Consensus
Via the information theoretic lens
(Part 1)

Ittai Abraham, VMware Research

Group blog: Decentralized Thoughts
A fundamental problem that captures the essence of coordination in the face of failures
  • Multi Party Computation
  • Used in many large-scale compute infrastructures
  • Cryptocurrency and blockchain disruption

Deep connections between (information theoretic) cryptography and (information theoretic) distributed computing
  • Lower bounds for consensus are lower bounds for MPC
  • Broadcast (consensus) is used for MPC
  • MPC techniques are used for obtaining efficient (randomized) consensus protocols

My background:
  • I do research in algorithms and distributed computing
  • Wannabe Cryptographer
“The proof-of-work chain is a solution to the Byzantine Generals’ Problem. I’ll try to rephrase it in that context”

Satoshi Nakamo, email archive, 2008
“Bitcoin is the first practical solution to a longstanding problem in computer science called the Byzantine Generals Problem”

Marc Andreessen, Why Bitcoin Matters, NYT, 2014
Consensus: Approach for today and tomorrow

Traditional way to learn distributed computing and fault tolerance: learning isolated Islands

Today: a foundational view on traditional (and new) protocols
  • Not a historical survey
  • Not islands, highlight connections
  • Understanding the connections allows better abstractions, theory, protocols, systems

Why via the information theoretic lens?
  • Everything should be made as simple as possible, but not simpler

On Learning
  • First via intuition then via rigor
  • Learning by asking
  • Learning by doing (no shortcuts)
Consensus: plan for today and tomorrow

Focus on information theoretic solutions

A call for multidisciplinary research

Adversary and Network Models

Consensus: definitions, upper and lower bounds

Paxos (Synchrony) → GradeCast

Byzantine Paxos (Synchrony) → Multiword + MVSS + RandElect

O(1) exp time Byzantine Paxos (Synchrony)

Paxos (Partial Synchrony) → Reliable Broadcast

Byzantine Paxos (Partial Synchrony) → A-MW + A-VSS + ARandElect

O(1) exp time Byzantine Paxos (Asynchrony)
Political science, Public policy, Law
Study social norms, institutions, regulations

Study of protocols that replace Trusted Third Parties

Economics
Distributed Computing
Cryptography
Governance

Study of decision making with scarce resources and in response to incentives

Study how multi party and large-scale systems can overcome network delays and tolerate failures
Distributed Computing 101

Synchrony, Asynchrony and Partial synchrony and flavors of Partial Synchrony

Asynchrony: adversary can delay messages by any finite amount

Synchrony: adversary can delay messages by some known $\Delta$
  • lock step: all messages take exactly $\Delta=1$

[DLS88]: Partial Synchrony (Global Stabilization Time):
  • adversary can delay messages by any finite amount
  • until some unknown finite point in time called GST (Global Stabilization Time)
  • adversary can delay messages by some known $\Delta$

[DLS88]: Partial Synchrony (Unknown Latency):
  • adversary must set $\Delta$ at the beginning of the execution
Power of the Adversary

Passive adversary (semi honest, honest-but-curious)
Crash failure
Omission failure ("bubble adversary")
Byzantine failure (malicious)

- Covert (malicious but does not want to be detected)
- $\epsilon$-covert (malicious but only if probability of detection is low)
Parties have initial input

Can send messages via point-to-point channels

*Termination (Liveness):* In the end of the protocol each party must *decide* on a value

*Safety:* No two non-malicious parties decide on different values

Trivial: Always decide a default value

Make the problem not trivial:

- *Validity:* If all the non-faulty have the same input, then this must be the decision value
- *Fair Validity:* With constant probability an input of a non-faulty server is decided upon

Nor required:

- *Security:* that the view of the adversary in the ideal world is indistinguishable from a simulated view generated from the view of the adversary in the real world
Consensus: Broadcast vs Agreement

Safety: all non-malicious parties decide the same value

Liveness: all non-faulty parties eventually decide

**Broadcast.**
- Designated sender P*
- Validity: if the sender is non-faulty with input m, then m is the decision value

**Agreement.**
- *Validity*: If all the non-faulty have the same input, then this must be the decision value
- *Fair Validity*: With constant probability an input of a non-faulty server is decided upon

Broadcast from Agreement (in synchrony):
- Given agreement, sender sends to to all, then parties run agreement

Agreement from Broadcast (in synchrony):
- Given broadcast (and f<\(n/2\)), each party broadcasts its input, then use say majority

Goal:
- Upper bounds for Agreement
- Lower bounds for Broadcast
## Consensus results in one slide: deterministic

<table>
<thead>
<tr>
<th></th>
<th>Synchrony</th>
<th>Partial Synchrony</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash</td>
<td>n&gt;f (primary backup)</td>
<td>n≤2f (DLS “split”)</td>
</tr>
<tr>
<td>Ommision</td>
<td>n≤2f (uniform)</td>
<td>n&gt;2f (Sync Paxos)</td>
</tr>
<tr>
<td>Byzantine (cannot simulate)</td>
<td>n&gt;2f (Auth Byz)</td>
<td>n≤3f (DLS “split”)</td>
</tr>
<tr>
<td>Byzantine (unbounded)</td>
<td>n≤3f (FLM the “hexagon”)</td>
<td>n&gt;3f (Sync Byz)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>n&gt;3f (PBFT)</td>
</tr>
</tbody>
</table>

- **FLP85**: every protocol solving asynchronous consensus for 1 crash must have an infinite execution
- **LF82**: every protocol solving synchronous consensus for f crashes must have a f+1 round execution
- **DR82**: deterministic consensus needs $\Omega(f^2)$ messages
**Consensus results in one slide: randomized, with private channels**

<table>
<thead>
<tr>
<th></th>
<th>Synchrony</th>
<th>Partial Synchrony</th>
<th>Asynchrony</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Crash</strong></td>
<td>n&gt;f (primary backup)</td>
<td>n≤2f (DLS88 “split brain”)</td>
<td></td>
</tr>
<tr>
<td><strong>Omission</strong></td>
<td>n≤2f (uniform)</td>
<td>n&gt;2f, O(1) expected time</td>
<td>n&gt;2f, O(1) expected time</td>
</tr>
<tr>
<td><strong>Byzantine (cannot simulate)</strong></td>
<td>n&gt;2f, Auth, O(1) exp. Time (KK06)</td>
<td>n≤3f (DLS88 “split brain”)</td>
<td></td>
</tr>
<tr>
<td><strong>Byzantine (unbounded)</strong></td>
<td>n≤3f (FLM86 the “hexagon”)</td>
<td>n&gt;3f, O(1) expected time</td>
<td>VSS n≤4f must have error (BKR94)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1. n&gt;4f, O(1) expected “time” (BCG93)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2. n&gt;3f, error, O(1) (CR93)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3. n&gt;3f, no error, poly exp. “time” (ADH)</td>
</tr>
</tbody>
</table>
Primary-Backup in the omission model [Lamport, Oki Liskov, DLS,]

The omission model
- There are $n$ replicas
- The adversary corrupts $f$ replicas which can fail by not receiving or not sending each message

Systems works in views, in each view
- One replica is designated as Primary
- All the rest of the replicas are Backups

For simplicity: in view $i$ the primary is $(i \mod n)$

Many other options:
- Randomized leader election
- Back-off protocols
Primary-Backup in the omission model: Lower bound for $n \leq 2f$ (DLS 88)

$n=2$ and one omission failure

1. In Partial synchrony
2. In Synchrony, assuming *uniform* consensus
   - Safety for omission faulty parties
Learning by Doing

3 parties, each with input in \{0,1\}

Adversary controls one party (omission)

Write a protocol for consensus:

- (Uniform) Safety: no two decide different values
- Liveness: All non-faulty parties decide
- Validity: If all the non-faulty have the same input \(x\), then \(x\) is the decision value
Primary-Backup in the omission model: Foundations

The only math you will need:

• Quorum intersection (pigeonhole principle)
• Given a set of $n$ elements: two sub-sets of $n-f$ elements must intersect at $n-2f$ elements

• For $n=2f+1$, any two sets of $f+1$ must intersect at one element
• For $n=3f+1$, any two sets of $2f+1$ must intersect at $f+1$ elements
Primary-Backup in the omission model: What could possibly go wrong?

Primary chooses its input: $x$
- decide $x$
- Sends $<\text{decide } x>$ to all replicas

Primary chooses its input: $x$
- Sends $<\text{propose } x>$ to all replicas
- decide $x$

Main challenge: the first primary may decide $x$, but the next primary decides $x'$
Primary-Backup in the omission model: View Change protocol

Use a **view change** protocol to guarantee safety:
- Before a new primary starts, it runs a view change protocol
- If there is any possibility that some value was previously decided, the new primary must adopt that value

Three challenges:
1. Only decide a value after you are sure later primaries can recover and adopt this value
2. Make the view change safe: only choose safe values to adopt
3. Make the view change live: don’t get stuck waiting
Primary-Backup: Algorithm structure – three simple parts!

1. Normal case protocol
   – allow the primary to decide

2. View change trigger protocol
   – trigger the replacement of a primary

3. View change protocol
   – a way for a new primary to make safe choices
Primary-Backup in the omission model:
Normal case

1. Send:
   • Primary (of view v) sends \(<\text{propose } x \text{ in view } v>\) to all replicas

2. Ack:
   • Replica sends \(<\text{ack } x \text{ in view } v>\) to all
     • Unless it has moved to a higher view

3. Decide:
   • Replica wait for \(n-f\) messages of \(<\text{ack } x \text{ in view } v>\) to decide \(x\)
View Change Trigger: Revolving coordinator, random leader, stable leader

View change to replace a failed primary

• Use synchronized heartbeat mechanisms to have all replicas move to the next view
• For now: simple revolving coordinator
• Later: random leader election
• In practice: use a stable leader for many consensus decisions
View Change

Maybe the previous primary caused a decision?
Maybe one of the previous primary caused a decision?

New primary may need to adopt a value instead of choosing its own

Quorum intersection to the rescue:
- If some primary decided, then it used a write quorum (of n-f)
- So reading from a quorum of n-f:
  - Is safe: primary will see intersection (since n-2f>0)
  - Is live: can always be done
Primary-Backup in the omission model: Normal case

1. Send:
   • Primary (of view v) sends <propose x in view v> to all replicas

2. Ack:
   • Replica sends <ack x in view v> to all
     • Unless it has moved to a higher view

3. Decide:
   • Replica wait for n-f messages of <ack x in view v> to decide x
View Change:
from view $v$ to view $v+1$

New primary for view $v+1$:
• (Send message `<view change for view v+1> to all`)
• A replica responds with `<my maximal propose is x’ at view v’>`
  – Using the `propose` with maximal view $v’$ it heard
  – Or send `<null at view 0>` if heard no propose

Primary waits for $n-f=f+1$ responses:
• **Adopts** the proposed value associated with the *maximal view number*, or
• Uses its own value if every message is `<null at view 0>`
Primary-Backup in the omission model for n>2f

Three simple parts

1. Normal case protocol
   – Send: Primary (of view v) sends <propose x in view v> to all replicas
   – Ack: Replicas send <ack x in view v> to all (update their maximal propose)
     • Unless it has moved to a higher view
   – Decide: Replicas wait for n-f messages of <ack x in view v> to decide x

2. View change trigger protocol
   – Revolving coordinator: wait for enough time (4 rounds) to replace primary with next primary

3. View change protocol
   – Each replica sends to new primary <my maximal propose is x’ at view v’>
     • Using the propose with maximal view v’ it heard
     • Or send <null at view 0> if heard no propose
   – Primary waits for n-f responses:
     • Adopts the proposed value associated with the maximal view number; or
     • Uses its own value if every message is <null at view 0>
Safety

Let \( v^* \) be the first view that some replica decides, say on value \( x \)

Base case: all decisions in view \( v^* \) must be to \( x \)

By induction on \( v > v^* \): any primary must adopt the value \( x \)

- Set \( G \) of \( f+1 \):
  - Each member of \( G \): maximal propose is on value \( x \)
  - Each member outside of \( G \): has an equal or higher maximal propose than any member of \( G \), then it must be on value \( x \)

This argument does not use synchrony! It works for asynchrony
Termination (liveness)

Claim: Eventually all non-faulty replicas will learn the decision value
Any faulty primary that does not make progress will eventually be replaced
A non-faulty primary will cause termination
(here we use synchrony)
Primary-Backup in Partial Synchrony

Asynchrony: adversary can delay messages by any finite amount

Synchrony: adversary can delay messages by some known finite value $\Delta$

Partial Synchrony:
- adversary can delay messages by any finite amount
- until some unknown finite point in time called GST (Global Stabilization Time)
- adversary can delay messages by some known finite value $\Delta$

The Partial Synchrony paradigm:
- Safety holds in asynchrony
- Termination holds in synchrony
- Extremely successful in industry
- Gateway to asynchrony
Byzantine Adversaries!

Can we reach agreement in synchrony for $n=2f+1$?
Can we reach agreement in partial synchrony for $n=2f+1$?
Byzantine adversaries

n=3, f=1 is impossible

In partial synchrony, the split-brain attack [DLS]:

In synchrony, the hexagon [FLM]:
- Any edge defines a legal world with two non-faulty parties around edge
- Non-faulty party decide the same for left edge and right edge worlds
Byzantine Model in Partial Synchrony

Two primary attacks:
- Equivocate: tell different replicas different things
- Unsafe: adopt a non-safe value after view change
  - Invent a value
  - Choose a non-maximal value

Solution approach:
- Add a sub-protocol to force primary to act like omission (no equivocation)
- Add a sub-protocol to guarantee the primary will fail if using un-safe values
  - Key idea: replica that sent a value lock on it, primary has to prove value is real
Byzantine Primary-Backup (at view v):

Straw Man 1: with n=3f+1, what could possibly go wrong?
Primary can send different values to different replicas ⊗, need to block equivocation

1. Primary sends <send, (value, v)> to all

2. Replica accepts <send, (value, v)>, then
   • Set lock:=v; lock value:=value
   • Sends <lock, (value, v)> to all

3. Replica gathers n-f <lock, (value, v)>, then
   • Decide (value)

The good: cannot decide different values

The bad: If non-faulty commits, there may be conflicting locks for the view change
   • How do we choose which one?
   • Want all the locks to be the same
Non-Equivocation:

Goal: given a (potentially) Byzantine primary, transform its send-to-all to a (potentially) omission fault primary send-to-all

$n > 3f$

1. Primary sends $\langle \text{send} (value, v) \rangle$ to all

2. Replica sends $\langle \text{echo} (value, v) \rangle$ to all for the first $\langle \text{send} (value, v) \rangle$ it hears from primary

3. If a replica sees n-f $\langle \text{echo} (value, v), proof \rangle$ from different replicas, then it accepts $\langle \text{send} (value, v) \rangle$
Non-Equivocation: Proof

Claim: If a replica accepts \(<send\ (value,v) >\) then no replica will accept \(<send,(value',v)>\) with \(value \neq value'\)

Proof by contradiction:

1. One replica sees \(n-f<echo\ (value,v)>\) and another sees \(n-f<echo\ (value',v)>\)
2. The intersection is at least \(f+1\), so at least one non-faulty in the intersection
3. Non-faulty will send at most one echo per view
Byzantine Primary-Backup (at view $v$):

Straw Man 2: with equivocation
Primary can send any value it wants ☺, how can we protect a decision value?

1. Primary sends \(<send, (value, v)>\) to all

2. Replica receives \(<send, (value, v)>\), then
   - If first send from primary in view $v$, then
   - sends \(<echo, (value, v)>\) to all

3. Replica gathers \(n-f <echo, (value, v)>\), then
   - Set lock:=v; lock value:=value
   - Sends \(<lock, (value, v)>\) to all

4. Replica gathers \(n-f <lock, (value, v)>\), then
   - Decide (value)

The good: all locks will be the same

The bad: how do we force the new primary to choose the highest lock?
Recall: View Change from view $v$ to view $v+1$

New primary for view $v+1$:
- A replica responds with $<\text{my maximal propose is } x' \text{ at view } v'>$
  - Using the propose with maximal view $v'$ it heard
  - Or send $<\text{null at view } 0>$ if heard no propose

Primary waits for $n-f$ responses:
- Adopts the proposed value associated with the maximal view number, or
- Uses its own value if every message is $<\text{null at view } 0>$

Can we force new primary to adopt the maximum value?
- Information theoretically possible, a PBFT type view change (see Castro’s thesis)

Can Primary prove the (value) its using was indeed sent in some view $u<v$?
- Yes, this will allow a Tendermint, HotStuff type view change

Safety:
- Replica that is locked on (value, $v$) will Ignore primary with (value’, $v'$) if $v'<v$
- $f+1$ locked replicas will block a malicious primary
Force primary to prove: the (value) its using was indeed sent in some view u<v

Primary of view u could sign its message!
• We don’t have signatures 😞
We have non-equivocation on primary, would like stronger property:
• If I accept the primary message then all parties weakly accept (and eventually accept it)
• Bracha’s Reliable Broadcast, (Micali and Feldmans’s Gradecast)

1. Primary sends <send (value,v)> to all
2. Replica sends <echo1 (value,v)> to all for first <send (value,v)> it hears from primary
3. If a replica sees n-f <echo1 (value,v) ,proof> from different replicas,
   • then it sends <echo2 (value,v)> to all
4. If a replica sees n-f <echo2 (value,v) ,proof> from different replicas,
   • then it accepts <send (value,v)>
5. If a replica sees f+1 <echo2 (value,v) ,proof> from different replicas,
   • then it weakly accepts and sends <echo2 (value,v)> to all
Reliable Broadcast (at view v):

1. Primary sends `<send (value,v)>` to all
2. Replica sends `<echo1 (value,v)>` to all for first `<send (value,v)>` it hears in view v from Primary
3. If a replica sees n-f `<echo1 (value,v),proof> from different replicas`,
   • then it sends `<echo2 (value,v)>` to all
4. If a replica sees n-f `<echo2 (value,v),proof> from different replicas`,
   • then it accepts `<send (value,v)>`
5. If a replica sees f+1 `<echo2 (value,v),proof> from different replicas`,
   • then it weakly accepts and sends `<echo2 (value,v)>` to all

Claim 0: all accepted values are the same (non-equivocation)
Claim 1: If a non-faulty accepts (in synchrony), then all non-faulty will at least weakly accept
Claim 2: If a non-faulty accepts (in asynchrony), then all non-faulty will eventually accept
Byzantine Primary-Backup (at view v):

Straw Man 3: with Reliable Broadcast

Primary can prove its using a real value

1. Primary sends $<send, (value, v)>$ to all

2. Replica receives $<send, (value, v)>$, then
   • If first send from primary in view v, then
   • sends: $<echo1, (value, v)>$ to all

3. Replica gathers $n-f <echo1, (value, v)>$, then
   • Sends $<echo2, (value, v)>$ to all

4. Replica gathers $n-f <echo2 (value, v)>$, then (at view v)
   • Set lock:=v; lock value:=value
   • Sends $<lock, (value, v)>$ to all

5. Replica gathers $n-f <lock, (value, v)>$, then
   • Decide (value)

Replica gathers $f+1 <echo2, (value, v)>$, then
   • If did not send echo2
   • Sends $<echo2, (value, v)>$ to all

View change:
   • **Replica**:
     • Sends its lock and lock value
   
   • **Primary**:
     • accept a lock $(value',v')$ if also $n-f <echo2, (value, v)>$ arrive
     • Wait for $n-f$ such locks
     • Choose the value with the highest lock (view)
Byzantine Primary-Backup (at view v):
with Reliable Broadcast and locking

1. Primary sends <send, (value, v, u)> to all
2. Replica receives <send, (value, v, u)>,
   • If \( u \geq \text{lock} \), n-f <echo2, (value, u)> arrive, and first send from primary in view v, then
     - sends: <echo1, (value, v)> to all
3. Replica gathers n-f <echo1, (value, v)>, then
   • Sends <echo2, (value, v)> to all
4. Replica gathers n-f <echo2 (value, v)>, then (at view v)
   • Set lock:=v; lock value:=value
   • Sends <lock, (value, v)> to all
5. Replica gathers n-f <lock, (value, v)>, then
   • Decide (value)

Replica gathers \( f+1 <\text{echo2}, (value, v)> \), then
   • If did not send echo2
   • Sends <echo2, (value, v)> to all

View change:
   • **Replica:**
     • Sends its lock and lock value
   • **Primary:**
     • accept a lock (value',v') if also n-f <echo2, (value, v)> arrive
     • Wait for n-f such locks
     • Choose the value with the highest lock (view)
Safety

Let $v^*$ be the first view that any replica decided (value $X$, view $v^*$)

Prove by induction that any accepted send of view $v \geq v^*$ must be consistent with value $X$
  • for base case due to non-equivocation

Induction argument:
  • Existence of a *core* of $f+1$ non-faulty that have a lock on view at least $v^*$ with value $X$
    – Base case: core is the $n-2f$ out of the $n-f$ that sent a lock to decider
  • Any accepted value from a primary of view at least $v^*$ must be $X$
    – By induction, core will block any other value
    – Core members can only gain a higher lock but then primary uses the same value.
Liveness

If a non-faulty primary is elected and the system is synchronous
Primary will hear locks from *all* non-faulty and will choose the maximum one
All non-faulty replicas will also see same lock and hence will echo1 the primary
Responsiveness: liveness in asynchrony

In asynchrony the non-faulty primary can wait for n-f responses during view change.

May miss a lock of a non-faulty
  • Will cause a livenss problem!

Solution: add one more round 😊
  • After seeing n-f echo2, send **key**
  • After seeing n-f keys, send **lock**
  • If I have a lock then there are at least f+1 non-faulty that have a key
  • During view change, ask for keys
Responsive Byzantine Primary-Backup (at view $v$):

Information Theoretic HotStuff

1. Primary sends $<\text{send}, (value, v, u)>$ to all

2. Replica receives $<\text{send}, (value, v, u)>$,
   - If $u > \text{lock}$, $n-f <\text{echo2}, (value, u)>$ arrive, and first send from primary in view $v$, then
     - sends $<\text{echo1}, (value, v)>$ to all

3. Replica gathers $n-f <\text{echo1}, (value, v)>$, then
   - Sends $<\text{echo2}, (value, v)>$ to all

4. Replica gathers $n-f <\text{echo2} (value, v)>$, then (at view $v$)
   - Set $\text{key} := v$; $\text{key value} := \text{value}$
   - Sends $<\text{key}, (value, v)>$ to all

5. Replica gathers $n-f <\text{key}, (value, v)>$ and $n-f <\text{echo2} (value, v)>$, then (at view $v$)
   - Set $\text{lock} := v$
   - Sends $<\text{lock}, (value, v)>$ to all

6. Replica gathers $n-f <\text{lock}, (value, v)>$, then Decide (value)

Replica gathers $f+1 <\text{echo2}, (value, v)>$, then
- If did not send echo2
- Sends $<\text{echo2}, (value, v)>$ to all

View change:
- **Replica**:
  - Sends its key and key value
- **Primary**:
  - accept a key $(\text{value'},\text{v'})$ if also $n-f <\text{echo2}, (value, v)>$ arrive
  - Wait for $n-f$ such key
  - Choose the value with the highest key (view)
Revolving coordinator
• After f view changes (O(f) rounds) a non-faulty primary will be elected

Assume we have a **oblivious leader election** functionality
• At least f+1 honest must request for functionality to start
• Each party i outputs a leader L(i)=j
• With probability at least \( \frac{1}{2} \) (can use any constant):
  – all non-faulty output the same value j and,
  – j was non-faulty before functionality started

Good for a static adversary

Adaptive adversary will adaptively corrupt that chosen primary 😐
Byzantine Paxos: adaptive adversaries

Everyone is a Primary 😊

Adaptive adversary will shoot down the primary

Solution:
- Let everyone be a primary
- Then choose who the real primary is in hindsight (and all other are just decoys)

Liveness: with constant probability a good primary is chosen

Safety:
- In hindsight, looks like a single primary each view
- If a faulty primary or a confusion of primaries is chosen then this is just like a faulty primary
  - Safety is maintained!
Responsive Byzantine Primary-Backup (at view v):

**Deterministic version**

1. Primary sends \(<send, (value, v, u)>\) to all

2. Replica receives \(<send, (value, v, u)>\),
   - If \(u >= lock\), n-f \(<echo2, (value, u)>\) arrive, and first send from primary in view v, then
     - Sends \(<echo1, (value, v)>\)

3. Replica gathers \(n-f <echo1, (value, v)>\), then
   - Sends \(<echo2, (value, v)>\)

4. Replica gathers \(n-f <echo2 (value, v)>\), then (at view v)
   - Set key:=v; key value:=value
   - Sends \(<key, (value, v)>\)

5. Replica gathers \(n-f <key, (value, v)>\) and \(n-f <echo2 (value, v)>\), then (at view v)
   - Set lock:=v
   - Sends \(<lock, (value, v)>\)

6. Replica gathers \(n-f <lock, (value, v)>\), then
   - Decide (value)

   **Replica gathers** \(f+1 <echo2, (value, v)>\), then
   - If did not send echo2
   - Sends \(<echo2, (value, v)>\)

**View change:**

- **Replica**:
  - Sends its key and key value

- **Primary**:
  - accept a key \((value',v')\) if also n-f \(<echo2, (value, v)>\) arrive
  - Wait for n-f such key
  - Choose the value with the highest key (view)
Responsive Byzantine Primary-Backup (at view v):
with random leader election

1. Each party as Primary, sends \(<send, (value, v, u)\)> to all
2. Run oblivious leader election to decide who to listen to
3. Replica receives \(<send, (value, v, u)\>
   • If \(u\geq\text{lock}\), n-f \(<\text{echo2}, (value, u)\>) arrive, and first send from primary in view v, then
     - sends \(<\text{echo1}, (value, v)\>) to all
4. Replica gathers n-f \(<\text{echo1}, (value, v)\>\), then
   • Sends \(<\text{echo2}, (value, v)\>) to all
5. Replica gathers n-f \(<\text{echo2}, (value, v)\>\), then (at view v)
   • Set key:=v; key value:=value
   • Sends \(<\text{key}, (value, v)\>) to all
6. Replica gathers n-f \(<\text{key}, (value, v)\>) and n-f \(<\text{echo2}, (value, v)\>\), then (at view v)
   • Set lock:=v
   • Sends \(<\text{lock}, (value, v)\>) to all
7. Replica gathers n-f \(<\text{lock}, (value, v)\>\), then
   • Decide (value)

Replica gathers f+1 \(<\text{echo2}, (value, v)\>) ,\ then
   • If did not send echo2
   • Sends \(<\text{echo2}, (value, v)\>) to all

View change:
   • Replica:
     • Sends its key and key value
   • Primary:
     • accept a key (value’,v’) if also n-f \(<\text{echo2}, (value, v)\>) arrive
     • Wait for n-f such key
     • Choose the value with the highest key (view)
Oblivious Leader Election

Choosing a random leader is a simple MPC protocol
But MPC uses VSS, and VSS requires broadcast 😞

Solution:
• a notion that is weaker than VSS but strong enough for OLE
• Moderated VSS (KK06) and Graded VSS (MF88)
• Tailor made MPC (with a constant error probability)

Gradecast -> MVSS -> OLE -> O(1) time expected Byzantine Agreement
Gradecast (MF88, D81)

Dealer P* has input m
Each party outputs a value m and a grade in \{0,1,2\}
If the dealer is no-faulty then all non-faulty output (m,2)
If a non-faulty outputs (m’,2) then all non-faulty output (m’,g) with g>0
(If two non-faulty have grade 1 then have same value)
Gradecast protocol (MF88)

round 1: Dealer $P^*$ <sends $m$> to all

round 2: Party sends <echo1 $m$> to the first message it receives from the primary

round 3: If party gathers n-f echo1 it sends <echo2 $m$>

End of round 3:

- Grade 2: If party gathers n-f echo2; otherwise
- Grade 1: if party gathers f+1 echo2; otherwise
- Gradecast 0 (default value)
Gradecast proof (MF88)

round 1: Dealer P* <sends m> to all
round 2: Party sends <echo1 m> to the first message it receives from the primary
round 3: If party gathers n-f echo1 it sends <echo2 m>
End of round 3:
  • Grade 2: If party gathers n-f echo2; otherwise
  • Grade 1: if party gathers f+1 echo2; otherwise
  • Grade 0 (default value)

Echo1 causes non-equivocation -> any two grade 1 must have same value
Non-faulty dealer -> all non-faulty have (m,2)
Non-faulty has (m’,2) -> all nonfaulty have at least f+1 echo2 -> all non-faulty have (m,g) with g>0
Moderated VSS [KK06]

MVSS from VSS

Dealer P*

Moderator P**

Take any VSS that uses broadcast only in share phase

Replace <broadcast m by party j> with:
• Party j runs gradecast (m)
• The moderator P** takes the value m’ of the gradecast and runs gradecast (m’)

Outcome for party i:
• Let (m,g) be the outcome of the first gradecast
• Let (m’,g’) be the outcome of the first gradecast
• If g’<2 or (g’=2 and g=2 and m≠m’) then set OK=false
Proof for Moderated VSS

If OK=true for any non-faulty then VSS properties hold
  • Because all see the moderator’s value and the moderator's value is consistent with any non-faulty broadcaster

If the moderator is non-faulty then all non-faulty have OK=true
  • From the grade cast properties of an honest sender
Oblivious Leader Election

OLE from MVSS

For each i, j, do a MVSS with dealer i and moderator j (say random value in n^4)
The secret ballot for j will be the sum mod n^4 of all the VSS where j is a moderator
Reveal all the secret ballots for all parties
But if for some moderator j you see OK=false in any MVSS then set secret ballot to 0
Choose the leader to be the party with the highest secret ballot
With large probability there are no collisions, and then with constant probability a non-faulty is elected
Moving to asynchrony

Responsiveness: we added a key round

MVSS does not work:

- $n > 4f$, constant time [MF]
- AVSS constant time, but has non-zero deadlock [CR]
- Shunning AVSS no deadlock but polynomial time [ADH]

Attach $f+1$ secrets. Honest attach only after the RB works
Liveness even in asynchrony?

Primary-Backup and Byzantine Primary-Backup:
  • Always safe; live when system is synchronous

Problems with asynchrony:
  • Adversary can attack the primary
  • Adversary can delay the primary
  • Cannot tell the difference
  • Choose a random leader?
  • Works for a static adversary
  • Replace leaders quickly: works for an adaptive adversary that is slow
  • What about an adaptive adversary that is not slow?
Asynchrony: Lower bounds and solutions

1985: Fischer, Lynch and Patterson:
• Impossible to decide on one command even with f=1 crash failure
• For any safe protocol there is an adversary strategy (on delays) that forces the protocol to make an infinite number of steps (never terminate)

Solutions:
• Assume eventually the system is synchronous (so no progress in DDoS)
• Use randomization so the infinite execution have probability (measure) 0
• In fact O(1) expected rounds!

Building State Machine Replication
• Weak validity: is not enough assuming asynchronous client communication
• Binary agreement is not enough (can be a building block)
• External Validity [CKPS 01] is key for SMR implementation
Start the election after \( n-f \) are done [AMS PODC19]

Primary i gets a proof that \( n-f \) learned it’s commit decision
  • call this a \textit{done-proof}, sends singed \(<\textit{done } (\textit{done-proof})>\),

Barrier: Start leader election after seeing \( n-f \) valid \(<\textit{done } (*)>\) messages

Safety does not change

Liveness:
  • With constant probability we chose a primary that made progress!
Thank you