to the decision value of probabilistic agreement. Doing so does not jeopardize agreement because if a process decides v in a round, all processes will start probabilistic agreement with v and hence will decide v (in probabilistic agreement). Another difference between the algorithms is that our algorithm is not restricted to binary consensus: initial values can come from any domain.

Our complete algorithm is shown in Figure 3. Each process maintains an estimate of the decision in variable v_i, which is initially the process initial value. Processes proceed in rounds $k = 1, 2, \ldots$, where each round has three phases. In phase 1 of round k, a process sends a (REPORT, k, v_i) message with its estimate

Code for each process p_i with initial value v_i:

1. $k := 0$
2. while true do
3. $k := k + 1$
   (phase 1 *)*
4. send (REPORT, k, v_i) to all
5. wait to receive $(\text{REPORT}, k, *)$ from $n-f$ processes
   (phase 2 *)*
6. if all received (REPORT, k, w) are for the same w
7. then send $(\text{PROPOSAL}, k, w)$ to all
8. else send $(\text{PROPOSAL}, k, ?)$ to all
9. wait to receive $(\text{PROPOSAL}, k, *)$ from $n-f$ processes
10. if received some $(\text{PROPOSAL}, k, w)$ with $w \neq ?$ then $v_i := w$
11. if all received $(\text{PROPOSAL}, k, w)$ are for the same w then decide w
    (phase 3 *)*
12. $v_i :=$ *probabilistic-agreement*(k, v_i) (* run k-th instance of probabilistic agreement with input v_i *)

Fig. 3. Algorithm for consensus and $n \geq 2f+1$, which uses probabilistic agreement as a subroutine. By using a fast probabilistic agreement algorithm, we obtain a fast consensus algorithm.

v_i to all, waits to receive $n-f$ reports of round k, and checks whether a majority of processes reported the same estimate w. If so, in phase 2, a process sends a $(\text{PROPOSE}, k, w)$ message to all, otherwise it sends a $(\text{PROPOSE}, k, ?)$, where "?" is a special value. There can be either 0 or 1 proposals different from ? in phase 2, because this proposal must have been reported in phase 1 by a majority of processes. Processes wait for $n-f$ proposals of round k. If all of them are for the same value $w \neq ?$ then the process decides on w. If one of them is for a value $w \neq ?$ then the process changes its estimate v_i to w. In phase 3, processes executes a new instance of probabilistic agreement using v_i as its initial value, and then changes v_i to the decision.

This algorithm can be easily extended to handle omission failures: we use the PKI to implement a private reliable send mechanism. Briefly, for p to send a message m to q privately and reliably, it performs the following. p encrypts and sends (m, q) to all processes and waits to receive acknowledgements from $f+1$ processes; a process that receives (m, q) from p sends an acknowledgement to p and forwards m to q.

6 Algorithms for Byzantine failures using weak sequence broadcast

We now consider an asynchronous system with Byzantine failures, where $n \geq 2f+1$. In this setting, it is easy to show that consensus (with probability one termination) cannot be solved even if processes have access to a public-key infrastructure. We therefore consider solutions that use weak sequenced broadcast as a primitive, described in Section 6.1. We show that this primitive can be implemented with a minimal TPM. Because the minimal TPM can be implemented in a system with crash failures, it follows that deterministic consensus is still impossible, even if processes have weak sequenced broadcast (otherwise, processes could solve consensus in a system with crash failures). Thus, as before, we have to resort to randomized solutions that guarantee termination with

probability one minus a negligible probability due to the use of cryptography. Using sequenced broadcast, we show how to implement probabilistic agreement with Byzantine failures, and then how to implement weak Byzantine agreement.

6.1 Sequenced broadcast and weak sequenced broadcast

Roughly speaking, sequenced broadcast is a type of broadcast that ensures, for a given sender p_i, that all processes deliver the messages of that sender in the same order. This provides an ordering *per sender* of messages, similar to FIFO broadcast [23].[2] This is useful because it prevents the problem of equivocation [12], in which a Byzantine process can send different values to different processes. In a system with $n \geq 3f+1$, equivocation can be avoided using Bracha's broadcast algorithm [5], but here we are concerned about systems with $n \geq 2f+1$.

We shall consider two versions of sequenced broadcast, where the stronger version ensures that all correct processes deliver the same set of messages, and the weaker version does not. More precisely, we define weak sequenced broadcast in terms of two primitives, *sbcast* and *sdeliver*. We are interested in the k-th message broadcast by a process, and the k-th message delivered from p by another process. To make this explicit, when a process p broadcasts m as its k-th message, we will say that p *sbcasts*(k, m). We note that k is determined by the order in which the process calls *sbcast*, and so it is not a real parameter; we make k explicit just to make it simpler to match a broadcast with its deliver in algorithms. When a process q delivers m as its k-th message from p, we will say that q *sdelivers*(k, m) from p. We assume that messages broadcast by correct processes are unique, which can be ensured via sequence numbers. Weak sequenced broadcast satisfies the following properties:

- *Integrity.* If processes p and q are correct, and q *sdelivers* (k, m) from p then p previously *sbcasts*(k, m).
- *Validity.* If correct process p *sbcasts*(k, m) then eventually all correct processes *sdeliver*(k, m) from p.
- *FIFO Agreement:* If two correct processes *sdeliver* (k, m) and (k, m') from the same process q then $m = m'$.

Weak sequenced broadcast allows one correct process to deliver a message from a sender, and another correct process not to. This is not allowed in *sequenced broadcast*, which provides an additional Agreement property similar to reliable broadcast [23]:

- *Agreement.* If a correct process *sdelivers* (k, m) then eventually all correct processes *sdeliver* (k, m).

It is easy to implement sequenced broadcast using weak sequenced broadcast. We now explain how to implement weak sequenced broadcast using a minimal TPM. The TPM has a secret TPM signing key and provides each process p_i with a private set of registers PCR_k (Platform Configuration Registers) initialized to zero [31]. The TPM allows a process to modify a PCR register only by using

[2] FIFO broadcast is defined for systems with crash failures. Sequenced broadcast can be seen as an extension of FIFO broadcast to systems with Byzantine failures.

the *tpm-extend*(k, v) function, which sets $PCR_k := hash(PCR_k \cdot v)$. Function *tpm-quote*$(k)$ allows p_i to sign (i, PCR_k) using the TPM secret key. There are many PCR registers, but we only use PCR_1.

To *sbcast*(m), process p_i calls *tpm-extend*$(1, m)$, and then calls *tpm-quote*(1) to obtain a TPM signature s on (i, PCR_1). Next, p_i sends m and s to all using FIFO-send (FIFO-send can be implemented using sequence numbers). Each process p_j keeps a vector $H_j[i]$ with the hash of the sequence of messages that p_j has seen from p_i for every i. When p_j receives (m, s) from p_i, it updates $H_j[i]$ to mimic the way the TPM of p_i updates PCR_1. Then p_j checks whether s is a valid signature on $(i, H_j[i])$. If so, p_j *sdelivers* m, otherwise it ignores m.

The key reason why the algorithm works is that a process cannot set its PCR_1 any way it wants, because *hash* is a one-way hash function. Once a process extends PCR_1 for k-th time using message m, it is restricted to send m as its k-th messages to all processes, otherwise m is rejected by correct processes.

6.2 Verifiable Secret Sharing

Our probabilistic agreement algorithm relies on VSS. We now give a simple VSS implementation for a system with Byzantine failures and $n \geq 2f+1$ using sequenced broadcast, and the cryptographic primitives of encryption, signature, and zero-knowledge proofs of knowledge. Roughly speaking, it works as follows. The dealer chooses a degree f polynomial $g(x)$ and sends via sequenced broadcast a message $\langle E_1(g(1)), \cdots, E_n(g(n))\rangle$ where $E_i(g(i))$ is an encryption of the share of player i using player i's public encryption scheme. The dealer then provides a zero-knowledge proof that the message sent is indeed an appropriate encryption of a degree f polynomial. A player that is convinced by the proof sends an acknowledgement via sequenced broadcast and the share phase ends once $n-f$ acknowledgements are received. The algorithm is given in Figure 4.

```
Code for process p_i:
1.    procedure dealer-share(v_i)                                   { for the dealer process }
2.      choose a random polynomial g of degree f such that g(0) = v_i
3.      sbcast(1, ⟨E_1(g(1)), ··· , E_n(g(n))⟩)        E_j(·) encrypts using p_j's public key *)
4.      do an interactive constant-round zero-knowledge proof with each process on the statement that
          E_1(g(1)), ..., E_n(g(n)) is an encryption of a degree f polynomial
5.    procedure share()                                             { for all processes }
6.      wait to sdeliver(1, ⟨EG_1, ..., EG_n⟩) from dealer   (* if dealer is correct EG_j = E_j(g(j)) *)
7.      fork {
8.        participate in constant-round zero-knowledge proof with dealer that e_1, ..., e_n is an encryption of
            a degree f polynomial
9.        if process is convinced of zero-knowledge proof
10.       then sbcast(2, DONE)      (* if necessary sbcast an empty messages so we can sbcast(2, ...) *)
11.     }
12.     wait until sdeliver(2, DONE) from n−f processes
13.   procedure recover():                                          { for all processes }
14.     send (i, g(i)) to all
15.     wait to receive (j, g_j) from n−f processes where E_j(g_j) = EG_j   (* g_j must match encrypted values
                                                                              broadcast by dealer *)
16.     find degree polynomial g of degree f going through the n−f values (j, g_j) gotten in line 15
17.     return g(0)
```

Fig. 4. Algorithm for securely implementing verifiable secret sharing and $n \geq 2f+1$ using sequenced broadcast in a system with Byzantine failures.

6.3 Binding gather with certified values

We now extend the definition of binding gather for a system with Byzantine failures. Clearly, we cannot require that Byzantine processes do anything, so we modify its properties as follows:

- *Validity.* Every value in every output set of a *correct* process is the input value of some process.
- *Binding Commonality.* The exists some value v such that v is in the output set of every *correct* process, and v can be determined by an external observer when the first *correct* process outputs its set.
- *Termination.* All correct processes eventually output some value.

(The italics indicate differences with respect to the definition in Section 4.) Even with these weaker requirements, there is still a problem with the Validity property: no implementation can provide this property, because if a Byzantine process acts correctly except that it changes its initial value to a bogus value, then this bogus value could appear in the output of correct processes.

To address this problem, we use the notion of *certified values*, which is similar to external validity [7]. Roughly speaking, a value v is certified if there is a legitimacy test that can be applied to v such that bogus values from Byzantine process cannot pass the test. For example, we may require a value to have signatures from $f+1$ processes, and the test verifies that the signatures are valid.

More precisely, we define a *certification scheme* as a set *Cert* of values and a set of Boolean procedures $check_i(v)$, one for each process p_i, such that a byzantine process cannot generate a value in *Cert* unless it is given that value (e.g., it receives the value from a correct process). Intuitively, values in *Cert* are the certified values and each $check_i(v)$ is a way for process p_i to check if $v \in Cert$. We require two properties: (a) if $v \notin Cert$ then $check_i(v)$ must return *false*; and (b) if $v \in Cert$ then there is a time after which $check_i(v)$ returns *true* (it may return *false* for a finite period until p_i sees evidence from other processes that $v \in Cert$). Since we rely on cryptographic primitives, we allow a negligible probability that properties (a) and (b) are violated.

By using certification schemes, we can provide a stronger validity condition. For example, with Byzantine agreement, we can require that initial values be certified, where the certification scheme is a parameter of the problem. Thus, each process has an unforgeable initial value and we can design algorithms that use the $check_i$ procedures to ignore bad values and ensure that correct processes decide on one of the initial values.

Similarly, we define *binding gather with certified values* via the three properties of binding gather with the requirement that initial values must be certified. Certification schemes are particularly useful in the composition of algorithms, when the input of an algorithm is an unforgeable output of another algorithm, as we demonstrate with probabilistic agreement and weak Byzantine agreement.

It is easy to implement binding gather with certified values, by using the same algorithm of Section 4 except that processes use sequenced broadcast to send values, and they ignore values that do not pass the certification check.

6.4 Fast probabilistic agreement with certified values

We now describe an algorithm for probabilistic agreement. To satisfy the Validity property of probabilistic agreement, we need to assume that initial values are certified according to some certification scheme, as in Section 6.3. We denote the certification test procedures by $checkInput_i$.

The algorithm for probabilistic agreement with Byzantine failures is similar to the one of Section 5.1, except that we replace send-to-all with the sequenced broadcast primitive (Section 6.1), we use the verifiable secret sharing algorithm for Byzantine failures (Section 6.2), and we use binding gather with certified values (Section 6.3). Furthermore, the algorithm accepts a message only if it is delivered by sequenced broadcast, and the value it carries has passed the secret sharing phase and the certification test of $checkInput_i$.

6.5 Fast weak Byzantine agreement

The algorithm for weak Byzantine agreement is similar to the algorithm for consensus and crash failures of Section 5.2. The differences are that we use the probabilistic agreement algorithm with certified values of Section 6.4, using a certification check that we will explain below. In addition, we replace send-to-all with sequenced broadcast. We must also check that messages received from processes follow the protocol. To do so, when a process *sbcasts* a message m, it must attach a proof that m follows the algorithm. This proof consists of the messages that the process *sdelivered* that causes it to send m, where those messages themselves must carry proofs that they are legitimate. Thus, in the end, each message m will contain a history of the execution that justifies m. When a process waits to receive $n-f$ messages, it ignores messages with incorrect proofs. At the end of each round, a process calls probabilistic agreement with certified values using the input $\langle v_i, P_i \rangle$, where P_i is a proof that v_i is legitimate. The certification test $checkInput_i$ used in probabilistic agreement is the function that verifies that v_i is legitimate according to P_i.

Acknowledgments

We would like to thank Sergey Yekhanin for helpful discussions and insights on the binding gather protocol.

References

1. I. Abraham, D. Dolev, and J. Y. Halpern. An almost-surely terminating polynomial protocol for asynchronous byzantine agreement with optimal resilience. In *ACM Symposium on Principles of Distributed Computing*, pages 405–414, New York, NY, USA, 2008. ACM.
2. H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience, March 2004.
3. M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *ACM Symposium on Principles of Distributed Computing*, pages 27–30, New York, NY, USA, 1983. ACM.
4. P. Berman and J. A. Garay. Randomized distributed agreement revisited. In *FTCS*, pages 412–419, 1993.
5. G. Bracha. An asynchronous [(n - 1)/3]-resilient consensus protocol. In *ACM Symposium on Principles of Distributed Computing*, pages 154–162, New York, NY, USA, 1984. ACM.

6. C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 88–97, New York, NY, USA, 2002. ACM.
7. C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 524–541, London, UK, 2001. Springer-Verlag.
8. C. Cachin, K. Kursawe, and V. Shoup. Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography (extended abstract). In *ACM Symposium on Principles of Distributed Computing*, pages 123–132, New York, NY, USA, 2000. ACM.
9. R. Canetti and T. Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *ACM Symposium on Theory of Computing*, pages 42–51, New York, NY, USA, 1993. ACM.
10. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.
11. B. Chor, M. Merritt, and D. B. Shmoys. Simple constant-time consensus protocols in realistic failure models. *J. ACM*, 36(3):591–614, 1989.
12. B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 189–204, New York, NY, USA, 2007. ACM.
13. M. Correia, N. F. Neves, L. C. Lung, and P. Veríssimo. Low complexity byzantine-resilient consensus. *Distrib. Comput.*, 17(3):237–249, 2005.
14. D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, Jan. 1987.
15. D. Dolev and H. R. Strong. Polynomial algorithms for multiple processor agreement. In *ACM Symposium on Theory of Computing*, pages 401–407, New York, NY, USA, 1982. ACM.
16. C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, Apr. 1988.
17. P. Feldman. Asynchronous byzantine agreement in constant expected time, 1989. Unpublished (copy availbale from M. Ben-Or).
18. P. Feldman and S. Micali. Optimal algorithms for byzantine agreement. In *ACM Symposium on Theory of Computing*, pages 148–161, 1988.
19. P. Feldman and S. Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM J. Comput.*, 26(4):873–933, 1997.
20. M. J. Fischer, N. A. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.
21. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *J. ACM*, 32(2):374–382, 1985.
22. S. Haber and W. S. Stornetta. How to time-stamp a digital document. *Journal of Cyptology*, 3(2):99–111, Jan. 1991.
23. V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report 94-1425, Computer Science Department, Cornell University, Ithaca, New York, May 1994.
24. J. Katz and C.-Y. Koo. On expected constant-round protocols for byzantine agreement. In C. Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 445–462. Springer, 2006.
25. R. Kotla and I. Roy, 2010. Personal Communication.
26. L. Lamport. The weak byzantine generals problem. *J. ACM*, 30(3):668–676, 1983.
27. D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. Trinc: small trusted hardware for large distributed systems. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 1–14, Berkeley, CA, USA, 2009. USENIX Association.
28. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996.
29. A. Patra, A. Choudhary, and C. Pandu Rangan. Simple and efficient asynchronous byzantine agreement with optimal resilience. In *ACM Symposium on Principles of Distributed Computing*, pages 92–101, New York, NY, USA, 2009. ACM.
30. M. Stadler. Publicly verifiable secret sharing. In *Advances in Cryptology EUROCRYPT 1996*, pages 190–199. Springer Verlag, 1996.
31. `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`, 2010. As of Feb 2010.