Distributed Algorithm - 2018

"Paxos better than pussy" - Umberto Sani

Legend:

- 💩 = useless
- 😍 = useful

TODO

- [ ] images
- [x] papers

System model

- set of $\sum = \{p_1, ..., p_n\}$ processes
- communicate by *message passing* (`send(m)` , `receive(m)`)
- crash failure model
- `f` faulty processes

Consensus

Definition

Consensus is used to allow a set of processes to agree on a value proposes. It ensures

- *Uniform integrity* : if a `p` decided on v, v was proposed by some `p`
- *Uniform agreement* : no two `ps` decide different values
- *Termination* : every correct `p` eventually decides on exactly one value

Synchronous model 💩

Operations are coordinated by one, or more, centralized clock signals. - message speed and delay are bounded - process keeps vector of values received - after f+1 rounds -> decide

Asynchronous model

No global clock, no strong assumptions about time and order of operations. (real world scenario) -

messages can take any time to arrive - **FLP** impossibility. Aka, we impossible to solve consensus - process keeps current status and sends msg to other processes

How to fix this shit. We can **strengthening** the model assumptions or **weakening** the problem definition or doing **booth**

Partially synchronous model 😍

Uses **failure detectors** with different accuracies (*strong*, *weak*, *eventually strong*, *eventually weak*) 💩 - *Strong* eventually each process crashed is always suspected by **every** correct process - *Weak* eventually each process crashed is suspected by **some** correct process

Paxos 😍😍😍

Paxos is love, paxos is life - Umberto Sani I'm the one in the room with the biggest c-rnd - Francesco Saverio Zuppichini

You know everything about it, pls ⭐ this

- uses `proposers`, `acceptors`, `learners` and `leader`
- to decide a `value` there must be a *quorum* of `acceptors`
- leader election to ensure that there is always a leader
- `n = 2f+1`
- latency =
  - 2δ for leader
  - 3δ for proposers

To speed up - ballot reservation (decide in advance which process will be the leader) - Leader can execute `PHASE_2B` directly to the learners - Leader among proposers and leader among learners

Ben-Or's 💩

Fault-tolerant broadcasts

broadcast : one → many relation

Reliable broadcast

- *Validity* : If a correct `p` broadcasts `m` then all correct `ps` eventually deliver `m`
- *Agreement* : If a **correct** `p` delivers `m` then all correct `ps` eventually deliver `m`
- *Integrity* For any `m`, every **correct** `p` delivers `m` at most 1 only if `m` was broadcast

Uniform Reliable broadcast

- *Uniform Validity* : If a correct `p` broadcasts `m` then all correct `ps` eventually deliver `m`
- *Uniform Agreement* : If `p` delivers `m` then all correct `ps` eventually deliver `m`
- *Uniform Integrity* For any `m`, every `p` delivers `m` at most 1 only if `m` was broadcast

FIFO broadcast

Deliver is done in the same order of the send - *FIFO order* : if a **correct** `p` broadcast `m` before `m'` then no **correct** `p` delivers `m'` before `m`

Uniform FIFO broadcast

- *Uniform FIFO order* : if a `p` broadcast `m` before `m'` then no `p` delivers `m'` before `m`

Casual broadcast

Same order of causally related deliver at all receivers - *Causal Order* : if the broadcast of a `m` **casually** precedes the broadcast of `m'`, then no **correct** `p` delivers `m'` unless it has `deliver(m)`

Uniform Casual broadcast

- *Uniform Causal Order* : if the broadcast of a `m` **casually** precedes the broadcast of `m1`, then no `p` delivers `m'` unless it has `deliver(m)`

FIFO reliable broadcast

CHECK implementation

Casual reliable broadcast

CHECK implementation

Atomic broadcast

Order is indipendent from the send order - *Uniform total order* : if `ps` `p` and `q` both deliver `m` and `m'`, then `p` delivers `m` before `m'` iff `q` delivers `m` before `m'` (they must do the same stuff)

Generic broadcast 😍

Pedone docet

Order by conflict relation `~`

- *Generic broadcast order* : if correct `ps` `p` and `q` both deliver `m` and `m'` and **m ~ m'**, then `p` delivers `m` before `m'` iff `q` delivers `m` before `m'` (they must do the same stuff)

Cheaper and faster delivery (like just eat) CODICESCONTO: `PAXOS50`

CHECK IMPLEMENTATION

Atomic Multicast

multicast : one → several relation

- Define $\Gamma = \{g_1, \ldots, g_k\}$ as the set of process groups
- They are disjoint
- `m.dst` = set of groups `m` is multicast to

Properties

- *Validity* : if `p` is correct and multicasts `m`, then eventually all correct `ps` `q` in `m.dst` deliver `m`
- *Uniform Agreement* : if `p` delivers `m` then all correct `ps` in `m.dst` eventually deliver `m`
- *Uniform Integrity* : for any `m`, every `p` delivers `m` at most 1 only if `p` was in `m.dst` and `m` was multicast
- *Uniform order* : if `p` delivers `m` and `q` delivers `m'`, either `p` delivers `m` before `m'` or `q` delivers `m'` before `m`

Atomic multicast can be reduced to atomic broadcast 💩

Impossibility of Genuine Multicast in 1 Communication step (proof)

TO ADD?

Non-fault tolerant atomic multicast

Fault-tolerant atomic multicast

Atomic Commitment

Every process has to **commit** in order to decide on action: ABORT / COMMIT

Properties

- *Agreement* : No two `ps` decides differently

- *Termination* : Every correct `p` eventually decides
- *Abort-Validity* : `ABORT` is the only possibile decision if some `p` votes `ABORT`
- *Commit-Validity*: `COMMIT` is the only decision if every correct `ps` votes `COMMIT`

Basically, *one for all and all for one*

Atomic Commitment vs Consensus 😍

	Atomic Commitment	**Consensus**
`COMMIT` decision	all `ps` proposed `COMMIT`	some `ps` proposed `COMMIT`
all `ps` proposed `COMMIT`	decide `COMMIT` or `ABORT`	decide `COMMIT`

Two-phase commit (2PC)

Uses one *Transaction Manager* (`TM`) and any number of **Resource Manager** (`RM`). Each process can be in state `PREPARED` or `COMMITED`

1. A `RM` enters in `PREPARED` and send `PREPARED` to the `TM`
2. upon `receive(PREPARED)` `TM` sends `PREPARE` to all `RM` s
3. upon `receive(PREPARE)` `RM` enters `PREPARED` and sends `PREPARED` to the `TM`
4. upon `receive(PREPARED)` from all `RM` s, `TM` sends `COMMIT`
5. upon `receive(COMMIT)` `RM` enters `COMMITED`

easy peasy

Problems

if `TM` ☠️ then the algorithm is blocked → **no fault tollerant**

Paxos commit 😍

- separete instance of Paxos for each `RM` 😍
- $2f + 1$ acceptors
- `TM` is the leader

It ensures : - *Stability* : every instance of paxos decides `PREPARED` or `ABORTED` - *Non-Blocking* : if the leader dies, then a new one is elected (paxos' *liveness*)

1. A `RM` sends `BEGIN_COMMIT` to the leader and `2A_PREPARED` to the `acceptors`
2. The leader sends `PREPARE` to all `RM` s
3. upon `receive(PREPARE)` `RM` sends `2A_PREPARED` to the `acceptors`
4. upon `receive(2B_PREPARED)` from a quorum of `acceptors` , the leader send `COMMIT`

5. upon `receive(COMMIT)` `RM` enters `COMMITED`

2PC vs Paxos commit

	2PC	Paxos Commit
Latency/Delay	4 δ	4δ
Messages	$3N - 1$	$3N$
Disk writes	$N + 1$	$N + 1$

Object Replication and Database replication

from the book [chp1,3,11]

consistency model is a property of a system designs, usually presented as a condition that can be `true` or `false` for a single execution.

execution = one pattern of events

Properties ACID:

- *Atomicity*: all transactions are executed or none of them is
- *Consistency*: a transaction transforms a a state correctly
- *Isolation*: serializability
- *Durability*: changes committed survive to future failures

A concurrent execution is **serializable** if it is equivalent to a **serial execution** of the same transactions

Sequential data type

Formalization of the sematics of the operations (What operations the client will do)

It is a six turple $< O, S, s_0, R, f_{ns}, f_{rv} >$

$$\begin{array}{lll} O & operations & \\ S & states & \\ s_0 & initial\ state & \\ R & return\ value & \\ f_{ns} & OxS \rightarrow S & next-state \\ f_{rv} & OxS \rightarrow R & return-value \end{array}$$

History

History H is a sequence of paris (operation, return-value)

Linearizable

Execution is the same as with a single site unreplicated system (the replication system gives the same functionality as the sequential data type)

DEF at pag 6 [chp1]

Strong consistency

- Replication is **hidden**
- execution is *linearizable*
- easier for the developer

Generic Functional Model

[chp11.2.1]

Has five phases

1. Client Request
2. Server coordination : before executing the operation, the servers may have to do some stuff
3. Execution
4. Agreement coordination: Servers may need to undergo a coordination phase to ensure that each executed operations has the same effect on all the servers
5. Client Reponse

Enviroment

- `ps` follows specs until it crashes
- a crashed `p` is eventually detected by **every** correct `ps`
- no process is suspected of being crashed if it is not really crashed

Fail stop failure model: - `ps` follow specs until they crash - crash of a `p` is detected by **every** correct `p` - no `p` is suspected of being crashed if it is not really crashed

Crash Failure model: - `ps` follow specs until they crash - crash of a `p` is detected by **every** correct `p` - `p` may be suspected erroneously

Basically, same as *Fail stop* just every `ps` may be suspected.

- let `s` be a server that executes transaction `t`. `t'` **precededs** `t`, denoted $t' \rightarrow t$, if `t'` committed at `s` before `t` started executing at `s`

- `t'` **conflicts** with `t` if `t'` and `t` access the same data item and at least one of them modifies the data item

Active replication (state machine replication)

Crash Failure model

(each number is a phase of the *generic functional model*)

1. client sends operations
2. client operations are ordered by an ordering protocol (Atomic broadcast)
3. each replica executes the operation
4. None
5. replies to the client

Keeping the replicas consistent requires the execution to be **deterministic** (given a client operation, same state updated is produced by each replica) → **costy** (hard to do in a multi-thread machine)

Passive replication (primary-backup replication)

Fail stop failure model.

A `p` that has not crashed and has the lowest identifier is designated *primary*.

A *primary* always exist thank to *Fail stop* model (failure are deteched and *primary* is replaced)

1. client sends operations
2. None
3. the *primary* execute the operation and sends **state updated** to all the replicas
4. replicas, passively, apply the state updates in the order received
5. replies to the client

These properties ensure **linearizability**

Multi-primary passive replication or deferred update😍

Multi is better than one - Umberto Sani

active and *passive* replicaiton are good high availability but not for high performance.

- fault-tolerace
- **high performance**
- suitable for databases

Similar to *passive* replication

- each operation executed by one machine (or a set of *primary* machines)

Transaction states = `EXECUTING` , `COMMITTING` , `COMMITTED` , `ABORTED`

1. - When receive the update, each replica checks **deterministically** if the update can be accepted → avoid **mutually inconsistency**

    - upon transaction termination
        - if is **read-only**, commit with no interaction between replica
        - if **update**, the transaction must be **certified** before be commit or abort

Termination must guarantess *transaction atomicity* (either all the servers commit it or none do it) and *isolation* (one-copy serializability)

Atomic commit based termination

[chp11.3]

New state = `PRECOMMIT`

- transaction `t` is commited if all servers precommit
- a server precommits `t` if each transaction it knows in `COMMITTED` or `PRECOMMITED` either

    - precedes `t`
    - or does not conflict with `t` . No `read` / `write` intersection

Problems

- if one server down → protocol is blocked (all servers need to precommit)
- high abort ratio

Atomic broadcast-based termination

Since atomic broadcast guarantees *Agreement* and *Total order* all servers reach the same outcome, `COMMIT` or `ABORT` . All replicas deliver in the same order, thus the certification test is deterministic

- transaction `t` is commited if no transaction `t'` that precedes `t` does update any data item read by `t`

No need to check **write-write** conflicts

Reordering-based termination

By reordering the transaction we can lower the abort ratio.

- uses a `ReorderList` contrains committed transactions not seen by transactions in execution since their order can change
- when we reach the `Reorder Factor` (max len of the array) one transaction is removed and its updated are applied to the db

Generic broadcast based termination

Uses Generic broadcast to taking care of the conflicts between operation.

- increase performance since ordering happens only when it is needed!
- conclict is defined for *write-write*, *write-read* and *read-write* conflicts

Papers

BFT-Smart

Problem:

Gap between existing software and research

Solution:

- Open source Java library implementing robuts **state-machine** replication
- support reconfigurations of the replica set
- provide **efficient** and **transparent** support for durable services

ByzCast

Problem:

No Byzantine Fault-tolerant Atomic Multicast exists

Solution:

- first BFT Atomic Multicast
- designed on top of existing BFT abstraction (BFT-Smart)
- scale with the number of group
- **partially genuine**
- uses two groups, all implements FIFO atomic broadcast
    - `auxilary`, help order the msg

  - `target`, the ones that can be in `m.dst`

- uses a **tree** of processes to re-route/order **efficiently** the msg to their destination (lowest common anchestor)

  ### Ceaser

  #### Problem

  Big performance degradation when there are **conflicting** request for **geographically replicated sites**

  #### Solution

- solve generic consensun to increase performance

- implements **Multi-Leader Generic Consensus**

- uses a *unique time-stamp* associated with every command `c` to decide if a **slow decision** is needed

FastCast

Problem

In Atomic-Multicast **distributed message ordering is challenging** since each message can be multicast to all destinations

Solution

- Genuine Atomic Multicast that uses only **four communication delays** (4δ)
- decompose the ordering in **two** execution paths, `FAST` and `SLOW`
  - `FAST` speculates about the order → if okay save time
  - `SLOW` path similar to BaseCast

GeoPaxos

Problem

Coordinating geographically distributed replicas

Solution

- decouples order from execution in a state machine replication
- *partial order* on the execution of operations (instead of *total order*) → save time

- exploit geographic location

Janus

Problem

Coordination between cross-data center is done **twice**, for *concurrency model* and *consensus*

(In a concurrent system different threads communicate with each other)

Solution

- *concurrency* control and *consensus* can be mapped to the same abstraction
- **unified** protocol do to booth at once:
    - **strict serializability** for transaction consistency
    - **linearizability** for replication consistency

(*Strict serializability* guarantees that operations take place atomically

Early Scheduling

Problem

Multi-core servers are not well exploited in fault-tolerant state machine-replication due to the deterministic execution of request that translates into a single-threaded replice leading to bad performance

Solution

- proposes **early scheduling** of operations. Decision are mode before the requests are ordered to schedule operations on worker threads at replicas
- outperform late scheduling

Spanner

Problem

Build a **scalable**, **globally-distributed** DB

Solution

- first system to distribute data at globally scale and support externally-consistent distributed transactions
- data is stored in a schematized **semi-relational** tables

- replica configuration can be controlled at fine grain
- provide **external consistency** reads-write
- **globally consistent** reads
- assign globally-meaningful **commit timestamps** for transactions

WREN

Problem

Transactional Causal Consistency (TCC) is not implemented well

Solution

- present the first TCC that implements **non blocking** reads, archieving **low latency** and allows application to scale out by sharding.

  ## Resources

- consistency