Practical Byzantine Fault-Tolerance

Pedro F. Souto (pfs@fe.up.pt)

January 17, 2022

Roadmap

Impossibility of consensus with a faulty process

Consensus problem Each process starts with an input value from a set V and must decide a value from V.

Safety properties

- **Agreement** no two correct processes decide differently
- **Validity** the decided value depends on the input values of the processes

Liveness every execution of a protocol decides a value

Theorem In an **asynchronous** system in which **at least one process may crash**, there is no **deterministic** consensus protocol that is **both** live and safe.

- Even if the network is reliable, i.e. does not loose messages
- This was proven by Fischer, Lynch and Patterson, and is often referred to as **FLP's impossibility result**

Intuition-based Argument

- In an asynchronous distributed system we cannot distinguish a slow process from a crashed process
- For every consensus algorithm, there are executions in which the algorithm reaches a state such that:
  - If a process takes no decision, it may remain forever undecided, thus violating liveness
  - If a process makes a decision, independently of the decision rule, it may violate one of the safety properties
- Paxos favors safety at the cost of liveness
  - It never violates safety
  - But it may block, if it is unable to elect a **single** leader

Proof (1/2)

Idea Show that a deterministic protocol, $\mathcal{P}$, that satisfies both:

Safety i.e. agreement and validity

Liveness i.e. that a non-faulty process eventually decides

under the model considered can execute forever without a process ever taking a decision (thus violating liveness)

Implementation

Specification of a very weak agreement problem

- The value to decide is a boolean variable
- Validity is very weak

Computation model basic, in which each process executes a sequence of atomic steps:

1. Receives 0 or 1 messages, previously sent, **event**
2. Performs some computation and sends a finite number of messages to other processes

Run is a sequence of steps executed by different processes

Schedule is a sequence of events associated to a run

Proof (2/2)

Configuration is essentially a global state

Bivalent if the set of decided values of reachable configurations is $\{0, 1\}$

Lemma 1 Two schedules, each by disjoint processes, applicable to a configuration commute and lead to the same configuration

Lemma 2 $\mathcal{P}$ has an initial bivalent configuration

Lemma 3 If an event e is applicable to a bivalent configuration C, there is a bivalent configuration D reachable from C by executing the steps corresponding to a schedule where e is the last event

Non-deciding run can be built, with the help of Lemmas 2 and 3, in stages, each starting/finishing from/with a bivalent configuration

- The first stage starts from an initial bivalent configuration
- In each stage, we pick a schedule that ends with some event applicable to the initial configuration of the stage
- Note that the final event of each stage is chosen such that processes take turns

Roadmap

System Model (Asynchronous system)

Network may

- fail to deliver messages
- delay messages
- duplicate messages
- deliver messages out of order

Nodes/processes may fail arbitrarily

- This is known as the Byzantine failure model

but independently

Processes use cryptographic techniques to:

- prevent spoofing and replays
- detect corrupted messages

Notation

$D(m)$ digest of message m

$\langle m \rangle_{\sigma_i}$ message m signed by process i

- Implemented by signing its digest ($D(m)$)
- Assumes that every process knows the others' public keys

Service Properties

State machine replication i.e. a replicated service with a state and some operations

- Each replica:
  - Maintains the service state (initially, the same for all of them)
  - Executes operations, which must be deterministic

Safety the replicated service satisfies linearizability

- Essentially, it behaves as a non-replicated service that executes operations atomically, one at a time

Liveness cannot be assured in an asynchronous system

Resiliency Tolerates f faulty replicas with $n = 3f + 1$ replicas. This is optimal:

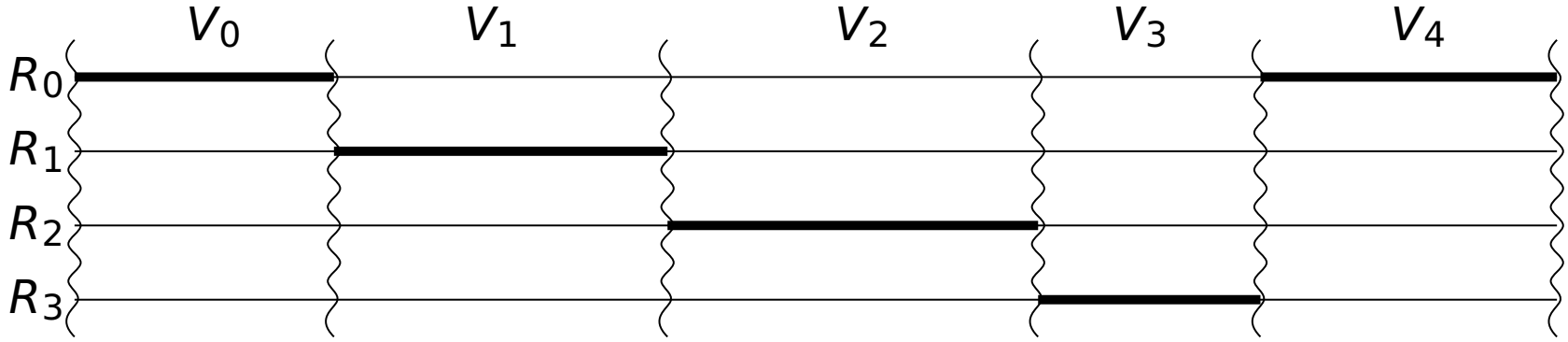
- Since f nodes may fail, it must be possible to proceed after communicating with $n - f$ replicas
- But the slower replicas may not be faulty, and therefore f of the replicas that responded may be faulty
- These must be outnumbered by the remaining replicas: $(n - f) - f > f$. Therefore, $n > 3f$

Roadmap

Views and Leaders

View is a numbered system configuration

- Replicas move through a succession of views



- Each view has a **leader**:

$$p = v \bmod n$$

where

v is the view number

$n = 3f + 1$ is the number of replicas

- View changes occur upon suspicion of the current leader
  - Which may be caused by network instability

Algorithm (SMR)

1. A client sends a request to execute a service operation to the leader
2. The leader **atomically broadcasts** the request to all replicas
   - Remember that atomic broadcast ensures a total order on the delivery of messages by non-faulty replicas
3. Replicas execute the request and send the reply to the client
4. The client waits for replies with the **same result** from $f + 1$ replicas

Client

1. A client c sends a request to execute a service operation o by sending a $\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$ message to the **leader**
   t is a timestamp to ensure **exactly once** semantics
   - Timestamps for c's requests are monotonically increasing
   - Every message sent by the replicas to a client includes the current view number, v
     - This allows the client to track the current leader
2. The leader **atomically broadcasts** the request to all replicas
3. Replica i executes the request and sends $\langle \text{REPLY}, v, t, c, i, r \rangle_{\sigma_i}$, with the result r of the execution of the operation
4. The client waits for $f + 1$ replies with **valid signatures** from different replicas, and with the same t and r, before accepting r
   - If the client does not receive replies on time, it **broadcasts** the request to all replicas
     - If the request has been processed, the replicas re-send the reply
     - Otherwise, the replicas relays the request to the leader
     - If the leader does not multicast the request to the group, it will eventually be suspected to be faulty by enough replicas

Roadmap

Quorums and Certificates

- PBFT uses quorums to implement atomic multicast
  - Quorums are based on the number of replicas
  - However, it would have been possible to use Byzantine dissemination quorums
    - Messages of PBFT protocols are self-verifying
- These quorums satisfy two properties

  Intersection any two quorums have at least a correct replica in common (as they intersect in f+1 replicas)

  Availability there is always a quorum with no faulty replicas
- Messages are sent to replicas
- Replicas collect **quorum certificates**

  Quorum certificate is a set with one message for each element in quorum, ensuring that relevant information has been stored

  Weak certificate is a set with at least $f + 1$ messages from different replicas
  - The set of $f + 1$ replies a client must receive before returning the result is a weak certificate, the **reply certificate**

Replicas

- The state of each replica comprises

  The service state

  A message log containing messages the replica has **accepted**
  - As usual, messages should be written to the log before sending subsequent messages
  - The log is garbage collected

  View id an integer with the replica's current view id
- When the leader, ℓ, receives a client request, m, it starts a three-phase protocol to atomically multicast the request to the replicas:

  Pre-prepare

  Prepare together with **pre-prepare** ensure total order of requests in a view

  Commit together with **prepare** ensure total order of requests across views

Pre-Prepare Phase

Upon receiving a client request m the leader

1. Assigns a monotonically increasing sequence number, n, to m
2. The leader multicasts message $\langle\langle \text{PRE-PREPARE}, v, n, d\rangle_{\sigma_\ell}, m\rangle$ to the other replicas, where:

   $\langle \text{PRE-PREPARE}, v, n, d\rangle_{\sigma_\ell}$ is the pre-prepare message
   m is the client's request with its signature
   $d = D(m)$ is m's message digest

   - Allows to reduce the size of the pre-prepare messages
   - Pre-prepare messages are used, in view changes, as a proof that message m was assigned sequence number n in view v

Upon receiving a PRE-PREPARE messsage a replica **accepts** it if:

- It is in view v
- The signatures in request, m, and in the PRE-PREPARE message are valid and d is the digest for m
- It has not accepted a PRE-PREPARE message for view v and sequence number n with a different digest d
- n is between a low water mark, h, and a high water mark, H
  - This prevents a faulty leader from exhausting the sequence number space by selecting a very large one

Prepare Phase

On accepting a PRE-PREPARE message replica i enters the **prepare phase**

- Replica i multicasts message $\langle \text{PREPARE}, v, n, d, i \rangle_{\sigma_i}$ to all other replicas,
  - The PREPARE msg is also logged with the PRE-PREPARE msg

On receiving a PREPARE message a replica, including the leader, **accepts** it provided that:

- The view v is the same as the replica's current view
- Its signature is correct
- The sequence number is between h and H

Prepared Certificate each replica collects

1. A PRE-PREPARE message
2. $2f$ PREPARES messages from different replicas

for request m in view v with sequence number n

- After collecting a prepared certificate a replica **prepared** the request
  - It knows the order of the request in the current view.

Total order within a view

Lemma The pre-prepare and prepare phases guarantee that a replica cannot obtain prepare-certificates for the same view and sequence number and requests with different digests

Proof sketch

- To obtain a prepare-certificate for a request in a view v with a sequence number n, $2f + 1$ replicas need to send/accept a PRE-PREPARE message in view v with number n.
- Because there are $3f + 1$ replicas, the quorums of two prepare-certificates must have at least $f + 1$ replicas in common

- Thus at least one non-faulty replica would have to send/accept two PRE-PREPARE messages with the same sequence numbers in the same view but different digests
- By the protocol, this is not possible

Ensuring order across view changes

- The pre-prepare and the prepare phases are not sufficient to ensure total order for requests across view-changes
- Replicas may receive PREPARE messages in the same view with the same sequence and different requests
  - Remember that the leader of a view may be faulty
- Replicas may collect prepared certificates in different views with the same sequence number and different requests.
  1. Non-faulty replica *i* collects a prepared certificate in view *v* for *m* with sequence number *n*
  2. The leader for *v* is faulty and there is a view change
  3. The new leader may not have the prepared certificate for *m* in view *v* and sequence number *n*
     - It may even have accepted a PRE-PREPARE message in *v* with the same sequence number but for a different request
  4. Even if the new leader tries to collect a prepare certificate it may not be able to. For example:
     - Replica *i* may be the only non-faulty replica with a prepared certificate for *m* in *v*, and may fail to timely respond to the new leader
     - Faulty replicas may refuse to cooperate

Commit Phase

On collecting a prepared certificate replica i enters the **commit phase** and

- Multicasts message $\langle \text{COMMIT}, v, n, d, i \rangle_{\sigma_i}$ to all replicas
  - The COMMIT message is also logged.

On receiving a COMMIT message a replica, including the leader, accepts it provided that:

- The view v is the same as the replica's current view
- Its signature is correct
- The sequence number is between h and H

Note that a replica may receive COMMIT messages before it has moved to the commit-phase

Commit certificate is a set of accepted $2f + 1$ COMMIT messages with the same view, sequence number and digest, received from different replicas (inlcuding the acceptor)

Committed request by a replica, if the replica has both the prepared and the committed certificates

Commit Phase Invariant

Invariant The commit phase ensures that if a replica committed a request that request is prepared by at least $f + 1$ non-faulty replicas

Proof sketch

- By definition, if replica i has committed a request it has collected a commit certificate, and therefore has accepted $2f + 1$ COMMIT from different replicas
- Since there are at most f faulty replicas, at least $f + 1$ non-faulty replicas sent COMMIT
- By the protocol, a non-faulty replica sends a COMMIT only if it is prepared

With the view change protocol this invariant guarantees that:

1. Non-faulty replicas agree on the sequence numbers of a committed request, even if different replicas may commit it in different views
2. Any request committed at a non-faulty replica, will commit at all non-faulty replicas, possibly in different views.
   - With the standard caveat (warning)...

Request Delivery and Execution

- Each replica *i* executes the operation requested by *m* after:
  - It has committed that request.
  - It has executed all requests with a lower sequence number
    - This ensures that all non-faulty replicas execute requests in the same order
- Replicas send a reply to the client after executing the requested operation
  - Replicas discard requests whose timestamp is lower than the timestamp in the last reply they sent to the client
    - This guarantees exactly-once semantics

Garbage Collection and Checkpoints

- To ensure **safety** a replica cannot discard the messages with a sequence number as soon as it executes that request
  - Some replicas may have missed some messages and are behind
- For replica repair or replacement, we need state synchronization
  - This is also useful, if a replica falls behind after a partition
  - If the log is pruned, a state transfer protocol is required
- A replica periodically, i.e. every K requests, **checkpoints** its state
  - And generates a **proof** of correctness of that checkpoint
- After generating such a proof, the checkpoint becomes **stable**
  - The replica can discard from the log earlier checkpoints and messages
- A checkpoint proof requires exchanging messages
- A replica maintains several copies of the service state:
  - The last **stable** checkpoint, i.e. a checkpoint with a proof
  - One of more checkpoints that are not stable yet
  - The current state

Checkpoint Proof Generation

Upon a checkpoint replica i multicasts a $\langle \text{CHECKPOINT}, v, n, d, i \rangle_{\sigma_i}$ message to all replicas, where:

- n is the sequence number of the last request whose execution is reflected in the state
- d is the digest of the state

Upon receiving a CHECKPOINT message a replica saves it in its log until it has collected a weak certificate, the **stable certificate**

Stable certificate is a set of $f + 1$ CHECKPOINT messages signed by different replicas (including itself) for sequence number n with the same digest d – this is the **checkpoint's proof**

- It proves that at least one non-faulty replica has generated a checkpoint with sequence n and digest d

Upon collecting a stable certificate a replica discards:

- all PRE-PREPARE, PREPARE and COMMIT messages with sequence number less than or equal to n
- all earlier checkpoints and respective CHECKPOINT messages

Updates to the low and high water marks

- A replica advances the low and high water marks every time it runs the checkpoint protocol

  h is set to the sequence number n of the last stable checkpoint
  H is set to $h + L$, where L is a small multiple (e.g. 2) of K, the checkpoint period
  - This makes it unlikely for replicas to stall waiting for the checkpoint to become stable

Roadmap

View Change Protocol: First Phase

Purpose to ensure liveness upon failure of the leader

- While ensuring safety

Leader failure is suspected with the help of a timer

- The timer prevents a replica from waiting indefinitely for requests to execute.
  - A replica is **waiting** for a request, if it received a valid request but has not executed it yet

Upon timeout in view v, replica i starts a view change to advance to view $v + 1$

1. It stops accepting messages (other than CHECKPOINT, VIEW-CHANGE and NEW-VIEW)
2. It multicasts a $\langle \text{VIEW-CHANGE}, v+1, n, \mathcal{C}, \mathcal{P}, i \rangle_{\sigma_i}$ message

   n is the seq. number of the last stable checkpoint s known to i

   $\mathcal{C}$ is that checkpoint's stable certificate

   $\mathcal{P}$ is a set with a prepared certificate for each request prepared at replica i with sequence number greater than n

View Change Protocol: Second Phase

New-view Certificate is a set with $2f + 1$ valid VIEW-CHANGE messages for view $v + 1$ each signed by a different replica

- New-view certificates contain prepared certificates for:
  1. All requests that committed in previous views
  2. Some request that prepared

  since the last stable checkpoint

On collecting a VIEW-CHANGE certificate the leader ℓ of view $v + 1$

1. Updates its log and/or service state, if necessary
2. Multicasts $\langle \text{NEW-VIEW}, v + 1, \mathcal{V}, \mathcal{O}, \mathcal{N} \rangle_{\sigma_\ell}$ to other replicas, where

   $\mathcal{V}$ is the new-view certificate

   $\mathcal{O}$ and $\mathcal{N}$ are sets of PRE-PREPARE messages (without the respective requests) that propagate sequence number assignments from previous views
3. Finally, ℓ enters view $v + 1$, and starts accepting messages

View Change Protocol: Computation of $\mathcal{O}$ and $\mathcal{N}$

1. The leader, ℓ, determines the sequence numbers
   h of the latest stable checkpoint in $\mathcal{V}$
   H the highest in a message in a prepared certificate in $\mathcal{V}$
2. ℓ creates a new PRE-PREPARE message for view $v+1$ and sequence number n, s.t. $h < n \leq H$. There are two cases:
   There is a prepared certificate in $\mathcal{V}$ for sequence number n, the leader adds a new message $\langle \text{PRE-PREPARE}, v+1, n, d \rangle_{\sigma_{ell}}$ to $\mathcal{O}$, where d is the digest in the prepared certificate with sequence number n and with the highest view number in $\mathcal{V}$
   Othewise it adds a new message $\langle \text{PRE-PREPARE}, v+1, n, null \rangle_{\sigma_\ell}$ to $\mathcal{N}$, where *null* is a special digest for a no-op request (this is similar to what happens in Paos upon a new leader election)
   IMP the leader appends the messages in $\mathcal{O}$ and $\mathcal{N}$ to its log, as they belong to the pre-prepare phase for these requests in the new view

View Change Protocol: State Update at New Leader

- If h is greater than the sequence number of its latest stable checkpoint, the leader adds the stable certificate for the checkpoint with sequence number h to its log
  - And discards log information (as usual, on checkpointing)
- If h is greater than the leader's current state, it also updates its current state to that of the checkpoint with sequence number h
- The leader may be missing:
  - The latest stable checkpoint, i.e. the service state
  - Request messages committed or prepared since that checkpoint

  These are not sent in NEW-VIEW messages
- The leader may obtain the missing information from another replica. E.g.:

  Missing checkpoint state, s can be fetched from a replica whose CHECKPOINT messages certified its correctness in $\mathcal{V}$
  - Since $f + 1$ replicas certified its correctness, at least one of these will be correct, and replica i will always be able obtain s or a later checkpoint

View Change Protocol: NEW-VIEW at replicas

- A replica accepts a NEW-VIEW message for view $v + 1$, if it:
  1. Is signed properly
  2. Contains a valid **new-view certificate**
  3. Contains correct $\mathcal{O}$ and $\mathcal{N}$ sets
     - A replica checks the correctness of the $\mathcal{O}$ and $\mathcal{N}$ sets, by recomputing them from the **new-view certificate**, just like the leader does
- It enters view $v + 1$
  - After updating its state as described for the leader, if necessary
- It adds the PRE-PREPARE messages in the $\mathcal{O} \cup \mathcal{N}$ to its log
  - They are need for prepared certificates for view $v + 1$
- It multicasts a PREPARE for each message in $\mathcal{O} \cup \mathcal{N}$ to all the other replicas
  - After adding these PREPARE to its log

IMP The atomic multicast protocol for each of these requests proceeds as described earlier

- Re-execution of client requests is prevented by using stored information about replies previously sent to clients

Roadmap

Correctness: Safety (1/2)

Safety depends on all non-faulty replicas agreeing on the sequence numbers of requests that commit locally

For local commits at the same view this is ensured by the pre-prepare and the prepare phases

For local commits at different views this is ensured by view-change protocol

- By the commit phase invariant, if a non-faulty replica commits locally request m with seq. no. n in view v, then
  - There is a set R_1 of at least $f + 1$ non-faulty replicas for which $prepared(m, v, n, .)$ is true
- A replica accepts messages in view $v' > v$ only after receiving a NEW-VIEW message for view v'
  - A NEW-VIEW includes VIEW-CHANGE messages from a set R_2 of at least $2f + 1$ replicas.
- Because there are $3f + 1$ replicas, at least one non-faulty replica in R_1 has sent a VIEW-CHANGE message for view v'
  - Let k be such a replica

Correctness: Safety (2/2)

For local commits at different views this is ensured by view-change protocol

. . .

- *k*'s VIEW-CHANGE message will ensure that the fact that *m* is prepared with sequence number *n* is propagated to subsequent views
  - Unless the NEW-VIEW message contains a VIEW-CHANGE message with a stable checkpoint with a sequence number higher than *n*
- In the first case, the algorithm will repeat the three phases of the atomic multicast protocol for *m* with the same sequence number *n* and the new view number
- In the second case, no replica in the new view will accept any message with a sequence number lower than *n*
- In either case, the replicas agree on the request that commits locally with sequence number *n*

Liveness (1/2)

Liveness goals

Replicas must move to a new view if they are unable to execute a request

Maximize the time interval with at least $2f + 1$ non-faulty replicas in the same view

- Additionally, ensure that the length of this time interval increases exponentially until the execution of some request

Liveness ensurance measures

Increasing view change timeouts to avoid starting a view change too early. A replica starts a timer upon receiving $2f + 1$ VIEW-CHANGE msg.'s for view $v + 1$ and increases the timeout value by T if the timer expires:

Either before the replica receives a NEW-VIEW for view $v + 1$

Or before the replica executes a request in the new view that it had not executed previously

Liveness (2/2)

Liveness ensurance measures (continued)

View change without timeout If a replica receives a set of $f + 1$ valid VIEW-CHANGE for views greater than its current view, it sends a VIEW-CHANGE for the smallest view in the set, even if its timer has not expired

- This speeds up view changes in progress

Faulty replicas cannot force too-frequent view changes unless they are leaders, because a non-faulty replica changes its view only

On timeout

On reception of $f + 1$ VIEW-CHANGE messages

Although the leader may force a view change by not sending messages (or sending bad messages)

The leader cannot be faulty for more than f consecutive views because the leader of view v is the replica with no. $v \bmod |\mathcal{R}|$

Claim These techniques guarantee liveness unless message delays grow faster than the timeout value indefinitely

- An unlikely event in a real system

Roadmap

Further issues

Fairness the implemenation guarantees that clients get replies to their requests even when there are other clients accessing the service

- Non-faulty leaders assign sequence numbers in FIFO order
- Replicas keep requests in a FIFO queue, and only stop the view change timer when the first request is executed
  - This prevents a faulty leader from starving clients

Speeding up cryptographic operations around the year 2000, computing a 1024-bit RSA signature was about 3 orders of magnitude slower than computing an MD5 message digest

- A new protocol based on MACs with essentially the same communication structure allows to speed up its execution

Other optimizations The thesis describes several other optimizations. Of these we considered only one:

- Replacing request messages by their digests in PRE-PREPARE messages
  - One of the other optimizations is the multicasting of requests from the client to all replicas

Byzantine Quorums vs. PBFT

- Compared with state machine replication, Byzantine Quorums appear to require fewer messages, by a large margin
- But the protocols we have seen assume that:
  - Either clients can be trusted
  - Or the information stored is self-verifiable
- To handle other cases, in Phalanx (the SRDS 1998 paper) Malkhi and Reiter use **consensus-objects**, which appear to require about the same number of messages as Byzantine SMR
- Furthermore, these quorum protocols support only read/write operations.
  - Although, we can build more complex operations on top of read/write operations, the number of messages will increase
  - Also, although read/write operations are atomic, and performed in the same order, consistency problems may arise when we build more complex operations on top of read/write
    - In Phalanx (the SRDS 1998 paper) Malkhi and Reiter use **mutual exclusion** objects

Roadmap

Further Reading

- Castro M. and Liskov B., *Practical Byzantine Fault Tolerance*, 3rd Symposium on Operating Systems Design and Implementation (OSDI), February 1999
- Miguel Castro, *Practical Byzantine Fault Tolerance*, MIT-LCS-TR-817, 01-31-2001
  - Covered only until Section 2.3 (inclusivé), this is less than 30 pages, less than 15 pages if you skip the first chapter
- van Steen and Tanenbaum, *Distributed Systems, 3rd Ed.*
  - Section 8.2(.5) *Consensus in faulty systems with arbitrary failures*