Distributed Systems Theory

Dependable Systems 2014

Lena Herscheid, Dr. Peter Tröger

Microsoft Cloud Outage Blamed On Faulty Update

by **Ken Presti** on March 14, 2013, 2:45 pm EDT

11 June 2014 Last updated at 12:09 GMT

Share

Feedly and Evernote struck by denial of service cyber-attacks

By Leo Kelion
Technology desk editor

Stack Overflow Goes Down, Programmers Around The World Panic (It's Back Up Now)

Posted Feb 16, 2014 by Colleen Taylor (@loyalelectron)

HBO Go Still Recovering From Outage During Game Of Thrones Premiere

Posted Apr 6, 2014 by Catherine Shu (@catherineshu)

WhatsApp Experiences Second Major Outage Following Facebook Acquisition

Posted Apr 2, 2014 by Sarah Perez (@sarahintampa)

Distributed Systems – Motivation

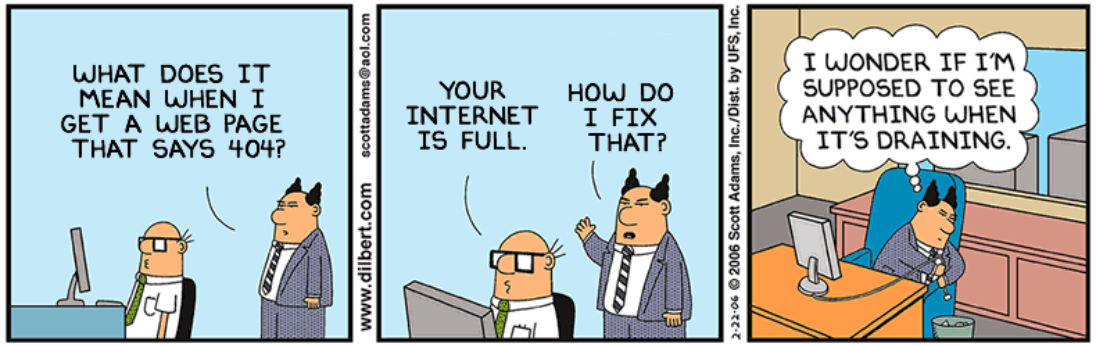
"Distributed programming is the art of solving the same problem that you can solve on a single computer using multiple computers."

- For Scalability
- For Performance / Throughput
- For **Availability**
  - Build a highly-available or reliable system using unreliable components
- Divide and conquer approach
  - **Partition** data over multiple nodes
  - **Replicate** data for fault tolerance or shorter response time

Transparency of Distribution

"A distributed system is a collection of independent computers that ***appears to its users as a single coherent system****"*

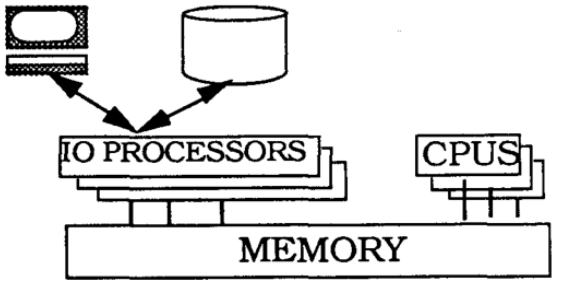
- Access transparency
- Location transparency
- Replication transparency
- Concurrency transparency
- <u>Failure transparency:</u>
  - Failures and recoveries of components are hidden from applications/users
- …



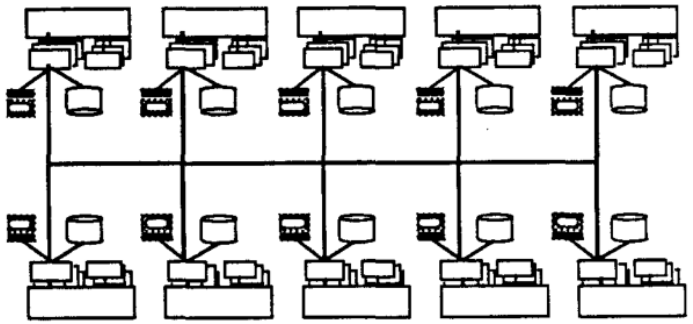
Fallacies of Distributed Computing
(L. Peter Deutsch, 1994)

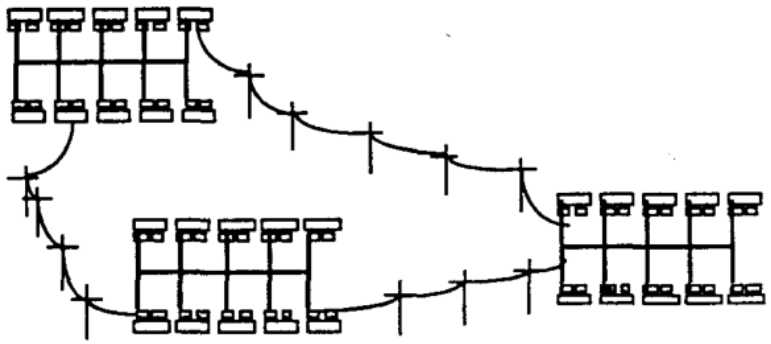
1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

Choose the right **system model**
(Weaker assumptions → more robustness and generality)

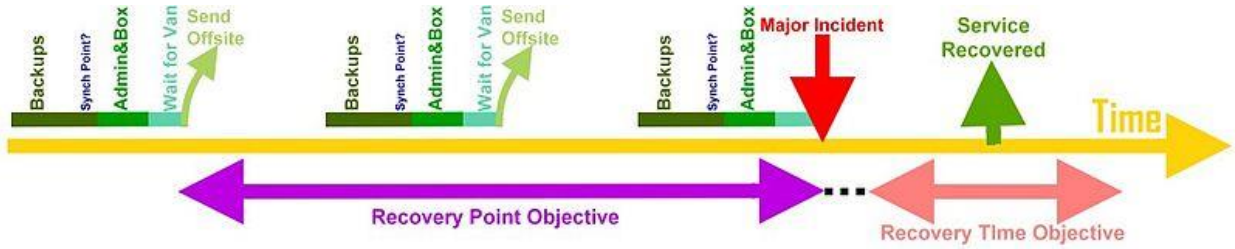


Jim Gray: "Transparency In Its Place: The Case Against Transparent Access to Geographically Distributed Data"





Buzzwords!

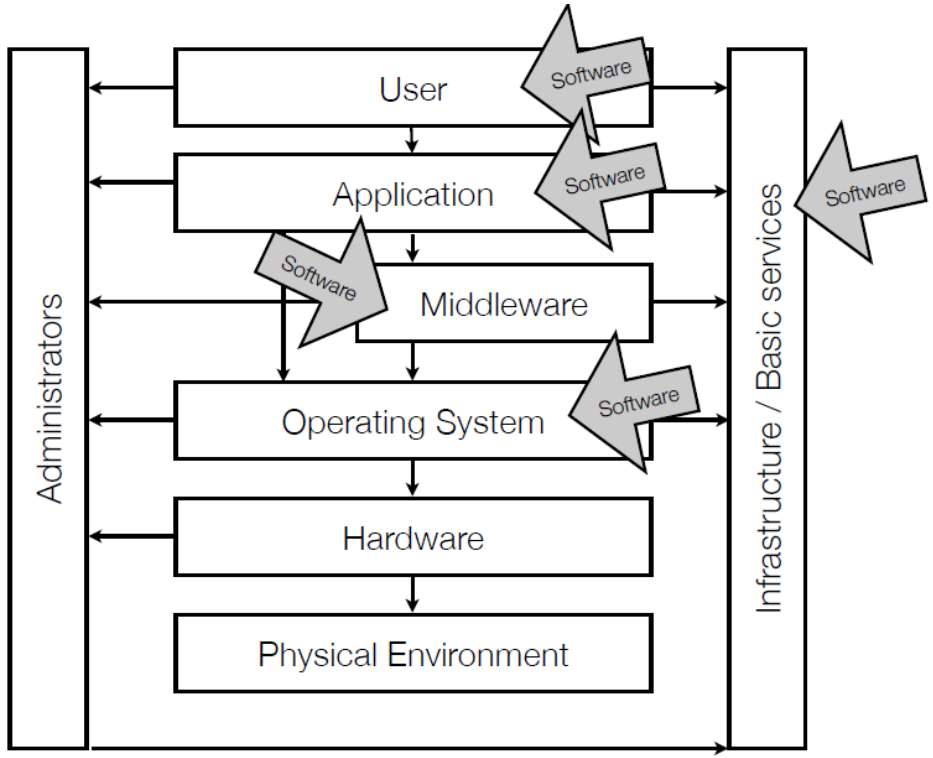


http://en.wikipedia.org/wiki/File:Schematic_ITSC_and_RTO,_RPO,_MI.jpg

- **High Availability**
  - 99.999% → 5.39 min downtime / year
- **Disaster Recovery**
  - Protect IT systems from large-scale disasters (natural or human made)
  - Part of business continuity planning
    - Recovery Point Objective (RPO): Maximum period in which data may be lost due to disaster
    - Recovery Time Objective (RTO): Maximum time until service restoration after disaster
- **Geographic Distribution**
  - Multiple data centres on different continents → increased *latency* and *consistency* challenges
- **Big Data**
  - Datasets getting so big that traditional implementations fail to handle them

Distributed Fault Tolerance

- High level of **concurrency**
  - Complex interaction patterns → increased likelihood for errors
- Many **levels** of potential failures
- **Heterogeneity** of components
- Increased **operation** effort
  - Operation errors have high impact
- **Latency** issues
- **Consistency** concerns



Divide and conquer

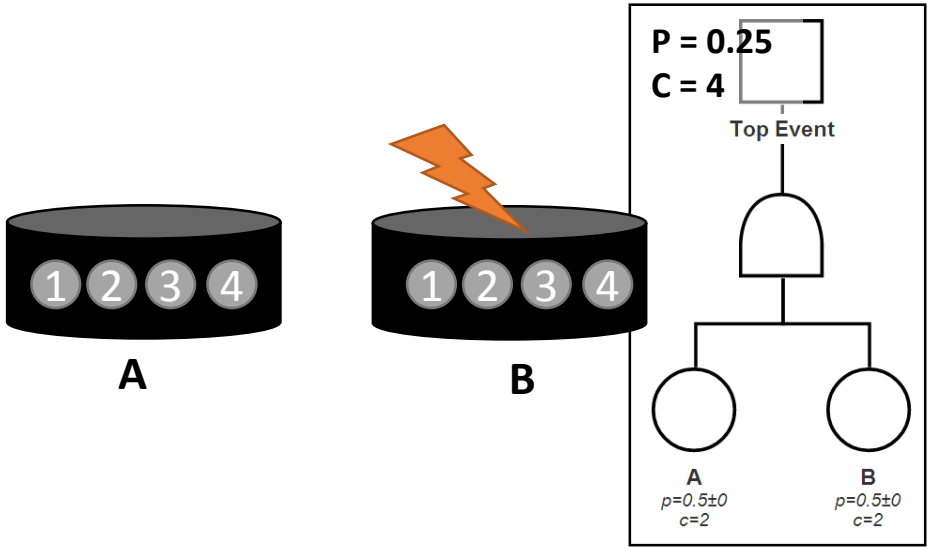
Distributed computing has two major challenges: *latency* and *failures*

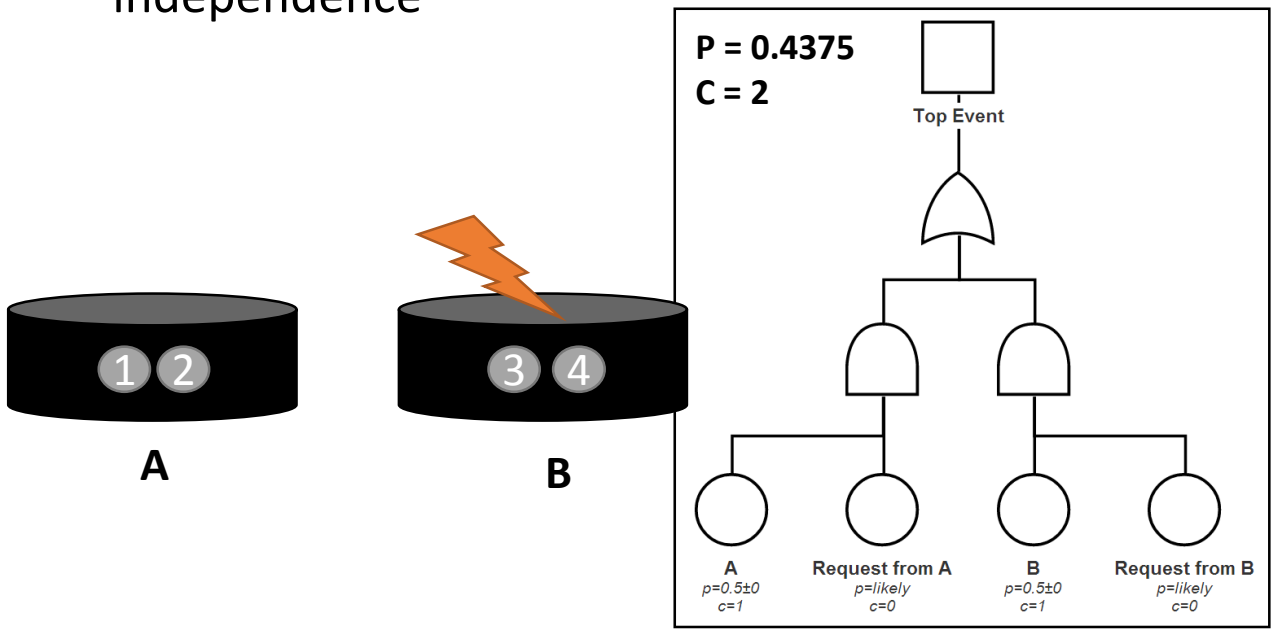
Replication

- Increase performance and availability through *redundancy*
- Increased infrastructure cost

Partition

- Decrease latency by co-locating related data
- Mitigate the impact of faults by increased independence





Distributed Systems – Abstractions

"You can't tolerate faults you haven't considered."

- **Timing** assumptions
  - Synchronous
    - Known upper bound on message delay
    - Each processor has an accurate clock
  - Asynchronous
    - Processes execute at independent rates
    - Message delay can be unbounded
  - Partially synchronous
- **Failure model**: Crash, Byzantine, …
- **Failure detectors**: strong/weakly accurate, strong/weakly complete
- **Consistency model**
  - Strong: the visibility of updates is equivalent to a non-replicated system
  - Weak: client-centric, causal, eventual…

Quiz

1. Why can it be problematic to measure only availability (e.g. in SLAs)?
2. Why does increasing the stochastic independence of faults increase availability?
3. Why is it harder to implement fault tolerance in geographically distributed systems?

Timing Model

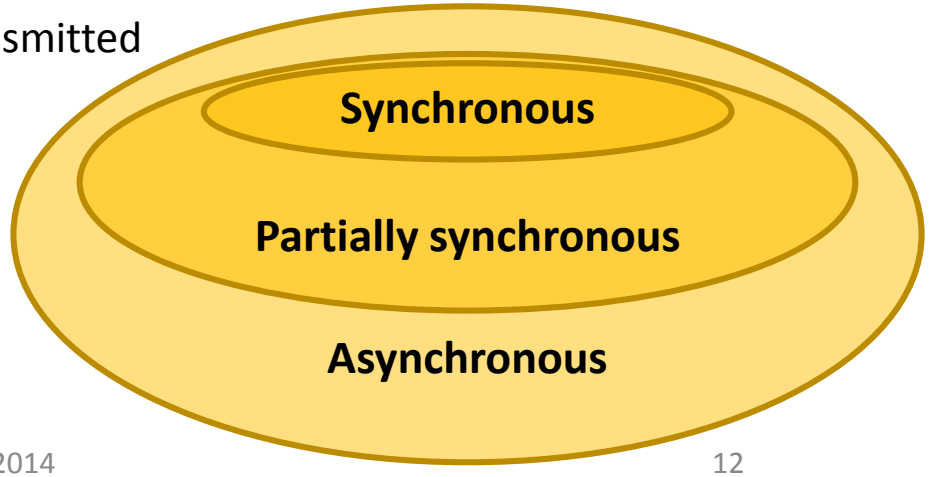
Lamport, Leslie. "Time, clocks, and the ordering of events in a distributed system." *Communications of the ACM* 21.7 (1978): 558-565.

Network Time Protocol, RFC 5905, http://tools.ietf.org/html/rfc5905

Corbett, James C., et al. "Spanner: Google's globally distributed database."*ACM Transactions on Computer Systems (TOCS)* 31.3 (2013): 8.

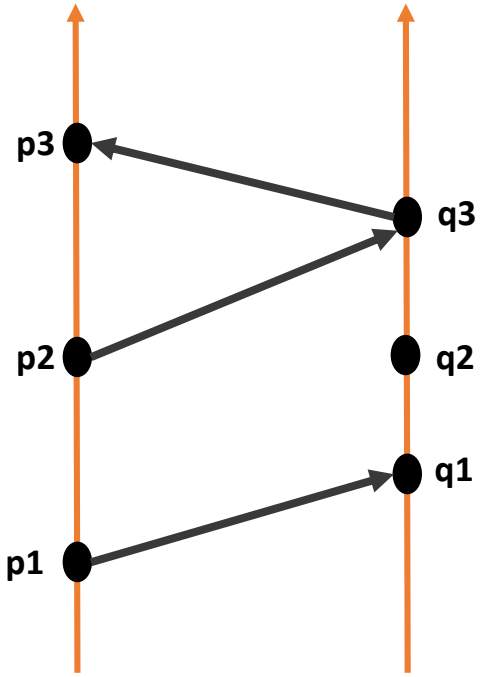
Timing Model

- **Synchronous**: known upper bound on message transmission
  - Time complexity: number of messaging rounds until a result is computed
- **Partially synchronous**: in practice, some assumptions can still be made
  - Clocks of limited accuracy, which may be out of sync
- **Asynchronous**: message delays are unpredictable
  - Algorithms for asynchronous systems work in any system
- Approaches to mitigate asynchrony
  1. Simulate a synchronous system in an asynchronous system
     - **Synchronizers**: send clock pulses within which messages are transmitted
     - High message overhead
  2. Simulate global time
     - **Virtual time**: Lamport clocks
  3. Simulate global state
     - **Snapshot algorithm**



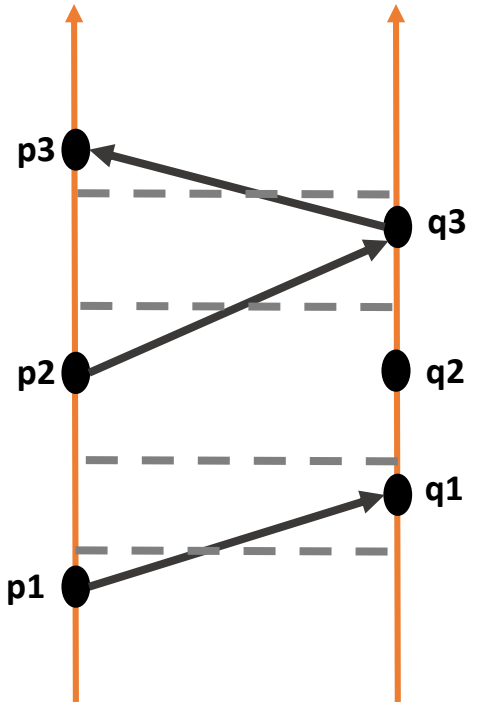
Logical Time

- How to order events in an asynchronous system without physical clocks?
  - Assume sequential events within each process
- **Happened before** relation ($\rightarrow$)
  - Mathematically, a partial ordering
  - $a \rightarrow b$, if
    - A and B are in the same process, and A comes before B
    - A is a send and B is a receipt of the same message
  - $a \rightarrow b \ \wedge b \rightarrow c \ \Rightarrow a \rightarrow c$ (transitivity)
  - Concurrency: b $\nrightarrow a \ \wedge a \nrightarrow b$
  - Denotes that an event causally affects another

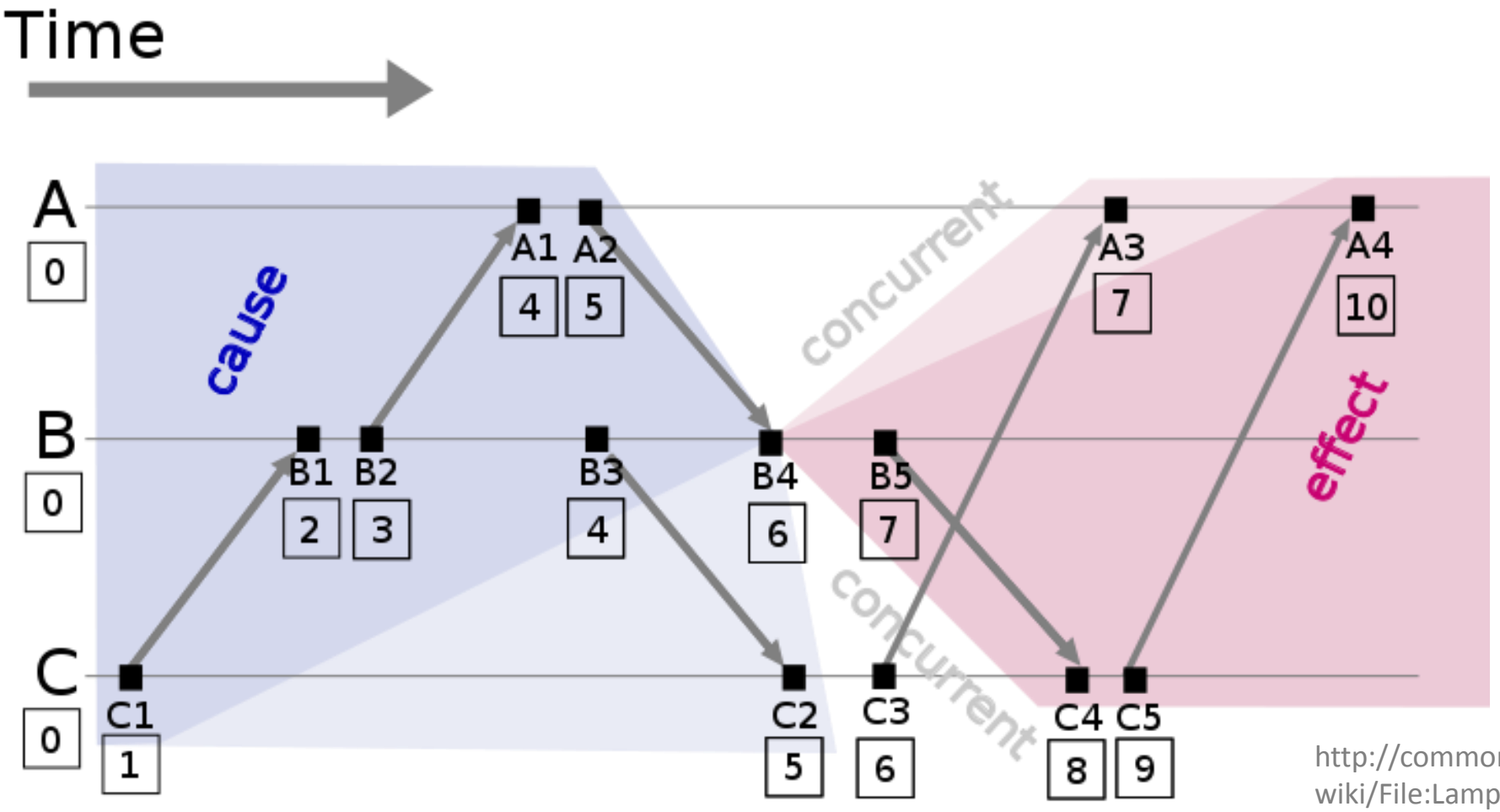


Lamport Clocks

- For each process, $\mathrm{C}\langle a\rangle$ assigns a number to any event
- **Clock condition**: $\mathrm{a} \rightarrow b \;\Rightarrow C\langle a\rangle \rightarrow C\langle b\rangle$
- Each process increments clock between events
- For message send from a to b:
  - Message contains logical timestamps $\mathrm{T} = \mathrm{C}\langle a\rangle$
  - Receiver sets C to a value >= its present value and > T
- Can be used to build a total order ($\Rightarrow$)
  - Use an arbitrary total ordering of the processes $\prec$
  - $\mathrm{a} \Rightarrow b$, if $C\langle a\rangle < C\langle b\rangle$ or $C\langle a\rangle = C\langle b\rangle \;\wedge\; P_a \prec P_b$



Lamport Clocks



http://commons.wikimedia.org/wiki/File:Lamport-Clock-en.svg

Timing in Distributed Systems

- **Clock drift**: distributed, slightly inaccurate clocks drift apart → need synchronization
- Christian's algorithm
  - Assumes central time server
  - Requires round trip time (RTT) << required accuracy
- Berkeley algorithm
  - Uses coordinator election
  - No central clock, global clock is the average of local slave clocks
  - Master sends clock adjustments (considering RTT) to the slaves
- **Network Time Protocol (NTP)**: synchronize clocks to UTC via the internet
  - Sub-millisecond accuracy
  - Implementations based on UDP port 123
  - Hierarchical approach
    - Each node has a stratum (0: physical clock, 1: primary computer connected to it, 2: secondary client, …)
  - Fault tolerant because of different network routes and redundant clocks and servers

Quiz

1. Why are logical clocks usually used together with physical clocks?
2. Why is it useful to have a partial order of events?
3. Why is it useful to have a total order of events?

Fault Model

Lamport, Leslie, Robert Shostak, and Marshall Pease. "The Byzantine generals problem." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982): 382-401.

Chandra, Tushar Deepak, and Sam Toueg. "Unreliable failure detectors for reliable distributed systems." *Journal of the ACM (JACM)* 43.2 (1996): 225-267.

Cristian, Flavin. "Understanding fault-tolerant distributed systems."*Communications of the ACM* 34.2 (1991): 56-78.

Fault Model

RECAP

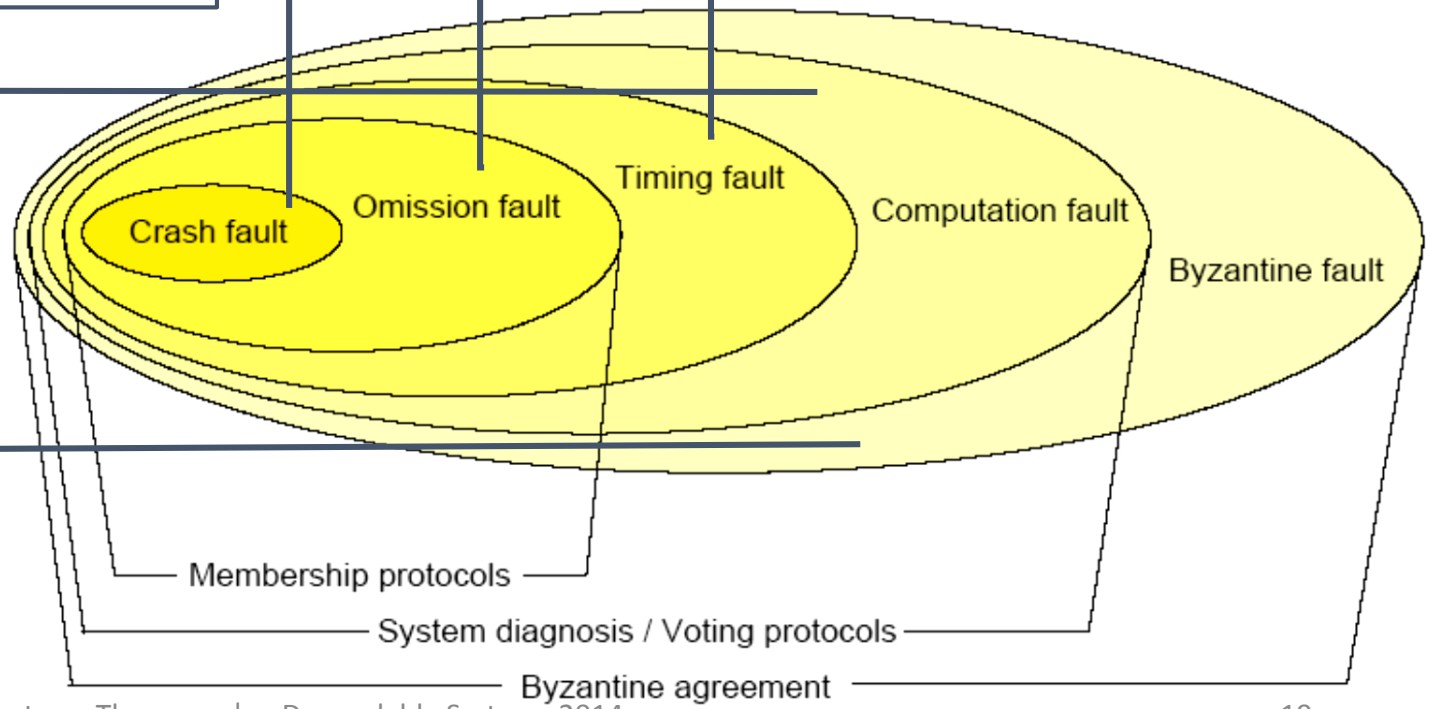
Processor stops without notification (until restart)

Processor breaks deadline

Processor does not react within its time window

Processor responds incorrectly

Any possible fault



Byzantine Generals Problem

- *"Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system"*
  - Formalized by Leslie Lamport (1982)
- Generals in the Byzantine Empire must agree on an attack time (consensus)
  - Traitors, which misbehave in arbitrary ways, may exist
  - Asynchronous, unreliable communication
- Solution criteria
  1. All generals decide upon the same plan of action
  2. A small number of traitors cannot cause the loyal generals to adopt a bad plan
- Equivalent formulation with commanding general
  1. All loyal lieutenants obey the same order
  2. If the commanding general is loyal, then every loyal lieutenant obeys the order he sends

Byzantine Generals Problem – Example

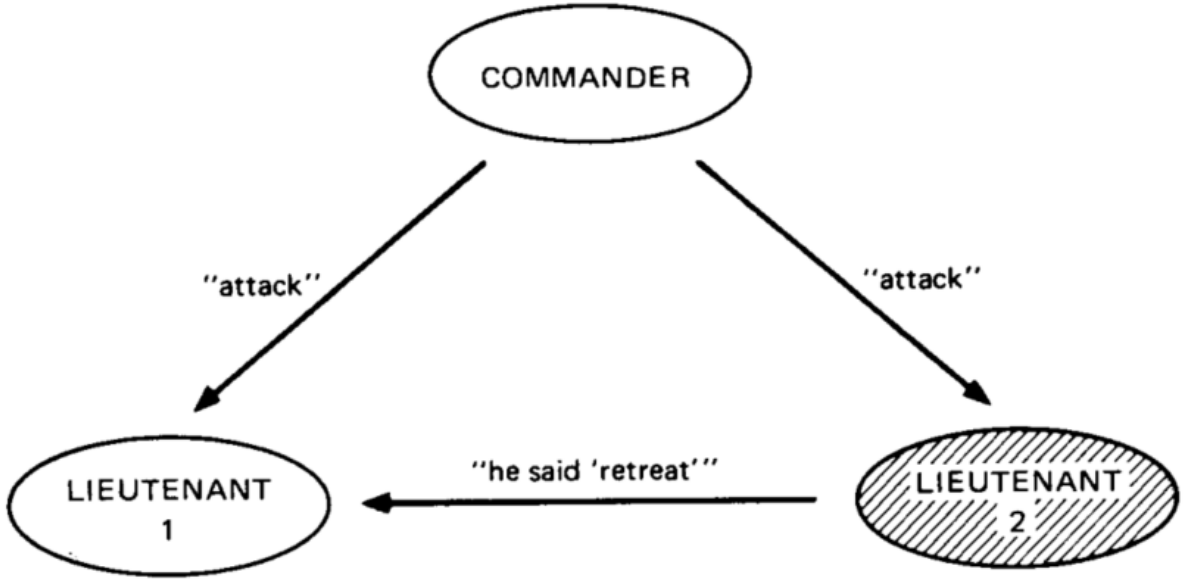


Fig. 1. Lieutenant 2 a traitor.

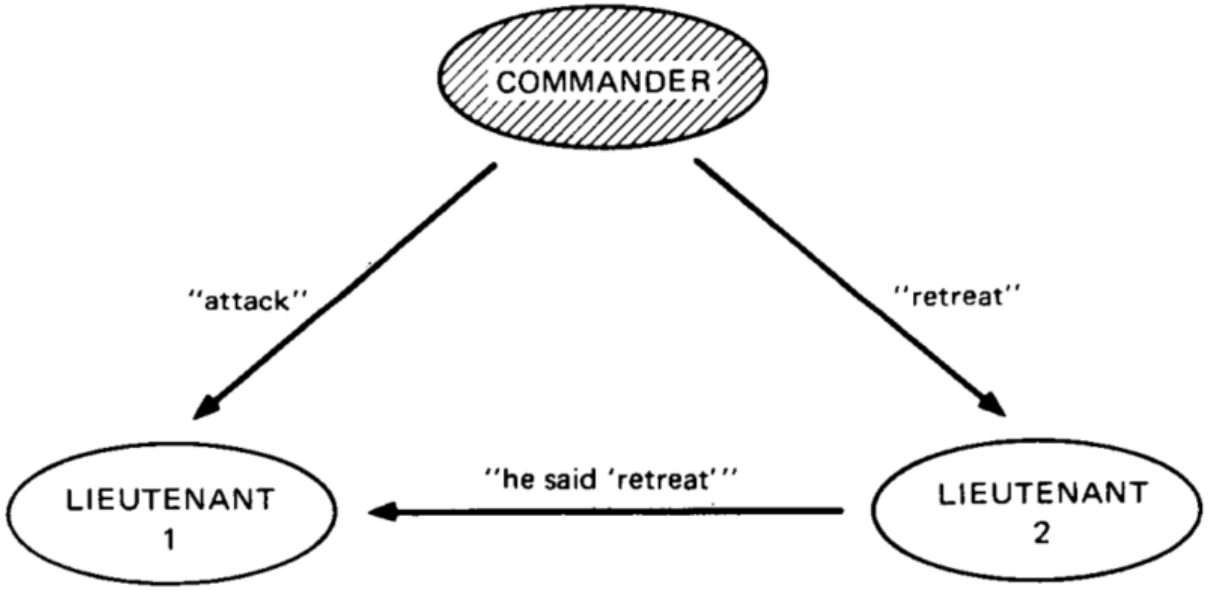
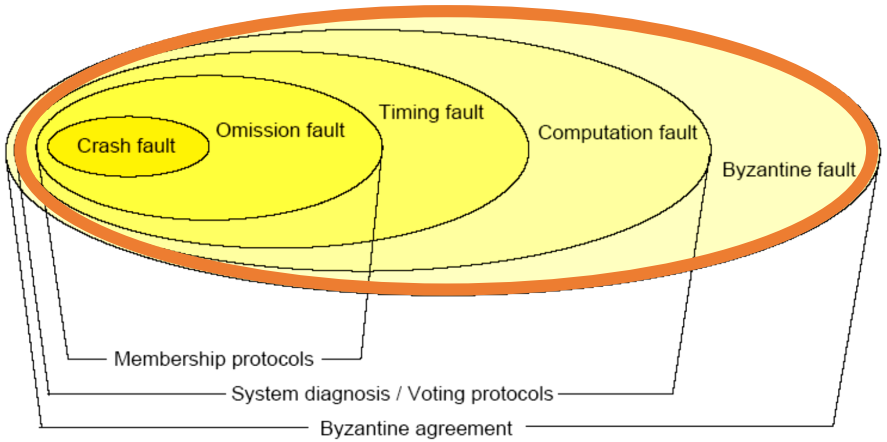


Fig. 2. The commander a traitor.

Leslie Lamport: "The Byzantine Generals Problem"

Byzantine Fault Tolerance



- Defend against arbitrary failures
  - Malicious attacks
  - Arbitrarily wrong, inconsistent return values
- Most realistic system model, but hardest to implement
  - Solutions require many message sends
  - Significant overhead for each decision
- To tolerate *F* byzantine failures, *3F+1* nodes are needed
  - Outlined proof by contradiction (Lamport)
    - Assume a solution for 3 generals (“Albanian generals”)
    - Derive generalized 3F solution from it
    - But: case enumeration proves 3 generals solution impossible

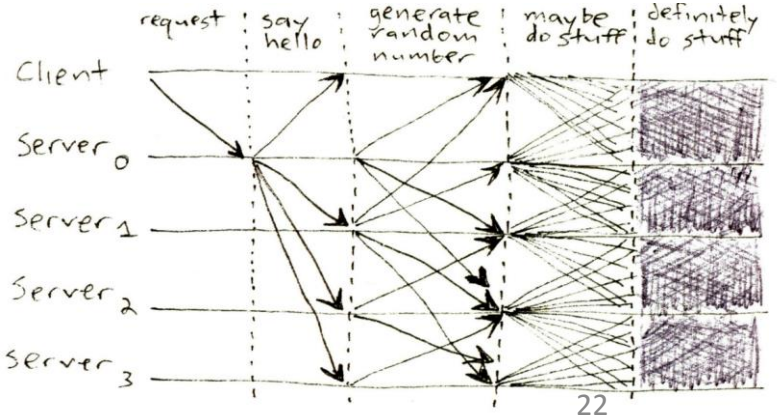
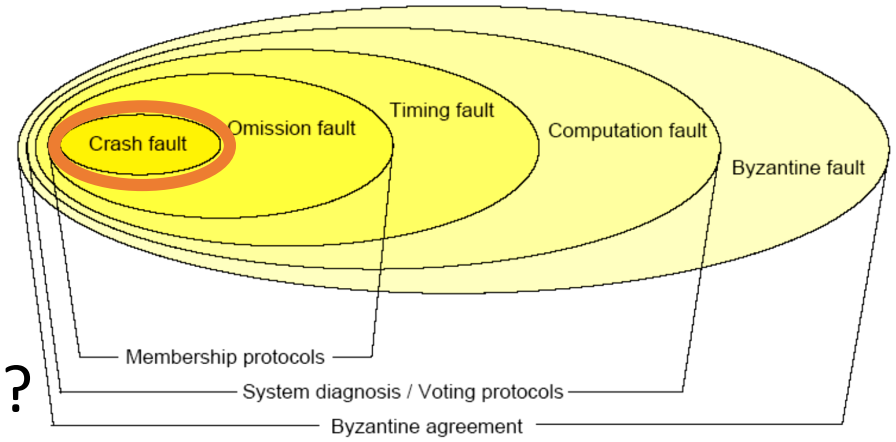


Figure 1: Typical Figure 2 from Byzantine fault paper: Our network protocol

Failure Detectors - Motivation



- How to find out when a node has failed (crashed)?
- Important for many fault tolerance paradigms (Consensus, Atomic Broadcast, …)
- Correct crash detection is possible only in synchronous systems
- In asynchronous systems, you never know if a process is just “very slow”
  - FLP impossibility
- Solutions: constrain the assumptions!
  - Constrained timing model (partial synchrony)
  - Weaker problems

→ To preserve asynchronous model: external, **unreliable failure detectors**

Failure Detectors - Model

- **Failure Detector:** *"Distributed oracle that provides **hints** about the operational status of processes" – Toueg*
  - May be incorrect
  - May change its mind
  - In synchronous systems: perfect detection using timeouts
- **Completeness**
  - Strong: Eventually, every crashed process is permanently suspected by *every* correct process
  - Weak: Eventually, every crashed process is permanently suspected by *some* correct process
- **Accuracy**
  - Strong: No process is suspected before it crashes
  - Weak: Some correct process is never suspected

Failure Detectors - Implementation

- ◊*W*: Eventually weak completeness & eventually weak accuracy
  - ◊*W* is the weakest failure detector for solving consensus in asynchronous systems with f < [n/2]
  - Consensus and atomic broadcast are equivalent problems
- In practice, ◊*W* can be implemented using timeouts
  - Increase timeout period after every mistake
  - Eventually, there are *probably* no premature timeouts
- **Reliable broadcast**
  - All correct processes deliver the same messages
  - All messages broadcasted by correct processes are delivered
  - No spurious messages are delivered

Quiz

1. How many nodes do you need to tolerate *crash faults*?
2. How many nodes do you need to tolerate *byzantine faults*?
3. Describe a failure detector satisfying strong completeness, but not weak accuracy.

Distributed Systems – Problems

- **Atomic broadcast**
  - All receive messages reliably in the same order
- **Consensus**
  - Multiple processors agree on a value
- **Leader election**
  - Find unique coordinator, known to every node
- **Atomic commit**
  - Apply a set of changes in a single operation
- **Terminating reliable broadcast**
  - All eventually receive the correct message

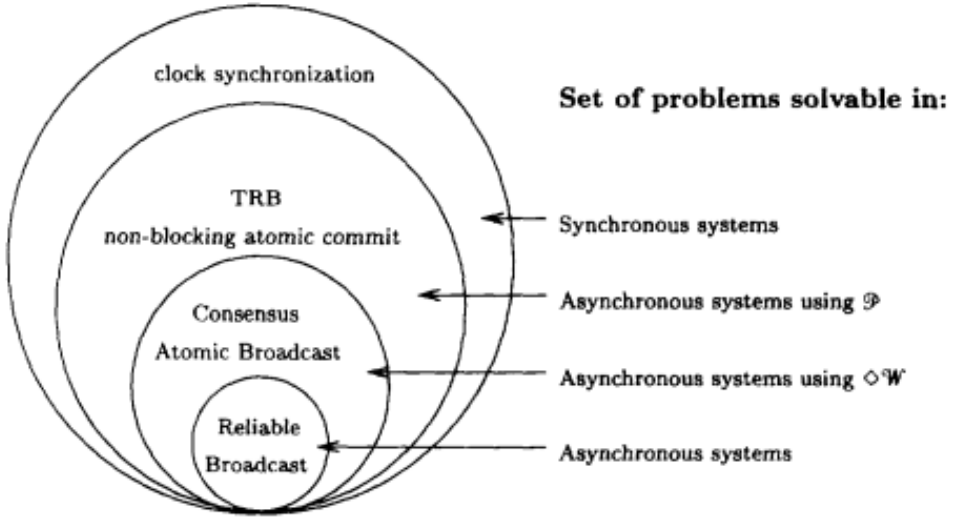
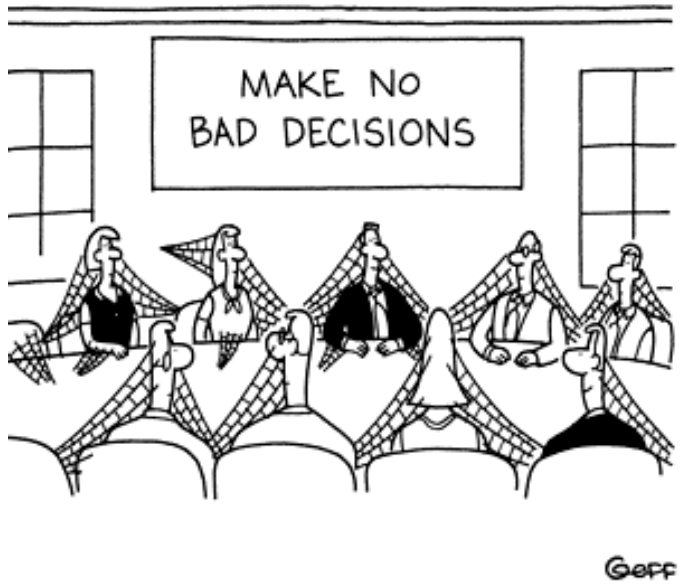


FIG. 9. Problem solvability in different distributed computing models.

Leader / Coordinator Election

- Goal: choose a coordinating process (leader) for a task
  - Coordinators are single points of failure
  - → Need to be replaced when they fail
- Complexity measures
  - Time complexity
  - Messaging complexity
- Algorithms tailored for different system assumptions
  - Topology: ring, complete graphs, undirected vs unidirectional
  - Timing: synchronous vs asynchronous systems
  - Anonymous systems vs per-process UID

Consensus



Fischer, Michael J., Nancy A. Lynch, and Michael S. Paterson. "Impossibility of distributed consensus with one faulty process." *Journal of the ACM (JACM)*32.2 (1985): 374-382.

Chandra, Tushar D., Robert Griesemer, and Joshua Redstone. "Paxos made live: an engineering perspective." *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM, 2007.

Gray, Jim. "A comparison of the Byzantine agreement problem and the transaction commit problem." *Fault-tolerant distributed computing*. Springer New York, 1990. 10-17.

Ongaro, Diego, and John Ousterhout. "In search of an understandable consensus algorithm." *Draft of October* 7 (2013).

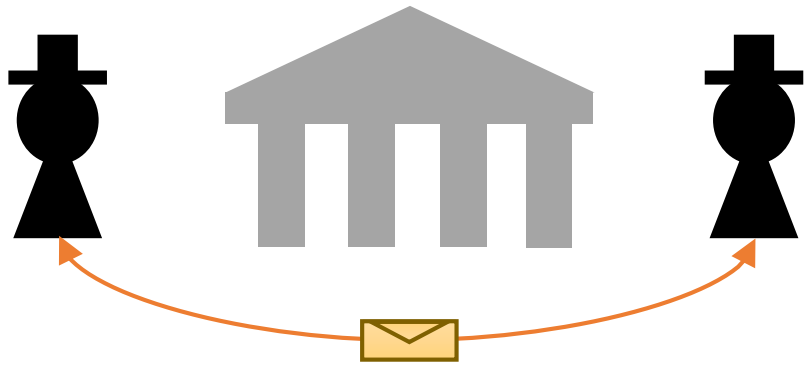
Two Generals / Coordinated Attack Problem

- Two spatially separated armies want to attack a city and need to **coordinate**
  - If they do not attack simultaneously, they die
  - Messengers sent between the armies may be captured
- Formally: consensus of two processes assuming unreliable network
- It is impossible to design a safe agreement protocol!

Proof by contradiction (informal):

1. Assume a protocol for correct coordination
2. This protocol contains a **minimal subset** of messages leading to coordinated attack: $m_1,\ldots m_n$
   - After m_n, a coordinated simultaneous attack occurs
3. Since the network is unreliable, m_n may not be delivered. Then,
   - The receiver of m_n will attack differently (minimal subset)
   - But the sender will act the same

   → The protocol was not correct after all



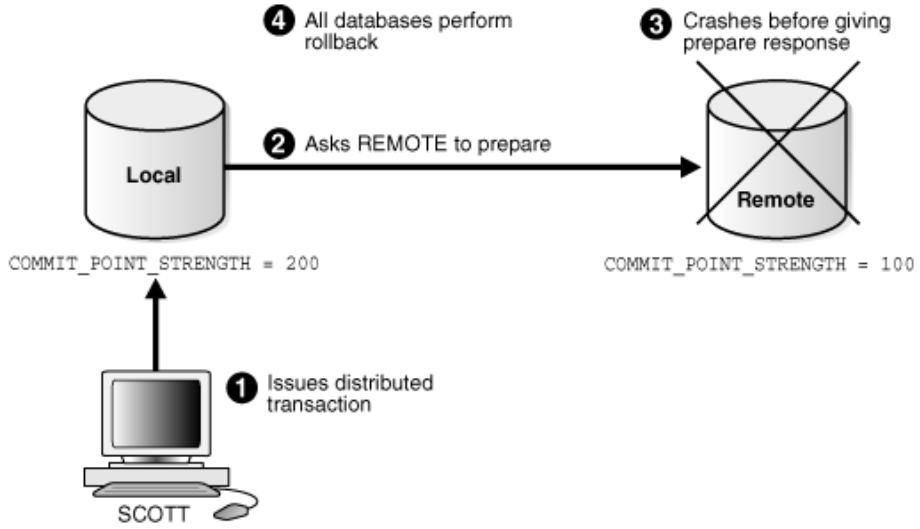
2 Phase Commit (2PC)

- Goal: coordinate decision whether to commit or abort a transaction
  - Special case of consensus protocol
  - Transaction: atomic, consistent, isolated, durable (ACID)
  - Transactional DBMS implement three phases
  1. Begin
  2. Execute
  3. Success ? Commit : Rollback
  - Assumes a global coordinator process

1. **Commit-request / prepare / voting phase**
   - Coordinator asks all nodes to *prepare* commit
     - Write to log
     - Lock modified tables, preventing reads
   - Nodes reply to coordinator: “Prepared” | “No modification” | “Abort (cannot prepare)”
2. **Commit phase**
   - If all nodes have prepared, coordinator asks for actual *commit*
   - Nodes reply with acknowledgement → coordinator closes log & *forgets* transaction

2PC – Fault Tolerance and Implementation

- What if nodes / the network fail?
  - After timeout or node failure, rollback is requested
  - No data change becomes visible externally
  - Rollback based on local logs
- Disadvantages
  - **2PC is a blocking protocol**
  - Coordinator: single point of failure
  - Costly message sends and logging
- Implementation
  - Typically there are multiple coordinators: Transaction Managers (TM)
  - Optimization: make assumptions about failure cases to send less messages
    - Presumed-abort: omit ack messages before aborting
    - Presumed-commit: omit ack messages before commiting



Consensus

- Goal: distributed processors agree on a value
  - **Termination**: all non-faulty processes eventually decide on a value
  - **Agreement**: all processes that decide do so on the same value
  - **Validity**: the value must have been proposed by some process
- Many applications in distributed systems
  - Decide whether a transaction commits
  - Agree on (logical) time
- In synchronous systems, solvable for any number of crash failures
- In eventually synchronous systems, solvable for < n/2 crash failures
- Algorithms: Paxos, Raft, Chandra-Toueg

FLP (Fischer, Lynch, Paterson) impossibility of consensus in asynchronous systems:

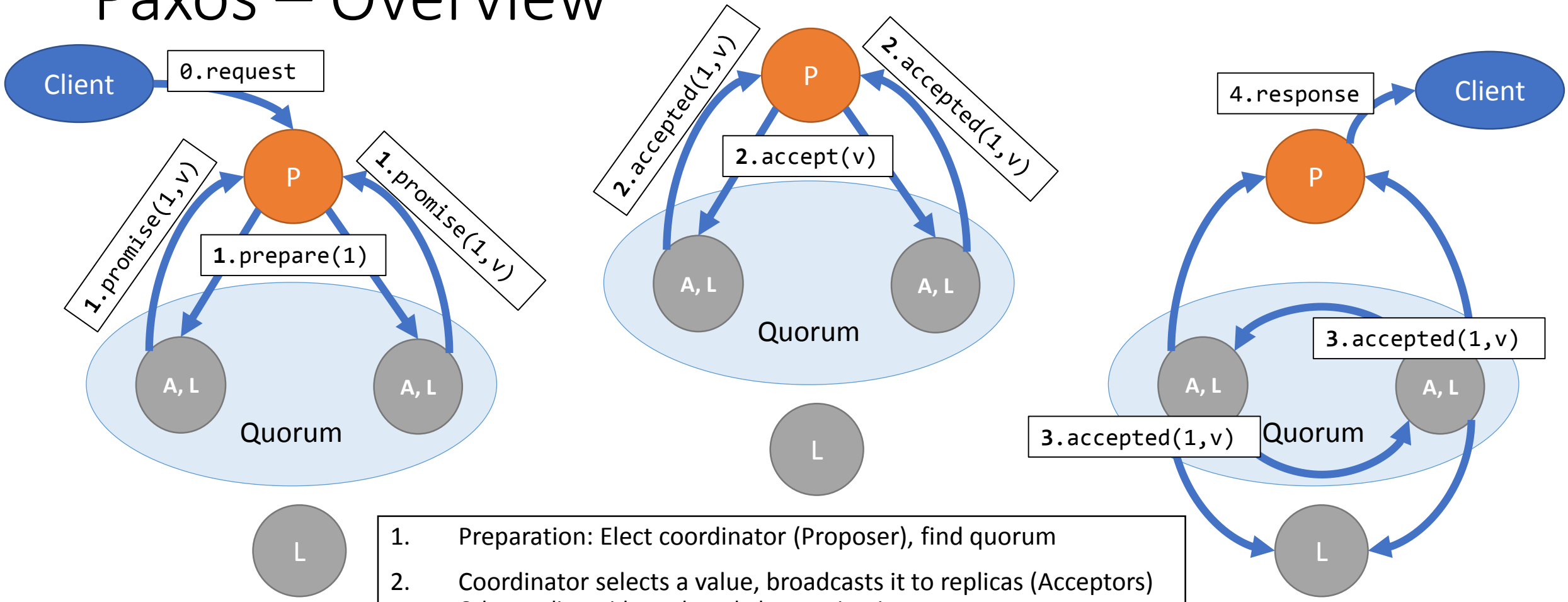
In an asynchronous system, where at most 1 process can fail (fail-stop), no safe consensus algorithm exists.

Paxos (Lamport, 1989)

Consensus: distributed processors agree on a value
- **Termination**: all non-faulty processes eventually decide on a value
- **Agreement**: all processes that decide do so on the same value
- **Validity**: the value must have been proposed by some process

- Proven correct, theoretical consensus protocol for unreliable processors + unreliable network
- System model
  - Crash faults
  - Asynchronous messaging, messages can be lost or duplicated, but not corrupted
  - Persistent processor-local storage
- Roles: Proposers, Acceptors, Learners
  - In reality, many roles are played by identical node → less messaging overhead
  - "Processes" are usually *replicas*
  - Frequently entwined with master-slave replication
    - Master = Coordinator, serves requests
    - When coordinator fails, additional re-election overhead is induced

Paxos – Overview



1. Preparation: Elect coordinator (Proposer), find quorum
2. Coordinator selects a value, broadcasts it to replicas (Acceptors)
   Other replicas either acknowledge or reject it
3. Majority acknowledged? → consensus reached
   Notify all replicas (Learners), so they update their states

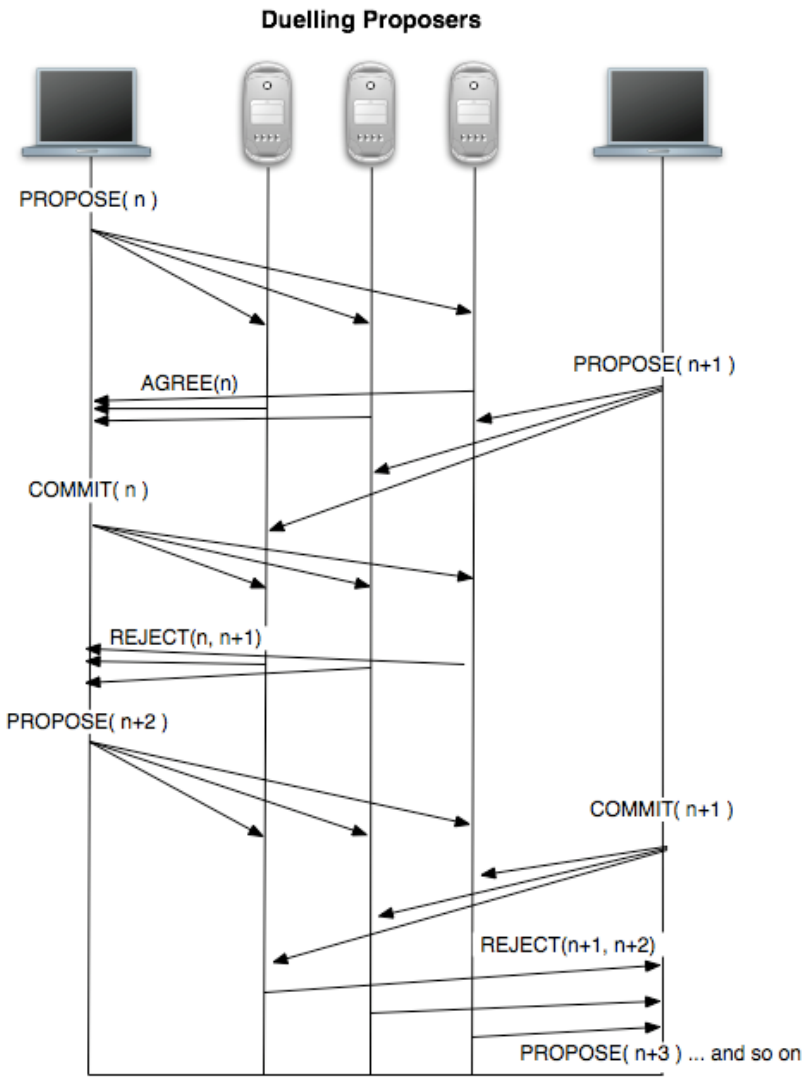
Paxos – Failures

1. Preparation: Elect coordinator (Proposer), find quorum
2. Coordinator selects a value, broadcasts it to replicas (Acceptors)
   Other replicas either acknowledge or reject it
3. Majority acknowledged? → consensus reached
   Notify all replicas (Learners), so they update their states

- Acceptor failure
  - Tolerated, but changes majority
  - To tolerate *F* crash faults, *2F+1* Acceptors are needed
- Proposer failure
  - Leader election may have to ensure that a new proposer exists
- Learner failure
  - Tolerated, due to assumption of persistent storage
  - Needs to synchronize data
- Protocol does not cover data synchronization after recovery or failure detection
- More fault tolerant than 2PC
  - Majority voting (in 2PC, all nodes need to agree)
  - Ordered proposals make redundant proposers possible

Paxos – Duelling Proposers

1. Proposer failure after accept
2. New Proposer
3. Old Proposer recovers
   - Doesn't know it is no longer the only Proposer
   - Gets rejected and re-proposes
   - "Duelling" with increasing proposal numbers
- Leader election *probably eventually* solves this
  - Potential violation of termination requirement!
- Corresponds to FLP impossibility:
  - For liveness, timeouts are needed
  - Solution: introduce partial synchrony



Paxos – Practical Challenges

- Determining roles: Paxos is only as live as its leader election protocol
- **Failure detection**
  - Not specified in the original algorithm
  - Non-trivial in asynchronous systems
- **I/O complexity**
  - Participants to log their state on disk before each message send (they may fail in between)
  - *#Acceptors * #Learners* messages in the accepted phase
  - Needs efficient broadcast implementation
- Consensus on single values only
  - Real systems need to apply chains of updates → Multi-Paxos
- Extensions: {Fast / Cheap / Egalitarian / …} Paxos

> There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. In order to build a real-world system, an expert needs to use numerous ideas scattered in the literature and make several relatively small protocol extensions. The cumulative effort will be substantial and the final system will be based on an unproven protocol.

Paxos Made Live - An Engineering Perspective

- **Chubby**: distributed locking, storage of metadata
  - Used in BigTable / GFS
- Design
  - Disk corruption handling:
    Node with corrupted disk participates in Paxos as Learner, but not Acceptor → rebuilds state
  - Only one master (Proposer), who sends heartbeats
- Engineering
  - For better understanding/correctness validation: DSL for state machines which compiles to C++
  - Runtime consistency checks (checksums, asserts)

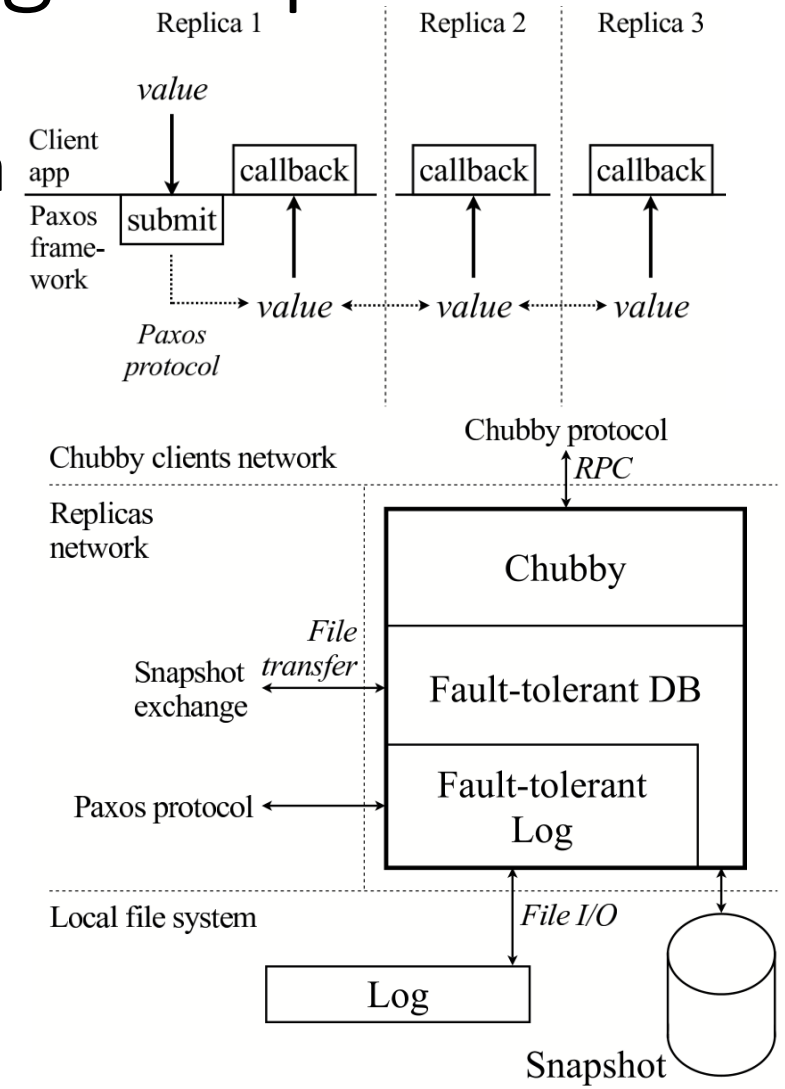
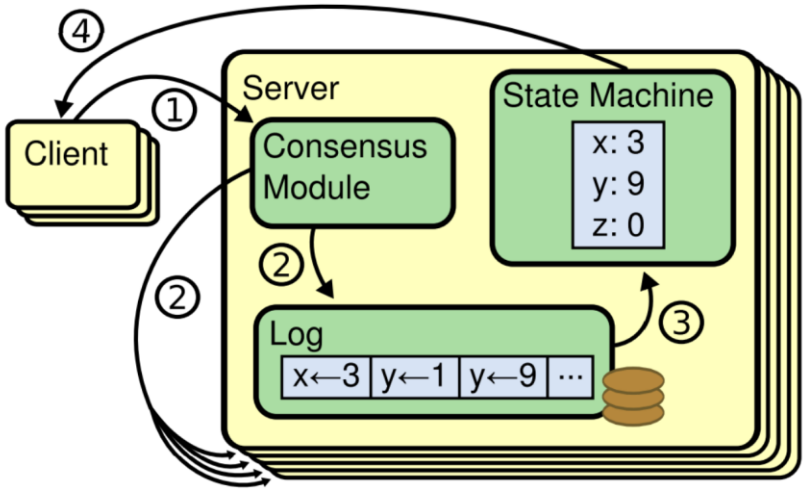


Figure 1: A single Chubby replica.

Raft

- Consensus algorithm designed for *ease of implementation*
- Designed for series of decisions (e.g., log entries)
- Administration by a strong **leader**
  - Receives requests
  - Sends updates + heartbeats
  - Leader election incorporated into the algorithm



Interactive visualization: http://thesecretlivesofdata.com/raft/

Quiz

1. What makes consensus in asynchronous systems hard?
2. What does the FLP impossibility state?
3. What timing model does Paxos assume?
4. What fault model does Paxos assume?

Further Reading

- **“Scalable Web Architecture and Distributed Systems”** http://aosabook.org/en/distsys.html
- **“Distributed Systems for Fun and Profit”** http://book.mixu.net/distsys/single-page.html
- Reading list (theoretical papers) http://courses.cs.washington.edu/courses/cse552/07sp/