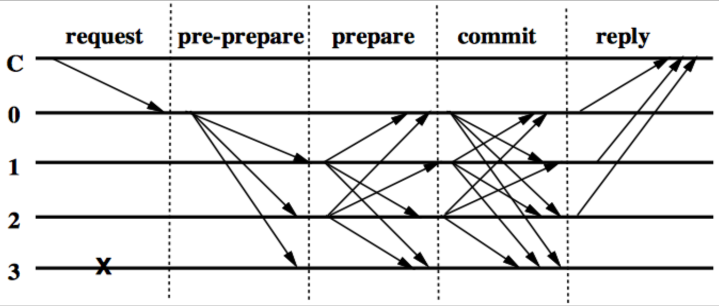
Dependable Distributed Systems – 5880V/UE

5. Byzantine Paxos / PBFT – 2021-10-11

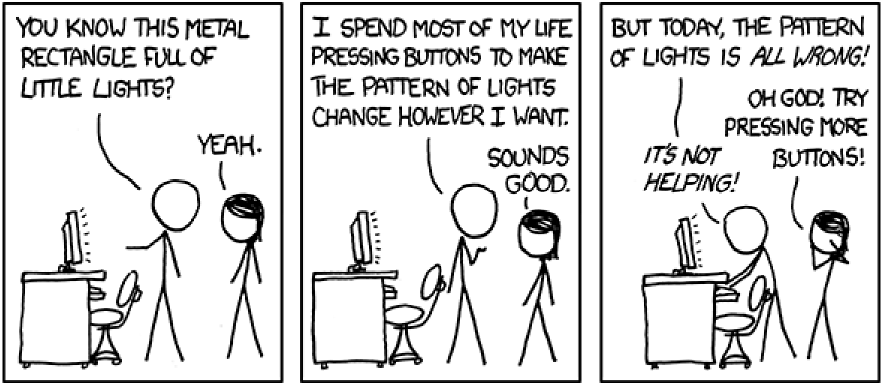
Prof. Dr. Hans P. Reiser | WS 2021/22

UNIVERSITÄT PASSAU



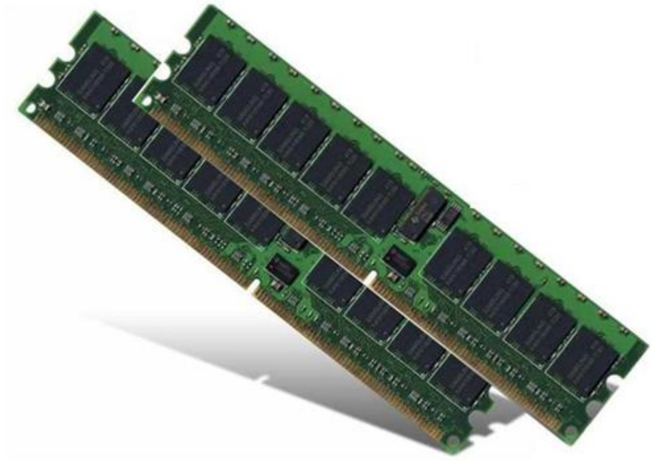
Overview

Motivation: Crash model sufficient?


YOU KNOW THIS METAL RECTANGLE FULL OF LITTLE LIGHTS?
YEAH.
I SPEND MOST OF MY LIFE PRESSING BUTTONS TO MAKE THE PATTERN OF LIGHTS CHANGE HOWEVER I WANT.
SOUNDS GOOD.
BUT TODAY, THE PATTERN OF LIGHTS IS ALL WRONG!
OH GOD! TRY PRESSING MORE BUTTONS!
IT'S NOT HELPING!

http://xkcd.com/722/

Motivation: Crash model sufficient?



ZDNet: “Nightmare on DIMM street” (2009): Google study: Bit errors up to 1000x more frequent than expected, 3751 correctable errors per year

Motivation: Crash model sufficient?

“Eve”

Byzantine Fault Tolerance (BFT): context

- The Byzantine Generals Problem
  - Lamport et al., 1982

Byzantine Fault Tolerance (BFT): context

- The Byzantine Generals Problem
  - Lamport et al., 1982
- Byzantine faults
  - A fault porcess may exhibit **arbitrary** behaviour
  - Usually some restrictions in practice
    - Often: correct sender identification, secure cryptography
    - Sometimes: Trusted Computing Base (TCB), “wormhole”
- System model:
  - Reliable channels
  - Asynchronous + . . .

Reliable broadcast with Byzantine faults

Remember Part 3?

Reliable broadcast with Byzantine faults

Remember Part 3?

- Simple echo + threshold $\Rightarrow$ best effort broadcast
  - Make sure that at most one value is delivered
- Two-phase echo + thresholds $\Rightarrow$ reliable broadcast
  - Make sure that all nodes (or none) deliver exactly one value

Overview

Byzantine consensus

Consensus properties

- *MVC1 Validity.* If a correct processes decides v, some process has previously proposed v
- *MVC2 Agreement.* No two correct process decided differently.
- *MVC3 Termination.* Every correct processes eventually decides.

Byzantine consensus

Consensus properties

- *MVC1 Validity.* If a correct processes decides v, some process has previously proposed v
- *MVC2 Agreement.* No two correct process decided differently.
- *MVC3 Termination.* Every correct processes eventually decides.

- Validity?
  - Problem: Faulty process can claim having proposed any arbitrary value

Byzantine consensus

- *Weak validity (CGR)*. If all processes are correct, the following condition must hold: (a) if all propose the same value v, then nobody decides a different value, and (b) if a correct process decides v, then v has been proposed by some process.
- *Strong validity (CGR)*. If all processes are correct and propose the same value v, then nobody decides a value different than v. Otherwise, a correct process may only decide a value that has been proposed by a *correct* process or some special value $\square$.
- *Validity**. If all correct processes propose the same value v, then each correct process must decide for this value.
- *Validity*** (endorsement). If a correct processes decides v, then v must be *endorsed* by a correct process.
  Requires (application-specific) rules for endorsement of “acceptable” values

Randomized binary Byzantine agreement

Bracha (1984): Algorithm executes in (logical) rounds

`https://dl.acm.org/doi/10.1145/800222.806743`

Round(k) by process p; up to t out of n processes faulty:

1. Byz-RBroadcast (i_p) and wait for $n - t$ messages.
   $i_p :=$ majority value of the messages
2. Byz-RBroadcast (i_p) and wait for $n - t$ messages.
   If more than $n/2$ of the messages have the same value v, then
   $i_p := (d, v)$ "tagged" state (d, v): node ready to decide for v
3. Byz-RBroadcast (i_p) and wait for $n - t$ validated messages.
   If at least $2t + 1$ (d, v) messages: Decide(v)
   If at least $t + 1$ (d, v) messages: $i_p := v$
   Otherwise, $i_p := 1$ or 0 with equal probability
4. Go to step 1 of round $k + 1$

Overview

Transforming Paxos into Byzantine Paxos

Challenges

- Leader election
  - Enforce leader change in case of faulty leader?
  - Prevent leader change if leader is correct?
- Leader operations
  - Leader needs to distributed same value to all nodes
  - Leader change: Promise not to accept any old messages any more
  - Leader change: Sufficient information about previous steps towards decision for some value to ensure consistency

PBFT – Literature

- Miguel Castro: Practical Byzantine Fault Tolerance. PhD thesis, MIT, 2001
- Miguel Castro, Nancy Lynch: Practical Byzantine Fault Tolerance. Proc. ACM Symp. on Operating System Design and Implementation (OSDI), pp 173-186, 1999
- Butler W. Lampson: The ABCD's of Paxos. Proc. 20th ACM Symp. on Principles of Distributed Computing (PODC'01), August 2001.

PBFT – context

Asynchronous system model

- for safety
- for liveness: additional timing properties required

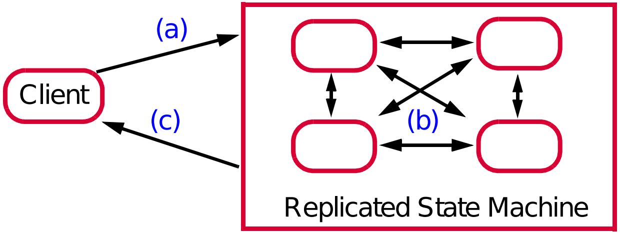
Byzantine faults

- $n = 3f + 1$ processes required for tolerating f faults

Use of cryptography

- Each node can sign messages with public key cryptography. Each node can verify these public key signatures.

PBFT – from a client perspective



- (a) Client sends a request
- (b) Consensus algorithm decides some value ($\rightarrow$ order of requests)
- (c) Client waits for reply

PBFT – from a client perspective

(a) Sending the request

- only to leader (problem if leader is faulty)
- to all nodes (more communication)

(b) Consensus algorithm

- Problem: Detecting fault leader
- Possible strategy: define maximum time between reception of request until decision

(c) Receiving the reply

- Client must be able to detect wrong replies
- Solution: wait for $f + 1$ identical replies

PBFT – normal-case operation

- Leader sends its value to all nodes
- Each nodes sends the received value to all nodes
- A node supports a value if sufficiently many nodes have sent a message for this value

$\Rightarrow$ Byzantine fault tolerant best-effort broadcast!

Properties?

PBFT – normal-case operation

- Leader sends its value to all nodes
- Each nodes sends the received value to all nodes
- A node supports a value if sufficiently many nodes have sent a message for this value

  $\Rightarrow$ Byzantine fault tolerant best-effort broadcast!

  - Correct leader: all nodes support value sent by leader
  - Faulty leader: all nodes support at most one value sent by leader

PBFT – normal-case operation

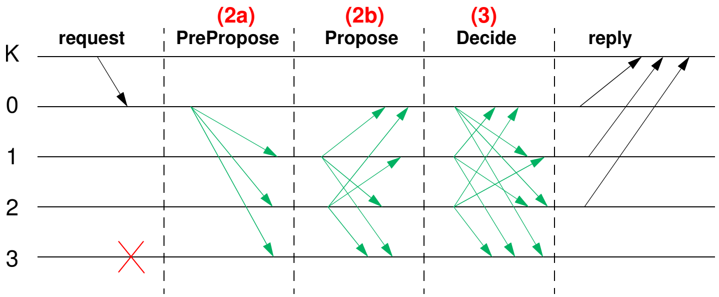
- Leader sends its value to all nodes
- Each nodes sends the received value to all nodes
- A node supports a value if sufficiently many nodes have sent a message for this value

  $\Rightarrow$ Byzantine fault tolerant best-effort broadcast!

  - Correct leader: all nodes support value sent by leader
  - Faulty leader: all nodes support at most one value sent by leader

- If $f + 1$ correct nodes support some value, the value is accepted
  - If $2f + 1$ nodes support some value, at least $f + 1$ correct nodes support it
  - Majority of correct nodes in the intersection of any two subsets containing $2f + 1$ nodes

PBFT – normal-case operation



- **PrePropose** and **Propose** ensures consistency within a single view even if leader is faulty
- **Propose** and **Decide** ensure consistency across view changes (similar to Paxos)

(*Note: **Decide** is the equivalent of the **Accept** message in our previous description of Paxos*)

PBFT – normal-case operation (Phase 2a)

Leader in view v with request number n and request m sends to all nodes:

$$\langle \texttt{PrePropose}(v, n, D(m))_{sig}, \quad m \rangle$$

$D(m)$: Hash value of m; sig: Signature of leader

A node accepts a `PrePropose` message, if

- Signatur and Hash are correct
- Node is in view v
- No other `PrePropose` message with same v and n was accepted before

PBFT – normal-case operation (Phase 2b)

If node P_i accepts a `PrePropose` message, it sends the following message to all nodes:

$$\texttt{Propose}(v, n, D(m))_{sig_i}$$

v: View; n: request number; $D(m)$: hash of request m, sig_i: P_i's signature

A node accepts a `Propose` message, if

- Signature sig_i is correct
- Node is in view v

PBFT – Predicate "prepared"

Predicate *prepared(m, v, n, i)* is true iff node P_i has accepted the following messages:

- Request *m* with request number *n*
- A `PrePropose` message for *m* in view *v*
- 2*f* `Propose` messages from other nodes which match the `PrePropose` message (same view, request number, hash)

prepared(m,v,n,i) = true $\Rightarrow$ *prepared(m', v, n, j) = false*
for all nodes *j* and all requests m' with $D(m) \neq D(m')$

- *prepared* implies that at least $f + 1$ correct nodes have sent `Propose` or `PrePropose` *m*, *v*
- This means that in a view *v* only a single value *m* may exist

PBFT – normal-case operation (Phase 3)

As soon as *prepared(m, v, n, i)* is true, node P_i sends the following message to all nodes:

$$\texttt{Decide}(v, n, D(m))_{sig_i}$$

v: View, n: request number, $D(m)$: hash value, sig_i: P_i's signature

A node accepts a `Decide` message, if

- Signature sig_i and hash value $D(m)$ are correct
- Node is in the same view v

(A node accepts a proposed value, if it can be sure that this is the unique value for this view id and if it is still in the same view, i.e., no view change with the promise not to accept messages for older view has occured)

PBFT – predicate “commited”

- *committed(m,v,n)=true* $\Longleftrightarrow$
  prepared(m,v,n,i)=true for $f + 1$ correct nodes P_i

- *committed-local(m,v,n,i)=true* $\Longleftrightarrow$
  prepared(m,v,n,i)=true and P_i has accepted $2f + 1$ `Decide` messages

- Invariant: *commited-local(m,v,n,i)* $\Rightarrow$ *commited(m,n,v)*
  - Proof (sketch): set of $2f + 1$ nodes certainly contains at least $f + 1$ correct nodes
  - As soon as *commited-local(m,v,n,i)* is true at node P_i, this node executes the request m and sends result to client

PBFT – checkpoints

Similar to Paxos:

- On view change (see later!) information about some actions in the past need to be exchanged

But:

- View changes are not initiated by new leader, but instead by all nodes
- Cannot trust leader to distributed correct information about the past to all nodes
- Naive approach requires the collection of information about *all* request numbers in the past at view change
- Storing all received messages would mean a continuously increasing storage space. . .

PBFT – checkpoints

Storage problem solved with periodic creation of stable checkpoints

- e.g., after each 100 decided requests
- Checkpoint contains all relevant information (i.e., resulting state) about all requests up to request number *n*
- All messages for request number *n* and smaller can be discarded

PBFT – checkpoints

Creating a checkpoint: Node P_i sends message

$\texttt{Checkpoint}(n, d)_{sig_i}$ to all, with

n: largest request number included in checkpoint
d: Digest (hash value) of system state after processing all requests up to and including request n

- As soon as a node has received $2f + 1$ `Checkpoint` message with matching n, d, sig_i, the checkpoint becomes stable
- These $2f + 1$ signed messages serve as a correctness proof for the checkpoint

PBFT – view change

Prevent that a faulty node can force the system to choose that node as leader

- Solution: Deterministic selection of leader: Leader in view v is node with id v mod n

View change is started only if majority of correct nodes suspect that the current leader is faulty

- Prevents that a faulty node can initiate a view change arbitrarily

PBFT – view change

Every node that suspects a faulty leader due to a timeout in view v starts a view change for view $v+1$

- Node stops processing messages for view v
- `ViewChange`$(v+1, n, C, P)_{sig_i}$ is sent to all nodes, with
  - n: Request number of last stable checkpoint s
  - C: Set of $2f+1$ message proofing the correctness of s
  - P: Set of messages P_m for all requests m, for which:
    prepared(m, v, n', i)=true, n'>n
    P_m contains a `PrePropose` and $2f$ `Propose` messages for m
  - sig_i is the signature of node P_i

PBFT – view change

As soon as the new leader of view $v + 1$ has received additional `ViewChange` messages from $2f$ nodes for this view, it sends the following messages to all nodes:

$$\texttt{NewView}(v + 1, V, O)$$

with:

- V: $2f + 1$ `ViewChange` messages serving as proof that a majority of correct processes has requested the view change
- O: set of `PrePropose` messages for all request numbers that have been started but potentially not decided by previous leaders

PBFT – view change

How to compute the set O:

- *min* := largest request number of a stable check point that was received in a `ViewChange` message
- *max* := largest known request number n' (from received `ViewChange` messages)

- For all request numbers n, $min < n \leq max$:

  If V contains messages for request number n:
  `PrePropose`$(v+1, n, d)$, $d :=$ Hash of `Propose` with largest view
  Otherwise: `PrePropose(v+1, n, null)`

PBFT – view change

Remarks:

- A new leader is selected only of at least $f + 1$ correct nodes suspect the current leader to be faulty
- If a new leader is selected, it is possible that the new leader does not yet know the largest existing checkpoint s in `NewView`. However, as each checkpoint is signed by at least $f + 1$ correct nodes, it is always possible to obtain the checkpoint from at least one of them

PBFT: Optimizations

Additional optimizations:

- Compression of `Reply` messages set to client: send complete reply m only once, otherwise send hash value $D(m)$
- Tentative execution of request
- Read-only-operations
- More efficient cryptography: Use MACs instead of public key signatures

Further improvements

Many research suggested improvements for PBFT

- Q/U (SOSP'05): Efficient in conflict-free executions, $n = 5f + 1$
- HQ (OSDI'06): Hybrid approach (Q/U + PBFT)
- Zyzzyva (SOSP'07): Speculative execution
- A2M (SOSP'07): Hybrid (Crash/BFT) architecture
- Aardvark (OSDI'09): Efficient in some problem situations

Challenges

BFT in practical applications: challenges

- Intrusion tolerance: heterogeneity of replicas
- Repairing faulty replicas: reactive and proactive recovery
- Resource efficiency
- Robustness under attacks
- Adaptation in dynamic environments