Distributed Systems and Fault Tolerance

Paweł Szałachowski

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

–Leslie Lamport

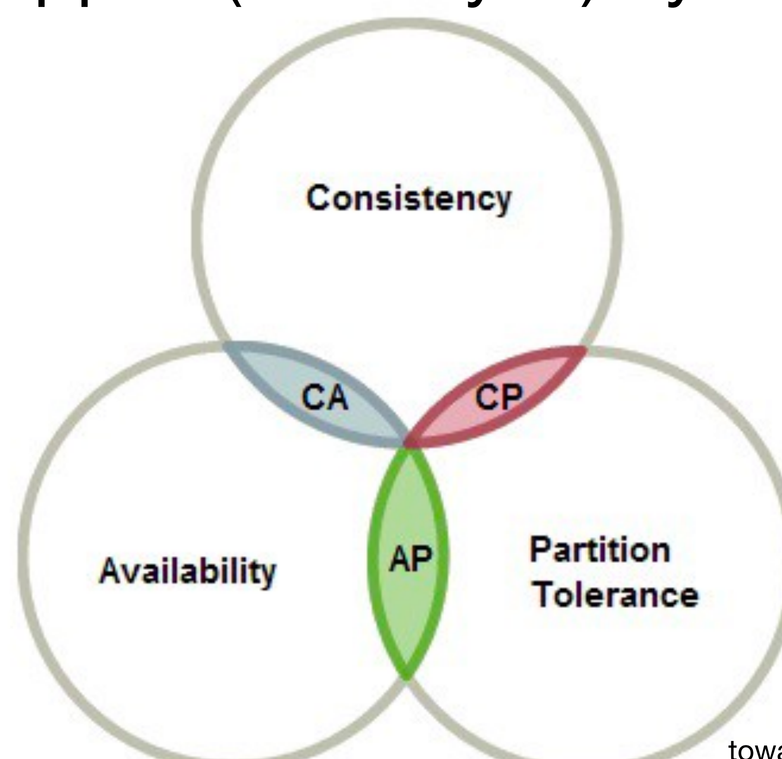
CAP theorem

- "*it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees*" - Eric Brewer
  - Consistency: Every read receives the most recent write or an error
  - Availability: Every request receives a (non-error) response – without guarantee that it contains the most recent write
  - Partition tolerance: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes
- Examples?

CP: MongoDB

AP: Cassandra

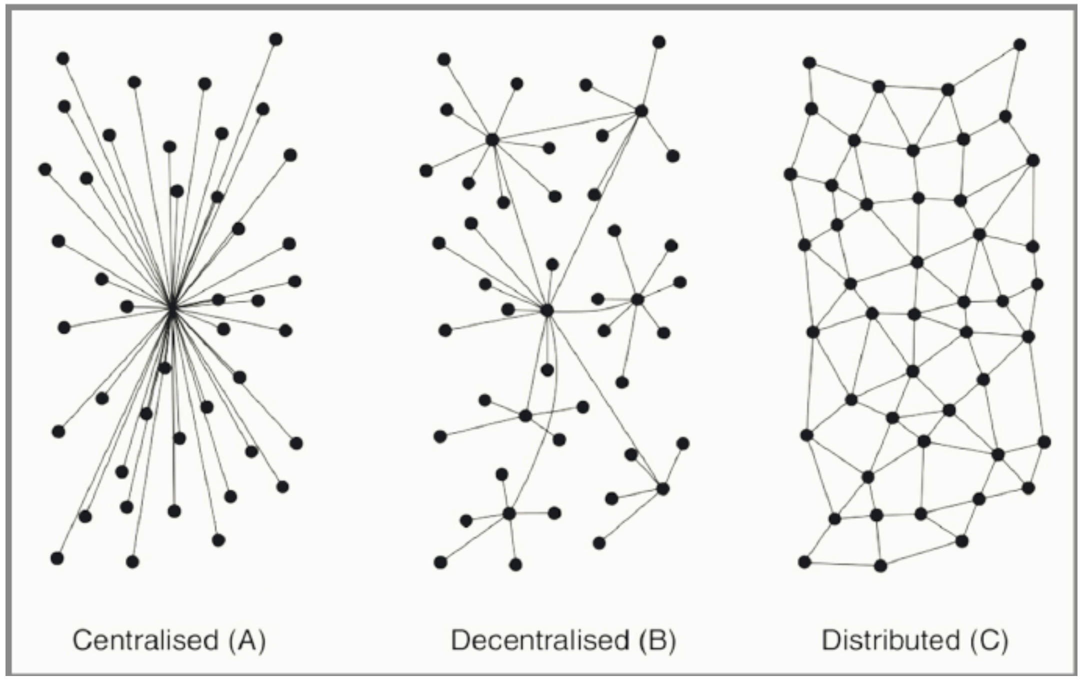
CA: MySQL



towardsdatascience.com

System Types

- Computing, Trust, Network, Decision making, …
- Centralized/Monarchy/Monopoly
    - One trusted node decides (can be replicated)
- Decentralized/Oligarchy/Oligopoly
    - Multiple trusted nodes can decide (each individually)
- Distributed/Open
    - Nodes collectively decide (no node is individually trusted)

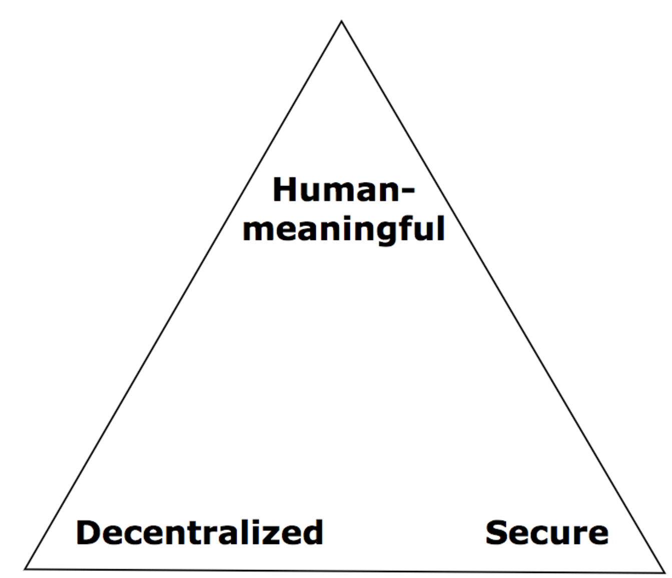


Distributed Naming

- Naming in distributed systems is hard
    - Especially for security
- How to name machines, organizations, persons, entities, … ?
    - Name collision may lead to authentication/authorization failures
- Names exist in contexts
- What namespaces do you know?

Zooko's Triangle

- No single kind of name can achieve more than two:
  - Human-meaningful: Meaningful and memorable (low-entropy) names are provided to the users.
  - Secure: The amount of damage a malicious entity can inflict on the system should be as low as possible.
  - Decentralized: Names correctly resolve to their respective entities without the use of a central authority or service.
- Examples
  - DNSSec
  - .onion and Self-certifying File System (SFS)



- It is believed that open and distributed consensus (blockchains) relaxes it

Consensus

Goals

- How N nodes can achieve consensus in the presence of faulty nodes?
    - (nodes are also called processes, actors, participants, hosts, …)
    - Different equivalent problem formulations (all about agreement)
- Properties
    - Safety: something bad will never happen
        - Agreement: all correct nodes select the same value
    - Liveness: something good will happen eventually
        - Termination: all correct nodes eventually decide

System Models

- Network: fully connected with message ordering controlled by adversary
- Timing
  - Synchronous: message sent at T is delivered by T+d, where d is known
  - Eventually Synchronous: message sent at T is delivered by max(T+d, T_g+d), where T_g is unknown
  - Asynchronous: sent messages are eventually delivered
- Faults: f nodes can be faulty (usually, relative to N)
  - Crash: would be honest but is (sometimes) unavailable
  - Byzantine: arbitrary (e.g., adversarial) behaving
- What would you assume for the Internet?

Many Bounds

- Two generals
  - Two-party agreement over unreliable medium is impossible
  - msg, ack, ack of ack, ack of ack of ack, …
- Fischer, Lynch, and Patterson (FLP) Impossibility
  - *No (deterministic) consensus can be guaranteed (liveness and safety) in an asynchronous communication system in the presence of any failures*
  - Intuition: cannot distinguish between failing and slow nodes
  - What to do then?
    - Tweak the model a bit (timing, randomness, failure detectors,…)
- … and much more: “A Hundred Impossibility Proofs for Distributed Computing”

Byzantine Consensus

Byzantine Nodes Bound

- Byzantine Failure Tolerance (Lamport, Shostak, and Pease)
  - *N* generals defending Byzantium, *f* of whom are malicious
    - Network with authentic and confidential (peer2peer) messages
  - What is max *f* that can be tolerated? N >= 3f + 1
    - Node with inconsistent info cannot determine who is faulty (note, however, that with digital signatures it can be solved easily.)

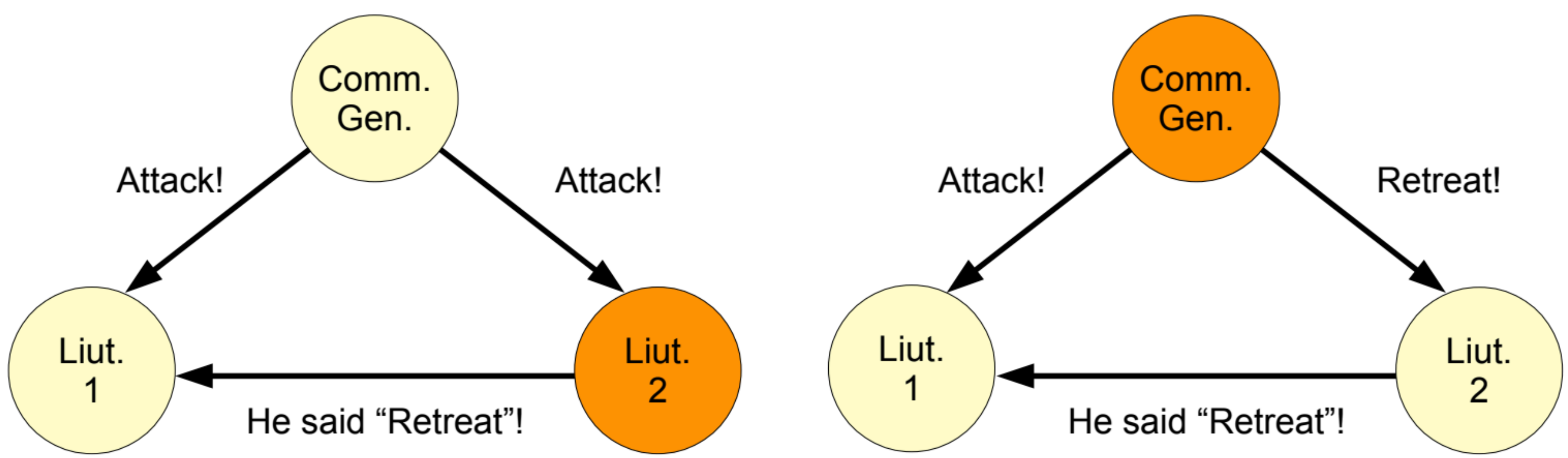


Fig: http://disi.unitn.it/~montreso/

Best-Effort Broadcast

<u>Sender:</u>

on input(x)
  multicast (send, x)

<u>Receiver:</u>

on receiving (send, x)
  output x

Issues?

- unreliable communication?
- how does sender know which nodes received x ?

Consistent Broadcast

- Sender has an input x to be broadcast
- Termination: if sender is honest, then every honest node outputs x
- Agreement: if any two nodes output x and y, then x=y
- Model
    - Asynchronous network
    - Byzantine faults with $f < N/3$

Consistent Broadcast

Sender:

on input(x)
  multicast (send, x)

Receiver:

on receiving (send, x)
  multicast (echo, x)

on receiving (echo, x) from >= (N+f+1)/2 nodes
  output x

Liveness:
With honest sender, N-f honest nodes receive (send, x)
  thus N-f correct nodes multicast (echo, x)
    thus each honest node receives N-f echo msgs

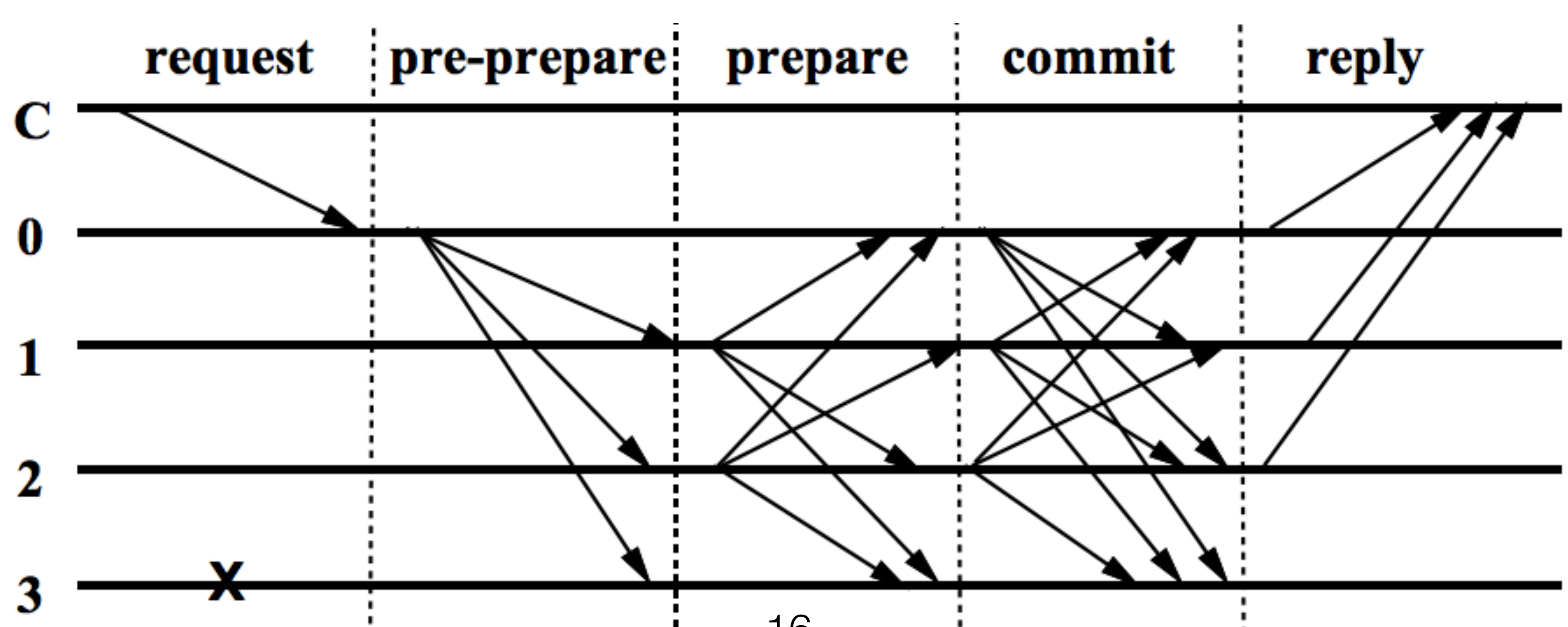
Safety:
Assume two honest nodes outputting a!=b
  Must have received (N+f+1) echo msgs in total
    At least f+1 nodes sent two conflicting echo msgs
    That cannot happen as only f nodes can be faulty

Do you see any problem(s) of that protocol?
Faulty sender?

PBFT

- Castro and Liskov "Practical Byzantine Fault Tolerance", 1999

    1. A client sends a request to invoke a service operation to the primary
    2. The primary multicasts the request to the backups
    3. Replicas execute the request and send a reply to the client
    4. The client waits for *f+1* replies from different replicas with the same result; this is the result of the operation.



Properties

- Scale to large # of transactions
  - Up to (several) thousands
- Scale only to small # of nodes
  - Only several to tens, due to $O(N^2)$ message complexity
- Resilient to 1/3 malicious nodes
- Needs known and static set of participants (identities)
  - Authority has to allow nodes to participate

How to use?

- Implement a distributed key:value storage (filesystem)
  - Universal (easy to implement other primitives on top)
    - Distributed locks
    - Leader election (often protocols provide it by default)
    - Membership enumeration
    - …
- Easy to combine with other services, load balancers, etc…
- Easy to implement a distributed ledger

Other protocols

- Paxos (prior PBFT)
  - Synchronization problem
  - Used by Google (Chubby, Spanner, Megastore, …)
  - OpenReplica, IBM SAN Volume Controller, …
- Raft
  - Designed as an alternative to Paxos
  - More understandable

Resilience and Availability

Resilience

- “*ability to provide and maintain an acceptable level of service in the face of faults*”
- Building “virtual servers” on top of a number of cheap machines
  - Reduce risk by introducing many vendors
  - Easier to detect that one component is attacked/malicious
- Distributed storage and redundancy

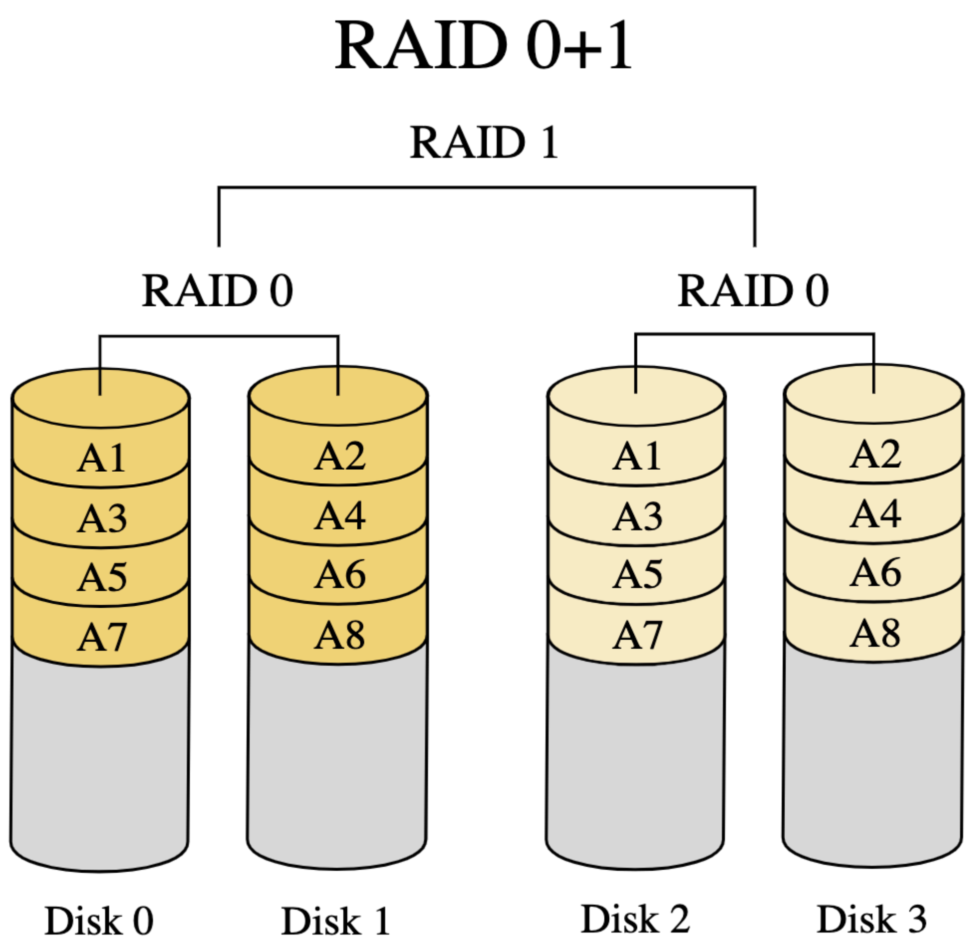
Redundancy

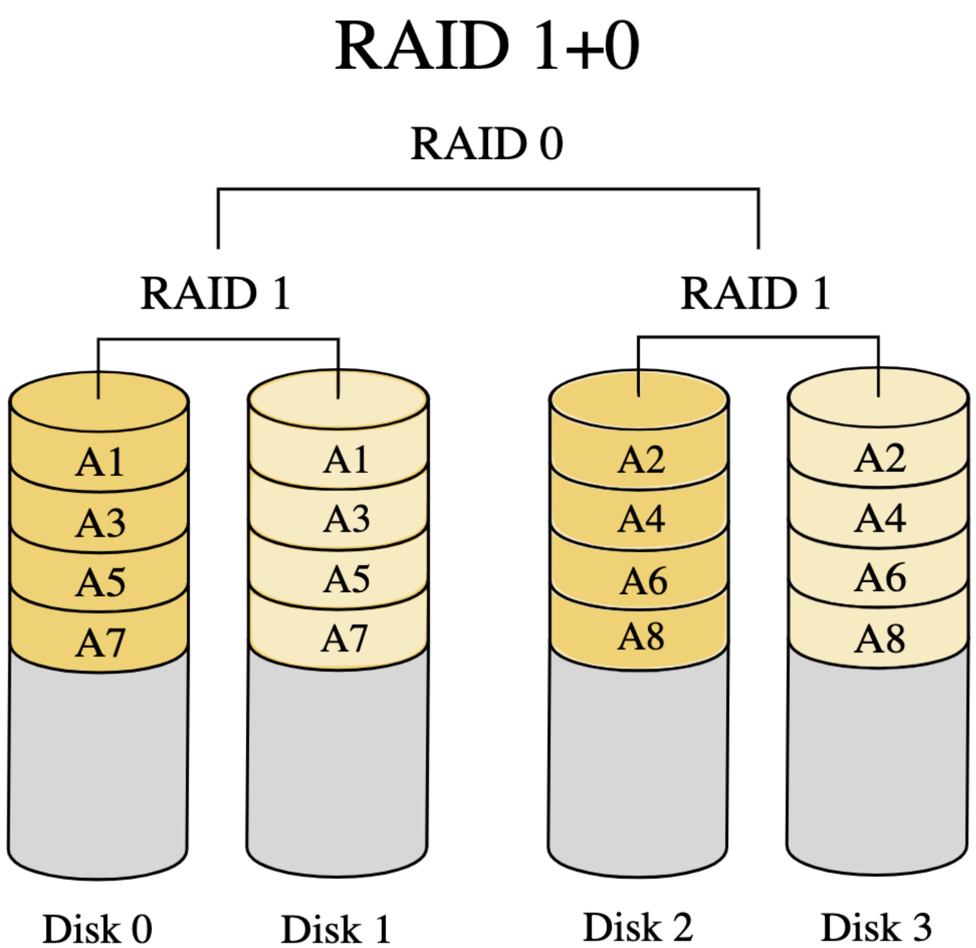
- Level of redundancy
  - Hardware (e.g., System/88)
    - disks, CPUs, mem, …
  - Process group redundancy (system level)
    - Multiple copies of a system (different locations)
  - Backup (system/data level)
    - Copy of a system taken and archived at regular intervals
    - Journals: keeps track of changes not yet committed to the file system
  - Fallback: limited functionality in application layer (less capable than backup)

RAID

- Redundant Arrays of Inexpensive Disks (RAID)
- Different levels
  - RAID 0: consists of striping, without mirroring or parity
  - RAID 1: consists of data mirroring, without parity or striping
  - RAID 2: consists of bit-level striping with dedicated Hamming-code parity
  - RAID 3: consists of byte-level striping with dedicated parity
  - …
- Hybrid
  - RAID 01: creates two stripes and mirrors them
  - RAID 10: creates a striped set from a series of mirrored drives

RAID Examples





Service Availability

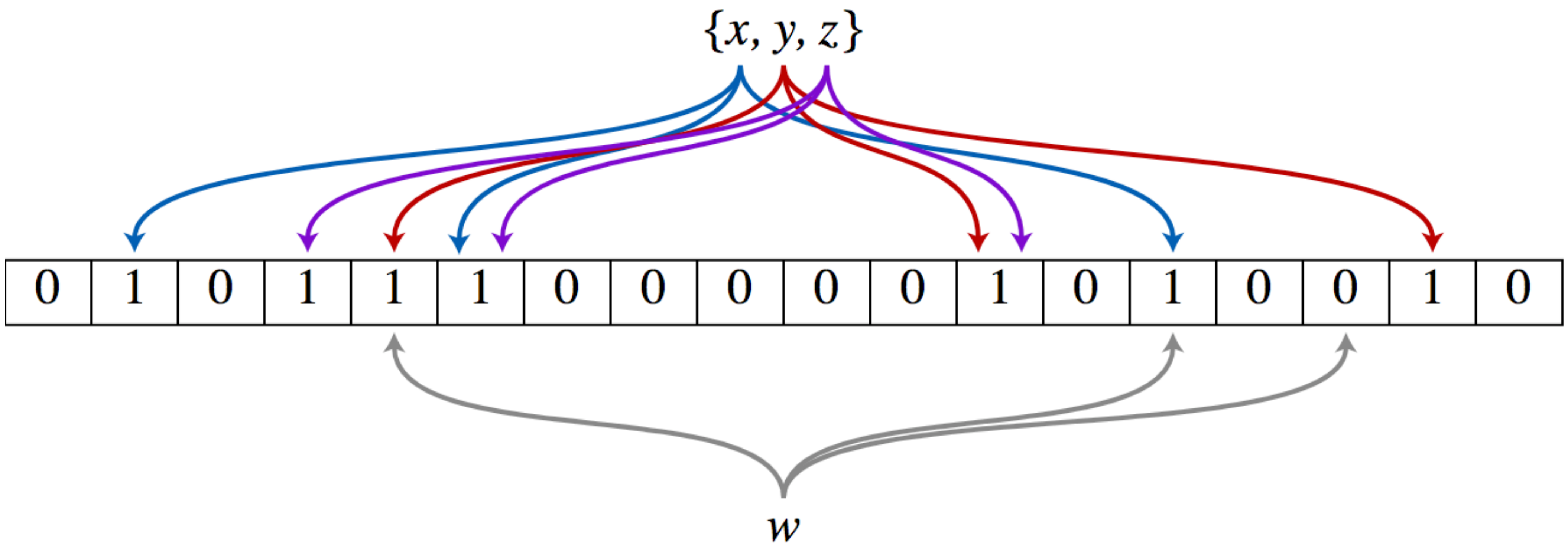
- Standard architecture of large databases
  - Cache + DB
    - Entries are taken from DB
    - Popular entries are cached (for efficiency reasons)
- Problem
  - Adversary can query database for non-existing entries
    - Search executes in the pessimistic time (often storage lookups)
- How to mitigate it?

Bloom Filters (BFs)

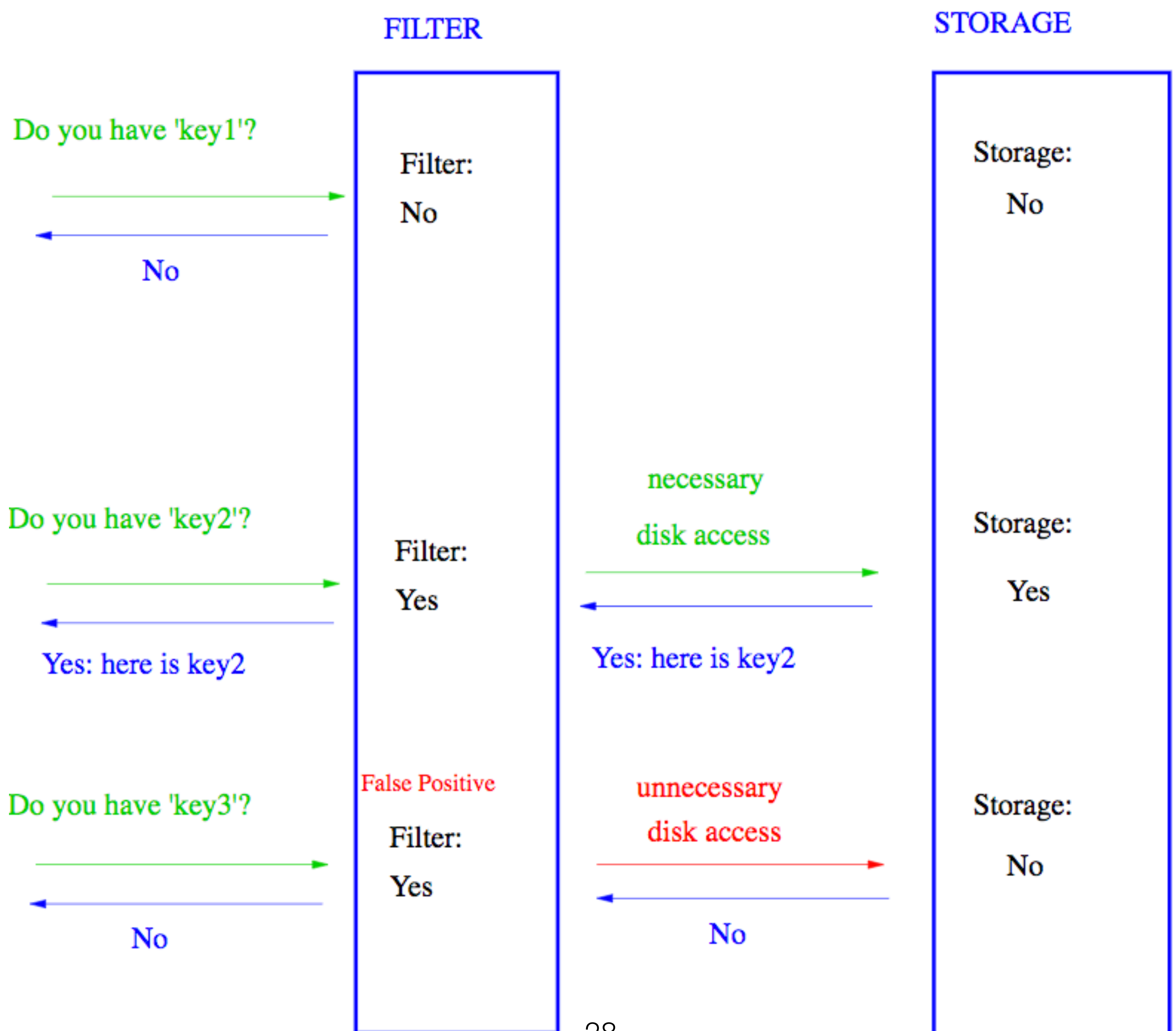
- Space-efficient probabilistic data structure
- Set membership (check quickly whether an element is in a set, without storing the element)
- m-bits long bit array (initially, all bits set to 0)
- k different hash functions, each maps set’s element to one of m array positions (usually $k << m$)
  - They do not have to be cryptographically-strong hash functions
- Adding element
  - Hash element with k hash functions and set 1 on the obtained positions
- Querying element
  - Hash element with k hash functions, if bits on all positions equal 1 return TRUE, o/w FALSE
  - False positives possible, no false negatives

BF Example

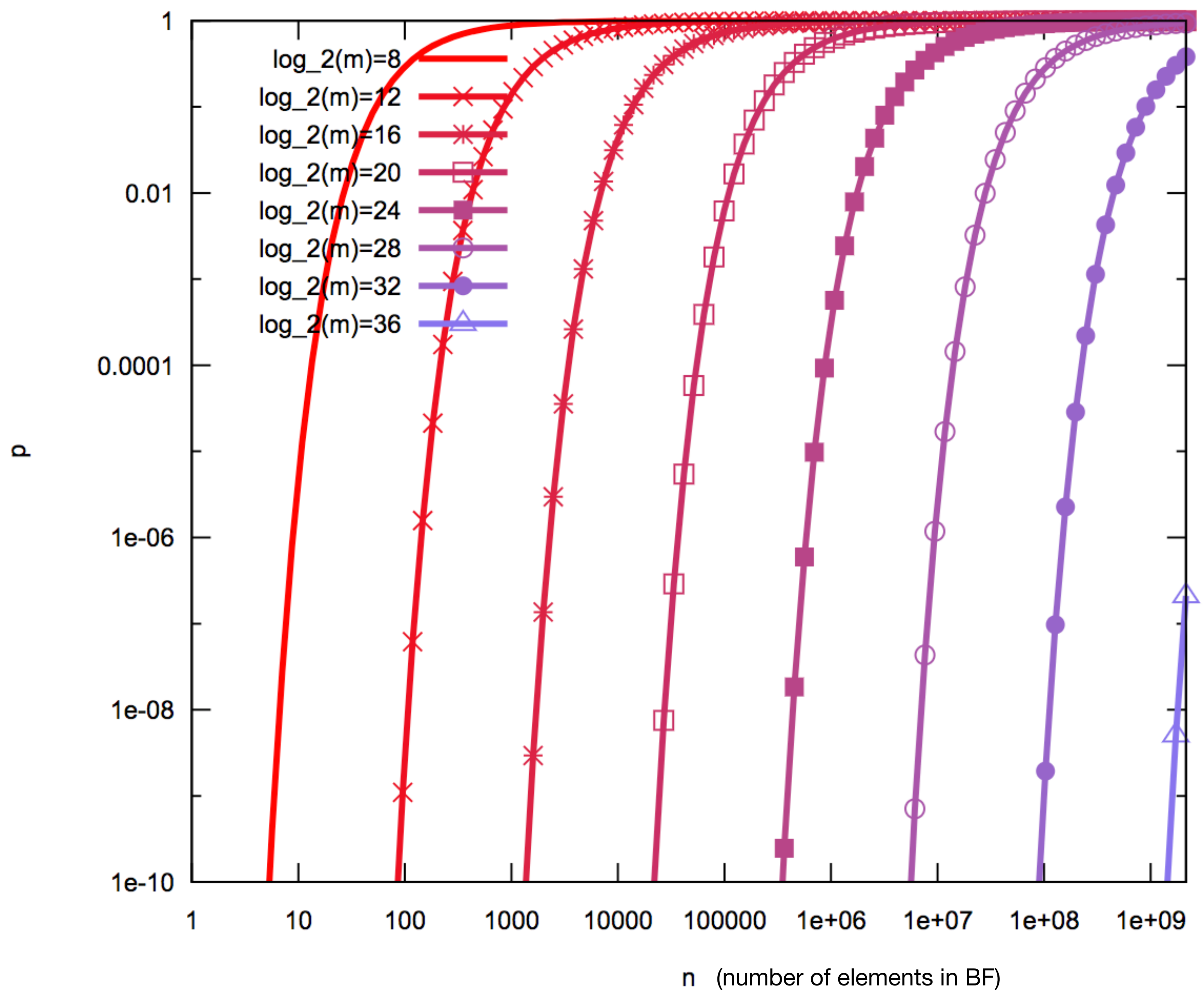
- Represent set {*x*,*y*,*z*}
- Query for *w*



BF-backed Database



False Positives Probability



k=(m/n)ln2 (optimal)

Reading

- Textbook: 1.4, 1.5, 2
- https://lpd.epfl.ch/site/_media/education/sdc_byzconsensus.pdf
- http://pmg.csail.mit.edu/papers/osdi99.pdf
- https://arxiv.org/pdf/1707.01873.pdf

Questions ?