Introduction to Consensus Algorithms

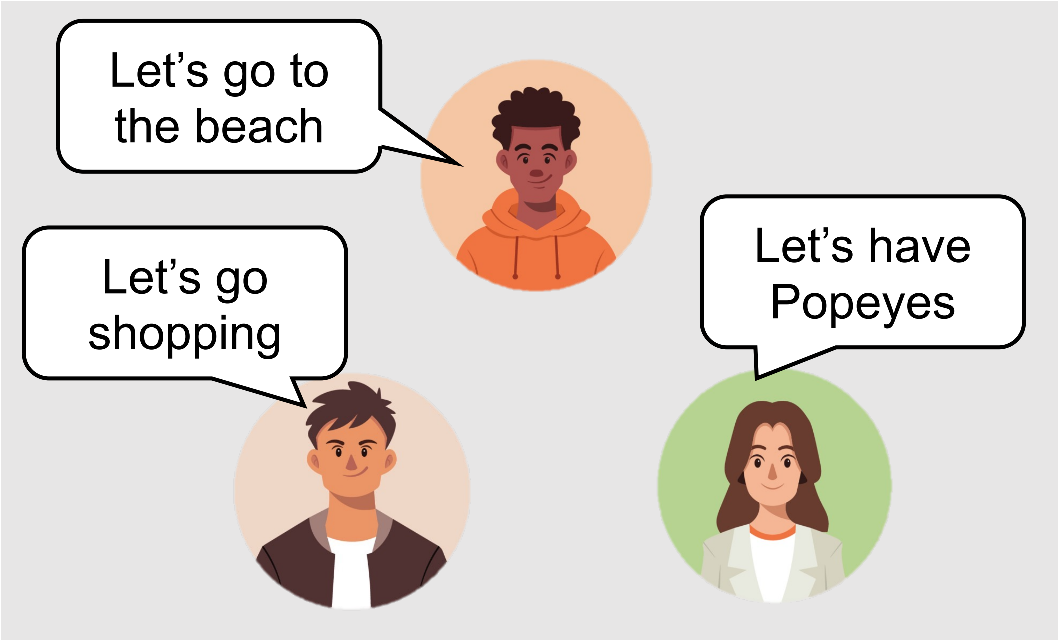
ECE1779 Guest Lecture

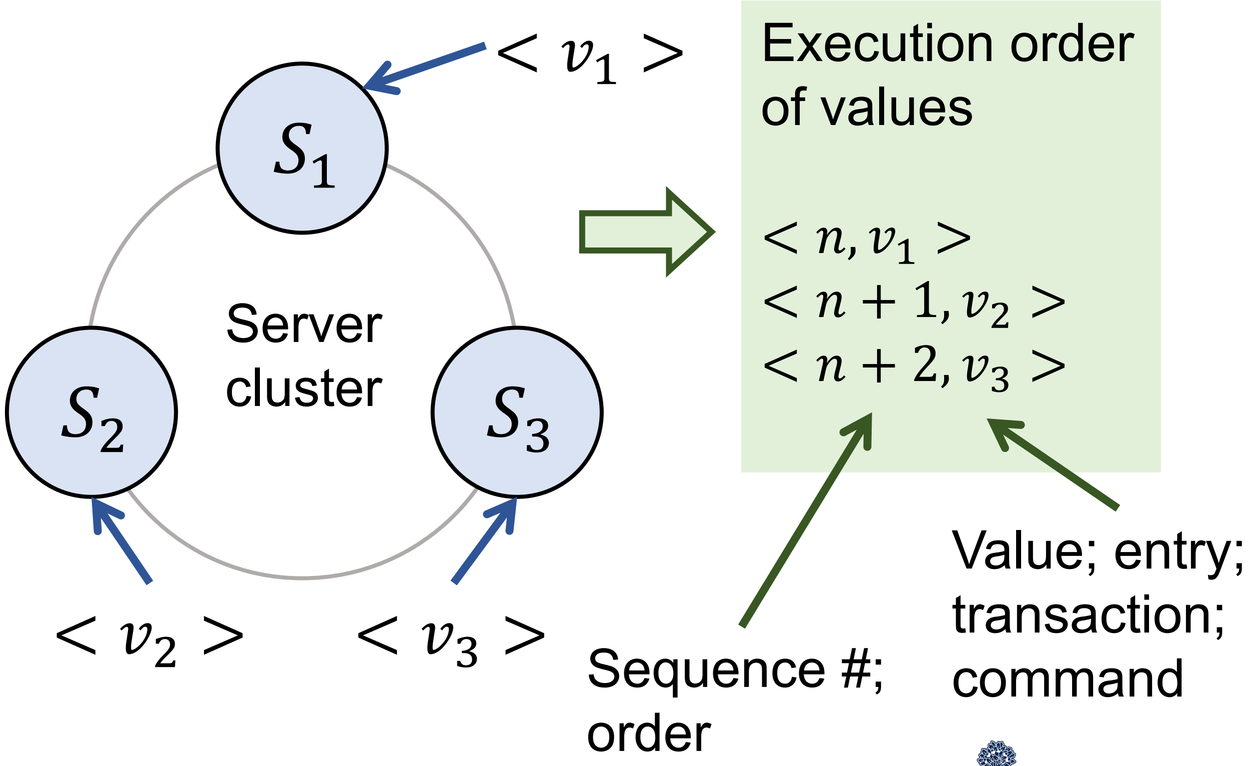
Edward (Gengrui) Zhang, PhD Candidate

ECE, University of Toronto

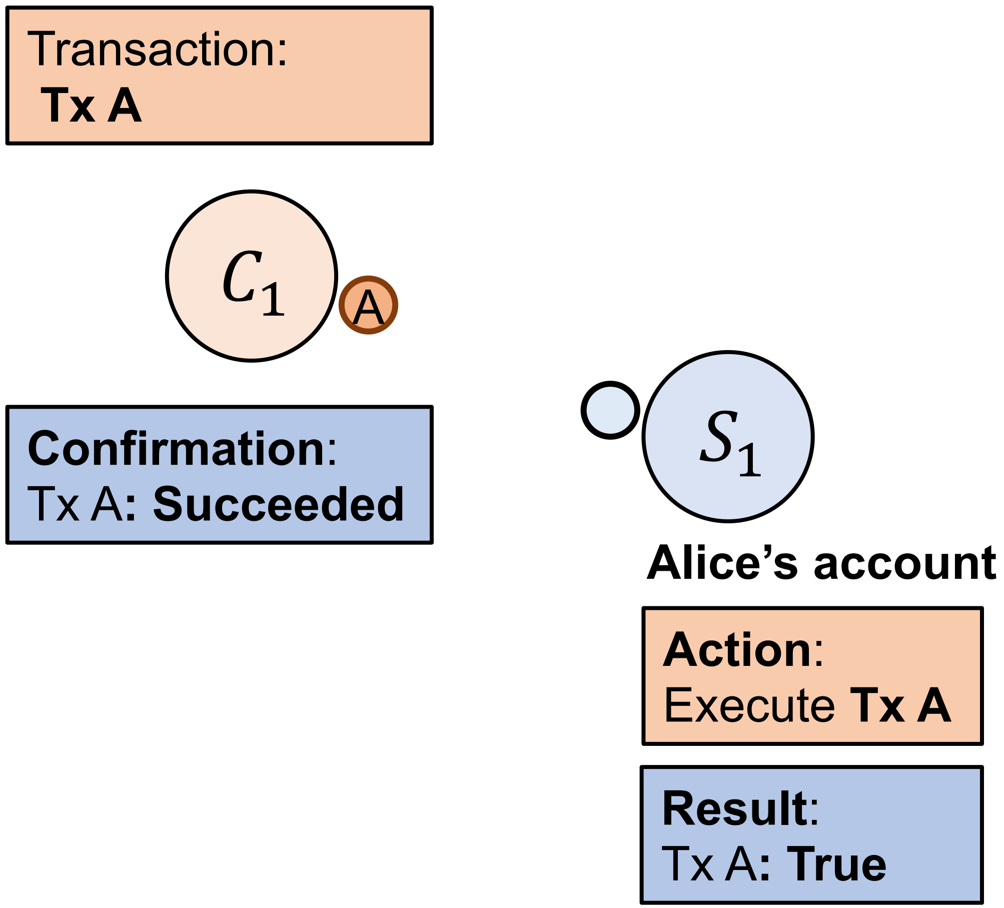
What is a consensus algorithm?

- ``Consensus'' means ``a general agreement''
- In distributed systems, consensus algorithms coordinate server actions to reach agreement on committing values/executing commands





Why do we need consensus algorithms?



- Clients invoke a service by sending a request to server
- Server replies to the client with the result of invocation

Question:
What if the server fails?

Does this model suffice to build safety-critical applications?

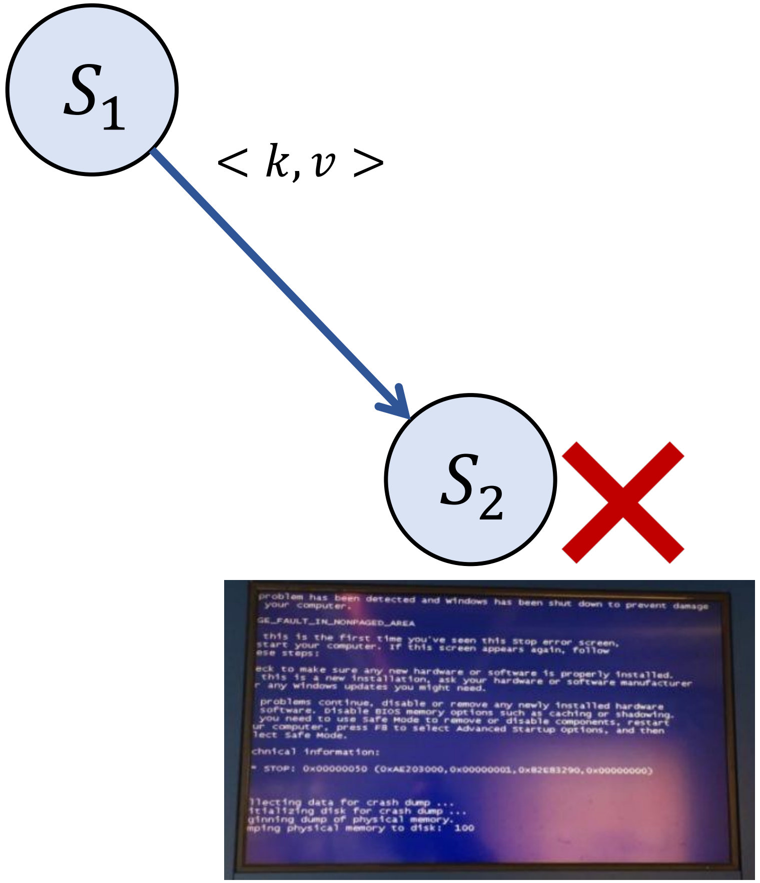
Failures are inevitable and ubiquitous

System Design Goal:

We need a system that can tolerate failures; i.e., a system that can function correctly when failures take place

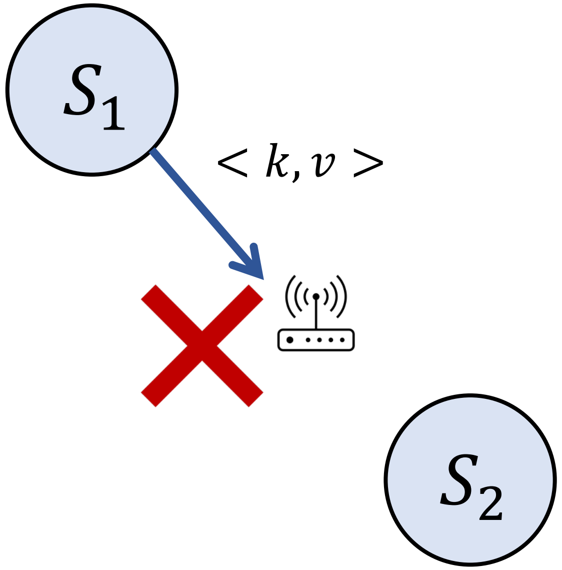
Credit: bennio. via Reddit

Family of failures



Crash failures

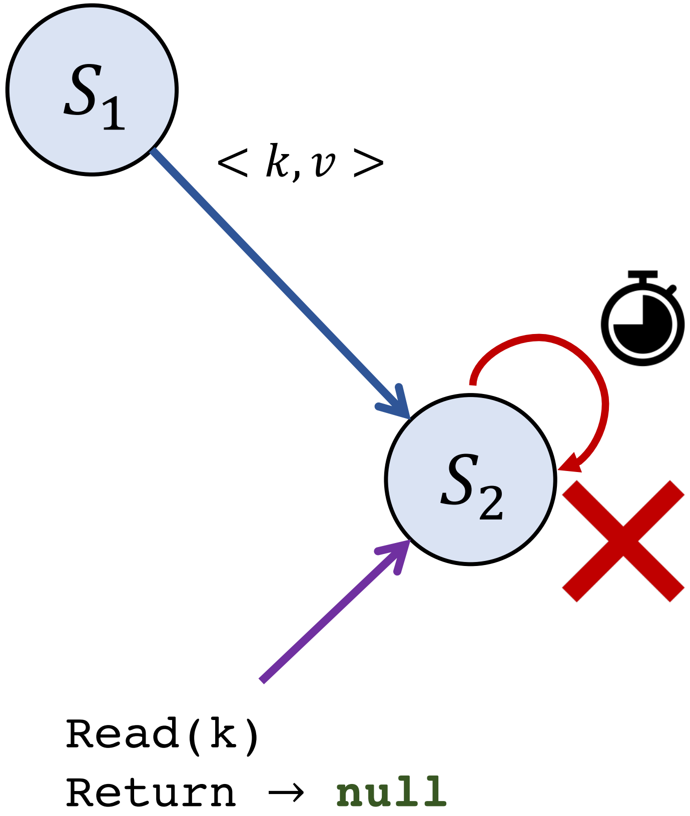
Family of failures



Crash failures

Omission failures

Family of failures

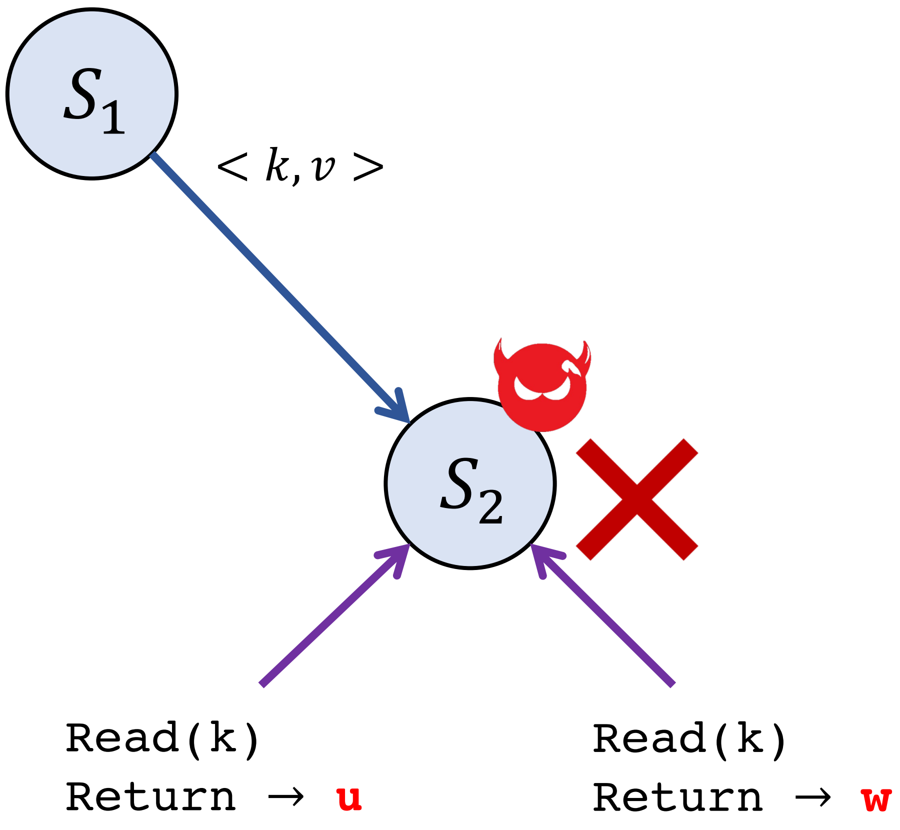


- Crash failures
- Omission failures
- Timing failures

Benign failures

In response to a failure, servers change to a state that permits other servers to detect that failure

Family of failures



Crash failures

Omission failures

Timing failures

Benign failures

In response to a failure, servers change to a state that permits other servers to detect that failure

Byzantine failures/
Internal attacks/
Intrusion attacks

Byzantine failures

Servers can exhibit arbitrary and malicious behavior; faulty servers can collude

Applying fault tolerance with redundancy

- Fault tolerance makes the system operate correctly even if some servers become faulty

Q: Recall how does the disk system tolerate failures?

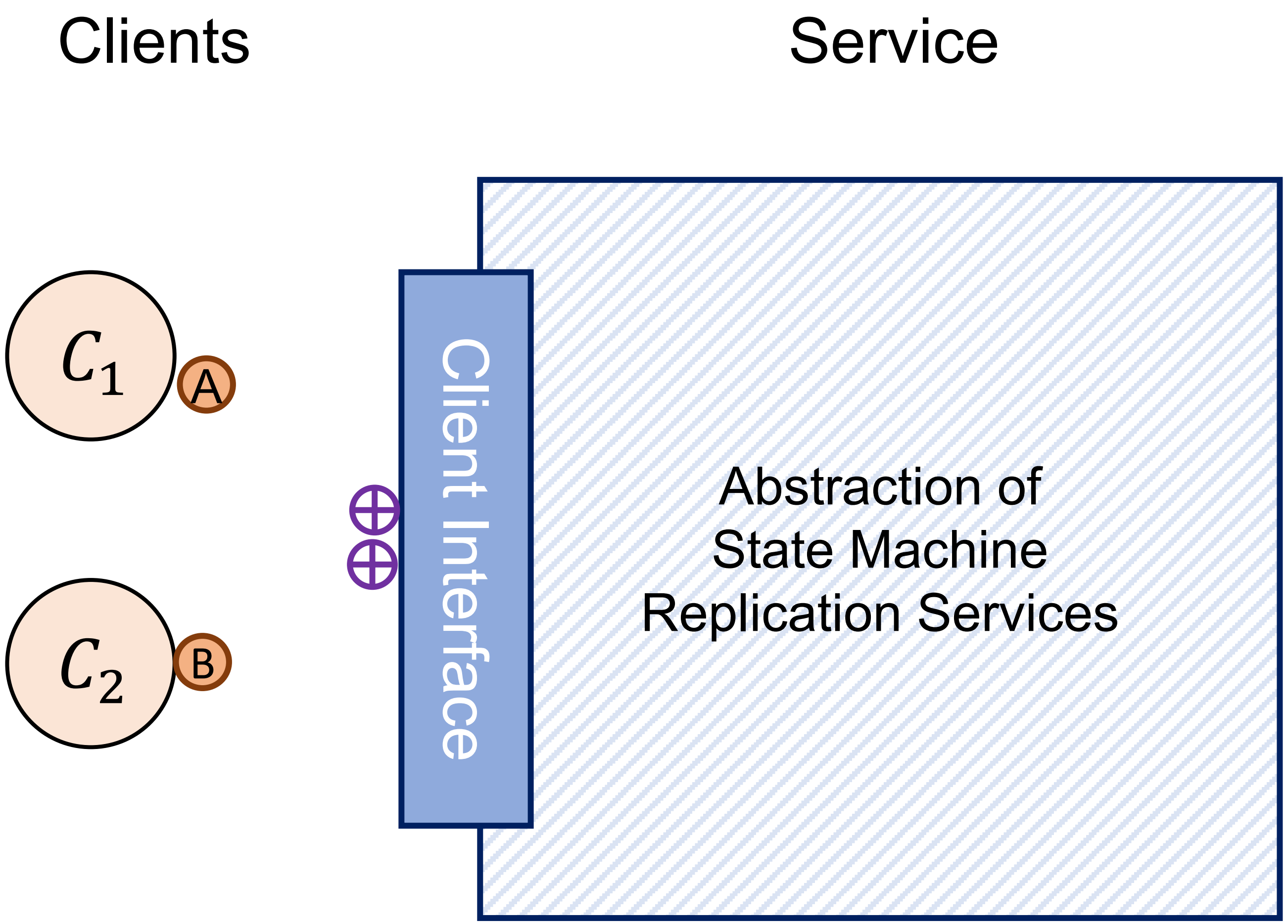
A: RAID (Redundant Array of Independent Disks)

- Redundancy in distributed systems: **a collection of independent servers**
  - Normally, a set of servers operate as a logically single server
  - If some servers are down, the remaining ones can still perform the task

Question

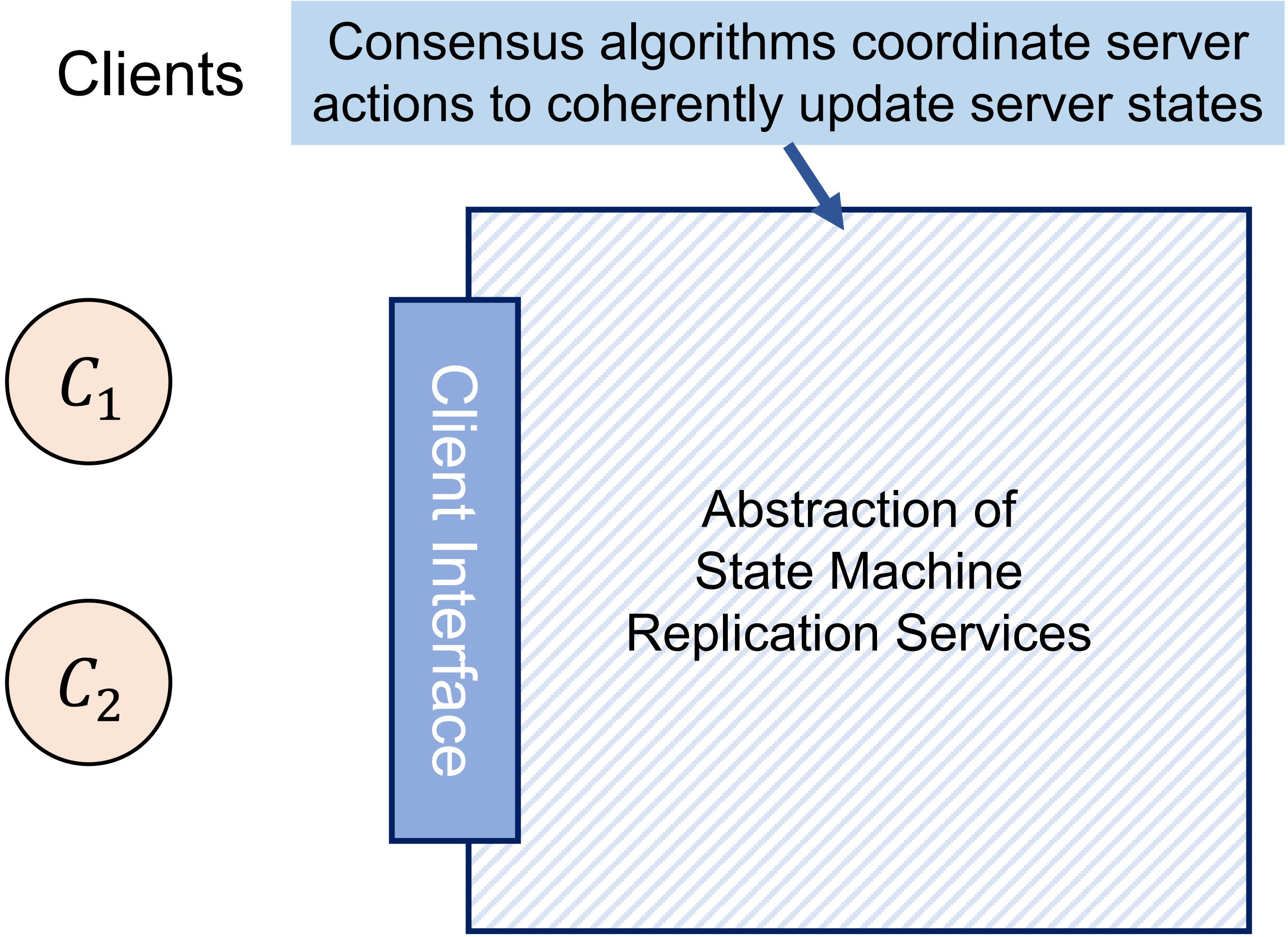
How can we make a set of servers operate like a logically single server? I.e., how can we manage the consistency among servers?

State machine replication (SMR)



- SMR is a replication service where a set of servers compute identical copies of the same state
- SMR provides an abstraction of its replication service with a client interface
- Clients treat SMR services as a black box
  - Send requests to the provided interface
  - Wait for replies to confirm their requests

Consensus algorithms in SMR



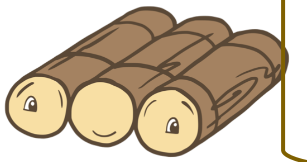
- Consensus algorithms guarantee two service properties: safety & liveness

Safety
All non-faulty servers agree on a total order for the execution of requests despite failures

Liveness
Clients eventually receive replies to their requests

- Exemplary algorithms:
  - Viewstamped Replication (1988)
  - Paxos (1998, 2001)
    - Leslie Lamport
  - **Raft (2014)**

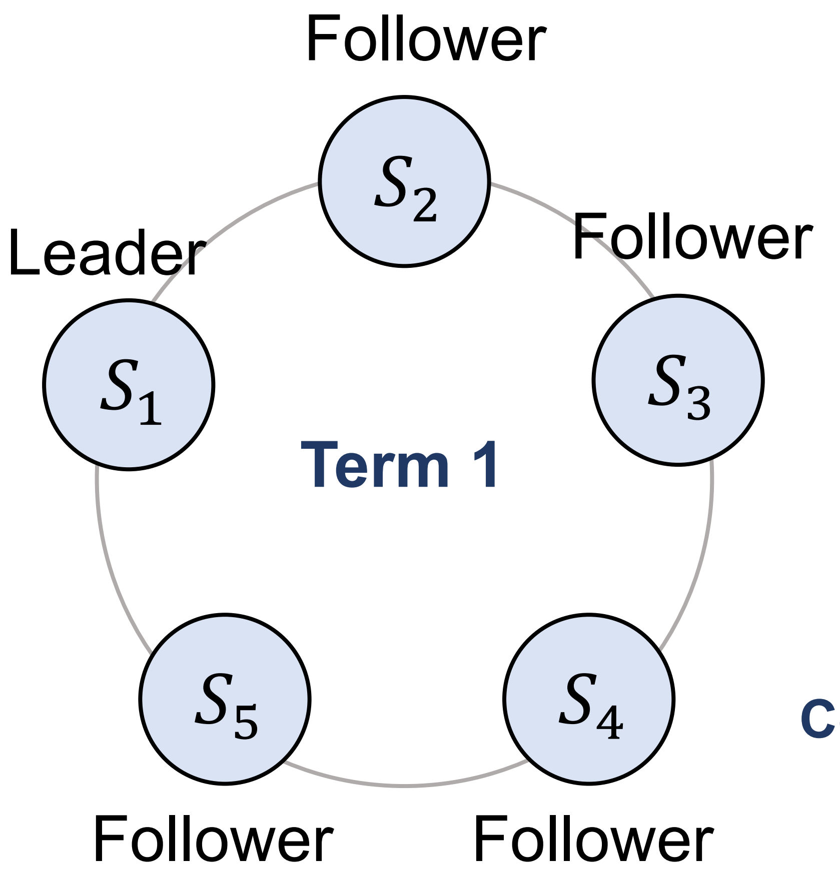
The Raft consensus algorithm



Replicated **A**nd **F**ault **T**olerant

- Published by Diego Ongaro et al. (from Stanford) and received Best Paper Award from 2014 USENIX Annual Technical Conference
- Raft is a leader-based consensus algorithm
  - More understandable than Paxos
  - Uses a designated server as a leader
  - Tolerates benign failures
    - E.g., server crash, packet loss, duplication, and reordering
- It has had a great impact on a wide range of applications:
  - File systems: PolarFS [VLDB'18]
  - Databases: CockroachDB[Sigmod'20], Etcd, and MongoDB (a Raft's variant)
  - Cloud computing: Docker (cluster state), Kubernetes (replication)

Raft basics 1: server states and terms



- Raft has two major operating stages
  - **Replication** and **leader election**
- Each server is in one of three states at any given time: **leader**, **follower**, **candidate**
- Time is divided into **terms** (logical time), which increase monotonically

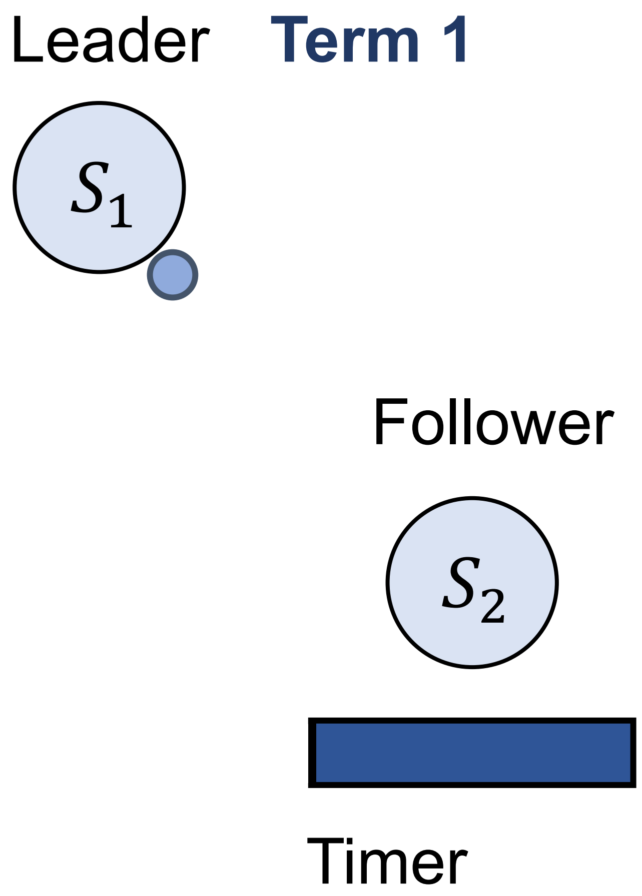
Under normal operation, there is **one leader** in a term and other servers run as followers; the leader coordinates server actions to conduct consensus

Consensus in replication

Consensus in leader election

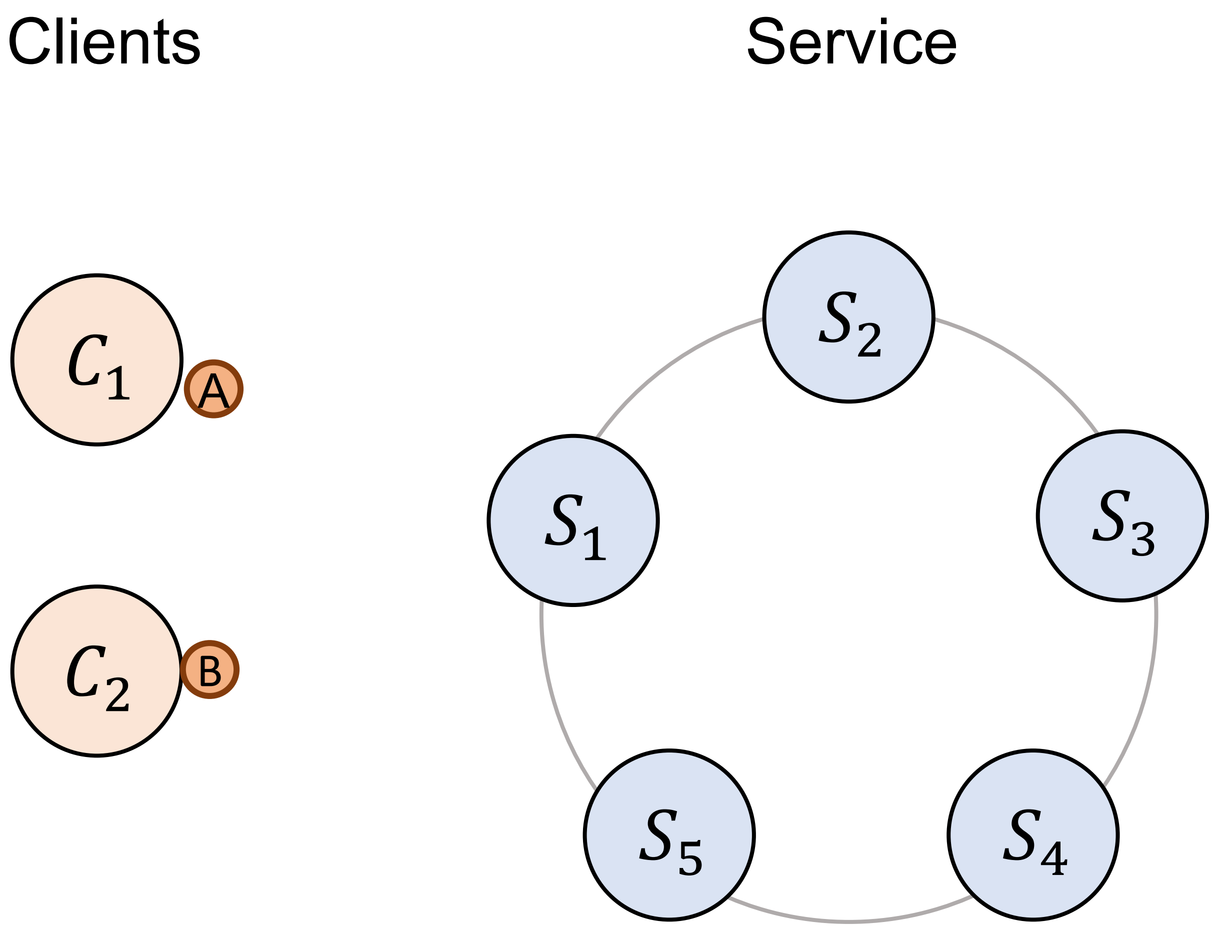
When a leader fails, the other servers start leader election to select a new leader from the remaining servers

Raft basics 2: timers and heartbeats



- Each follower uses a timer to monitor the health of the leader
  - Timer keeps counting down until follower receives a message from leader
  - Otherwise, timer expires; follower becomes a candidate and starts a leader election campaign
- Leader sends periodic heartbeats to reset followers' timers
  - Interval of heartbeats << Timer timeouts
    (e.g., 100 ms) (e.g., 1-2 s)

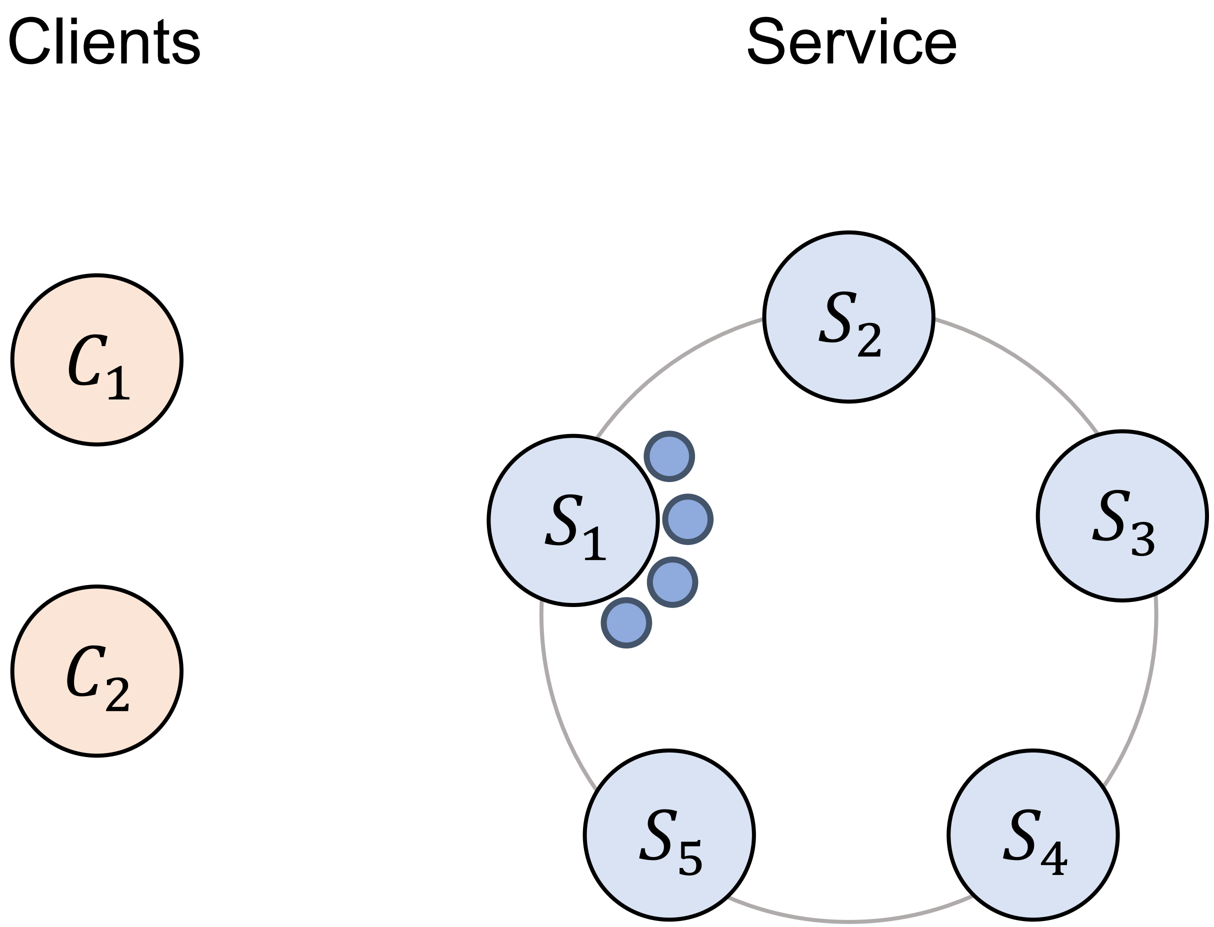
Replication in Raft (Phase I: ordering)



In term 1 (leader: S_1)	
Ordering	Commit
S_1: A B S_2: S_3: S_4: S_5:	S_1: S_2: S_3: S_4: S_5:

- Clients send requests to leader

Replication in Raft (Phase I: ordering)

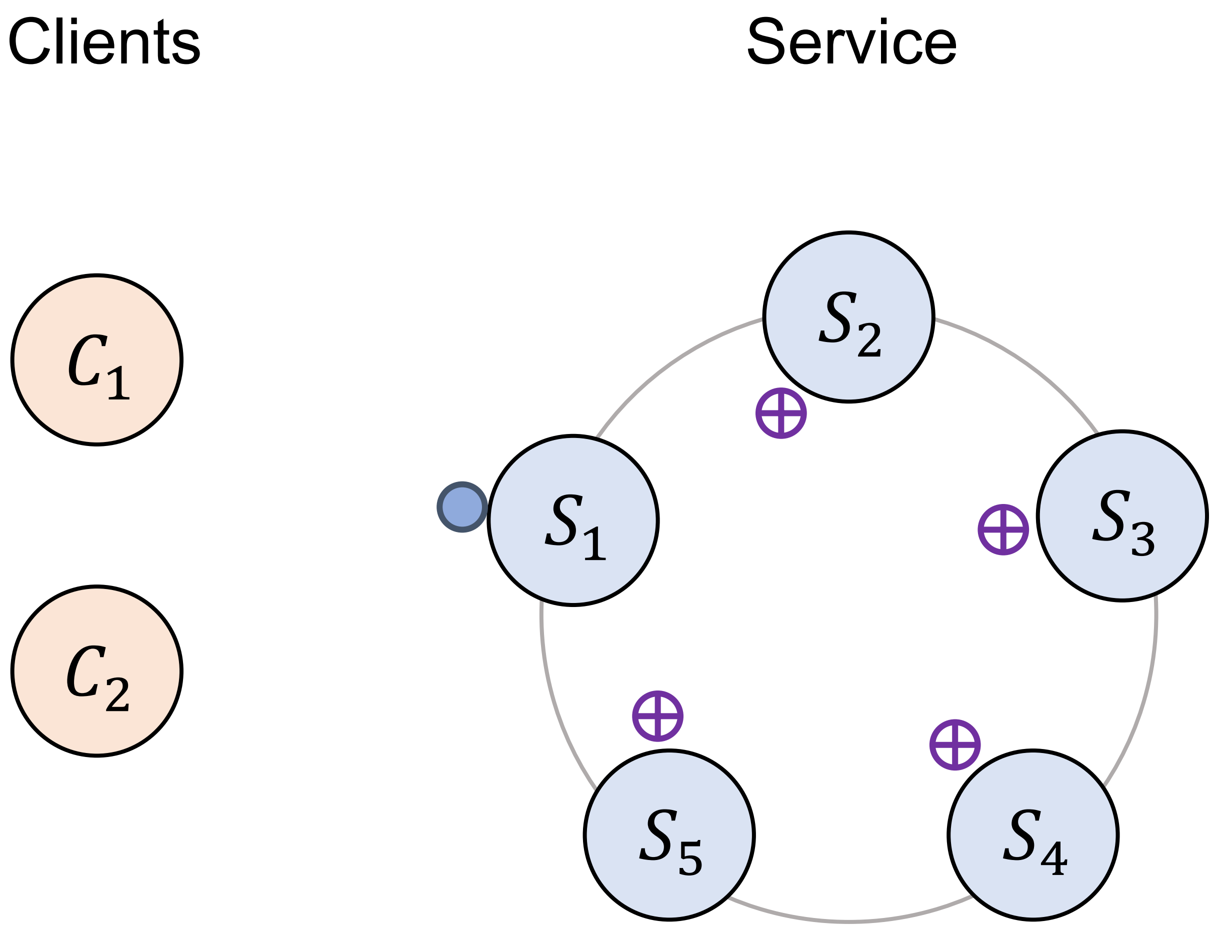


In term 1 (leader: S_1)	
Ordering	Commit
S_1: A B	S_1:
S_2: A	S_2:
S_3: A	S_3:
S_4: A	S_4:
S_5: A	S_5:

- Clients send requests to leader
- Leader assigns a sequence # to client request
  - E.g., $< n, a >$; $< n+1, b >$

UNIVERSITY OF TORONTO

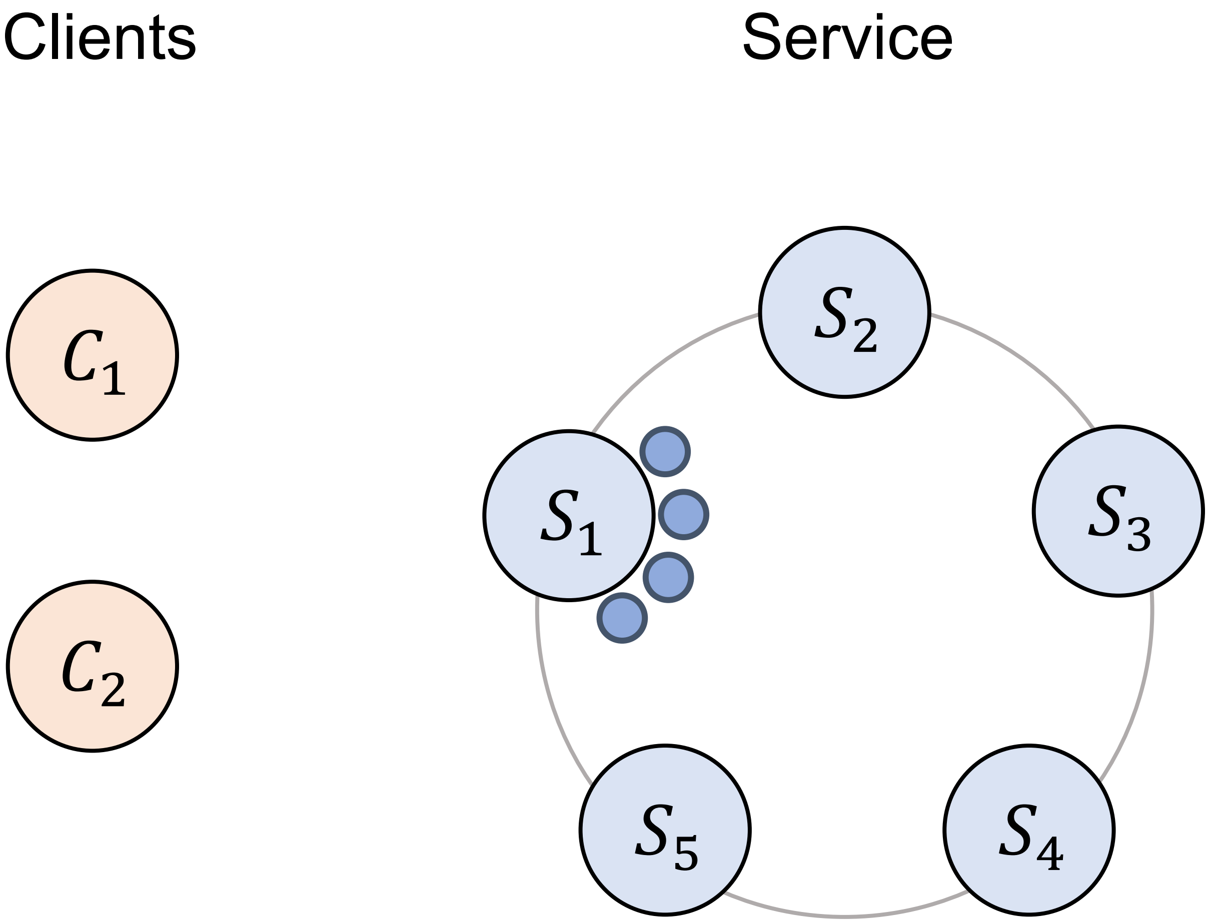
Replication in Raft (Phase II: committing)



In term 1 (leader: S_1)	
Ordering	Commit
S_1: A B	S_1: A
S_2: A	S_2:
S_3: A	S_3:
S_4: A	S_4:
S_5: A	S_5:

- Clients send requests to leader
- Leader assigns a sequence # to client request
  - E.g., $< n, a >$; $< n+1, b >$
- Leader commits the value if **a majority of replies** can be collected

Replication in Raft (Phase II: committing)



In term 1 (leader: S_1)	
Ordering	Commit
S_1: A B	S_1: A
S_2: A	S_2: A
S_3: A	S_3: A
S_4: A	S_4: A
S_5: A	S_5: A

- Clients send requests to leader
- Leader assigns a sequence # to client request
  - E.g., $< n, a >$; $< n+1, b >$
- Leader commits the value if **a majority of replies** can be collected

Summary of Raft's replication

- Strong leadership:
  - Log entries flow only **from the leader to followers**
  - Follower must synchronize its log according to leader's log

- Quorum replication:
  - In a system consisting of $n = 2f + 1$ servers, an action can be agreed upon by $f + 1$ servers (majority)
    - E.g., in a 5-server system, 3 servers form a majority
  - A minority ($\leq f$) of slow servers do not impact overall replication performance

Impact of failures

- Under a correct leader, as long as a majority of servers are correct, the system can operate correctly

However, what if the leader fails?

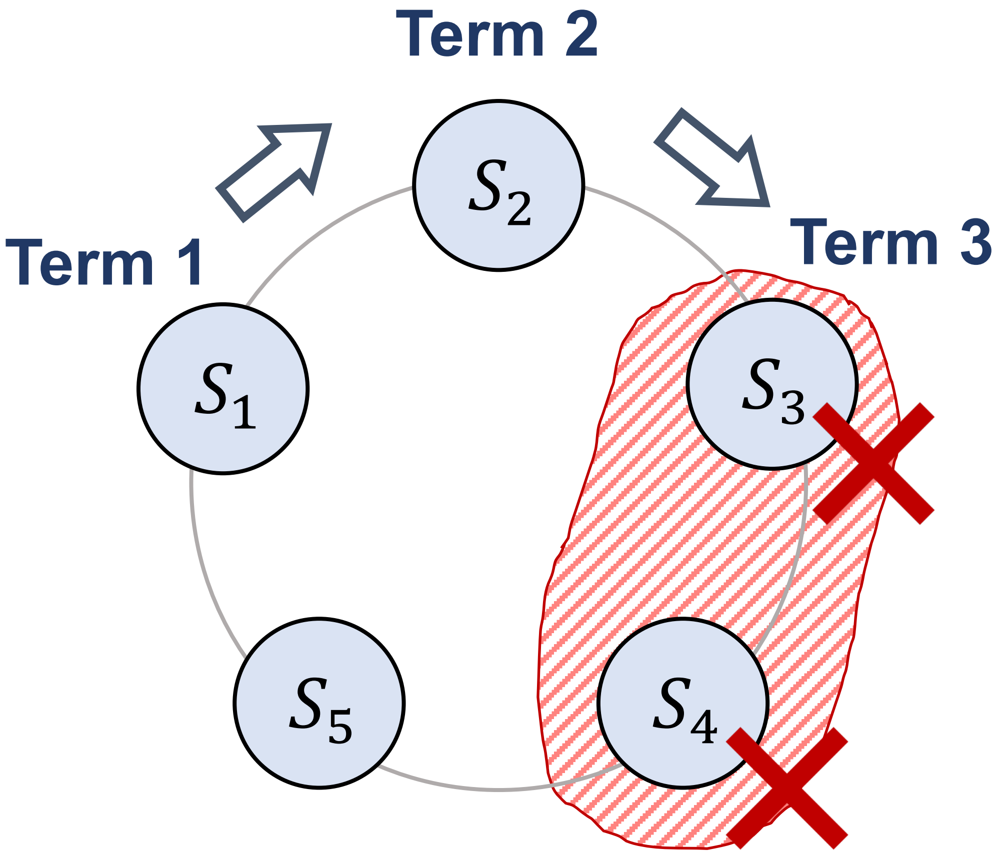
- Leader is the most crucial role
  - It interacts with clients and coordinates consensus with other servers
  - Other servers synchronize with the leader

A new leader should have
- **the highest term value**
- **the most up-to-date log**

→

Make sure the system **never falls back** to a previous state; i.e., not loosing log entries when leadership changes

Solution 1: Passive leadership rotation



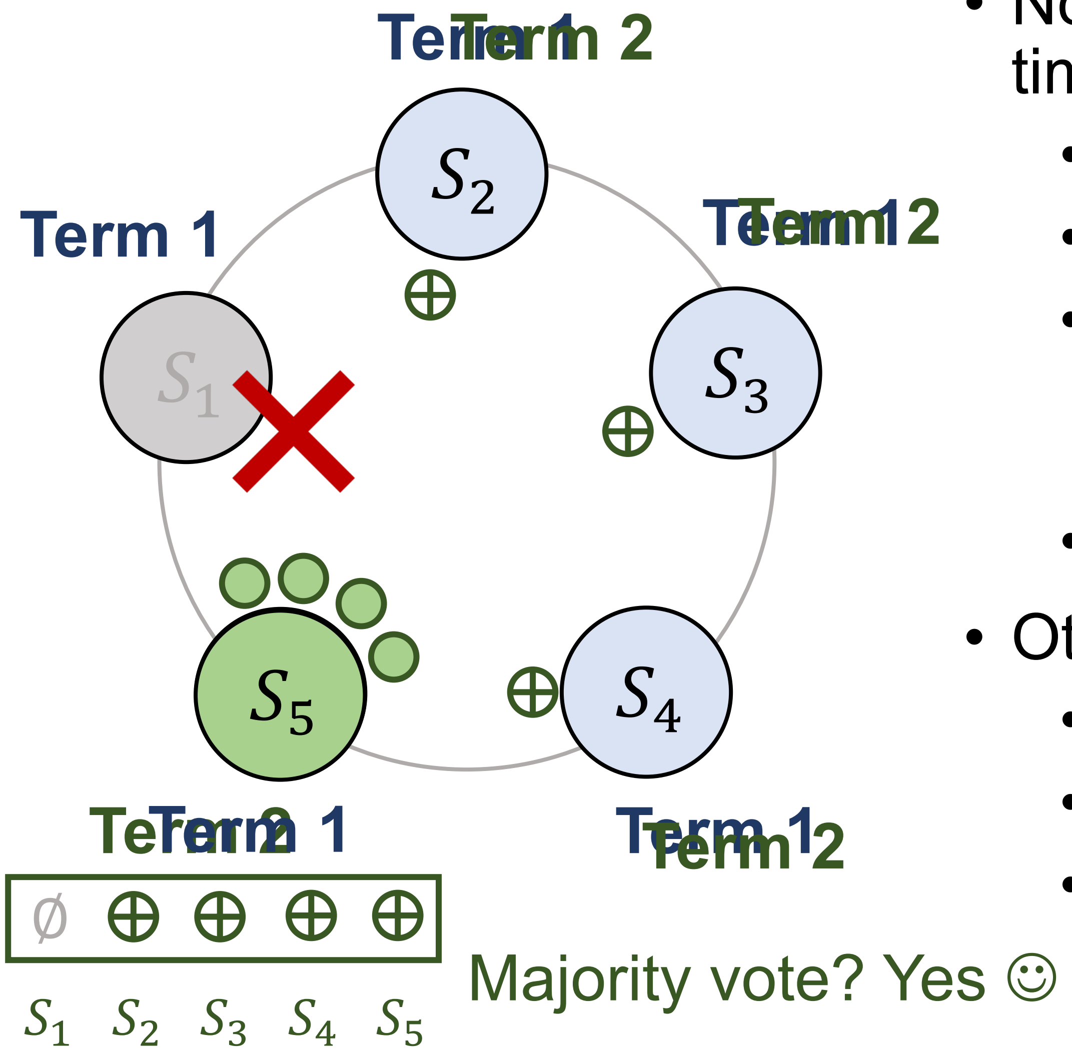
[Viewstamped Replication]

- All servers follow **a pre-defined leader schedule** to rotate leadership
  - Leader = Term *mod* # of servers
    - I.e., leadership is assigned to $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_1 \rightarrow \cdots$
- Pros:
  - Simple; easy to understand and implement
- Cons:
  - Cannot avoid already crashed servers
  - Cannot avoid slow servers

Can we do better?

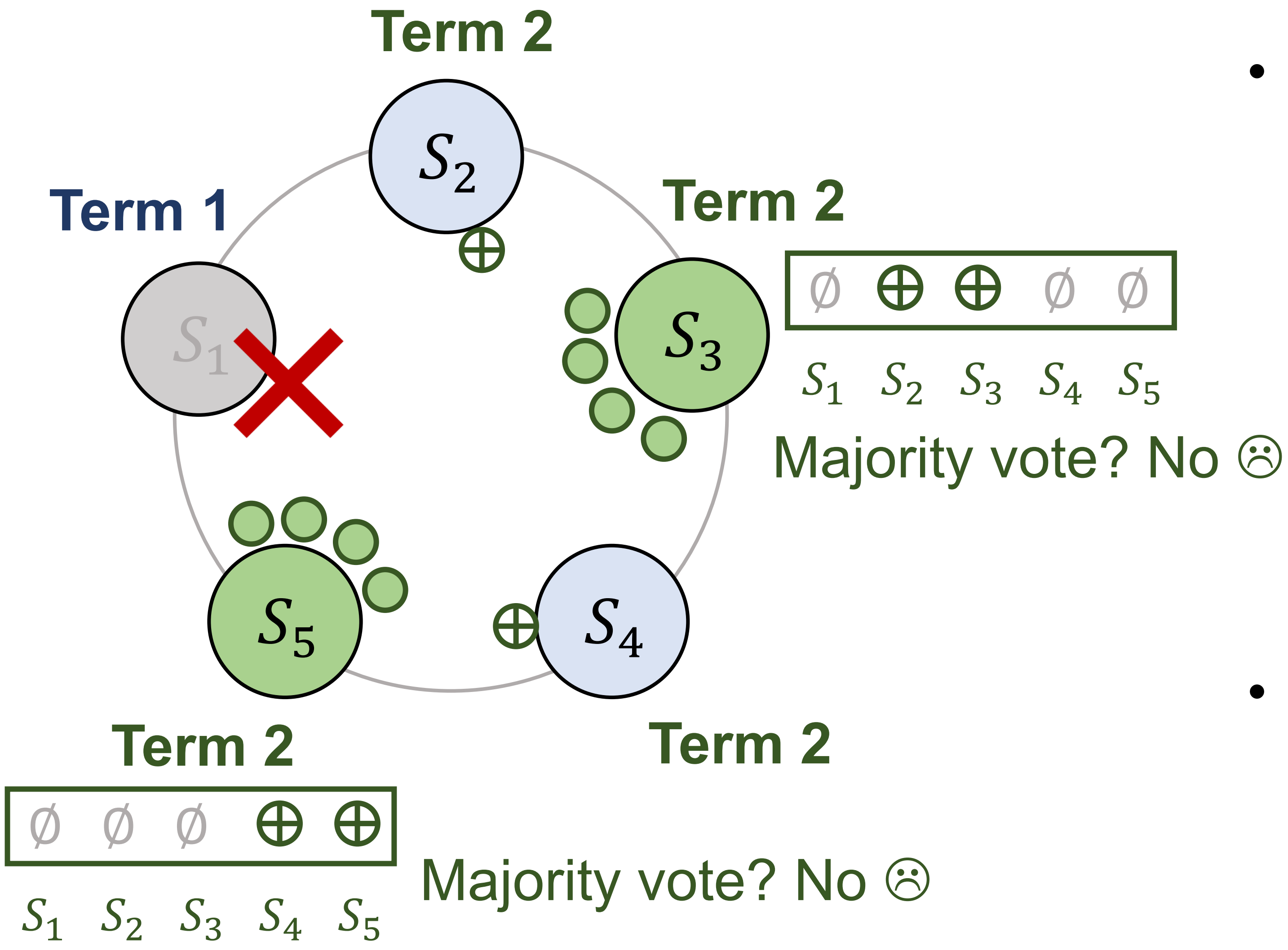
UNIVERSITY OF TORONTO

Raft's solution: Active leader election



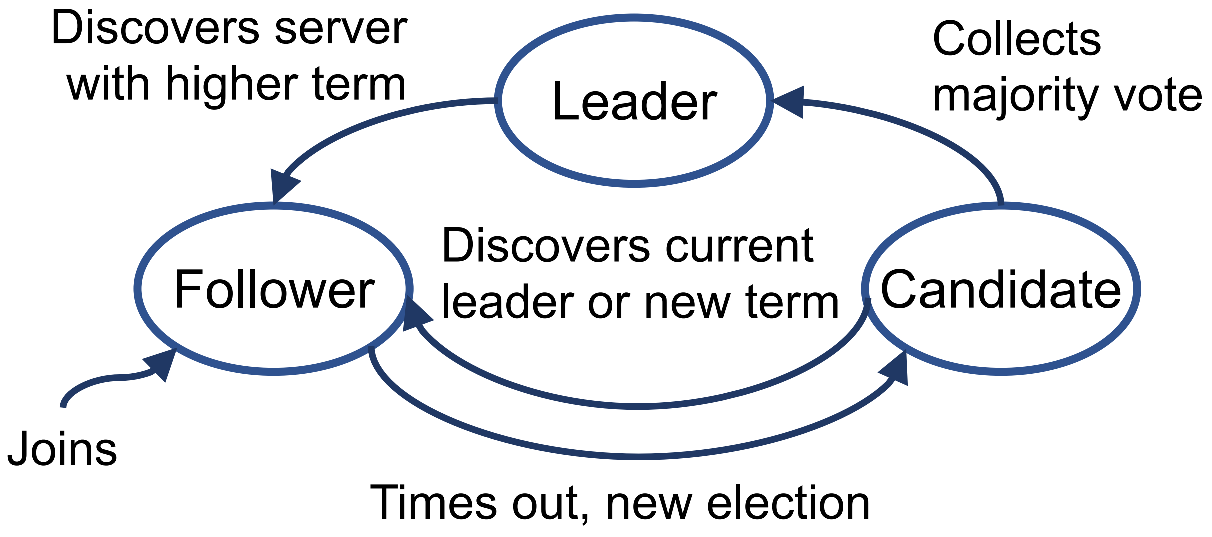
- No leader schedule; whoever triggers a timeout campaigns for leadership
  - Transitions to **candidate**
  - Increments its term
  - Sends out messages to request votes from other servers in the form of `<term, lastLogIndex, lastLogTerm>`
  - Votes for itself
- Other servers vote for the candidate if
  - Candidate's term >= receiver's term
  - Receiver has not voted in this term
  - Candidate's log is at least as up-to-date as receiver's log

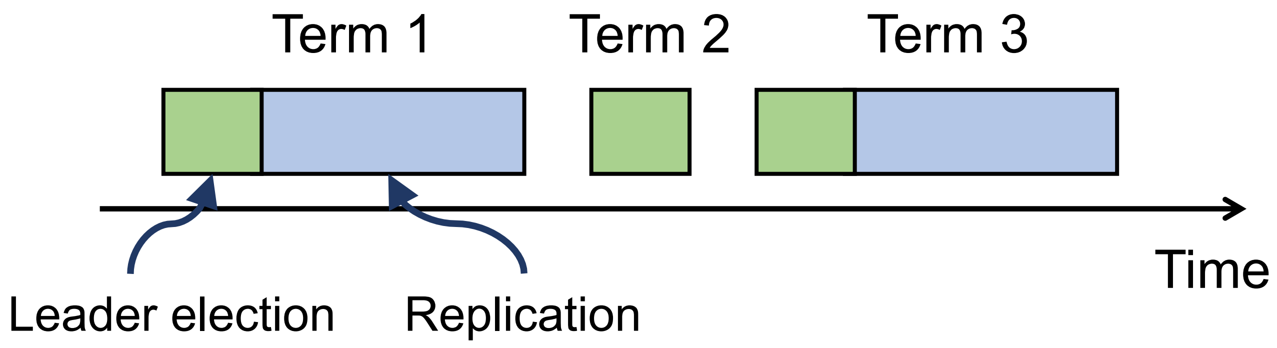
A problem: split votes in leader election



- Servers S_3 and S_5 both trigger timeouts and transition to candidate state
  - S_2 votes for S_3 and will not vote for S_5
  - Similarly, S_4 votes for S_5 and will not vote for S_3
  - **No candidate can collect majority vote**
- Raft's solution:
  - Randomized timers
  - Waiting for new elections

Raft’s server state transition





- Only a qualified server will be elected as a new leader
  - Crashed servers will not be assigned with leader duty
  - Slow servers will not be elected
- At most one leader can be elected as a leader in a given term
  - Elected leader conducts consensus in its term
  - Some terms may result in no leader being elected

Summary

- Raft operates in a succession of **terms** with two major components
  - Leader election: the consensus to agree on a leader
  - Replication: the consensus to agree on client requests

- Raft is fast and efficient
  - It can tolerate up to $f + 1$ benign failures among a total of $2f + 1$ servers
  - Under normal operation, it can achieve consensus by collecting replies from a majority of servers ($f + 1$)
  - Its leader election mechanism allows servers to proactively campaign for leadership, thereby avoiding unqualified servers to be elected

Additional resources

- This lecture does not cover all the details of Raft
  - Check out the full paper at: https://raft.github.io/raft.pdf
  - Raft's visualization: https://raft.github.io/

- Solving Raft's split vote problem:
  - G. Zhang and H. -A. Jacobsen, "ESCAPE to Precaution against Leader Failures," *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS).* https://ieeexplore.ieee.org/document/9912172

Gengrui Zhang | home page
gengruizhang.github.io

Learn more about consensus algorithms at:
https://gengruizhang.github.io/

MSRG Linkedin