Information-Theoretically Secure Byzantine Paxos

Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Harry C. Li
The University of Texas at Austin
{anand, lorenzo, aclement, harry}@cs.utexas.edu

Abstract

We present Information Theoretically secure Byzantine Paxos (IT ByzPaxos), the first deterministic asynchronous Byzantine consensus protocol that is provably secure despite a computationally unbounded adversary. Previous deterministic asynchronous algorithms for Byzantine consensus rely on unproven number theoretic assumptions (i.e., digital signatures) to maintain agreement. IT ByzPaxos instead uses secret sharing techniques that are information theoretically secure to ensure that all correct processes agree. Our protocol guarantees safety in an asynchronous system and provides progress under eventual synchrony. IT ByzPaxos matches the 3f+1 lower bound on the number of processes for Byzantine consensus.

1 Introduction

We present Information Theoretically secure Byzantine Paxos (IT ByzPaxos), the first deterministic asynchronous Byzantine consensus protocol that is provably secure despite a computationally unbounded adversary. Previous deterministic asynchronous algorithms for Byzantine consensus [6, 13, 8] rely on unproven number theoretic assumptions (i.e., digital signatures) to maintain agreement. IT ByzPaxos instead uses secret sharing techniques that are information theoretically secure to ensure that all correct processes agree. Our protocol guarantees safety in an asynchronous system and provides progress under eventual synchrony. IT ByzPaxos matches the 3f+1 lower bound on the number of processes for Byzantine consensus.

IT ByzPaxos does not assume that an adversary has a limited number of CPU cycles or that problems like factoring or discrete logarithms are hard. With the increasing connectivity of today's computers, adversaries have more computing power at their disposal than ever before. Additionally, unforeseen advances in number theory have led to algorithms that crack ciphers in a matter of months, compared to the originally estimated millions of years. In 2005, for example, Bahr et al. [1] factored the 640-bit number (RSA-640) in the RSA Factoring Challenge in less than six months using 30 2.2 GHz computers.

IT ByzPaxos provides an alternative to previous algorithms that rely on computational bounds or unproven number theoretic assumptions. We show that it is possible to design a deterministic Byzantine consensus algorithm despite a computationally unbounded adversary. Furthermore, we believe that our techniques generalize to other consensus protocols, like Fast Byzantine Paxos [13] and Dutta-Gerraoui [8], to create alternatives to cryptographic solutions.

IT ByzPaxos implements a write-once register

over a set of processes by restricting the values that can be written according to the values that can be read [12]. In Byzantine versions of such a register, a process can only write to the register after presenting a proof that that process first read from the register. As shown by Li et al., guarding every write with such proofs makes the register write-once, a property ideal for maintaining agreement.

In previous works, these proofs are implemented as quorums of signed messages providing two properties. First, that a quorum of processes believes it is safe to decide some value. And second, who the members of that quorum are. For consensus, only the first property is necessary. Our insignt is that the first property can be implemented using secrets. In IT ByzPaxos, a process can only write a value if it can reconstruct an appropriate secret. We carefully structure the protocol such that processes can only reconstruct secrets that will not lead to safety violations.

For clarity, we present IT ByzPaxos in three stages. First, we present IT ByzPaxos as an implementation of a write-once register whose writes are guarded by proofs of read operations. Second, we show how to implement such proofs using secret sharing with an honest dealer. Third, we relax the requirement on the dealer and show how the processes can generate and disseminate secrets independently.

2 Related Work

IT ByzPaxos follows a long line of work in asynchronous consensus algorithms. Deterministically solving consensus is impossible in an asynchronous system with process failures [10]. Researchers have dealt with this impossibility by proposing protocols that either guarantee liveness only during synchronous periods or guarantee liveness with high probability. IT ByzPaxos falls in the latter category.

There are a wealth of deterministic asynchronous consensus protocols that always maintain safety and provide liveness only during synchronous times. These protocols share a common technique inspired by Lamport's Paxos protocol [11], a deterministic asynchronous consensus algorithm for crash failures. Castro and Liskov extended the Paxos protocol to Byzantine failures in their Practical Byzantine Fault-tolerance (PBFT) work [6]. Since then, researchers have proposed a number of variants to the original algorithm in PBFT [13, 8]. All of these approaches, however, rely on digital signatures to maintain safety.

The second class of asynchronous Byzantine consensus protocols sacrifices determinism for probabilistic termination. Interestingly, many of the results in this second class are resilient to computationally unbounded adversaries, thus not relying on digital signatures for safety. Bracha introduced a Byzantine consensus protocol that terminated in expected exponential time in the number of processes [2]. Canetti and Rabin refined this result to terminate in constant expected time [5]. Common to these randomized Byzantine consensus protocols is secrets.

Robust secret sharing, as described by Shamir [15], enables any group of t out of n processes to reconstruct a secret while any group of size less than t gains no information about the secret. Our approach leverages existing secret sharing work that enables processes to detect if a reconstructed value is indeed the actual secret or a false one. Shamir's scheme allows a process to be fooled into believing a falsified value is the secret. Verifiable secret sharing [7, 14] schemes prevent such situations but they rely on cryptographic mechanisms. In IT ByzPaxos, we use Tompa et al.'s technique [16] to detect when a reconstructed value corresponds to the actual secret or not. We explain Tompa's appoach in more detail in Section 4.3.

3 Model

3.1 Problem Statement

In this paper, we focus on binary consensus. Each process in the system starts with a binary input value, which it can propose. A binary consensus protocol guarantees the following properties:

- **Agreement :** No two correct processes decide different values.
- **Validity :** If a correct process decides a value v, then some process proposed v.
- **Termination :** Each correct process eventually decides.

3.2 Model/Assumptions

We consider n processes at most $f < \lceil \frac{n+1}{3} \rceil$ of which may be Byzantine and can deviate arbitarily from the protocol. The remaining processes are correct, but may be running at arbitrary speeds. We also assume that each process has access to random bits, a source of randomness that Byzantine processes cannot affect.

3.2.1 Network assumptions

The processes may communicate with each other via authenticated point-to-point links that are asynchronous but reliable. We assume that links are reliable to make the presentation simpler. To handle reliable links, we could implement mechanisms to resend messages.

We also assume that point-to-point channels are private, an assumpition that is consistent with previous works [3, 9, 4] that solve randomized Byzantine agreement and provide information-theoretic guarantees. Thus, an adversary cannot eavesdrop on the private communication between two correct nodes[1]. Private channels do not have to be implemented using cryptography. A secured and verified switch can prevent adversaries from eavesdropping on private communication.

3.2.2 Synchrony assumptions

Since deterministically solving consensus is impossible in an asynchronous setting prone to crash failures, we require that a consensus protocol always be safe and be live only during sufficiently long synchronous periods. Consensus protocols typically provide this condition using exponentially increasing timeout intervals in which a distinguished process proposes a value for other process to decide.

4 IT ByzPaxos (Honest Dealer)

IT ByzPaxos implements a write-once register over n processes. Each process is responsible for both issuing read and write requests and acknowledging such requests. Processes write value-timestamp pairs to the register and read such pairs out.

The leader for each timestamp is responsible for writing a value to the register during that timestamp. Process i is the leader of timestamp ts if $i \equiv ts \bmod n$. Processes move from one timestamp to the next if it seems a decision is stalled, suggesting that the leader of the current timestamp is faulty. Timestamps are similar to the views employed in PBFT.

To ensure that only one value is ever written, processes read from the register before writing. If a leader reads a value $v \neq \perp$, then that leader attempts to write v. To guard against Byzantine processes that may try to write values different from the

[1]The private channel is not required for the protocol in Section 4

ones they read, each read returns a proof of the value read. Such proofs guard writes, limiting the values that can be written and ensuring that only one value is ever written.

When writing a value v for timestamp ts, a proof is *appropriate* if it guarantees that the last written value is v or that no value has or ever will be written for a timestamp ts', $ts_0 \leq ts' < ts$, where ts_0 is the initial timestamp. Processes ignore write requests that are not accompanied with appropriate proofs.

We implement these proofs as sequences of secrets. For each timestamp ts, a trusted dealer generates three random secrets, S^0_{ts}, S^1_{ts}, and $S^\perp_{ts}$, corresponding to the values that can be written or when no value is written. The dealer splits each secret into n parts and sends each part to a different process such that $n - f$ parts are necessary and sufficient to reconstruct each secret. We denote process i's part of secret S^v_{ts} as $S^v_{ts}[i]$. Processes collect parts for reconstructing secrets to prove that specific events have or have not taken place.

- If a process reconstructs a secret S^v_{ts}, $v \neq \perp$, then no process can reconstruct a secret $S^{v'}_{ts}$, $v' \neq \perp$ and $v' \neq v$.
- If a process reconstructs a secret $S^\perp_{ts}$, then no value $v \neq \perp$ can be written for timestamp ts.

A process reconstructs an appropriate sequence of secrets to write value $v \in \{0, 1\}$ for timestamp ts by either

- reconstructing the secret $S^v_{ts'}$ for some $ts' < ts$ and reconstructing all secrets $S^\perp_{ts''}$ such that $ts' < ts'' < ts$ **or**
- by reconstructing all secrets $S^\perp_{ts''}$ such that $ts_0 \leq ts'' < ts$

To detect faulty actions, the dealer also distributes a verification part $V^v_{ts}[i]$ to processs i for each secret S^v_{ts}. Process i uses $V^v_{ts}[i]$ to check other processes' claims about reconstructing S^v_{ts}.

4.1 The Algorithm (first write)

We now discuss how processes implement the write-once register and how processes reconstruct secrets during read and write operations. For reference, Figure 1 gives the protocol in greater detail.

The leader p for timestamp ts writes a value v to the register by sending $\langle \texttt{PRE-WRITE}, v, ts, proof \rangle$ to all processes. When ts is the initial timestamp ts_0, $proof$ can be `NULL` since no value can be written for an earlier timestamp. We defer a discussion of the proofs and checking for the next subsection (Section 4.2).

A process i that receives $\langle \texttt{PRE-WRITE}, v, ts, proof \rangle$ from p accepts it only if the following conditions are met.

1. i is currently in timestamp ts
2. the proof is appropriate for value v and timestamp ts
3. i has not sent a `WRITE` message for ts yet

A process that accepts $\langle \texttt{PRE-WRITE}, v, ts, proof \rangle$ sends the message $\langle \texttt{WRITE}, v, ts, S^v_{ts}[i] \rangle$ to all processes. If i receives `WRITE` messages for v, ts from $n - f$ processes, i uses $V^v_{ts}[i]$ to verify the secret reconstructed from the secret parts. If i successfully verifies the reconstructed secret, then i sends $\langle \texttt{WRITE-ACK}, v, ts \rangle$ to all processes. If i receives $n - f$ `WRITE-ACK` messages for value v all with the same timestamp, i decides v.

4.2 The Algorithm (subsequent writes)

IT ByzPaxos provides liveness despite a faulty primary by allowing processes to timeout for each

Process i's high-level protocol:

let $currTS$ be a non-negative number initialized to 0
let $last$ have three fields: v, ts, and $shrs$
  $last.v := \perp, last.ts := -1, last.shares :=$ NULL
  let read-acks be an empty dictionary

while true
  wait until $i \equiv currTS \bmod n$
    v, ts, secret-shares := read()
    if read did not return error
      write(v, ts, secret-shares)
    endif

Implementation of read and write

procedure read()
  broadcast $\langle$READ, $currTS\rangle$
  wait until read-acks$[currTS]$ is non-NULL
    let TS' be the highest ts' among the read acks
    let V' be the value of the read ack for TS'
    let $shares$ be a map such that $shares[ts]$ are the shares
      used to reconstruct $S^{V'}_{TS'}$ if $ts = TS' \geq 0$ and
      $S^{\perp}_{ts}$ for $TS' < ts < currTS$
    return $(V', currTS, shares)$
    on timeout
      return error

on receive $\langle$READ-ACK, $ts, *, *, *, *\rangle_j$ **from** j
  if already received READ-ACK for ts from j
    discard
  else if there exists $n - f$ read acks for ts such that
      a read ack reveals $S^{v}_{ts'}$ for $ts' < ts$ and $v \neq \perp$ AND
      the read acks reveal all secrets $S^{\perp}_{ts''}$ for $ts' < ts'' < ts$
    read-acks$[ts] := n - f$ read acks
  endif

procedure write(v,ts,secret-shares)
  broadcast $\langle$PRE-WRITE, $v, ts,$ secret-shares$\rangle$

when receive $\langle$WRITE-ACK, $v, ts\rangle$ **from** $n - f$ processes
  decide v

Responding to read and write messages:

on receive $\langle$READ, $ts\rangle_p$
  if ($ts = currTS$ AND $p \equiv ts \bmod n$)
    let $\perp$-shares be a map such that $\perp$-shares = $S^{\perp}_{t}s'$
      for $last.ts < ts' < ts$
    send $\langle$READ-ACK, $ts, last.v, last.ts, last.shrs, \perp -$ shares$\rangle_i$ **to**
      p
  endif

on receive $\langle$PRE-WRITE, $v, ts,$ secret-shares$\rangle_p$
  if (($p \equiv ts \bmod n$) AND
    ($ts \geq currTS$) AND
    (have not sent WRITE for ts) AND
    (($V^{v}_{ts'}[i]$ verifies secret-shares$[ts']$ for some $ts' < ts$ AND
     $V^{\perp}_{ts''}$ verifies secrets-shares$[ts'']$ for all $ts' < ts'' < ts$) OR
    ($V^{\perp}_{ts''}$ verifies secrets-shares$[ts'']$ for all $0 \leq ts'' < ts$)))
    if $ts > currTS$
      reset timeout
      $currTS := ts$
    endif
    broadcast $\langle$WRITE, $v, ts, S^{v}_{ts}[i]\rangle_a$
  endif

on receive $\langle$WRITE, $v, ts, S^{v}_{ts}[j]\rangle$ **from** j
  if already received WRITE for ts from j
    discard
  else if there exists $n - f$ writes for ts such that
    $V^{v}_{ts}[i]$ verifies that the shares in those writes reveal S^{v}_{ts}
    if ($ts \geq currTS$)
      reset timeout
      $currTS := ts$
    endif
    $last.v := v$
    $last.ts := ts$
    $last.shrs :=$ shares in the $n - f$ writes
    broadcast $\langle$WRITE-ACK, $v, ts\rangle$
  endif

at time $timeoutVal$
  $currTS := currTS + 1$
  $timeoutVal := 2 \times timeoutVal$
  $p := currTS \bmod n$
  send $\langle$TIMESTAMP-CHANGE, $currTS\rangle$ **to** process p

when receive $\langle$TIMESTAMP-CHANGE, $ts\rangle$ **from** $n - f$ processes
  if $ts > estTS$ AND $p \equiv ts \bmod n_p$
    $currTS := ts$
  endif

Figure 1: Information-Theoretically Secure Byzantine Paxos with an honest dealer.

timestamp, moving to the next timestamp and changing the primary role to a hopefully correct process. Because processes clocks may run at different speeds, processes exponentially increase the length of their timeouts so that eventually enough processes agree on a timestamp long enough to write a value.

Before a primary p for timestamp $ts > ts_0$ attempts to write a value, p first reads from the register. Process p sends $\langle \texttt{READ}, ts \rangle$ to all processses and then waits for $n - f$ acknowledgments, potentially abandoning the read if the timeout for ts expires. Process i accepts $\langle \texttt{READ}, ts \rangle$ only if i is currently in timestamp ts.

If i accepts $\langle \texttt{READ}, ts \rangle$, i responds with $\langle \texttt{READ-ACK}, ts, v', ts', S, Seq \rangle$ where:

- If i has never sent a WRITE-ACK then $S = v' = ts' = \texttt{NULL}$, and Seq is a sequence such that $Seq[ts''] = S^{\perp}_{ts''}[i]$ for $ts_0 \leq ts'' < ts$.
- If i has sent a WRITE-ACK then let v' and ts' be the value and timestamp, respectively, such that $\langle \texttt{WRITE-ACK}, v', ts' \rangle$ is the highest time stamped WRITE-ACK message that i sent. S is the set of $n - f$ parts that allowed i to reconstruct $S^{v'}_{ts'}$, and Seq is a sequence such that $Seq[ts''] = S^{\perp}_{ts''}[i]$ for $ts' < ts'' < ts$.

A leader p waits to receive $n - f$ read acknowledgments for timestamp ts that allow p to do either of the following:

case 1: Reconstruct $S^{v}_{ts'}$, $v \in \{0, 1\}$ and $ts' < ts$, and reconstruct every $S^{\perp}_{ts''}$ such that $ts' < ts'' < ts$

case 2: Reconstruct every $S^{\perp}_{ts''}$ such that $ts_0 \leq ts'' < ts$

If p accomplishes case 1 or 2 then p bundles the corresponding $n-f$ read acknowledgments into $proof$. If p accomplished case 1 then the read's value is v, and p sends $\langle \texttt{PRE-WRITE}, v, ts, proof \rangle$ to all processes. If p accomplished case 2 then the read's value is $\perp$ and p sends $\langle \texttt{PRE-WRITE}, inp, ts, proof \rangle$, where inp is p's input value.

A set of read acknowledgements $proof$ is *appropriate* in a message $\langle \texttt{PRE-WRITE}, v, ts, proof \rangle$ if they could have led p to send $\langle \texttt{PRE-WRITE}, v, ts, proof \rangle$. If a process i receives a PRE-WRITE with timestamp $ts > ts_0$, i checks that the attached proof is appropriate using the verification parts handed out by the dealer.

4.3 Secret Parts & Verification Parts

We now explain how the dealer splits each secret into n secret parts and n verification parts. We use the secret sharing of Tompa et al. [CITE] in which a participant who does not reveal his share can determine whether a secret corresponds to the actual secret

The intuition behind the technique of Tompa et al. is that each secret corresponds to a polynomial and each share corresponds to a random point on that polynomial. Even if an adversary knows the polynomial, that adversary has a negligible chance of guessing a second polynomial that happens to go through a participant's point. However, once a participant reveals his share, an adversary can fool that participant.

The dealer splits each secret S^{v}_{ts} into $n(k + 1)$ shares and parcels $k + 1$ shares out to each of the n processes. Each process i assembles k of those shares into its secret part $S^{v}_{ts}[i]$ and uses the one remaining share to implement $V^{v}_{ts}[i]$, never divulging that one share to any other process.

We now show that to meet the lower bound on the number of processes necessary for Byzantine consensus, $3f + 1$, our protocol requires that $k > f$. Let t be the threshold number of shares necessary and sufficient to reconstruct a secret. From the protocol, $n - f$ correct processes should be able to recon-

struct any secret S^v_{ts}, so $k(n-f) = t$. At the same time, f faulty processes should be unable to reconstruct $S^{v'}_{ts}$, $v' \neq v$, despite having the corresponding secret parts from the other f correct processes, so $kf + (k+1)f < t$.

Solving for n, we obtain $n > 3f + \frac{f}{k}$, showing that for any $k > 0$, there exists a lower bound on n necessary and sufficient for correctness. And when $k > f$, our protocol can match the $3f + 1$ lower bound on the number of processes for Byzantine consensus.

4.4 Correctness

We now show that the protocol presented in Section 4 solves the consensus problem. We define few predicates, as in [6], that are useful in the proof.

prepared(i, v, ts): Process i has received $(n - f)$ WRITE messages with value v and timestamp ts such that the secret reconstructed from the $(n - f)$ shares can be successfully verified using the $V^v_{ts}[i]$.

committed(v, ts): prepared(i, v, ts) is true for at least $(n - 2f)$ correct processes.

committed-local(i, v, ts): prepared(i, v, ts) is true and process i has received $(n - f)$ WRITE-ACK messages for value v and timestamp ts.

Lemma 1. *S^v_{ts} can be reconstructed only if at least $(n - 2f)$ non-faulty processes send the $\langle \mathtt{WRITE}, v, ts, S^v_{ts}[i] \rangle$*

Proof. S^v_{ts} is shared among processes such that at least $(n - f)$ shares are required to reconstruct the secret. If S^v_{ts} is to be reconstructed then at least $(n - 2f)$ non-faulty processes must reveal their secret parts, $S^v_{ts}[i]$. Non-faulty processes reveal the secret part $S^v_{ts}[i]$ if and only if they send $\langle \mathtt{WRITE}, v, ts, S^v_{ts}[i] \rangle$. □

Lemma 2. *if S^v_{ts} can be reconstructed, then $S^{v'}_{ts}$ cannot be reconstructed for $v' \neq v$.*

Proof. If both S^v_{ts} and $S^{v'}_{ts}$ are to be reconstructed, then there must be at least $(n - 2f)$ correct processes sending each of $\langle \mathtt{WRITE}, v, ts, S^v_{ts}[i] \rangle$ and $\langle \mathtt{WRITE}, v', ts, S^{v'}_{ts}[i] \rangle$. Since $n \geq 3f + 1$, this means that at least one correct process must both$\langle \mathtt{WRITE}, v, ts, S^v_{ts}[i] \rangle$ and $\langle \mathtt{WRITE}, v', ts, S^{v'}_{ts}[i] \rangle$. This cannot happen if $v \neq v'$ since correct processes only send one WRITE message for each timestamp. □

Lemma 3. *If prepared(i, v, ts) is true for a non-faulty i, then prepared(j, v', ts) is false for every non-faulty j and $v' \neq v$.*

Proof. prepared(i, v, ts) can hold for a non-faulty process only if S^v_{ts} can be reconstructed. Since $S^{v'}_{ts}$ cannot be reconstructed for $v' \neq v$, prepared(j, v', ts) is false for every non-faulty j. □

Lemma 4. *If committed-local(i, v, ts) is true for a non-faulty process i, then committed(v, ts) is true.*

Proof. By definition, committed-local(i, v, ts) requires that i must have received WRITE-ACK messages for value v and timestamp ts from at least $n-f$ processes. Since at least $n - 2f$ of these processes must be correct and would only send a WRITE-ACK message if prepared is true, committed(v, ts) should be true. □

Lemma 5. *If committed(v, ts) is true, then the secrets $S^{v'}_{ts}$ or $S^{\perp}_{ts}$ cannot be reconstructed for $v' \neq v$.*

Proof. If committed(v, ts) is true then prepared(i, v, ts) holds for at least $n - 2f$ correct processes. Since these processes will not give out their shares $S^{\perp}_{ts}[i]$ or $S^{v'}_{ts}[i]$, it follows that $S^{v'}_{ts}$ or $S^{\perp}_{ts}$ cannot be reconstructed. □

Lemma 6. *If committed(v, ts) holds for some v and ts then in any timestamp $ts' > ts$, a non-faulty process accepts a pre-write message only if the proposed value is v.*

Proof. For a pre-write to be accepted by a non-faulty process, the message must include a proof that the non-faulty process accepts.

We show that the primary cannot gather such a proof by induction on ts'.

Base case $ts' = ts + 1$: $S^{v'}_{ts}$ cannot be reconstructed. So, the proof will have to consist of either

- $S^{v'}_{ts"}$ for some $ts" < ts$, and $S^{\perp}_{t}$ for all $ts" < t < ts + 1$, or
- $S^{\perp}_{t}$ for all $ts_0 \leq t < ts + 1$,

In either case, the primary will have to reconstruct $S^{\perp}_{ts}$, which is not possible by lemma **??**.

Induction step: Assume that no correct non-faulty accepts any pre-write message for timestamps $ts"$ such that $ts < ts" \leq ts + k$.

The primary in $ts + k + 1$ cannot send all $S^{\perp}_{ts"}$ for $ts_0 \leq ts" < ts'$ because it cannot possibly reconstruct $S^{\perp}_{ts}$.

Since $S^{v'}_{ts}$ and $S^{\perp}_{ts}$ cannot be reconstructed, the only way to generate a proof would require $S^{v'}_{ts"}$ be reconstructed for some $ts" > ts$. This is not possible as no non-faulty processes would reveal $S^{v'}_{ts"}[i]$ unless it accepts a pre-write message for v' in timestamp $ts"$.

Thus, by induction, it follows that a primary cannot gather a proof to propose v' in any timestamp $ts' > ts$. □

Theorem 1 (Validity)**.** *If a non-faulty process decides v then v must have been proposed.*

Proof. Suppose that a non-faulty process decides on value v in timestamp ts_1. Then predicates committed-local and prepared must have been true for value v in timestamp ts_1.

Let t_0 be the smallest timestamp in which at least $(n-2f)$ correct processes send the WRITE message for value v. Since, non-faulty processes only send the WRITE message on receiving the PRE-WRITE message from the primary, primary must have either proposed v or should have sent a proof reconstructing S^v_t for some $t < t_0$. Since $t = t_0$ is the smallest timestamp when at least $(n - 2f)$ correct processes reveal their shares $S^v_t[i]$ (along with the WRITE message), it follows that the secret S^v_t cannot be reconstructed for any $t < t_0$.

Thus the primary in timestamp t_0 must have proposed v. □

Theorem 2 (Agreement)**.** *If a non-faulty process decides v then no non-faulty process can decide a different value.*

Proof. WLOG, let process i be the earliest process to decide. Say it decides on v in timestamp t. Then it follows that committed-local(i, v, t) is true.

Thus, by lemma **??** no non-faulty process can decide $v' \neq v$ in timestamp t. Also, by lemma **??** non-faulty process will not accept a pre-write message for $v' \neq v$ for any timestamp $ts' > t$. So no non-faulty process can decide on a different value v' in any timestamp $ts' > t$. □

Theorem 3 (Termination)**.** *If the system behaves synchronously in timestamp ts and the primary is non-faulty, then all processes will be able to decide on a value by the end of timestamp ts.*

Proof. When the system is synchronous, we assume that all correct messages sent by non-faulty processes to other non-faulty processes would be delivered before timeout.

Since the system is synchronous, the primary will be able to collect responses from all the non-faulty processes. Thus, the primary will either be able to reconstruct one of S^0_t or S^1_t or $S^{\perp}_t$ for every $ts_0 \leq t < ts$. Thus, the non-faulty primary will be able to gather the proof to be able to propose the appropriate value, v.

All non-faulty processes will send the WRITE message in response to the PRE-WRITE message, and all these messsages will be received by the non-faulty processes causing them to send out the WRITE-ACK message. Since the system is synchronous, the WRITE-ACK messages will also reach the non-faulty processes within the time out and all the non-faulty processes will be able to receive $n - f$ WRITE-ACK messages to be able to decide on v in timestamp ts if they have not already decided. □

5 Poly-Secret Protocol

5.1 Removing The Honest Dealer

We now show how to remove the honest dealer from our protocol.

We could entrust this role to an arbitrary process, but if that process were faulty, then it could distribute inconsistent shares and impede progress. One idea is to use verifiable secret sharing to check that distributed shares are consistent. These solutions, however, are costly and are still vulnerable if the dealer process divulges the secret to an adversary.

We sidestep this issue by observing that our protocol does not require the full power of secret sharing. Secret sharing protocols are traditionally used to share values, with the implicit assumption that the value has some significance. IT ByzPaxos is oblivious to a secret's value; we only care that the secret is hard to guess and whether it has been reconstructed or not.

We show how to implement a protocol that satisfies this weaker version of secret sharing, a version that we used in Section 4 to build IT ByzPaxos. Our protocol uses *poly-secrets*.

A poly-secret is a vector of n secrets, each secret being generated by a process in the system. To avoid confusion, we refer to these n secrets as *individual secrets*. We denote the poly-secret as S^v_{ts} and the i^{th} individual secrets of S^v_{ts} as $S^{v,i}_{ts}$. Process i generates the i^{th} individual secret for every poly-secret.

Individual secrets are random numbers. After generating an individual secret, a process divides that secret into secret and verification parts as per the honest dealer scheme (Section 4.3). Remember that a process distributes these parts over point-to-point private channels.

We refer to a vector of secret parts as a *poly-secret part*, and a vector of verification parts as a *poly-verification part*. Consistent with the description from Section 4, we denote the poly-secret part as S^v_{ts}i and the poly-verification part as $V^v_{ts}[i]$.

Poly-secret and poly-verification parts are either *complete* or*incomplete*. We say a part (secret or verification) is complete if it contains all the individual parts generated by correct nodes. Otherwise, the part is incomplete. Note that a process may not necessarily know whether the poly-part it has is complete or incomplete. If a process receives subsequent secret-parts for an already divulged poly-secret part, then that process adds the new parts to the poly-secret part and sends an update to its original message. These resends imply that a correct process who divulges a poly-secret-part eventually sends a complete poly-secret part.

A process accepts a poly-secret as *reconstructed* only if at least $f + 1$ individual secrets can be reconstructed correctly, i.e. can be verified using the corresponding verification parts. At least one secret that is reconstructed must have been generated and shared by a correct process, ensuring that the poly-secret can only be revealed if enough processes divulge their poly-secret parts.

The condition for a correct process to *claim* it has reconstructed a poly-secret is stronger than the condition to actually consider one reconstructed. A correct process makes such a claim only if it can reconstruct at least $2f + 1$ individual secrets. Thus, if a

correct process claims to have reconstructed a poly-secret, all other correct nodes will accept the claim.

5.2 Using poly-secrets

The high-level consensus protocol using poly-secrets is similar to the one presented in Section 4 that uses an honest dealer. Using poly-secrets introduces the folloming changes:

- Instead of requiring that an honest dealer share secrets processes divulge those shares, processes generate and distribute the poly-secret and divulge their poly-secret parts.
- Instead of verifying that a reconstructed secret is the correctvalue or not, we now require that processes check that enough individual secrets are revealed or not.

Since a poly-secret can be only be reconstructed if a correct processes individual secret is reconstructed, it follows that the poly-secret algorithm satisfies the safety requirements that the honest dealer protocol satisfies.

In terms of liveness, we rely on periods of synchrony to ensure that all correct processess receive the complete poly-shares before the processess time-out on the leader. This is not an additional constraint as the algorithm in Section 4 provides liveness guarantees only during periods of synchrony.

However, one issue is that in the honest dealer protocol, if a correct leader accepts that the secret is reconstructed then all correct servers will accept the secret as reconstructed. With poly-secrets, however it is possible that the correct leader accepts a poly-secret as reconstructed but other correct servers do not believe the same. If the process claiming to have revealed the poly-secret is faulty, it could make the leader accept this claim by reconstructing fewer than $f + 1$ individual secrets from correct processess, while ensuring that the remaining number of required secrets can only be accepted by the leader[2]. In this case, the correct leader may not be able to make progress as the processess will not accept the proof from the leader.

To solve this, we allow leaders to detect and henceforth ignore faulty processes. If a process rejects a proof from the leader because more than f individual secrets that were reconstructed are wrong, that process responds to the leader that it does not accept the poly-secret to be revealed. If more than f nodes (i.e. at least one correct node) do not accept the poly-secret to be revealed, then the leader infers that the process claiming to the leader that the poly-secret was revealed must have been faulty.

Correct leaders ignore processes they have detected as faulty. Thus, after at most f such failures for each leader, the leader ignores all faulty processes. If a leader then accepts a poly-secret to be revealed, then all other correct processes also accept the same.

References

[1] F. Bahr, M. Boehm, J. Franke, and T. Kleinjung. Rsa-200 is factored! http://www.rsa.com/rsalabs/node.asp?id=2879.

[2] G. Bracha. An asynchronous [(n - 1)/3]-resilient consensus protocol. In *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 154–162, New York, NY, USA, 1984. ACM Press.

[2]This can happen if these individual secrets are shared inconsistently by faulty processess

[3] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[4] R. Canetti and T. Rabin. Optimal Asynchronous Byzantine Agreement. Technical Report 92-15, TR 92-15, Dept. of Computer Science, Hebrew University, 1992.

[5] R. Canetti and T. Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 42–51, New York, NY, USA, 1993. ACM Press.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. 3rd OSDI*, pages 173–186, Feb. 1999.

[7] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. verifiable secret sharing and achieving simultaneity in the presence of faults. In *FOCS*, pages 383–395, 1985.

[8] P. Dutta, R. Guerraoui, and M. Vukolić. Best-case complexity of asynchronous Byzantine consensus. Technical Report EPFL/IC/200499, EPFL, Feb. 2005.

[9] P. Feldman and S. Micali. An optimal probabilistic algorithm for synchronous byzantine agreement. In *ICALP '89: Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, pages 341–378, London, UK, 1989. Springer-Verlag.

[10] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

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

[12] H. C. Li, A. Clement, A. Aiyer, and L. Alvisi. The Paxos Register. Technical Report TR-07-25, The University of Texas at Austin, May 2007.

[13] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 05), DCC Symposium*, pages 402–411, Yokohama, Japan, June 2005.

[14] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 73–85. ACM Press, 1989.

[15] A. Shamir. How to share a secret. *Comm. ACM*, 22(11):612–613, 1979.

[16] M. Tompa and H. Woll. How to share a secret with cheaters. In *Proceedings on Advances in cryptology—CRYPTO '86*, pages 261–265, London, UK, 1987. Springer-Verlag.