TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Consensus in Distributed Systems

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2025/26

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Tipi di failure

- In che modo possono fallire i componenti di un SD?
- Diversi tipi di failure:
  - Crash: il componente si arresta, ma aveva funzionato correttamente fino a quel momento
  - Omissione: il componente non risponde ad una richiesta
  - Fallimento nella temporizzazione: il componente risponde ma il tempo di risposta supera l'intervallo specificato
  - Fallimento nella risposta: la risposta del componente non è corretta
    - Fallimento nel valore
    - Fallimento nella transizione di stato
  - Fallimento arbitrario (o *bizantino*): il componente può produrre una risposta arbitraria con tempi arbitrari
    - Guasto bizantino: sintomi diversi ad osservatori diversi
- I crash sono i fallimenti più innocui, quelli bizantini i più gravi

Modelli di failure

- *Problema nei SD*: distinguere tra componente che ha subito un crash ed uno che è solo troppo lento
  - Esempio: il processo P attende dal processo Q una risposta, che tarda ad arrivare
    - Q è soggetto ad un fallimento nella temporizzazione o ad una omissione?
    - Il canale di comunicazione tra P e Q è soggetto ad un guasto?
- Modelli di failure: dal meno al più grave
  - Fallimento **fail-stop**: Q ha subito crash e P può scoprire il fallimento (tramite timeout o preannuncio)
  - Fallimento **fail-silent**: Q ha subito crash o omissione, ma P non può distinguerli
  - Fallimento **fail-safe**: Q ha subito un fallimento arbitrario, ma senza conseguenze
  - Fallimento **fail-arbitrary**: Q ha subito un fallimento arbitrario non osservabile

Rilevare i fallimenti

- Per mascherare i fallimenti, bisogna innanzitutto rilevarli
- **Failure detection** per rilevare il fallimento di un processo
  1. *Attiva*: invio di un messaggio e timeout per rilevare se un processo è fallito
     - Soluzione più usata, adatta per fallimento fail-stop
  2. *Passiva*: attesa di ricevere un messaggio
  3. *Proattiva:* come effetto collaterale dello scambio di informazioni tra vicini (ad es. disseminazione delle informazioni basata su gossiping)
- Difficoltà con timeout
  - Come impostare il valore del timeout? Ok nei SD sincroni, ma in quelli asincroni?
  - Inoltre, timeout dipende anche dall'applicazione
  - Come distinguere tra fallimenti dei processi o della rete?

Practical failure detection

- How to reliably detect that a process has crashed
- Model
  - Each process has a failure detector, that provides it with a list of processes it suspects to have crashed
  - Process P probes process Q and waits for a response
  - If Q responds, Q is considered alive by P
  - If Q does not respond within timeout *t*, Q is suspected crashed
    - Synchronous system: suspected = crashed
- Practical implementation
  - If P does not receive a heartbeat from Q within timeout *t,* P suspects Q
  - If Q later responds to P, P stops suspecting Q and increases *t*
  - Note: if Q has crashed, P will keep suspecting Q

Distributed failure detection

- How to efficiently detect failures in large-scale DS
- Goal: design failure detectors that are both scalable and efficient
  - Efficiency: failures are detected quickly and accurately (without too many mistakes)
- Idea: exploit gossip protocols, where nodes periodically exchange failure information
- Hierarchical approach
  - Nodes are grouped hierarchically to reduce the number of messages passed during failure detection
  - ✓ Improves scalability and reduce message overhead

Gupta et al., On Scalable and Efficient Distributed Failure Detectors, PODC 2001
https://dl.acm.org/doi/epdf/10.1145/383962.384010

Distributed failure detection

- Failure detection algorithms:
  - Quorum-based detection: requires a subset of nodes (quorum) to agree before declaring a failure
    - ✓ Balances reliability and communication cost
  - Eventual detection: guarantees detection over time but not immediately after a failure
    - ✓ May involve temporary false negatives, but is suitable for large systems

Resilienza dei processi

- In ingegneria: resilienza = capacità di un materiale di resistere a forze di rottura
- Nei SD: capacità del SD di fornire e mantenere un livello di servizio accettabile in presenza di guasti e minacce alla normale operatività

- Come mascherare in un SD il guasto di un processo?

  Replicando e distribuendo la computazione in un gruppo di processi

Gruppi e mascheramento dei guasti

- Consideriamo un gruppo di processi
- Un gruppo composto da *N* processi è ***k*-fault tolerant** se può mascherare *k* guasti concorrenti
  - *k* è detto grado di tolleranza ai guasti

- Quanto deve essere grande un gruppo *k*-fault tolerant?
  - **Dipende** dal modello di failure

Gruppi e mascheramento dei guasti

- Quanto deve essere grande un gruppo *k*-fault tolerant?
  - Fallimento fail-stop o fail-silent → $N >= k+1$ processi
    - Nessun processo del gruppo produrrà un risultato errato, quindi è sufficiente il risultato di un solo processo non guasto
  - Fallimento arbitrario e il risultato del gruppo è definito tramite un meccanismo di voto → $N >= 2k+1$ processi
    - Abbiamo bisogno di $2k+1$ processi non guasti in modo che il risultato corretto possa essere ottenuto con un voto a maggioranza
- *Assunzioni* importanti:
  1. I processi sono identici
  2. I processi eseguono i comandi nello stesso ordine
  - Per essere certi che tutti i processi facciano esattamente la stessa cosa

Consenso nei sistemi distribuiti

- *Assunzione*: consideriamo ora che i processi del gruppo *non* siano identici, ovvero che ci sia una computazione distribuita
- *Obiettivo*: i processi non guasti del gruppo devono raggiungere un **consenso** (accordo) unanime su uno stesso valore (es. il prossimo comando da eseguire) in un numero finito di passi, nonostante la presenza di processi guasti
  - Esempi di consenso: elezione di un coordinatore, mutua esclusione, commit di una transazione
- Che tipo di guasti?
  - Guasti non bizantini (es. crash, omissioni)
  - Guasti bizantini

Consensus = agreement?

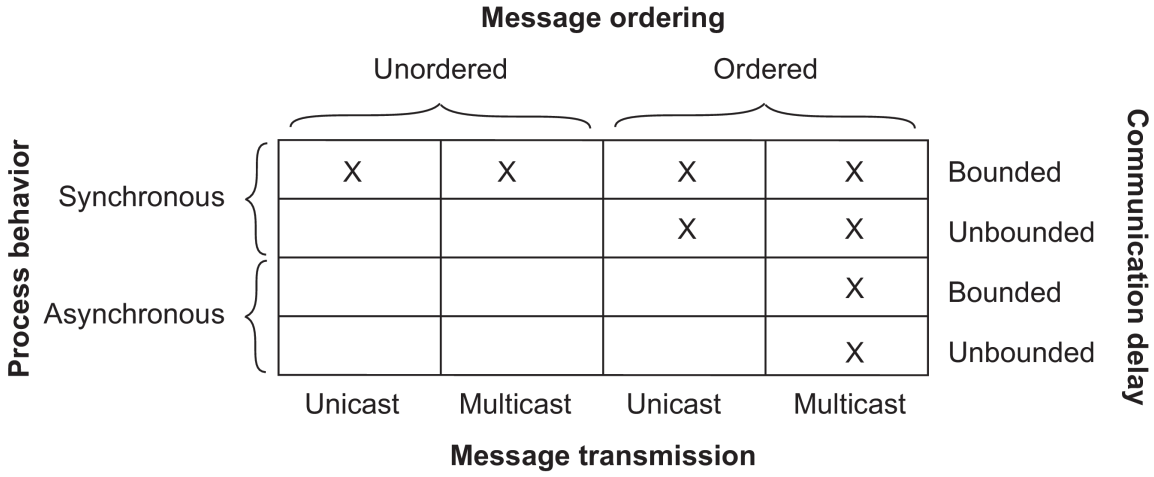
- In the course we treat these terms as synonyms
- But for the theoretical DS community the two terms refer to very similar but not identical problems
  - Agreement problem: a single process has the initial value
  - Consensus problem: all processes have an initial value
- We'll consider 3 consensus algorithms
  1. **Paxos** (only overview)
  2. **Raft**
  3. **Byzantine generals**
  - The first two focuns on consensus, the last on agreement (see original algorithm by Lamport)
- Let us first examine the necessary conditions for consensus and the FLP impossibility result

Consenso distribuito: quando può essere raggiunto

- Consideriamo le diverse tipologie di:
  - processi
  - ritardi di comunicazione
  - ordinamento dei messaggi
  - trasmissione dei messaggi
- Processi: *sincroni* o *asincroni*?
  - Processi sincroni: operano in modalità *lock-step*, ovvero esiste *c* >= 1 tale che se un processo ha eseguito *c*+1 passi, ogni altro processo ha eseguito almeno 1 passo
- Ritardo nella comunicazione: *limitato* o *illimitato*?
- Ordinamento dei messaggi: messaggi consegnati nello stesso *ordine* in cui sono stati inviati oppure *senza ordine*?
- Trasmissione dei messaggi: *unicast* o *multicast*?

Consenso distribuito: quando può essere raggiunto

- Quali sono le condizioni necessarie per poter raggiungere il consenso in un SD soggetto a guasti?



FLP impossibility result

- FLP impossibility result:

Fischer, Lynch and Patterson, "Impossibility of distributed consensus with one faulty process", 1985

  - In an asynchronous model with an unbounded but finite message delay, where only one processor might crash, there is no distributed algorithm that solves the consensus problem
  - They prove that no asynchronous algorithm for agreeing on a one-bit value can guarantee that it will terminate in the presence of crash faults
    - And this is true even if no crash actually occurs!

- The good news: impossibility means "is not always possible"
  - FLP proves that any fault-tolerant algorithm solving consensus has runs that never terminate, but these runs are extremely unlikely ("probability is zero")

FLP impossibility result

- What makes consensus hard? *Membership* in an asynchronous environment:
  1) we can't detect failures reliably because process speeds and channel delays are not bounded
  2) a faulty process stops sending messages but a "slow" message might confuse us
- Are distributed and Cloud systems asynchronous?
  - Not in the sense of the definition used in the FLP result, where asynchronous systems have no common clocks and make no use of time, and networks never lose messages but can delay them arbitrarily long
  - In practice we have partially synchronous systems: most of the time, we can assume the system to be synchronous, yet there is no bound on the time that a system is asynchronous
    - And we can reliably detect crash failures (see slide 4)

Paxos

- Fault-tolerant consensus protocol run by *N* processes to tolerate up to *k* failures (where *N* >= 2*k*+1) in an asynchronous system
- Important and widely studied/used protocol
  - Perhaps the most important consensus protocol
  - The dominant offering for consensus in asynchronous systems
- Rather a family of consensus protocols
  - Cheap Paxos, fast Paxos, generalized Paxos, byzantine Paxos, …
- We consider the basic version:

  L. Lamport, "Paxos made simple", *ACM SIGACT News*, Vol. 32, No. 4, Dec. 2001.

The history of Paxos

- Elegant and relatively simple algorithm
  - but "the original presentation was Greek to many readers" because presented through the allegory of a fictitious ancient Greek parliamentary government on the island of Paxos
- See the original paper The part-time parliament
  - "Inspired by my success at popularizing the consensus problem by describing it with Byzantine generals, I decided to cast the algorithm in terms of a parliament on an ancient Greek island."
  - "To carry the image further, I gave a few lectures in the persona of an Indiana-Jones-style archaeologist."
  - "My attempt at inserting some humor into the subject was a dismal failure. People who attended my lecture remembered Indiana Jones, but not the algorithm. People reading the paper apparently got so distracted by the Greek parable that they didn't understand the algorithm." (L. Lamport)
- "The Paxos algorithm, when presented in plain English, is very simple." (L. Lamport)

Paxos: distributed system model

Assumptions (rather weak and realistic)

- Partially synchronous system (it may even be asynchronous) with non-arbitrary failures
- Processes communicate with one another by sending messages
- Communication between processes may be unreliable, indeed messages:
  - Take arbitrarily long to be delivered
  - May be duplicated or lost
  - Corrupted messages can be detected and thus subsequently ignored

Paxos: distributed system model

Assumptions (rather weak and realistic)

- Processes:
  - Set of processes that run Paxos is known a-priori
  - Operate at arbitrary speed
  - Are fail-stop: may exhibit crash failures but **not** arbitrary failures
  - Can be restarted and rejoin if they fail
  - Can remember some information if restarted after failure (accessing a persistent storage that survives crashes)
  - Do not collude (i.e., do not team up to produce a wrong result)

Paxos: a quorum-based protocol

- Problem: get a set of processes to reach consensus on a single proposed value
  - no value proposed → no value chosen
  - value chosen → processes should learn the chosen value

- Main idea: a **quorum-based** protocol
  - Proposals are associated with a *unique sequence number*
  - Processes vote on each proposal
  - A proposal approved by *a majority will get passed*
  - Size of majority is "well known" because potential membership of system was known a-priori
  - A process considering two proposals approves the one with the larger sequence number

Paxos requirements

- Safety requirements
  - Only a value that has been proposed may be chosen
  - Only a single value is chosen
  - A process never learns that a value has been chosen unless it really *has* been chosen
- Don't care *what value is chosen, just as long* as it satisfies those three requirements

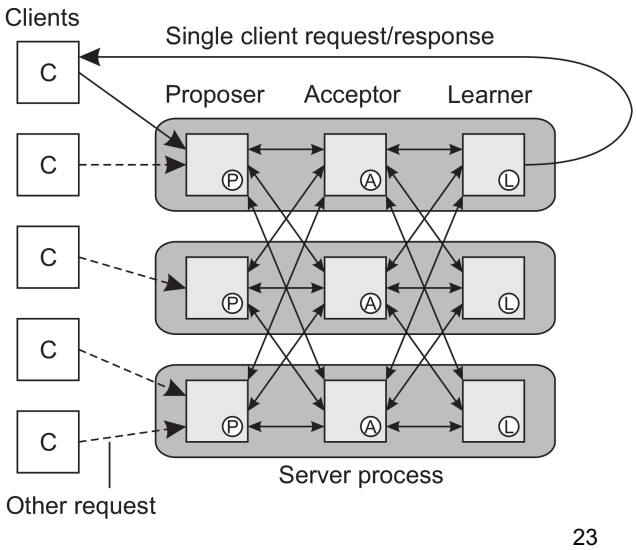
- Liveness requirements
  - Some proposed value is eventually chosen
  - If a value has been chosen, a process can learn the value

Paxos compromise

- Remember FLP impossibility result
  - No asynchronous consensus algorithm can guarantee both liveness and safety
- However, asynchrony of FLP is *overly pessimistic*
- Real systems are usually partially synchronous
  - System behavior is close to synchronous “most of the time”, sometimes goes asynchronous
- Practical compromise
  - Compromise liveness when system behaves asynchronously
  - But never safety
- Therefore
  - Paxos **does not guarantee liveness**
  - Might never terminate, but in practice it does terminate

Paxos roles

- A process may play 3 different roles:
  - *Proposer*, *acceptor*, *learner*
- Proposers *propose* a value to be chosen on behalf of clients
- Acceptors (i.e., voters) *decide* which value to choose
- Learners *learn* which value was chosen and report final decision back to clients
- Processes can play 1, 2, or all 3 roles
  - Thinking of these roles as being separate makes Paxos easier to understand



Paxos: some ideas

1. One acceptor makes things really easy:
   - But this isn't distributed at all: if acceptor fails, game over!
   - So, let's have multiple acceptors, each of which can accept a proposed value
2. A value is chosen once a simple majority of acceptors accepts it
   - If *m* acceptors, then > *m*/2 need to accept
   - Why does this work?
     - Any two majorities of acceptors must have at least one acceptor in common
     - An acceptor can accept only one value at a time
     - Therefore, any two majorities that choose a value must choose the same value
   - Just need to make sure acceptors do not accept something else once a value is chosen

Paxos: some ideas

3. Acceptors need a way to distinguish one proposal from another
   - Proposers assign a unique sequence number to each proposal they make
   - A proposal has two parts:
     - Proposal number (i.e., the unique identifier)
     - Proposed value (could be a decision value or some other information, such as "Frank's new salary" or "Position of Air France flight 21")
   - There can be *multiple distinct* proposals for the same value
     - But they differ by the proposal number
4. Stable storage, preserved during failures, is used to maintain information that must be remembered in case of failure

Paxos: some ideas

5. Paxos use a multiple-round approach
   - Once a decision on a value is reached in a round, decisions in all subsequent rounds must agree
     - Once decision is reached on a value, Paxos must force future proposers to select that same value
   - *Within each round*, finding consensus is a two-phase process, where each phase consists of:
     - a request sent from a proposer to a group of acceptor
     - a reply from the acceptors to the proposer
   - The two phases are:

     1) **Prepare**

     2) **Accept**

Paxos: the two phases

- At high level:

  1) **Prepare**

  - A proposer asks a majority of acceptors whether anyone already received a proposal
  - If the answer is no, propose a value

  2) **Accept**

  - If a majority of acceptors agree to this value, then that is our consensus
- Goals of the two phases:

  Prepare
  - Proposers check whether a value has already been chosen
  - Older proposals that have not yet completed are blocked

  Accept
  - Ask acceptors to accept a specific value

Paxos algorithm: phase 1

- Phase 1 (**prepare**):
  a) A proposer selects a proposal number *n* and sends a *prepare request* with number *n* to a majority of acceptors

  b) If an acceptor receives a prepare request with number *n*:
     - If *n* is greater than the proposal number of *any* prepare request the acceptor has responded to, the acceptor promises not to accept any lower-numbered proposals and replies with the highest-numbered proposal and the proposed value the acceptor has accepted, if any
     - Otherwise the acceptor does not respond (or responds with a negative ack)

Paxos algorithm: phase 2

- Phase 2 (**accept**):
  a) If the proposer receives a response to its prepare requests for the proposal numbered *n* from a majority of acceptors, it sends an *accept request* to each of those acceptors for a proposal numbered *n* with a value *v* which is the value of the highest-numbered proposal among the responses of phase 1 (if no acceptor had accepted a proposal up to this point, then the proposer may choose any value for its proposal)
  b) If an acceptor receives an accept request for *n*, it accepts the proposal unless it has already responded to a prepare request having a number greater than *n*

- Definition of chosen
  – A value is chosen at proposal number *n* *iff* majority of acceptors accept that value in phase 2 of the proposal number

Paxos properties

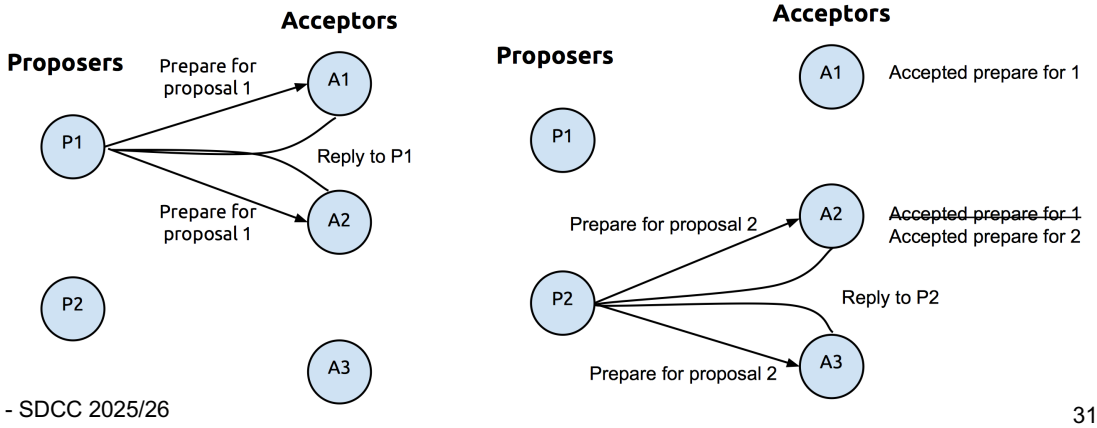
- Any proposal number is unique
- Any two sets of acceptors have at least one acceptor in common
- The value sent out in the accept phase is the value of the highest-numbered proposal of all the responses received in the prepare phase

Paxos: example (without failures)

- Proposers are p_1 and p_2
- Acceptors are a_1, a_2, and a_3

1° round, prepare phase

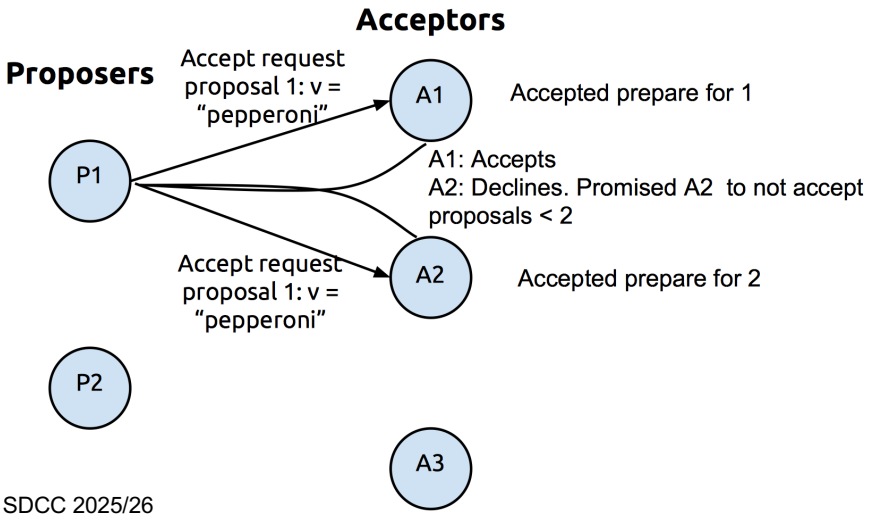
- p_1 sends prepare request for proposal 1 to a_1 and a_2
- a_1 and a_2 reply to p_1
- p_2 sends prepare request for proposal 2 to a_2 and a_3
- a_2 and a_3 reply to p_2



Paxos: example (without failures)

1° round, accept phase

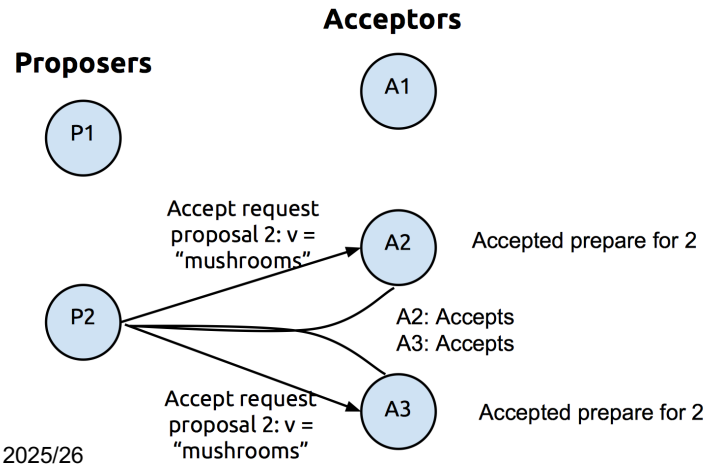
- p_1 sends accept request to a_1 and a_2 for proposal 1 with value "pepperoni"
  - p_1 got to select which value to propose
- a_1 accepts proposal 1
- a_2 does not accept proposal 1 (the older proposal is blocked)
  - a_2 promised p_2 it would not accept proposals < 2



Paxos: example (without failures)

1° round, accept phase (continued)

- p_2 sends accept request to a_2 and a_3 for proposal 2 with value "mushrooms"
  - p_2 also got to select which value to propose
- a_2 accepts proposal 2
- a_3 accepts proposal 2
- $\{a_2, a_3\}$ is a majority of acceptors, so proposal 2 is chosen
  - The chosen value is "mushrooms"



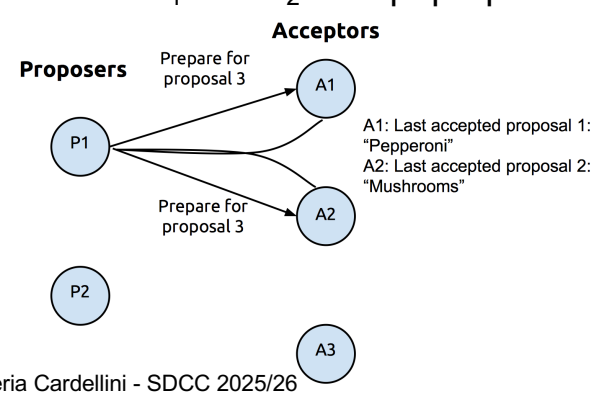
Paxos: example (without failures)

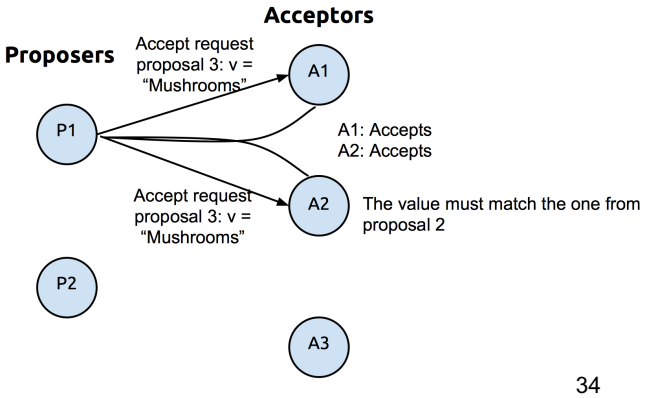
2° round, prepare phase

- p_1 sends prepare request for proposal 3 to a_1 and a_2
- a_1 replies; it last accepted proposal 1 for “pepperoni”
- a_2 replies; it last accepted proposal 2 for “mushrooms”

2° round, accept phase

- p_1 sends accept request to a_1 and a_2 for proposal 3 with value “mushrooms”
  - Value must match the one from proposal 2
- a_1 and a_2 accept proposal 3





Paxos: what about learners?

- There are some options to learn a chosen value:

a) Each acceptor, whenever it accepts a proposal, informs all the learners

✗ Lots of messages to be sent

b) Acceptors inform a distinguished learner (usually the proposer) and let the distinguished learner broadcast the result

✗ Single point of failure

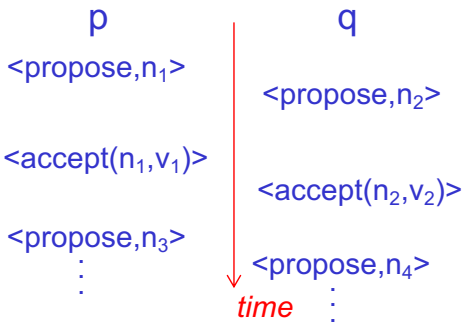
c) Compromise with a set of distinguished learners?

✓ Limits number of messages needed

✓ All distinguished learners need to fail to cause a problem

Paxos: distinguished proposer (or leader)

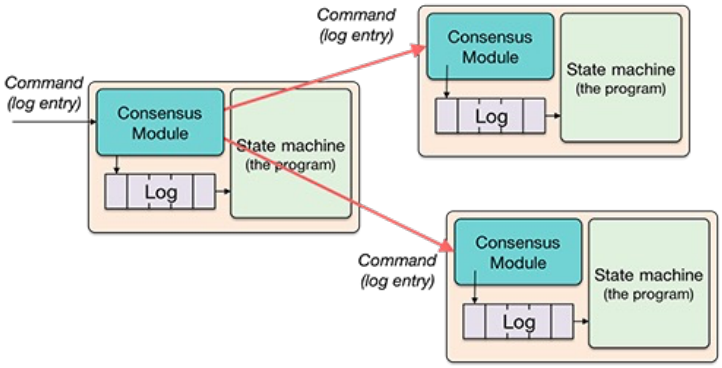
- Multiple dueling proposers that propose conflicting values may stall the protocol (because of FLP result)



p completes phase 1 for proposal number n_1. Another proposer q then completes phase 1 for proposal number $n_2 > n_1$. p's phase 2 accept requests for proposal numbered n_1 are ignored because at least one acceptor has promised not to accept any new proposal numbered less than n_2. So, p then begins and completes phase 1 for new proposal number $n_3 > n_2$, causing the second phase 2 accept requests of q to be ignored. And so on.

- Paxos guarantees progress (i.e., liveness) **if** only one of the proposers is eventually chosen as *leader*
- Therefore, in many Paxos implementations there is only one active proposer (i.e., leader)
  - Other proposers send proposals only when the current leader fails, and a new one needs to be elected

State machine replication and consensus protocols



- State machine replication (SMR): general approach to build fault-tolerant systems based on replicated servers
  - Each replica has a state machine (SM) and we want to make it fault-tolerant
  - A log is a list of commands that are received and stored; this list is read sequentially and used as input by SM
  - Using a consensus protocol, each SM processes the same list of commands in the log and thus produces the same series of results and arrives at the same series of states

SMR and Paxos

- Paxos is applied to achieve SMR
  - SM commands and their sequencing (the order in which they appear) are the values to agree
  - But requires one instance of Paxos per command: many instances of Paxos are executed simultaneously!
- Multi-Paxos is a more efficient solution to reduce the number of messages
  - Why multi? Multiple rounds from a stable leader
  - Prepare phase only in first round, then only accept phase in next rounds
    - After first round, leader enters into to a galloping mode where it sends successive accept messages when it receives a majority of acks for previous accept request
    - Galloping mode may be interrupted by leader crashing, in that case new leader must be elected

Paxos: other common use patterns

- Besides SMR, there are other common use patterns of Paxos, including:
- Log replication
  - To duplicate data across different nodes (different from SMR whose goal is to make copies of server state)
- Synchronization service
  - To control concurrent access to shared data
- Configuration management
  - Leader election, group membership, service discovery, and metadata management

Paxos in practice

- Some DSs that use Paxos
  - The first ones: Petal (distributed virtual disks) and Frangipani (scalable distributed file system)
  - Chubby: Google's distributed lock service used in BigTable, Google Analytics and other Google products
    - Zookeeper uses a Paxos-variant protocol called Zab
  - Spanner: Google's globally distributed NewSQL database
  - XtreemFS: fault-tolerant distributed file system for WANs
  - Mesos: uses Paxos to manage its replicated log
  - LibPaxos: implementations for your app
- However, getting Paxos right in practice is hard
  - E.g., how to implement a globally unique proposal number
  - See Paxos made live paper by Google researchers

> While Paxos can be described with a page of pseudo-code, our complete implementation contains several thousand lines of C++ code. The blow-up is not due simply to the fact that we used C++ instead of pseudo notation, nor because our code style may have been verbose. Converting the algorithm into a practical, production-ready system involved implementing many features and optimizations – some published in the literature and some not.

Raft

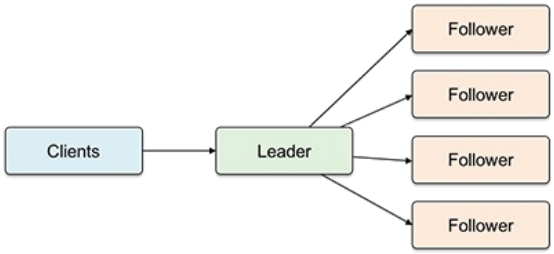
- Consensus algorithm developed in 2014 at Stanford University https://raft.github.io
  - RAFT: Replicated And Fault Tolerant
- Goals:
  - Designed to be easier to understand, implement and validate than Paxos
  - Complete foundation for implementation
    - Paxos not complete enough for real implementations, the algorithm is specified in a way that is detached from real-world implementation issues and use cases
    - Paxos implementations need extensive proofs and verification of their own, detaching them further from the original theoretical results
  - Equivalent and as efficient as Paxos for log replication

Raft: features

- System model similar to Paxos' one
  - Delayed/lost messages, fail stop (not Byzantine)
- Equivalent to (multi-)Paxos in fault-tolerance and performance
- Differences from Paxos
  - Problem decomposed into relatively independent sub-problems (leader election and log replication)
  - Addresses all major pieces needed for practical systems
- Rapid and widespread adoption
  - Many implementations currently listed on Raft home page, >10 versions in production
  - Some systems using Raft: etcd, CockroachDB, Consul, Hazelcast

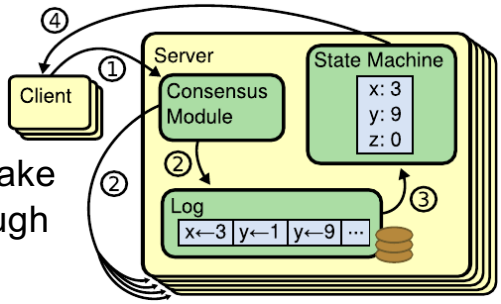
Raft: overview

- Raft is implemented on a cluster of servers, each of which hosts:
  - State machine (provided service)
  - Log that contains inputs fed into state machine
  - Raft protocol
- One of the servers is *elected* to be the *leader*, the others function as *followers*
- Clients send commands only to the leader, who forwards them to followers
- Each of the servers stores received commands in a log



Raft: overview

- Raft is a *state machine replication* protocol
  - Each server has a state machine and a log: state machine is what we want to make fault-tolerant (e.g., key-value pairs) through replication



  - The state machine is a deterministic program that specifies the desired behavior of the cluster as a whole
  - The state machine processes a sequence of *commands*, given by external clients; they interact with the system as if it were a single node running a single copy of the state machine
  - Each server simulates a copy of the state machine
  - Protocol goal: to maintain consistency across the copies of the state machine by ensuring proper log replication

Raft: overview

- Two major components of Raft

1. Leader election

  - Select one of the servers to act as leader
  - Leader is responsible for log replication to followers
  - In case of leader crash, choose new leader

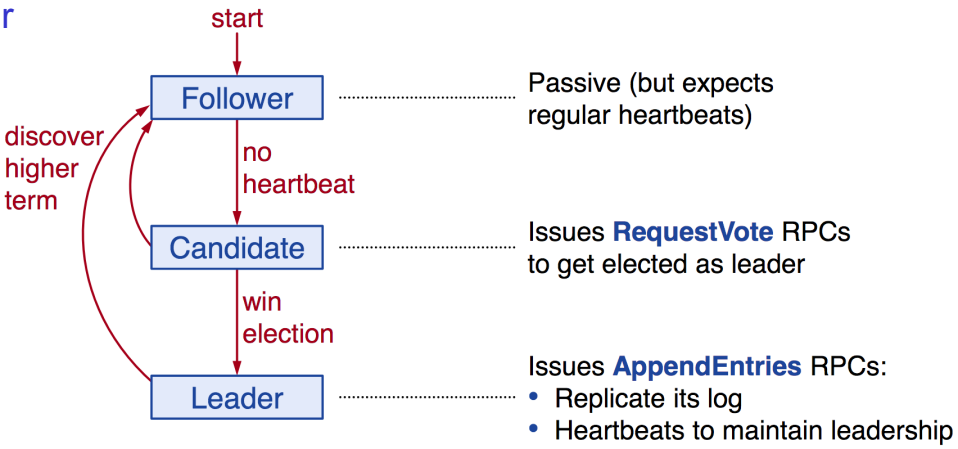
2. Log replication (normal operation)

  - Goal: make sure that replicated state machine is up to date across a majority of servers in the cluster
  - Leader accepts commands from clients, appends them to its log (note that log is append-only)
  - Leader replicates its log to other servers (overwriting inconsistencies)

- Let's examine Raft using http://thesecretlivesofdata.com/raft/

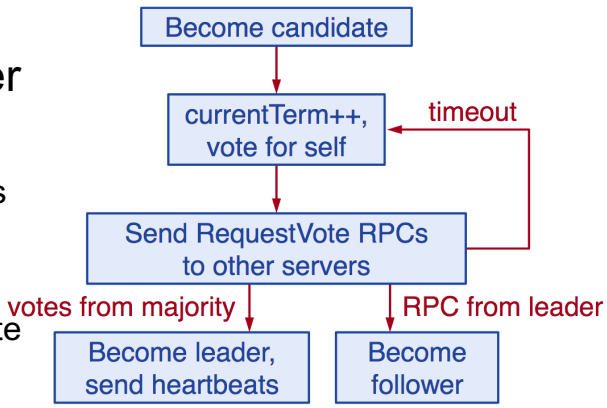
Raft: server state

- A server can be in 1 of 3 states:
  - Leader (at most one leader per term)
  - Candidate
    - If followers don't hear from a leader (**heartbeat**) by their own **election timeout** then they can become a candidate
    - Request votes from other nodes
    - Can become the leader if gets majority vote (**leader election**)
  - Follower



Raft: leader election

- At most one leader per term
  - Some term has no leader because of failed election (*split vote*)
- Each server maintains *currentTerm* value (no global view)
  - Each server has its own local view of time that is represented by its *currentTerm*; it increases monotonically over time
- **Election timeout**
  - Follower waits until become candidate
- **Election term** starts if follower doesn't see a leader
  - Candidate votes for self, and sends out *RequestVote* RPCs to all the other servers
  - Receiving servers vote on candidate iff they haven't voted yet this term
  - Election timeout reset



Raft: log replication

- Client sends command to leader
- Leader appends command to its log
- Leader sends *AppendEntries* RPCs to all followers in order to replicate all changes to all servers
  - *AppendEntries* message sent by leader to followers at periodic intervals specified by **heartbeat timeout**
  - Followers acknowledge *AppendEntries* message
  - Once the leader receives acks from a majority of servers, it executes the command on its state machine and returns result to client, and the log entry is considered *committed*

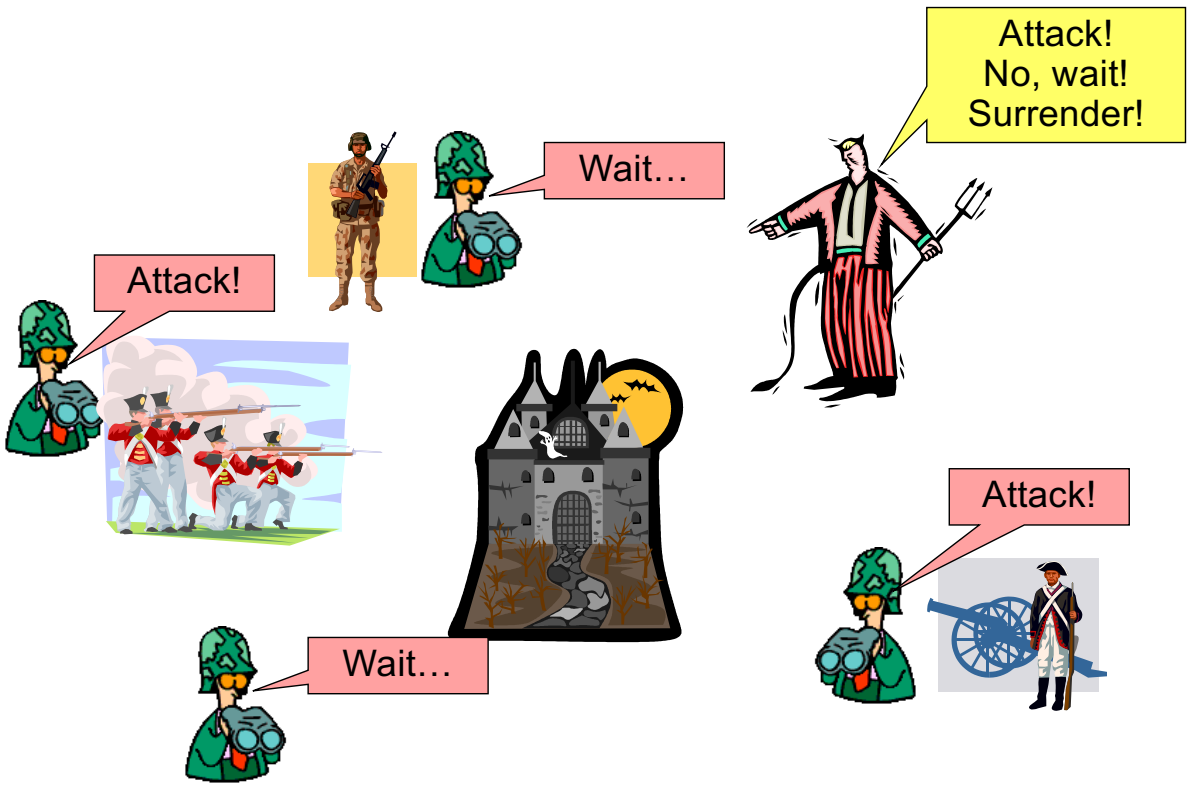
Raft: log replication

- Once new log entry is committed:
  - Leader notifies followers of committed entries in subsequent *AppendEntries* RPCs
  - Followers execute committed commands in their state machines
- Election term will continue until a follower stops receiving heartbeats and becomes a candidate

Raft: properties

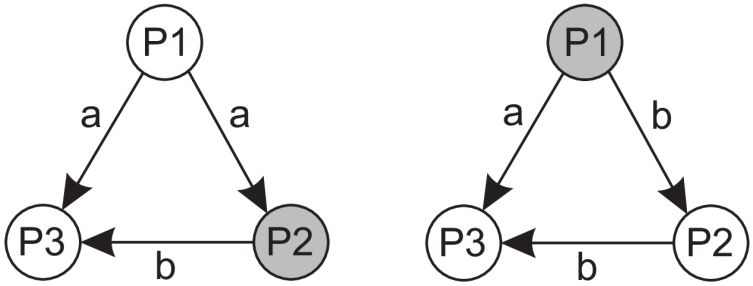
- Raft guarantees safety
  - At most one leader per term
  - Logs are kept consistent
  - Only a node with an up-to-date log can become a leader
- Raft also provides a liveness guarantee
  - If there are “sufficiently few failures”, then the system will eventually process and respond to all client commands
- Raft is tolerant to network partitions
  - Log uncommitted so long as no majority
  - Majority as seen in face of partition, e.g. 2+3 partition
  - Recovery
    - Old leader steps down when sees higher term
    - Rolls backs uncommitted entries and matches new leader’s log

Byzantine scenario



Accordo in presenza di fallimenti bizantini

- In presenza di fallimenti bizantini, per sopravvivere ad attacchi di *k* processi guasti e raggiungere l'accordo distribuito occorre avere N >= 3*k*+1 processi
  - E' il problema dei generali bizantini (definito da Lamport)
  - *Idea*: si vuole raggiungere l'accordo se attaccare una città oppure ritirarsi tra un gruppo di generali fedeli, essendoci *k* generali traditori → occorrono 2*k*+1 generali fedeli
    - Se non ci sono più dei 2/3 di generali fedeli, non è possibile raggiungere l'accordo
    - P3 non riesce a capire chi sia il traditore tra P1 e P2

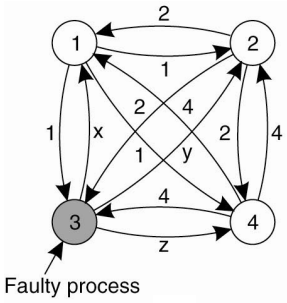


Problema dell'accordo bizantino

- *Assunzioni*:
  - Processi sincroni
  - Comunicazione unicast
  - Ordinamento dei messaggi
  - Ritardi limitati
- Ci sono *N* generali (ovvero processi) ed ogni processo *i* fornisce un valore v_i agli altri processi
  - v_i rappresenta la forza della truppa del generale *i*
- *Obiettivo*: far costruire ad ogni processo un vettore *V* di dimensione *N* tale che se il processo *i* è non guasto allora *V[i]* = v_i, altrimenti *V[i]* è non definito

Problema dell'accordo bizantino

- Come raggiungere l'accordo nel caso di 3 processi che funzionano correttamente ed 1 fraudolento (N=4, k=1)?
  - Per semplicità assumiamo $v_i = i$



Passo 1: ogni processo P_i invia il suo valore v_i agli altri

Passo 3: ogni processo invia il proprio vettore agli altri

Passo 2: i risultati del passo 1 sono riuniti in vettori V

1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

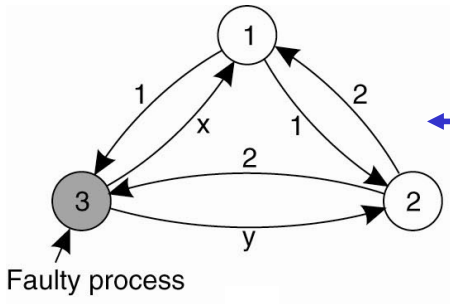
1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

Passo 4: voto di maggioranza sui valori ricevuti: se P_i funziona correttamente, $V[i] = v_i$

1 2 ? 4

Problema dell'accordo bizantino

- Perché con 2 processi che funzionano correttamente ed 1 processo fraudolento (N=3, k=1) non si riesce a raggiungere l'accordo?



Passo 1: ogni processo invia il suo valore agli altri

Passo 3: ogni processo invia il proprio vettore agli altri

Passo 2: i risultati del passo 1 sono riuniti in vettori

1 Got(1, 2, x)
2 Got(1, 2, y)
3 Got(1, 2, 3)

1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

Passo 4: voto di maggioranza sui valori ricevuti

? ? ?

Lamport's algorithm for Byzantine agreement (oral message)

Algorithm OM(0)

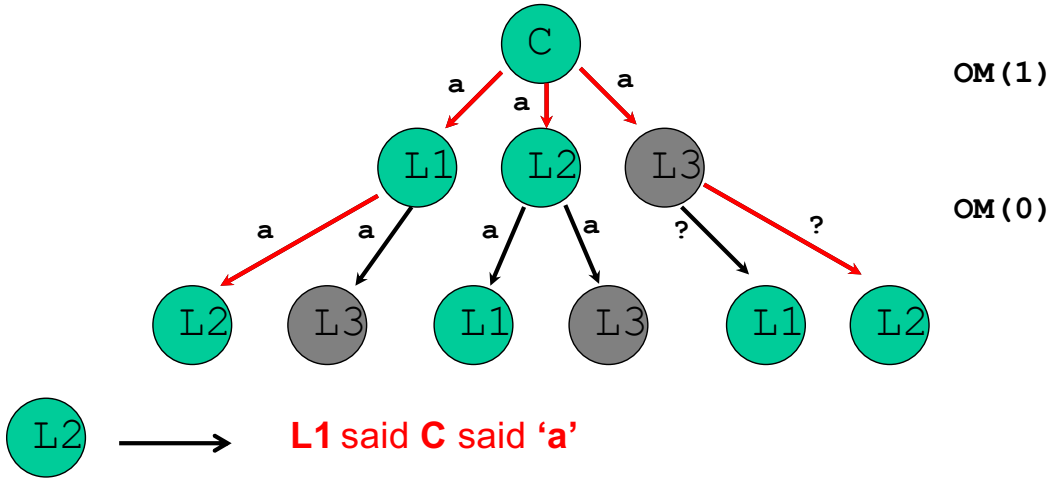
1. The commander sends his value to every lieutenant
2. Each lieutenant uses the value he receives from the commander, or uses the value RETREAT if he receives no value

Algorithm OM(k), k>0

1. The commander sends his value to every lieutenant
2. For each i, let v_i be the value Lieutenant i receives from the commander, or else be RETREAT if he receives no value. Lieutenant i acts as the commander in Algorithm OM(k-1) to send the value v_i to each of the N-2 other lieutenants
3. For each i and each $j \neq i$, let v_j be the value Lieutenant i received from Lieutenant j in step 2 (using Algorithm OM(k-1)), or else RETREAT if he received no such value. Lieutenant i uses the value majority (v_1, …, v_{N-1})

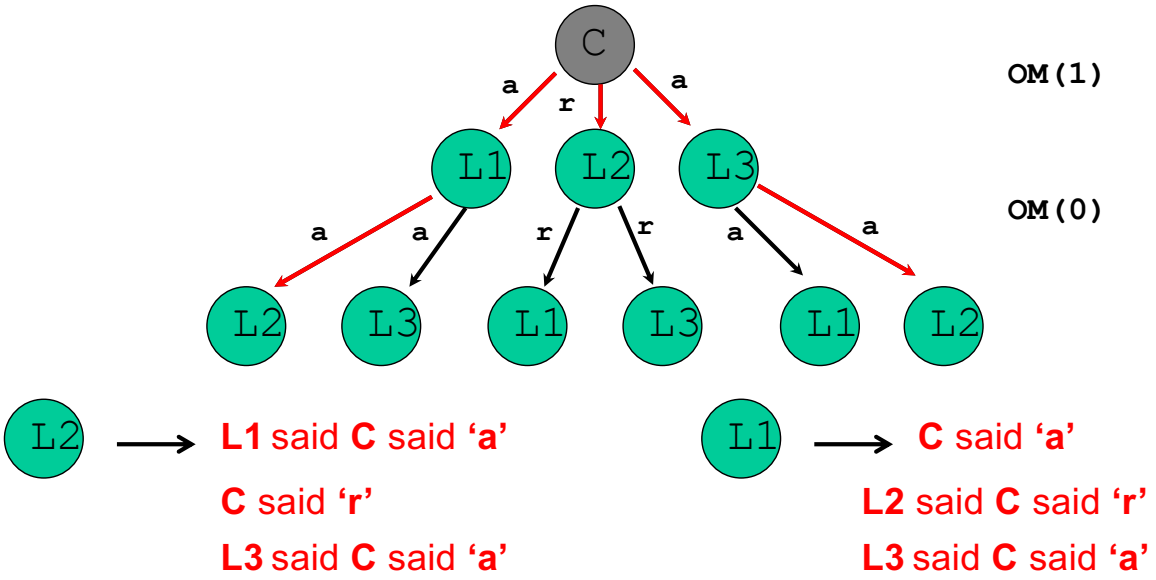
Lamport et al, The Byzantine Generals Problem, ACM Transactions on Programming Languages and Systems, 1982

Demo: OM(1), L3 as traitor



Result: majority(a, a, ?) = a

Demo: OM(1), C as traitor



L2 result: majority(a, r, a) = a; L1 result: majority (a, r, a) = a

Byzantine fault tolerance (BFT) in practice

- What about the performance of Byzantine generals algorithm?
  - k+1 synchronous rounds: quite slow
  - $O(N^k)$ messages: high traffic
- BFT protocols were long considered too expensive to be practical
- In 1999 Practical Byzantine Fault Tolerance (PBFT) algorithm was proposed
  - Thousands of requests per second with only sub-millisecond increases in latency
  - PBFT triggered a renaissance in BFT research

Byzantine fault tolerance and blockchains

- Blockchain systems require BFT to ensure consensus in the presence of faulty or malicious node
- Bitcoin uses Nakamoto Consensus protocol
  - Leader is elected through Proof of Work (PoW): nodes compete to solve a puzzle
  - First node to solve it becomes the leader, generating a new block and appending it to the blockchain
- Proof of Stake (PoS):
  - Used by Ethereum 2.0 and others
  - Nodes (validators) are selected based on the amount of cryptocurrency they hold and stake
  - More energy-efficient than PoW

References

- Sections 8.1, 8.2, 8.3 and 5.4.4 of van Steen & Tanenbaum book

- Lamport, Paxos made simple, *ACM SIGACT News*, 2001.
- The Paper Trail, Consensus Protocols: Paxos, 2009. (html)

- References on Raft
  - Raft paper: In Search of an Understandable Consensus Algorithm (extended version), 2014
  - Talk on Raft by John Ousterhout, 2016 https://youtu.be/vYp4LYbnnW8
  - Another Raft visualization: https://raft.github.io/raftscope/

- Lamport et al, The Byzantine Generals Problem, ACM Transactions on Programming Languages and Systems, 1982