Distributed Computing in Shared Memory and Networks

Class 3: Fault-Tolerant Distributed Services
Paxos, BFT, Blockchains

WEP 2018
KAUST

So far…

Shared-memory computing:

- Wait-freedom and linearizability
- Lock-based and lock-free synchronization
- Consensus and universality

Message-passing

- Consider a network where every two processes are connected via a reliable channel
  - ✓no losses, no creation, no duplication

- Which shared-memory results translate into message-passing?
- Implementing a distributed service

Implementing message-passing

Theorem 1 A reliable message-passing channel between two processes can be implemented using two one-writer one-reader (1W1R) read-write registers

Corollary 1 Consensus is impossible to solve in an asynchronous message-passing system if at least one process may crash

ABD algorithm:
implementing shared memory

Theorem 2[ABD] A 1W1R read-write register can be implemented in a (reliable) message-passing model where a majority of processes are correct

Implementing a 1W1R register

```
Upon write(v)
  t++
  send [v,t] to all
  wait until received [ack,t] from a majority
  return ok

Upon read()
  r++
  send [?,r] to all
  wait until received {(t',v',r)} from a
  majority
  return v' with the highest t'
```

Implementing a 1W1R register, contd.

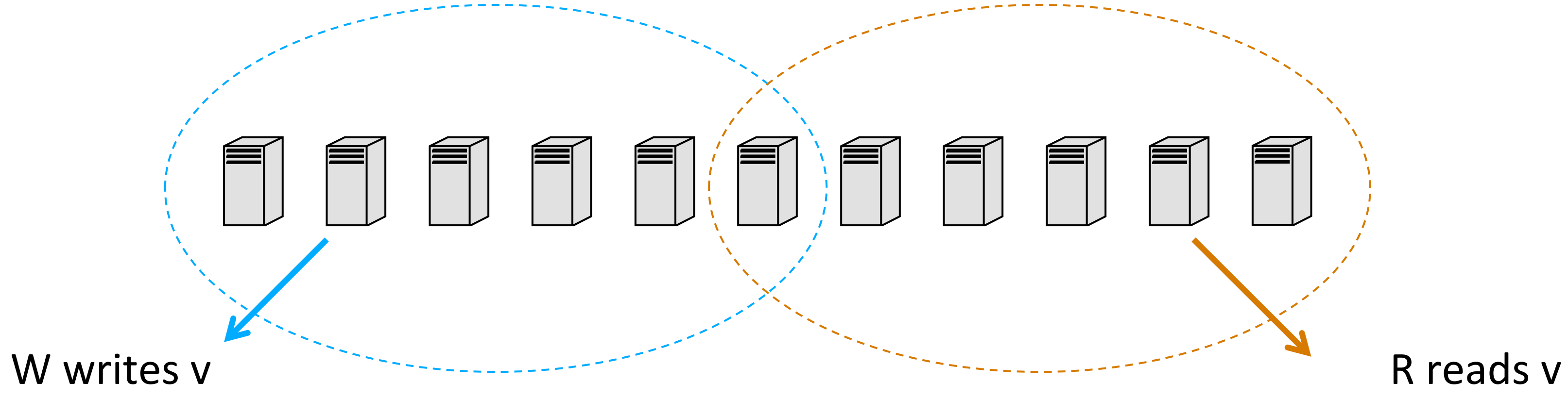
```
Upon receive [v,t]
  if $t>t_i$ then
        $v_i$ := v
        $t_i$ := t
        send [ack,t] to the writer

Upon receive [?,r]
  send [$v_i$,$t_i$,r] to the reader
```

A correct majority is necessary

Otherwise, the reader may miss the latest written value

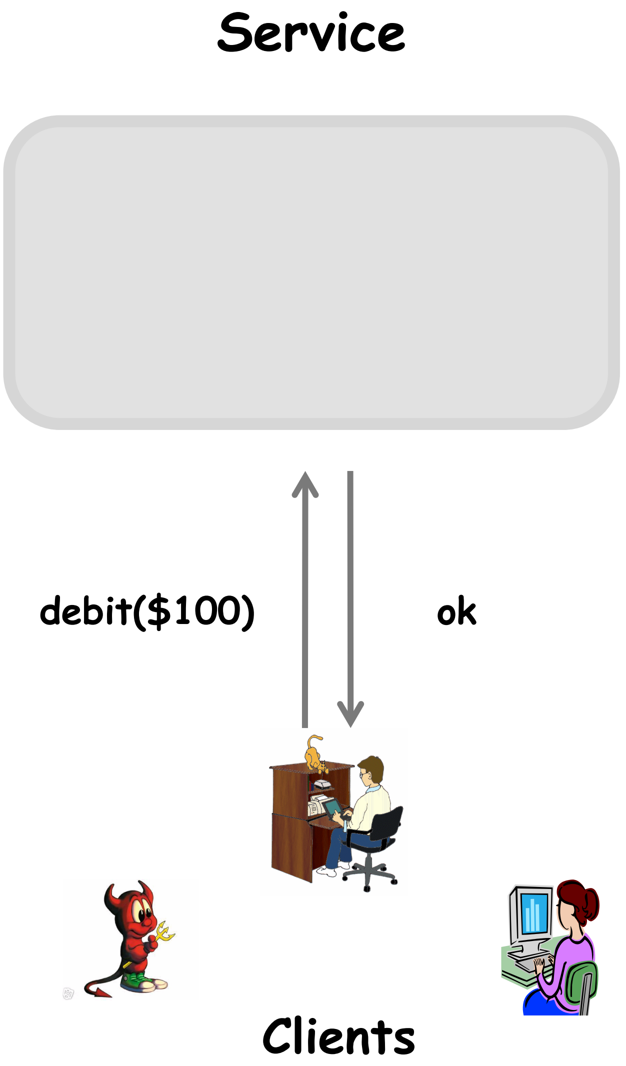
The quorum (set of involved processes) of any write operation must intersect with the quorum of any read operation:



How to build a consistent and reliable system?

Service accepts requests from *clients* and returns responses

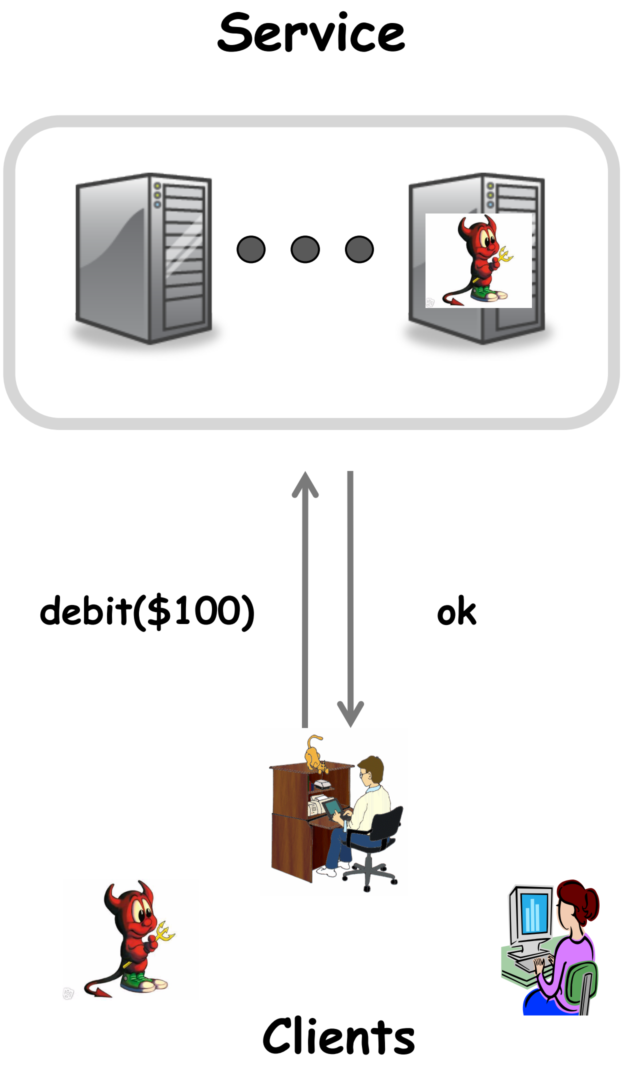
- Liveness: every persistent client receives a response
- Safety: responses constitute a total order w.r.t. the service's *sequential specification*



How to build a fault-tolerant system?

Replication:

- Service = collection of *servers*
- Some servers may *fail*



“CAP theorem” [Brewer 2000]

No system can combine:

- Consistency: all servers observe the same evolution of the system state
- Availability: every client’s request is eventually served
- Partition-tolerance: the system operates despite a partial failure or loss of communication

Sounds familiar, no?

Strongly consistent replicated state machine

Universal construction in message-passing:

- Clients access the service via a standard interface
- Servers run replicas of the (sequential) service
- (A subset of) faulty servers do not affect consistency and availability

Leslie Lamport: The Part-Time Parliament. ACM Trans. Comput. Syst. 16(2): 133-169 (1998)

Paxos: some history

- Late 80s: a three-phase consensus algorithm
  - ✓A Greek parliament reaching agreement
- 1989: a Paxos-based fault-tolerant distributed database
- 1990: rejected from TOCS

> "All three referees said that the paper was mildly interesting, though not very important, but that all the Paxos stuff had to be removed."

This submission was recently discovered behind a filing cabinet in the TOCS editorial office. Despite its age, the editor-in-chief felt that it was worth publishing. Because the author is currently doing field work in the Greek isles and cannot be reached, I was asked to prepare it for publication.

The author appears to be an archeologist with only a passing interest in computer science. This is unfortunate; even though the obscure ancient Paxon civilization he describes is of little interest to most computer scientists, its legislative system is an excellent model for how to implement a distributed computer system in an asynchronous environment.

…

Keith Marzullo
University of California, San Diego
(preface for the TOCS 1998 paper)

Paxos today

- Underlies a large number of practical system when strong consistency is needed
  - ✓Google Megastore, Google Spanner
  - ✓Yahoo Zookeeper
  - ✓Microsoft Azure
  - ✓….
- ACM SIGOPS Hall of Fame Award in 2012
- Turing award 2015

Consensus: recall the definition

A process *proposes* an *input* value in V (|V|≥2) and tries to *decide* on an *output* value in V

- *Agreement:* No two process decide on different values
- *Validity:* Every decided value is a proposed value
- *Termination:* No process takes infinitely many steps without deciding

  (Every correct process decides)

Model

- Asynchronous system
- Reliable communication channels
- Processes fail by crashing
- A majority of correct processes

But we proved that 1-resilient consensus is impossible even with shared memory!

“CAP theorem” is violated!

Where is the trick?

Ω: an oracle

- Eventual leader failure detector
- Produces (at every process) events:
  - ✓‹Ω, leader, p›
  - ✓We also write p=leader()
- Eventually, all correct processes output the same correct process as the leader

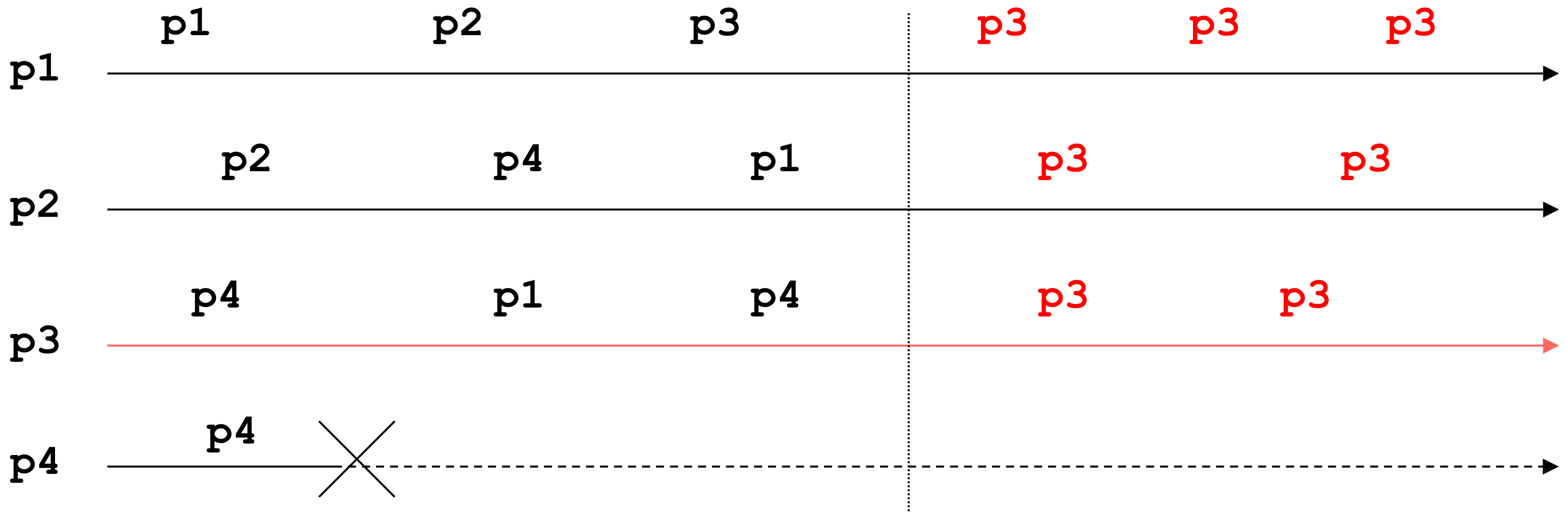
Can be implemented in eventually synchronous system:

- ✓There is a bound on communication delays and processing that holds only eventually
- ✓There is an a priori unknown bound in every run

Leader election Ω: example

There is a time after which the same correct process is considered leader by everyone.

(Sufficient to output a binary flag leader/not leader)



Paxos/Synod algorithm

- Let’s try to decouple liveness (termination) from safety (agreement)

- Synod made out of two components:
  - ✓Ω - the eventual leader oracle
  - ✓(ofcons) obstruction-free consensus

Obstruction-free Consensus (ofcons)

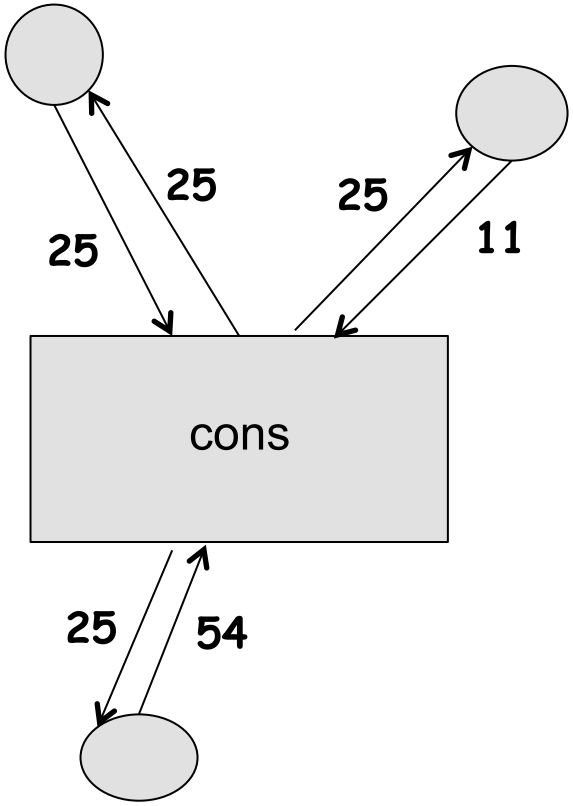
- Similar to consensus
  - ✓except for Termination
  - ✓ability to abort

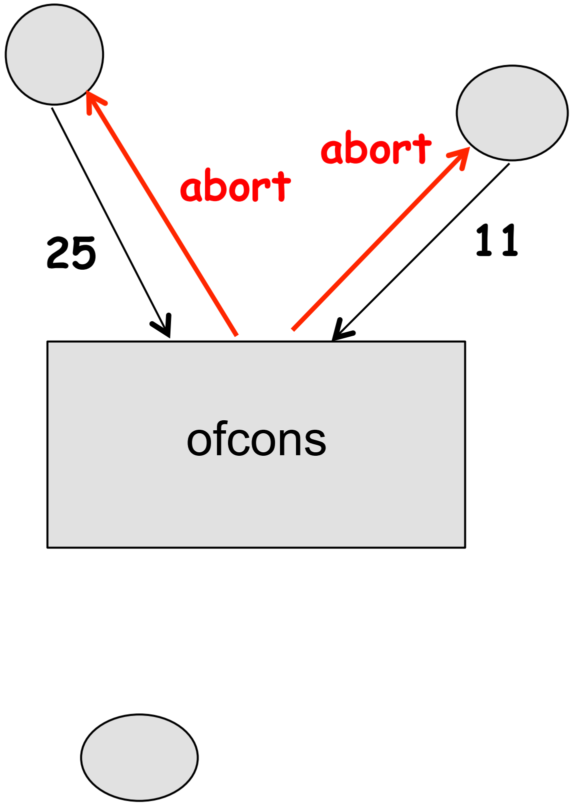
- Request:
  - ✓‹ofcons, propose, v›
- Indications:
  - ✓‹ofcons,decide, v’›
  - ✓‹ofcons,abort›

Obstruction-free Consensus

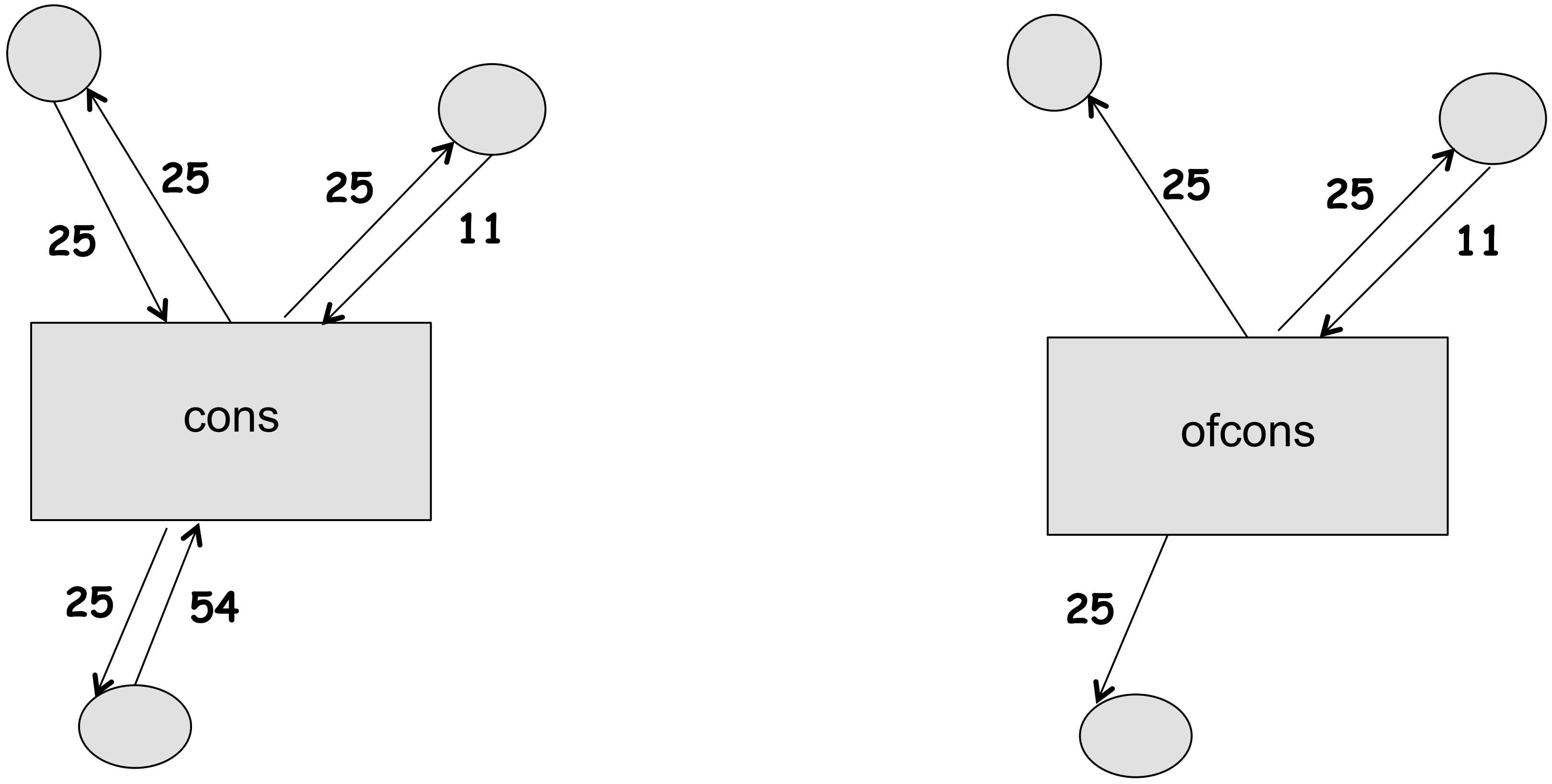
- *C1. Validity*:
  - ✓Any value decided is a value proposed
- *C2. Agreement:*
  - ✓No two correct processes decide differently
- *C3. Obstruction-Free Termination:*
  - ✓If a correct process p proposes, it eventually decides or aborts.
  - ✓If a correct process decides, no correct process aborts infinitely often.
  - ✓If a single correct process proposes a value sufficiently many times, p eventually decides.

Consensus vs. OF-Consensus





Consensus vs. OF-Consensus



Consensus using Ω and ofcons

- Straightforward
  - ✓Assume that in cons everybody proposes

```
upon ‹cons, propose, v›
  while not(decided)
      if   self=leader() then
       result = ofcons.propose(v)
       if  result=(decide,v’) then
           return v’
```

Link to Paxos/Synod

- External cons.propose events come in a state machine replication algorithm as requests from clients
  - ✓As in universal construction

- Focus now on implementing OFCons

OFCons

- Not subject to FLP impossibility!
- Can be implemented in fully asynchronous system
  - ✓Using the correct-majority assumption
  - ✓Or read-write
- Synod OFCons: a 2-phase algorithm

Synod OFCons I

```
Code of every process pi:

Initially:
   ballot:=i-n; proposal:=nil; readballot:=0; imposeballot:=0;
   estimate:= nil; states:=[nil,0]^n

upon ‹ofcons, propose, v›
   proposal := v; ballot:=ballot + n; states:=[nil,0]^n
   send [READ, ballot] to all

upon receive [READ,ballot’] from p_j
   if readballot ≥ ballot’ or  imposeballot ≥ ballot’ then
            send [ABORT, ballot’] to pj
   else
            readballot:=ballot’
            send [GATHER, ballot’, imposeballot, estimate] to pj

upon receive [ABORT, ballot] from some process
   return abort
```

Synod OFCons II

```
upon receive [GATHER, ballot, estballot, est] from pj
  states[pj]:=[est,estballot]

upon  #states ≥ majority //collected a majority of responses
  if ∃ states[pk]≠[nil,0] then
     select states[pk]=(est,estballot) with highest estballot
     proposal:=est;
  states:=$[nil,0]^n$
  send [IMPOSE, ballot, proposal] to all

upon receive [IMPOSE,ballot’,v] from $p_j$
   if readballot > ballot’ or  imposeballot > ballot’ then
          send [ABORT, ballot’] to $p_j$
  else
         estimate := v; imposeballot:=ballot’
         send [ACK, ballot’] to $p_j$
```

Synod OFCons III

```
upon received [ACK, ballot] from majority
  send [DECIDE, proposal] to all

upon receive [DECIDE, v]
   send [DECIDE, v] to all
   return [decide, v]
```

Correctness

- Validity
  - ✓Immediate

- Agreement (try to do it yourselves)
  - ✓When is the decided value determined?

- OF Termination
  - ✓Show that a correct process that proposes either decides or aborts
  - ✓If a single process keeps going
    - It will eventually propose with a highest ballot number not seen so far
    - This process will not abort with such a ballot number

Quiz 3.1

- Does the ABD algorithm run by one writer and multiple readers implement an atomic (linearizable) register?
- Prove that Synod satisfies Agreement:
  - ✓No two processes decide differ

Time Complexity

- Fault-free time complexity: 4 message delays

  + 1 communication step for decision relaible broadcast

- Optimizations

  ✓Getting rid of the first READ phase

- Allow a single process (presumed leader, say p1) to skip the READ phase in its 1st ballot

  ✓Reduces fault-free/sync time complexity to 2

From Synod to Paxos

- Paxos is a state-machine replication (SMR) protocol
  - ✓i.e., a universal construction given a sequential object

- Implemented as totally-ordered broadcast: exports one operation toBroadcast(m) and issues toDeliver(m') notifications

From Synod to Paxos: TO-Broadcast

- Every message m (to)broadcast by a correct process p_i is eventually (to)delivered by p_i
- Every message m delivered by a process p_i is eventually delivered by every correct process
- No message is delivered unless it was previously broadcast
- No message is delivered twice
- The messages are delivered in the same order at all processes

Implies totally ordered (linearizable) execution of clients' requests

From Synod to Paxos

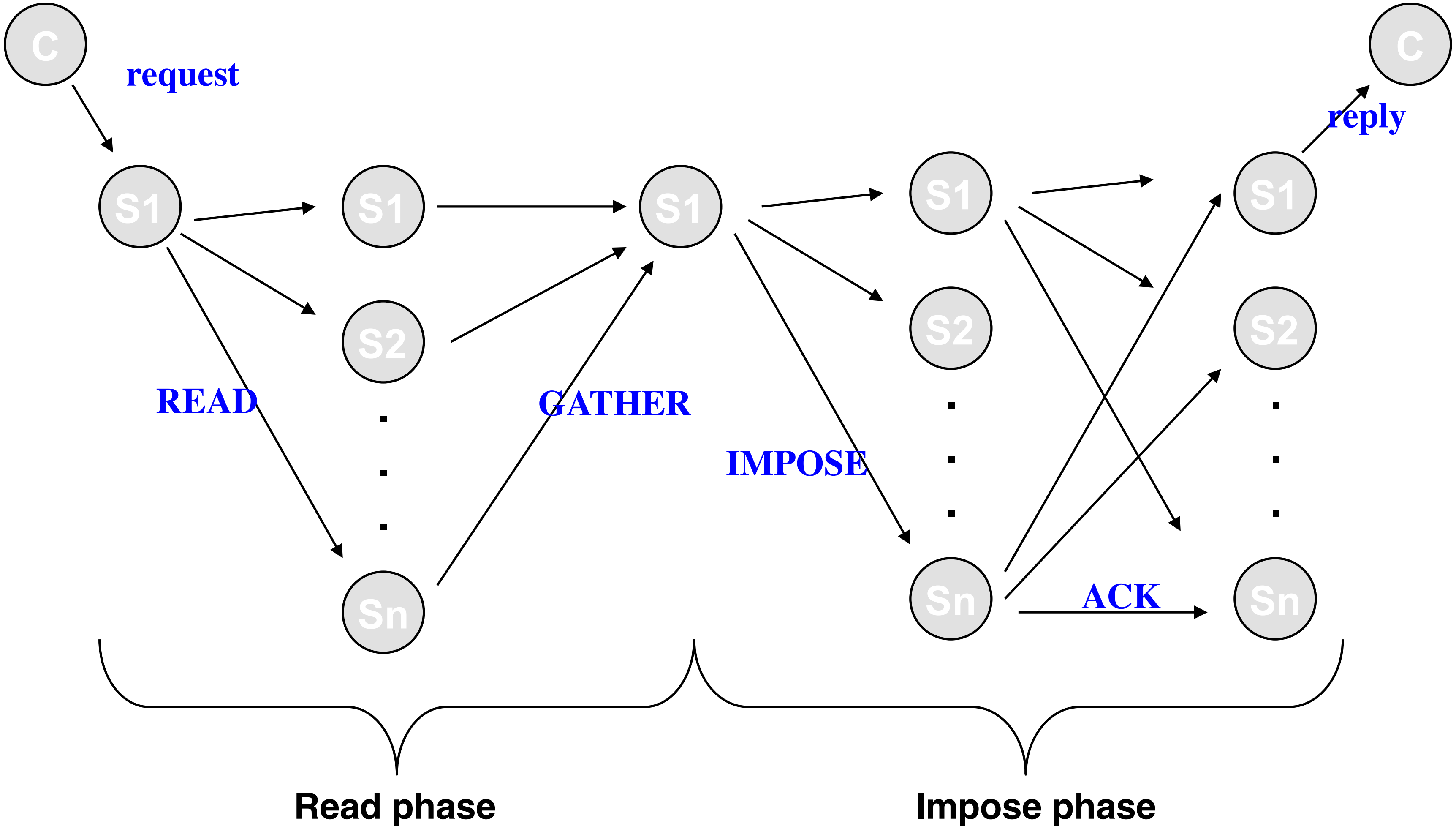
- But consensus (Synod) is one shot…
  - ✓How to most efficiently transform Synod to toBroadcast (Paxos)?

- Shared-memory universal construction?

Paxos SMR

- Clients initiate requests
- Servers run consensus
  - ✓Multiple instances of consensus (Synod)
  - ✓Synod instance 25 used to agree on the 25th request to be ordered
- Both clients and servers have the (unreliable) estimate of the current leader (some server)
- Clients send requests to the leader
- The leader replies to the client

Paxos failure-free/sync message flow



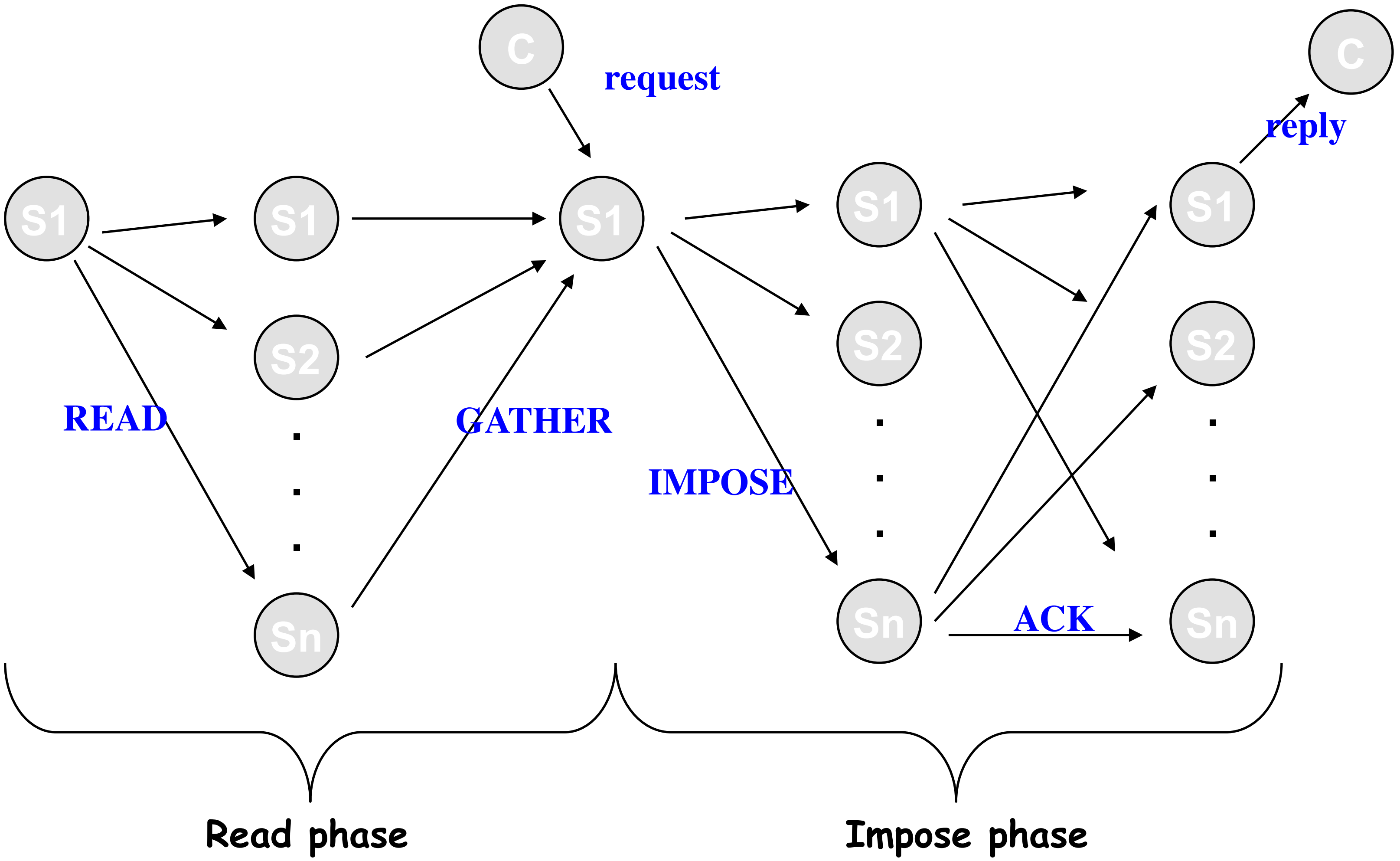
Observation

- READ phase involves no updates/new consensus proposals
  - ✓Makes the leader catch up with what happened before
- Most of the time the leader will remain the same
  - ✓+ nothing happened before (e.g., new requests)

Optimization

- Run READ phase only when the leader changes
  - ✓and for multiple Synod instances simultaneously
- Use the same ballot number for all future Synod instances
  - ✓run only IMPOSE phases in future instances
  - ✓Each message includes ballot number (from the last READ phase) and ReqNum, e.g., ReqNum = 11 when we're trying to agree what the 11th operation should be
- When a process increments a ballot number it also READs
  - ✓e.g., when leader changes

Paxos Failure-Free Message Flow



Potential Issues?

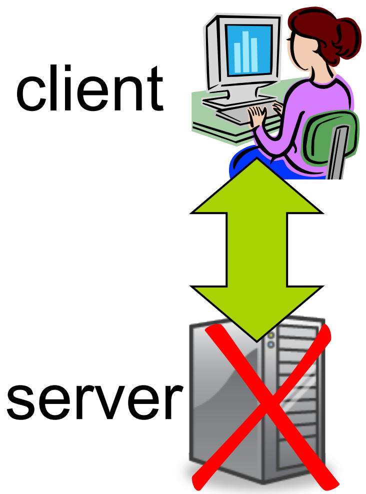
- Holes/gaps detected in the READ phase
  - ✓The leader detected a value in READ/GATHER for requests 1-12, 14, and 17
  - ✓but not for 13, 15 and 16
- The leader then runs the IMPOSE phase for instances 13, 15 and 16 with a special proposal
  - ✓A noop value ("do nothing")

Fault-Tolerant Distributed Services
BFT

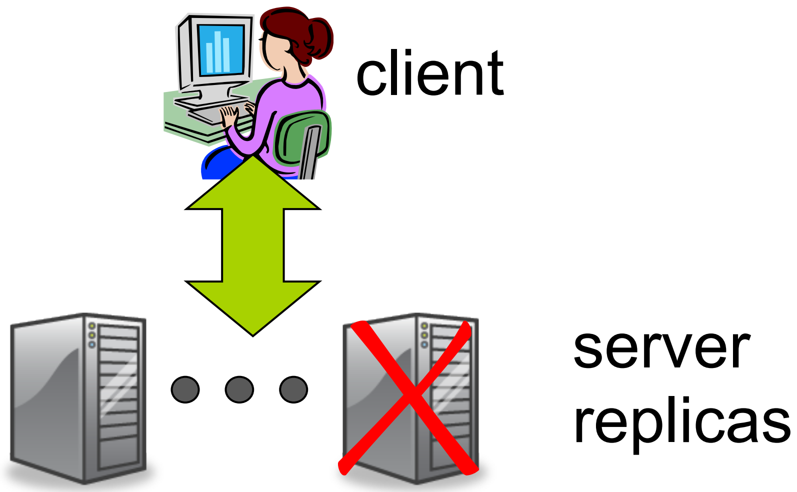
WEP 2018
KAUST

Context: Replication

unreplicated service



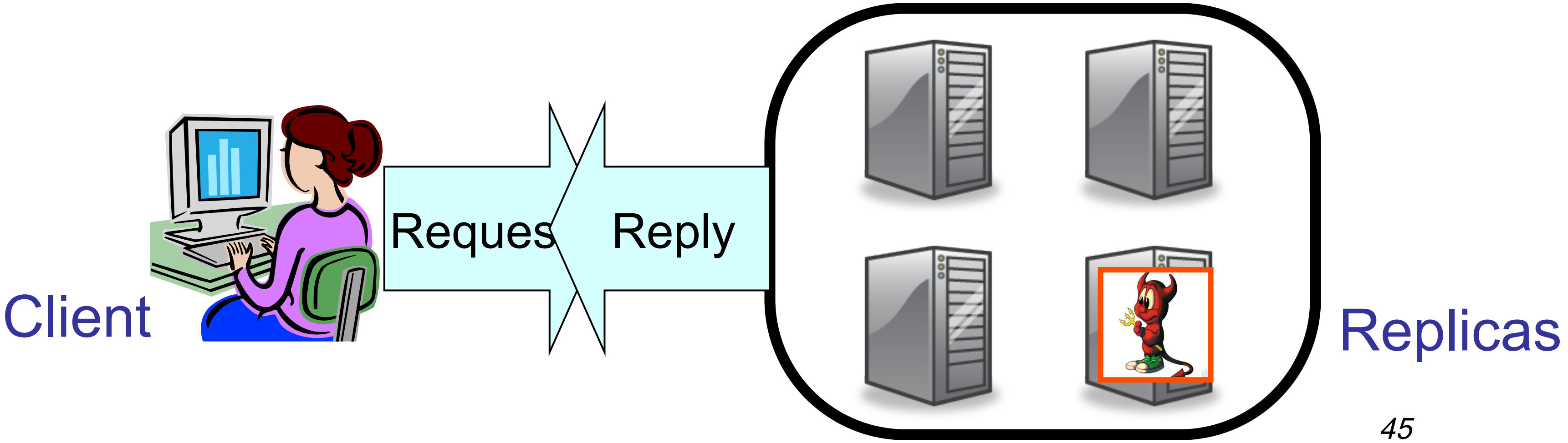
replicated service



- Assumptions
  - ✓ Network: synchronous/asynchronous
  - ✓ Digital signatures (availability trusted CA)
  - ✓ Failure Model – Benign (stopping) vs. Byzantine (arbitrary)

State-Machine Replication

- Replicated deterministic state machine
- Correct clients “see” replicated service as one correct server
  - ✓Requests are totally ordered
  - ✓Every request by a correct client is eventually served



Byzantine Generals
[Lamport, Shostak, Pease, 1982]

N armies face an enemy: an agreement should be reached on attack or retreat

- Agreement: no two correct processes decide differently
- Validity: if every correct process propose v, then v must be decided
- Termination: every correct process decides

Model: Byzantine faults (some generals can be traitors), synchronous, no crypto

The 2/3 bound

Split the armies in three groups: Commander, Lieutenant 1, Lieutenant 2.

Without signatures, the traitor may lie about received messages.

The two runs are indistinguishable to Lieutenant 1:

- Commander is faulty
- Leutenant 2 is faulty

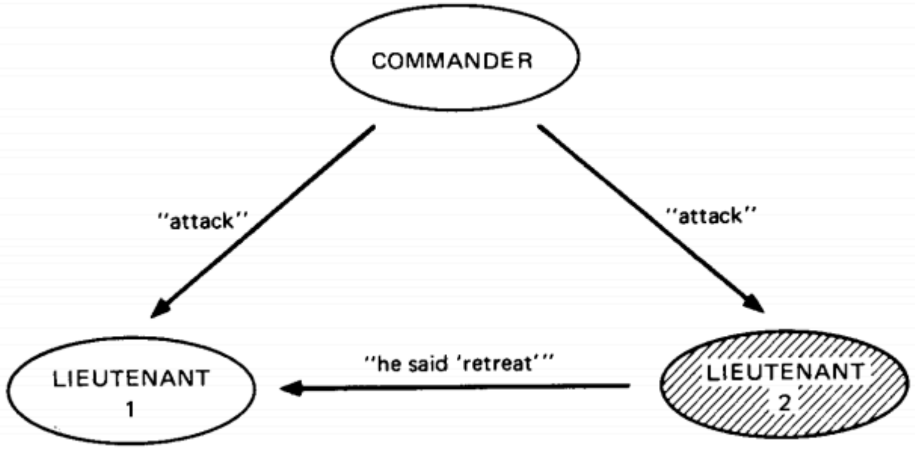


Fig. 1. Lieutenant 2 a traitor.

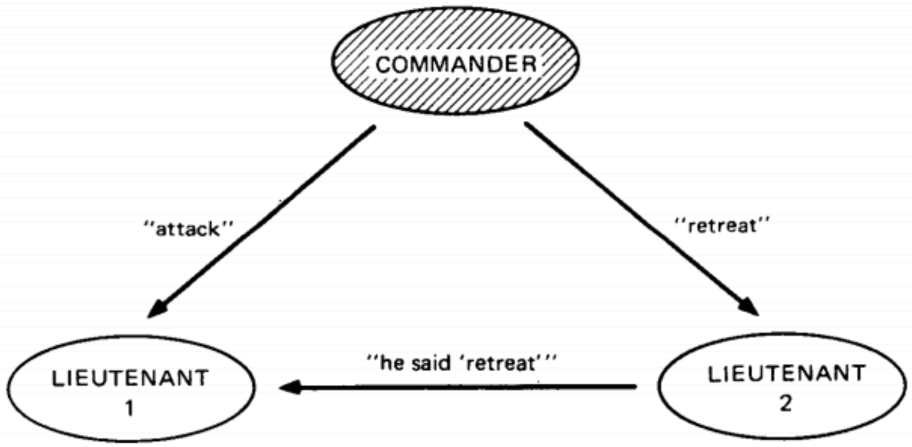
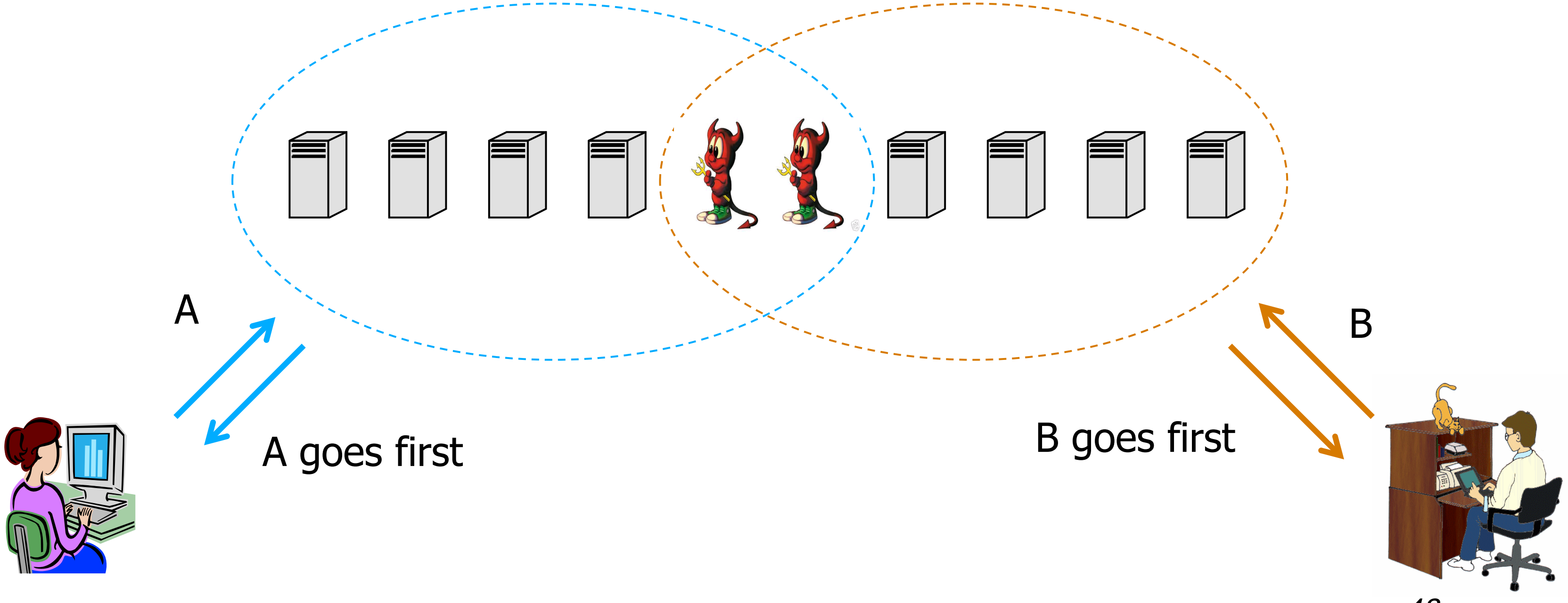


Fig. 2. The commander a traitor.

Signatures?

- Without crypto: both synchrony and >2/3 correct servers are needed
- With crypto: only 2/3
  - ✓Why? Every two requests should involve at least one common *correct* server



Safety vs. liveness

n – number of servers

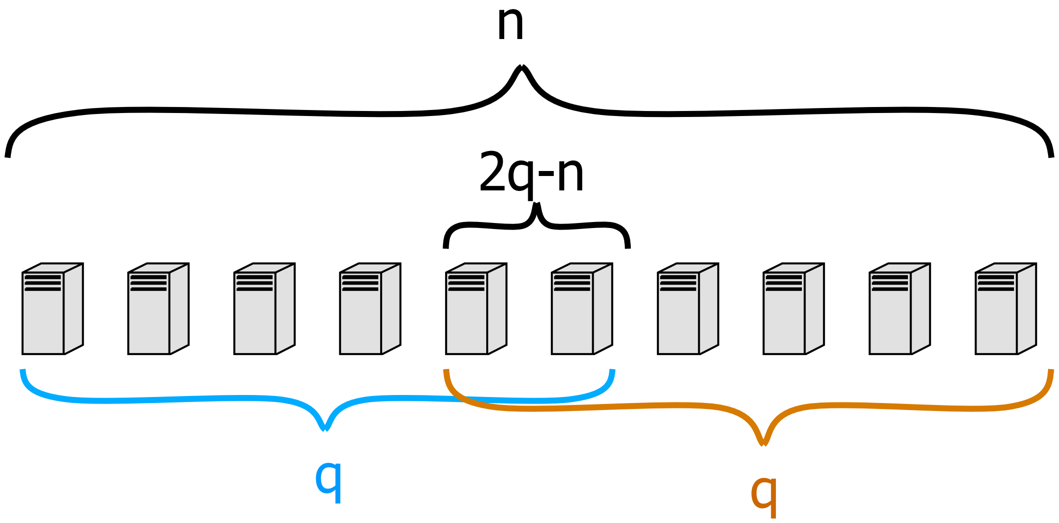
q – quorum size (number of servers involved in processing a request)

f – upper bound on the number of faulty servers

$2q-n \geq f+1$ **or** $q \geq (n+f+1)/2$ **(safety)**

$\Rightarrow n \geq 3f+1$

$n-f \geq q$ **(liveness)**

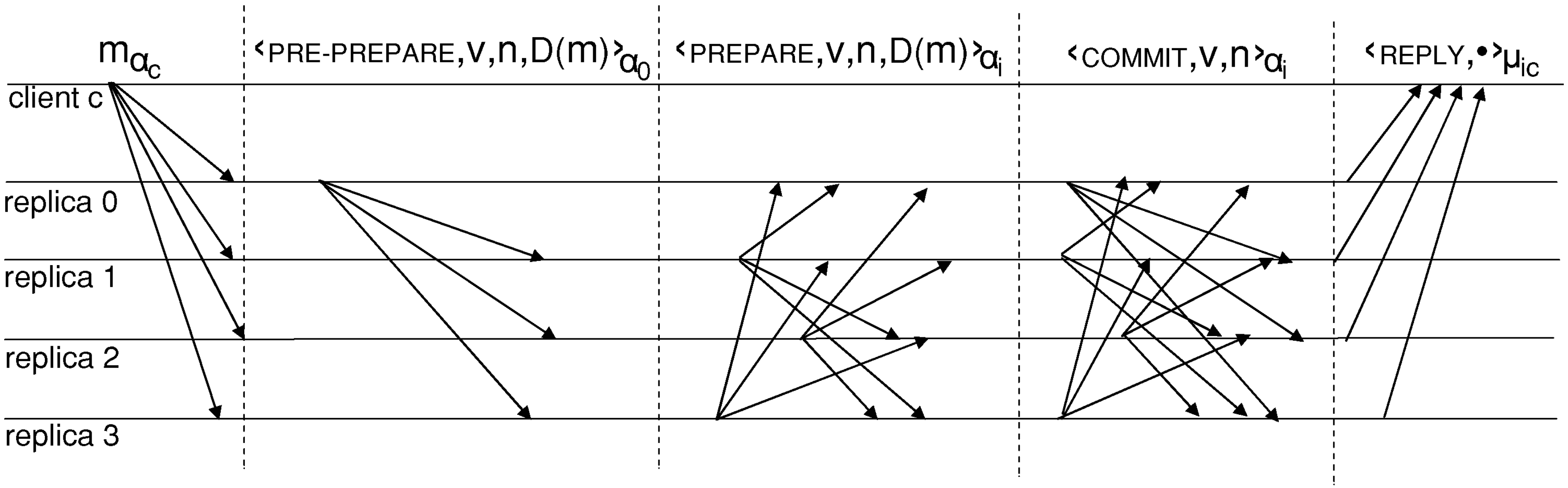


PBFT: Castro-Liskov

Pracrical Byzantine Fault-Tolerance (with Proactive Recovery), OSDI 1999

- A request (a batch of requests) involves a three-phase agreement protocol
- The system is *eventually* synchronous
- >2/3 of the service replicas (servers) must be correct

BFT: normal mode of operation


m_{α_c}
$\langle$PRE-PREPARE,v,n,D(m)$\rangle_{\alpha_0}$
$\langle$PREPARE,v,n,D(m)$\rangle_{\alpha_i}$
$\langle$COMMIT,v,n$\rangle_{\alpha_i}$
$\langle$REPLY,•$\rangle_{\mu_{ic}}$
client c
replica 0
replica 1
replica 2
replica 3

- Client sends request to all servers
- Primary broadcasts a pre-prepare request (sequence number, view, message hash)
- Servers exchange prepare messages
- Servers exchange commit messages
- Servers send commited tuple to client
- Client computes the outcome

All phases require a quorum (>2/3) to terminate and all messages are signed

BFT issues

- >2/3 assumption is reasonable if faults are independent
- Questionable for software bugs or attacks
- An obstacle for scalability: unlikely to hold for large number of replica groups [Farsite, OceanStore]

Another perspective

- Prepare for the worst and hope for the best

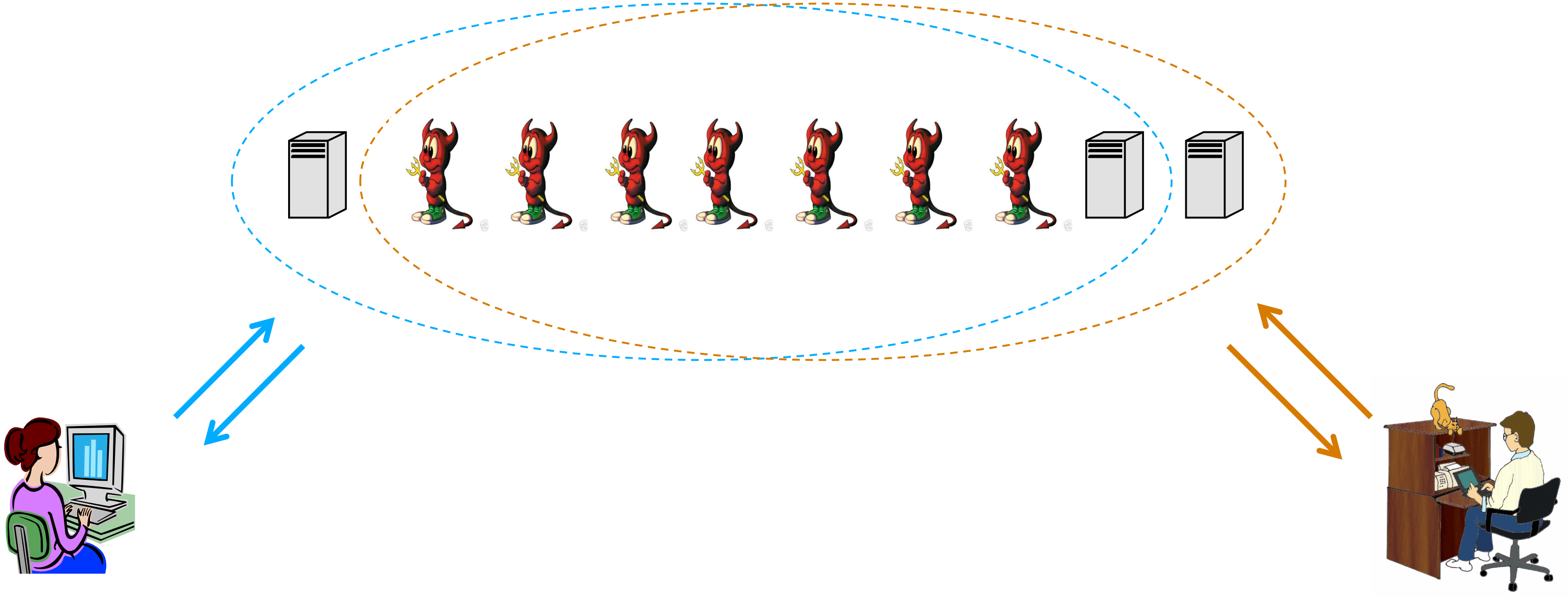
✓Best case – small fraction of faulty nodes → ensure safety+liveness

✓Worst case – some groups may have very large fraction of faulty nodes (beyond 1/3) → ensure safety

✓Rare case – a few nodes unavailable → lose liveness

Trading off liveness for safety

- Every request involves at least (n+f+1)/2 servers ⇒ safety is ensured as long as f or less servers fail
- Liveness will be provided if not more than n-(n+f+1)/2 = (n-f-1)/2 servers fail
- n=10, f=7: liveness tolerates at most one failure



Quiz 3.2

- The Byzantine generals setting assumes synchronous
- BFT assumes asynchronous system and digital signatures
- Both protocol assume >2/3 correct servers

Can you devise a synchronous state machine replication protocol with signatures that tolerate any number of faulty servers?

Fault-Tolerant Distributed Services
Blockchain basics

WEP 2018
KAUST

Chronology

1982 Byzantine Generals

1990 Paxos

1992 “ProofOfWork”

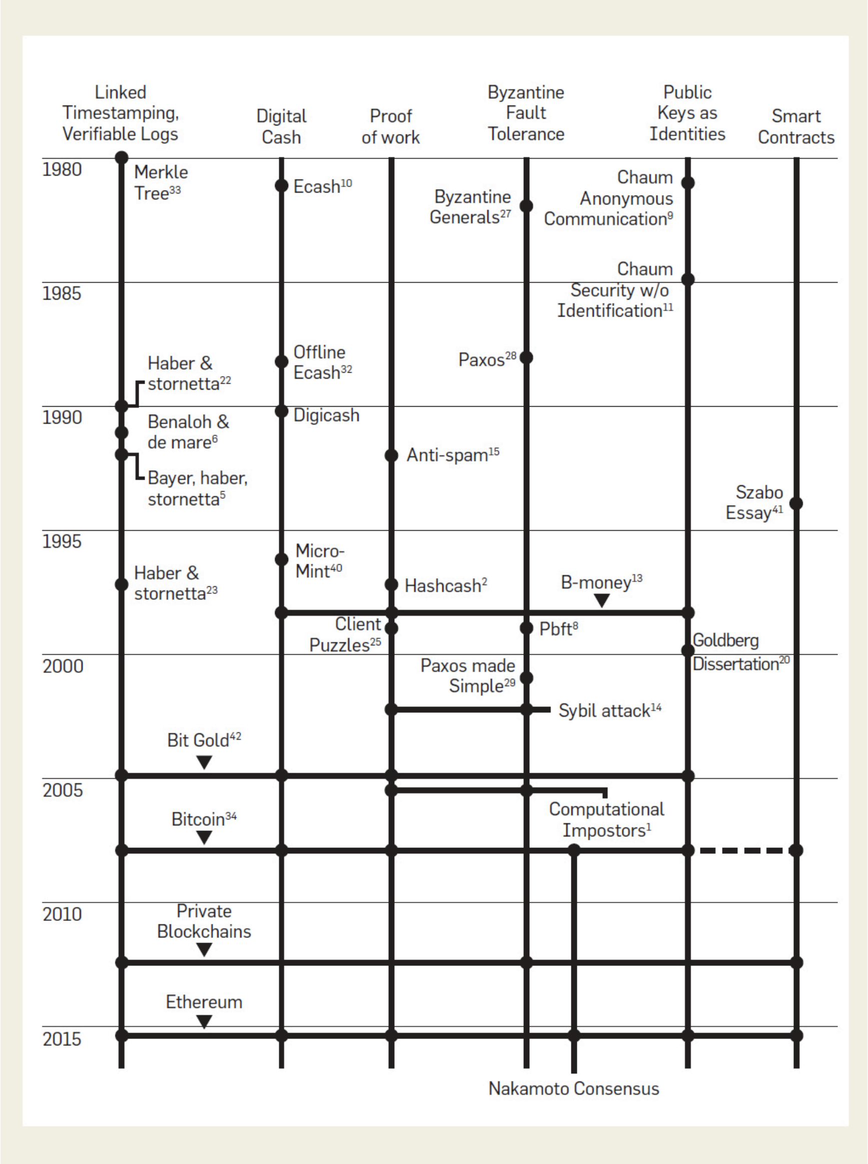
1999 PBFT

1995 Hashcash

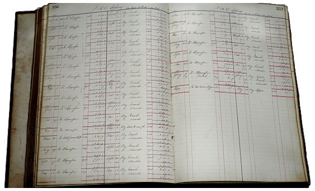
2002 Sybil attack

2009 Bitcoin

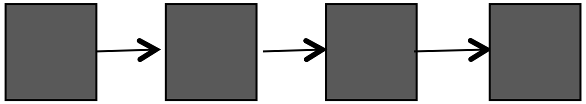
…



Distributed ledger?



Shared data structure: linear record of (blocks of) transactions



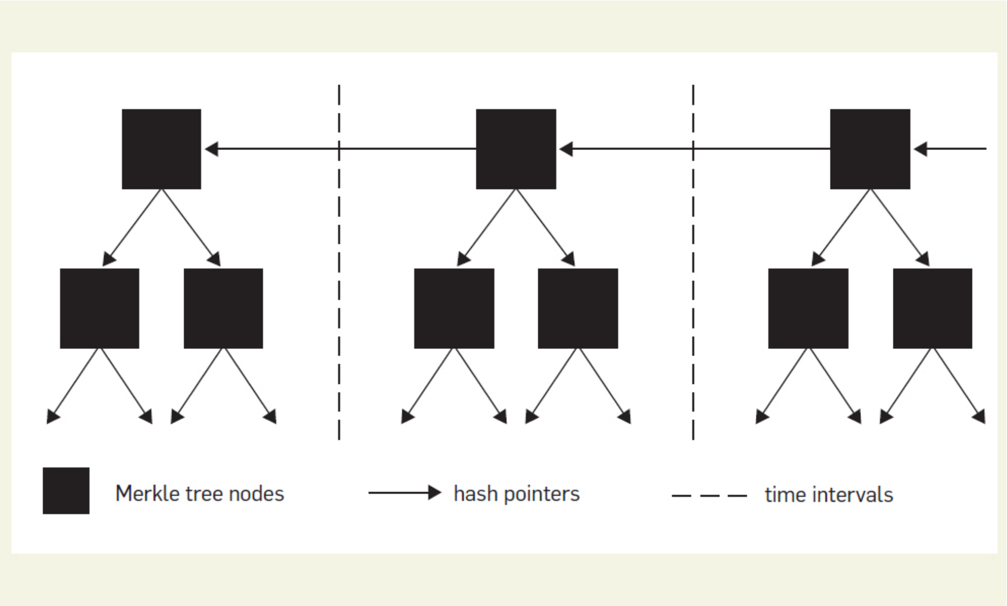
- Append-only
- Backtrack verifiable
- Consistent?

Open environment:

- No static membership
- No identities (public keys)
- Asynchronous?

Verification: linked timestamping

- A change in a block affects all following blocks
  - ✓ Originally with signatures: each block contains its signed predecessor
  - ✓ Now: hashchains
- Bitcoin: Merkle trees
  - ✓ Leafs: transactions
  - ✓ Intermediate: hashes of children
  - ✓ Roots: hashes of predecessor roots



Consistency?

- Sybil attack: the adversary can own an arbitrarily large fraction of participants
  - ✓ Why don't good guys do the same? ☺
- Classical consistent protocols don't work

- Assume a synchronous system
  - ✓ Message delays are bounded by δ
  - ✓ Need to "slow down" updates (wrt δ)

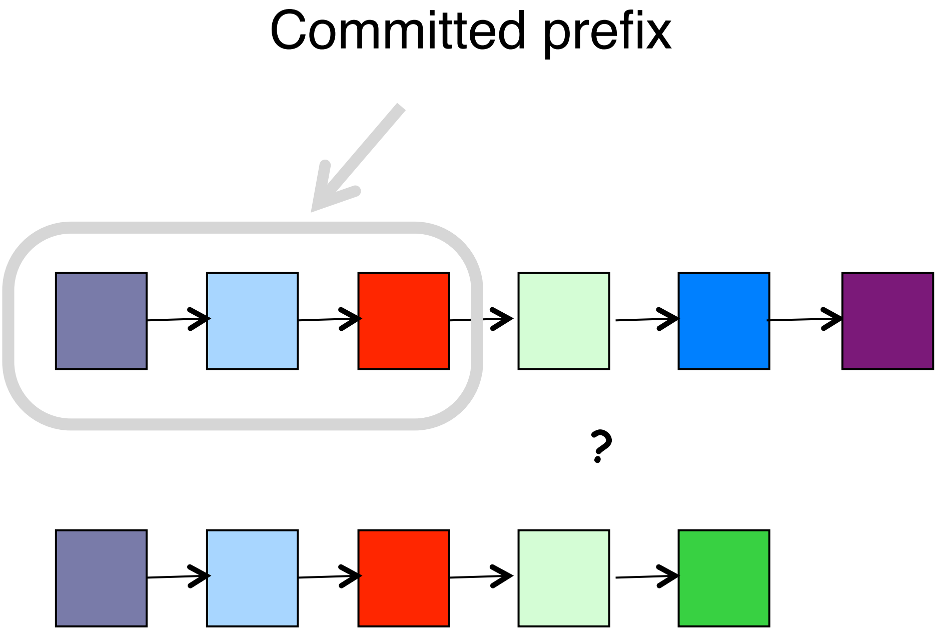
Proof of work

Need to solve a (time-consuming) puzzle to be able to affect the state of the ledger (blockchain)

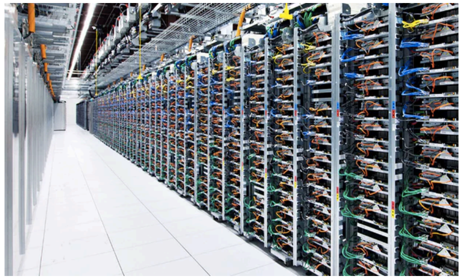
- Every process maintains a locally consistent copy of the ledger
  - ✓ Hashchain/Merkle tree
- To update (to “mine” a new block of transactions): broadcast a new block B=<s,x,ctr> containing a puzzle solution
  - ✓ H(ctr,G(s,x))<d (difficulty)

(Bitcoin) blockchain

- Clients broadcast an update
- Dedicated clients (miners) collect updates solve puzzles, update and broadcast their local ledgers
- Clients always choose the longest (verifiable) ledger
- Old enough blocks are considered consistent



When it works



“Nakamoto consensus”

- Expected time to solve the puzzle >> δ
- The adversary does not possess most of computing power

The probability of a fork drops exponentially with the staleness of blocks

When it does not work

- Asynchronous/ eventually synchronous communication, or
- An adversary controls half of computing resources, or
- Even a small probability of error cannot be tolerated, or
- Energy consumption is an issue

Bitcoin Consumes More Electricity Than Iceland

November 15, 2017 9:00 by Rahul Nambiampurath

The work of bitcoin miners all over the world are contributing to a massive rise in electricity consumption. Recent data reveals that current levels of consumption surpass those of the country of Iceland.<

When it is not needed?

- No Sybil attacks
  - ✓ Participation under control
- No need for consensus
  - ✓ Updates commute
  - ✓ Eventual consistency is good enough
  - ✓ Storage-like systems [ABD]

What's next? CAPES theorem?

Consistency /Availability/Partition-tolerance/ Energy-efficiency/Sybil-tolerance

Which combinations are possible?

- Relaxing consistency: from strong universal (Paxos) to application-specific to eventual (Amazon's Dynamo)
- Allowing energy waste (bitcoin)
- Relying on social studies (incentivizing)

Wrapping up

- Distributed computing becomes mainstream
- But it is hard: CAPES arguments pop up in one way or another
- Resources (time, memory, bandwidth, energy) are bounded
- We need to understand systems we devise!