SCC.311: Fault Tolerance

Prof. Barry Porter

Overview for today

- Introduction to fault tolerance: concepts and terminology
- Failure detection for fail-stop and Byzantine failures
- Server replication schemes for fault tolerance (active vs. passive)

Terminology

- We need to be able to differentiate different kinds of problem in fault tolerance, so we define specific terms:

  - **Failure**: inability of a system (or subsystem) to perform its required function

  - **Error**: transition of internal state into an invalid state

  - **Fault**: the cause of an error

Not all faults and errors cause a failure, but all failures and errors are caused by a fault

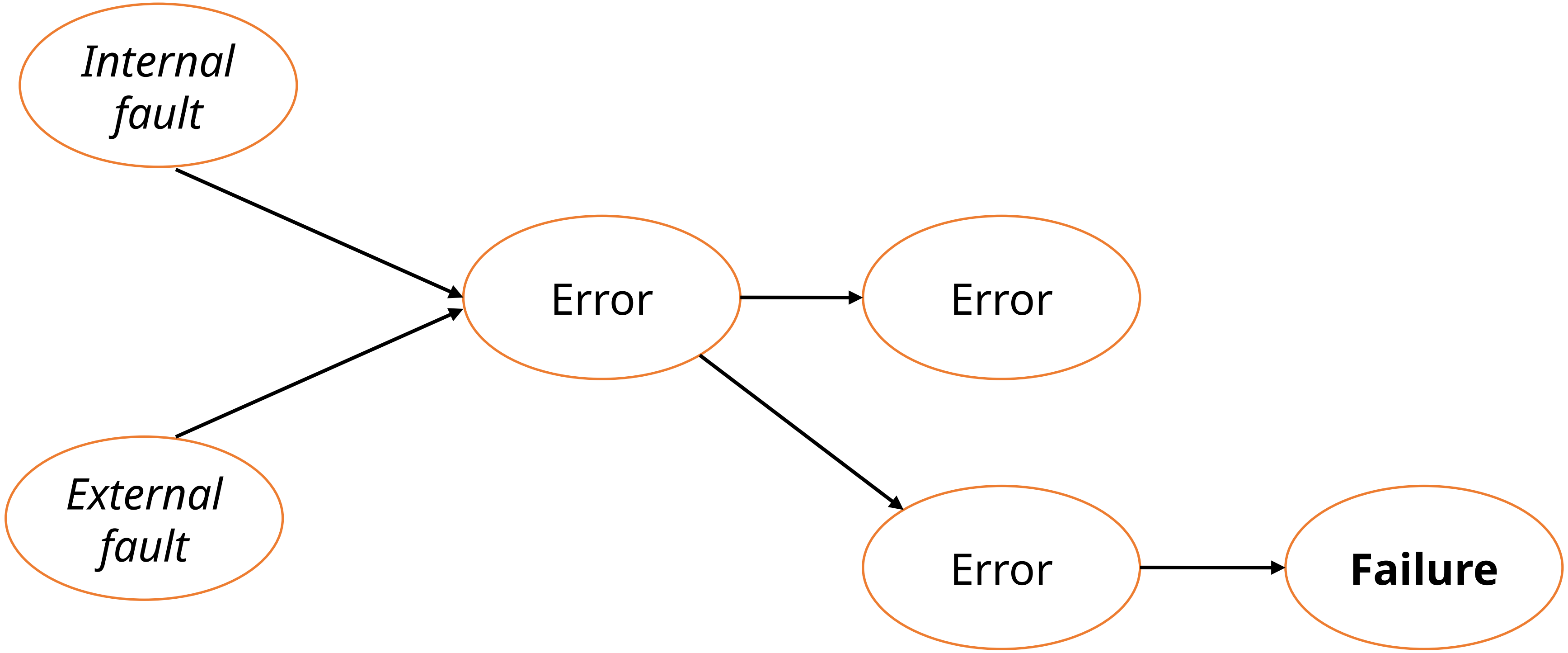
Terminology

Easy to detect

- Types of fault:
  - **Omission**: a specific response / expected event does not arrive
  - **Crash**: system stops – a kind of omission where *all* responses fail to arrive
  - **Timing**: a response arrives outside of its expected window (early or late)
    - an omission fault (above) can be seen as a timing fault with infinite time
  - **Byzantine (or arbitrary)**: a response occurs but with unexpected / invalid / malicious contents

Hard to detect

Propagation of faults → errors → failures



Not all faults and errors cause a failure, but all failures and errors are caused by a fault

How is a service performing?

- At a high level we can model a service in terms of its **availability** and its **reliability**

Availability

Readiness to offer a service (*service responds when requested*)

Reliability

Continuity of correct service (*service operates without failures*)

Examples:
system crashes for 1s every hour: 99.9% available but unreliable
system never crashes but undergoes maintenance once per week: 100% reliable but 98% available)

How is a service performing?

- We can also model *quality of service* on a scale
  - Here the notion of a failure varies
    - A web server takes 5 minutes to respond with the correct reply
    - This is functionally correct, but is it an acceptable level of service?

- A service exhibits *graceful degradation* if it avoids total failure with potentially reduced service
  - This is a key principle which gives dev ops / administrator teams **time to react** to a failure without loss of all service to users

How common are failures?

- Modern data centres use cheap, commodity server hardware; as a result, machine failures are common

Google experiences over 1,000 **total server failures** per year, with thousands of hard drive failures and multiple power distribution unit failures per data centre

Microsoft sees an average of 5 network device (switch/router) total hardware failures **per day** in its datacentres, with a large number of transient failures

Common fault tolerance approaches

Replication
Run multiple copies of a service on different hardware units, either for availability of reliability

N-version design
Design a system in multiple different ways, making it less likely that all versions will experience the same error

Checkpointing and operation logs
Save the state of a system periodically so that we can recover from a failure by re-loading the most recent checkpoint

Fault tolerance can generally be enhanced with **heterogeneity**: e.g. of hardware, software, physical location; distributed systems are very well placed to take advantage of this effect

Coming up…

- Failure detection (crash-stop and Byzantine)

- Replication (passive and active)

Failure detection in distributed systems

- The *impossibility* of detecting crash failures in a distributed system is a key defining proof which affects the design of fault-tolerant systems[1]

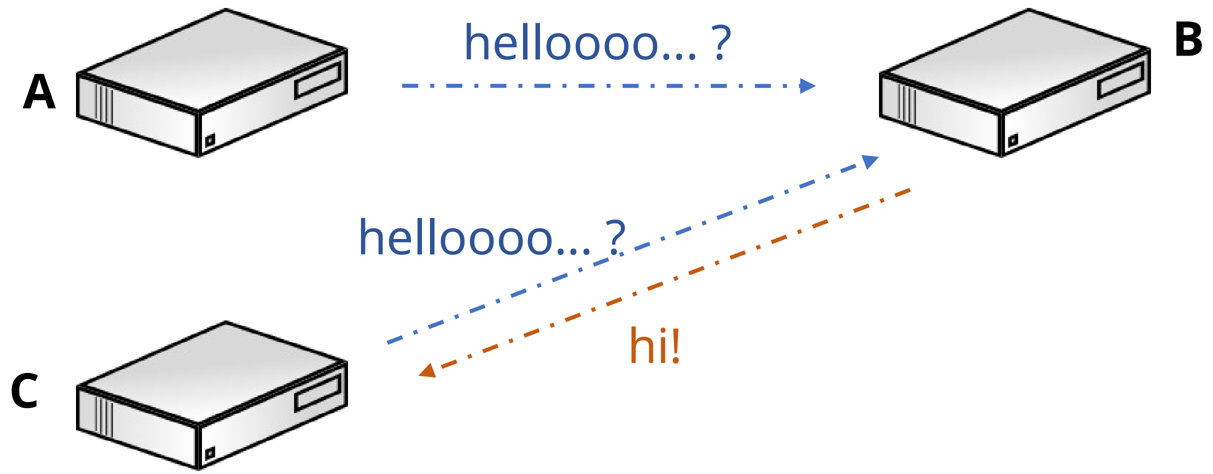
- This proof says that the only way to detect a crash failure is to ask a computer if it is still alive, and wait for a reply
  - However, it is impossible to decide at what point we have waited *long enough* to declare a computer has crashed, vs. being slow / busy / network delays

We typically therefore wait for a set amount of time to declare a computer failed, but ***be prepared to be wrong*** if we later get a valid response from that computer

[1] Impossibility of distributed consensus with one faulty process, Fischer, Lynch, Paterson, 1985

Failure detection in distributed systems

- In some cases temporary network errors or partitions can cause *disagreement* about the crash status of remote computers
  - Server A may think that server B has crashed, but server C thinks B is alive
  - If A and C talk to each other about B, they're going to disagree...



Byzantine failures

- If a server crashes we can detect its failure with a timeout and design a protocol to continue as normal afterwards

- If a server starts sending us ***garbage data*** or ***malicious data***, this can disrupt the orderly flow of a communication protocol and cause an entire distributed system to enter erroneous states

- These conditions are much harder to detect than a server crash

Byzantine failures

- Handling this kind of failure is often called the *Byzantine Generals* problem
  - based on a fictional story developed by Leslie Lamport to help explain the distributed systems consensus problem

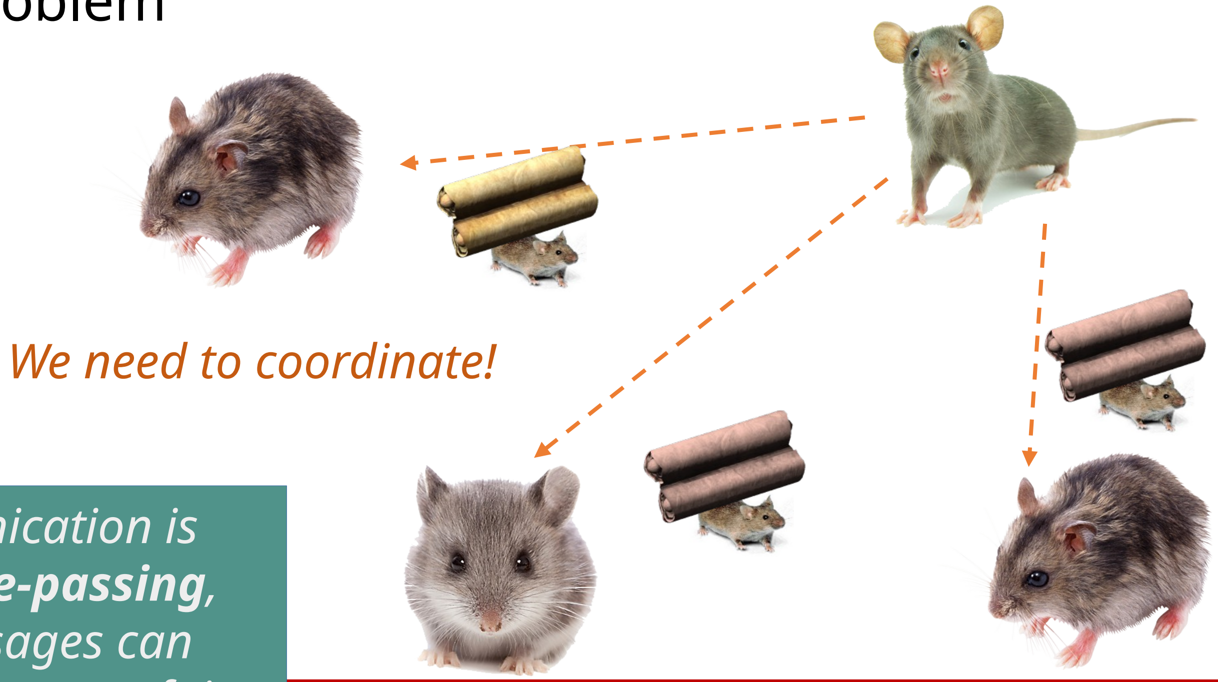
Byzantine failures

- Handling this kind of failure is often called the *Byzantine Generals* problem
  - based on a fictional story developed by Leslie Lamport to help explain the distributed systems consensus problem

We need to coordinate!

Byzantine failures

- Handling this kind of failure is often called the *Byzantine Generals* problem
  - based on a fictional story developed by Leslie Lamport to help explain the distributed systems consensus problem



*All communication is by **message-passing**, where messages can take any amount of time to arrive, and may be corrupted in transit*

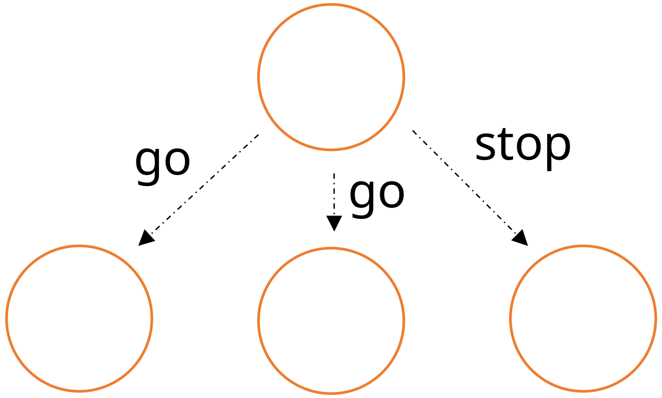
Byzantine failures

- Handling this kind of failure is often called the *Byzantine Generals* problem
  - based on a fictional story developed by Leslie Lamport to help explain the distributed systems consensus problem

A malicious actor can exploit message-passing to send different things to different nodes; this can cause catastrophic failures

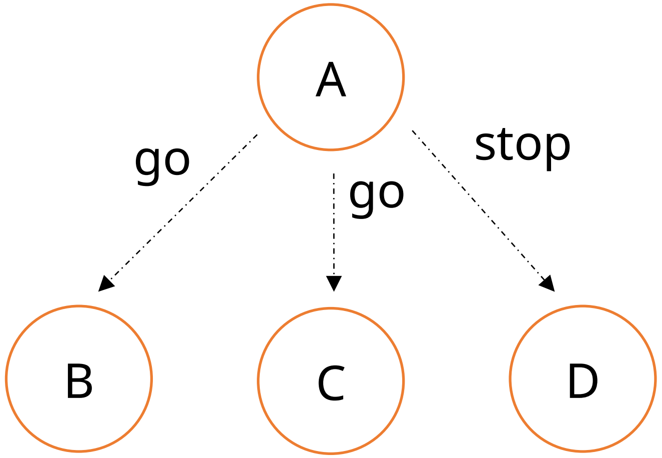
Byzantine failures

- Handling this kind of failure is often called the *Byzantine Generals* problem
  - it is now known that solving this problem for *n* malicious or misbehaving computers requires **3n+1** computers (this is proven to be *necessary* and *sufficient*[1])
  - with this many computers, and only *n* malicious ones, we can solve the problem if *everyone tells everyone else* everything that they know



[1] *Reaching Agreement in the Presence of Faults, Marshall, Shostak, Lamport, 1980*

Byzantine failures



B: A_{go}

C: A_{go}

D: A_{stop}

Each node notes what it has heard from every other node

A is malicious; there are 3n + 1 = 4 nodes in total

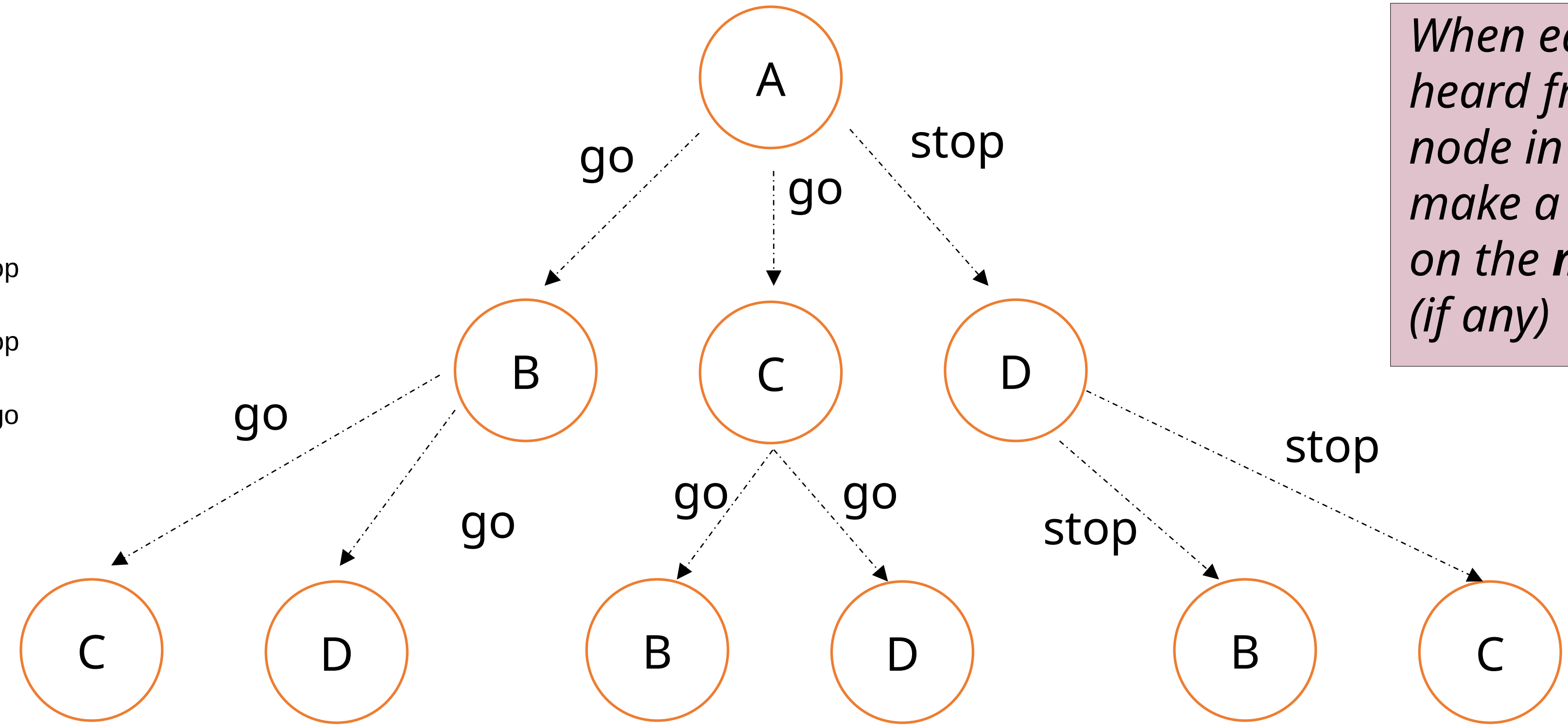
[1] Reaching Agreement in the Presence of Faults, Marshall, Shostak, Lamport, 1980

Byzantine failures

B: A_{go}, C_{go}, D_{stop}

C: A_{go}, B_{go}, D_{stop}

D: A_{stop}, B_{go}, C_{go}



*When each node has heard from every other node in the group, we make a decision based on the **majority view** (if any)*

A is malicious; there are 3n + 1 = 4 nodes in total

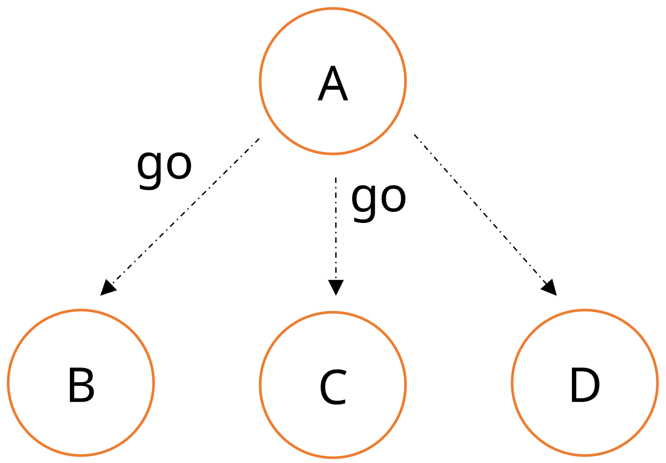
[1] Reaching Agreement in the Presence of Faults, Marshall, Shostak, Lamport, 1980

Byzantine failures

B: A_{go}

C: A_{go}

D: $A_{\text{<non>}}$



What if we hear nothing for a long time?
- we time-out & use a "non" value from that node (we assume good node are responsive)

A is malicious; there are 3n + 1 = 4 nodes in total

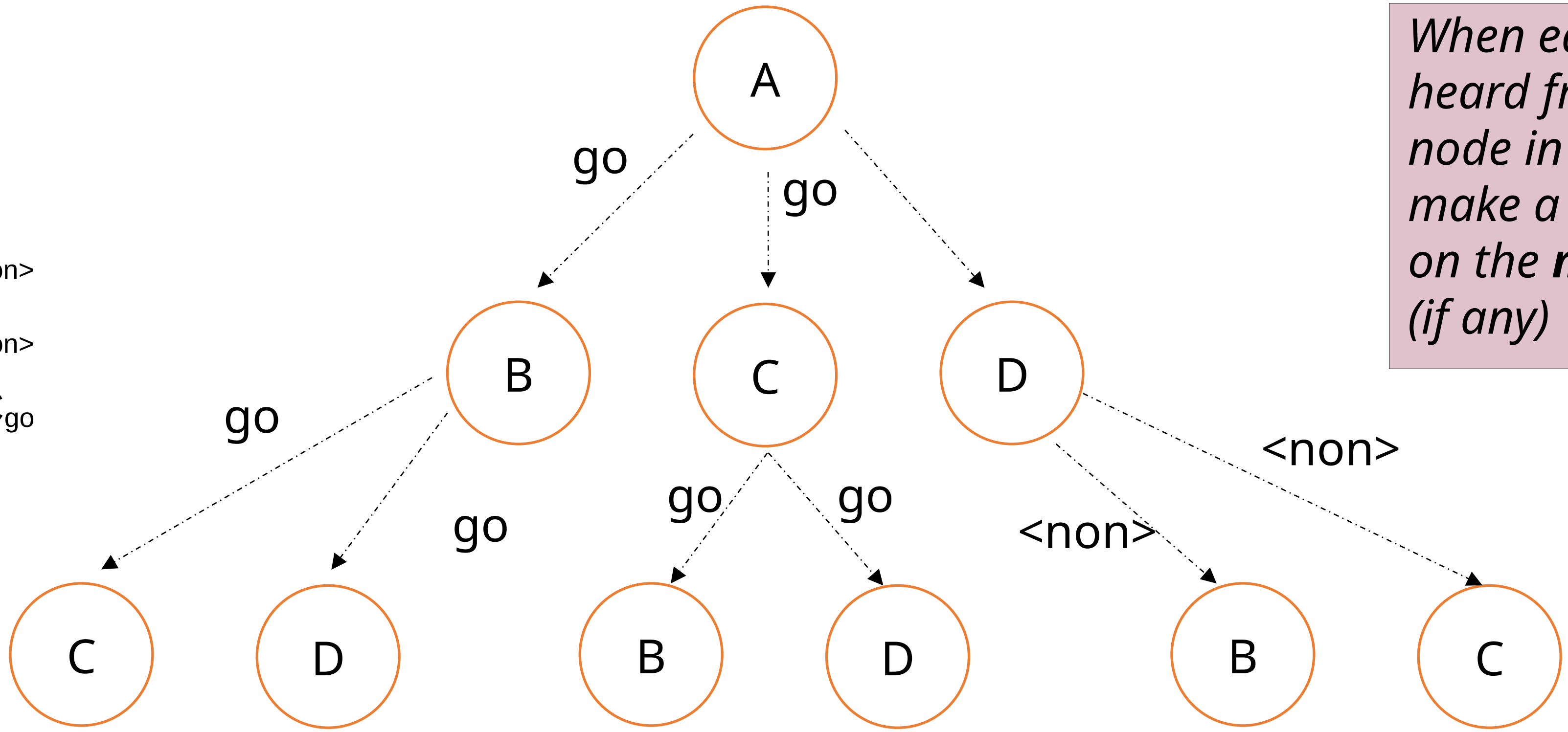
[1] Reaching Agreement in the Presence of Faults, Marshall, Shostak, Lamport, 1980

Byzantine failures

B: A_{go}, C_{go}, $D_{<non>}$

C: A_{go}, B_{go}, $D_{<non>}$

D: $A_{<non>}$, B_{go}, C_{go}

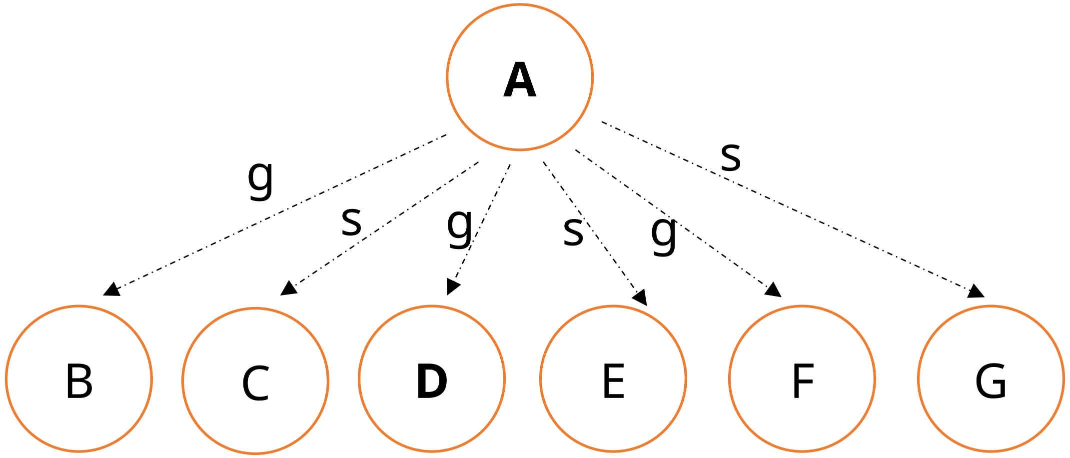


*When each node has heard from every other node in the group, we make a decision based on the **majority view** (if any)*

A is malicious; there are 3n + 1 = 4 nodes in total

[1] Reaching Agreement in the Presence of Faults, Marshall, Shostak, Lamport, 1980

Byzantine failures



B: A_g

C: A_s

D: A_g

E: A_s

F: A_g

G: A_s

A and **D** are malicious; there are 3n + 1 = 7 nodes in total

[1] Reaching Agreement in the Presence of Faults, Marshall, Shostak, Lamport, 1980

Byzantine failures

B: $A_g, C_s, D_g, E_s, F_g, G_s$

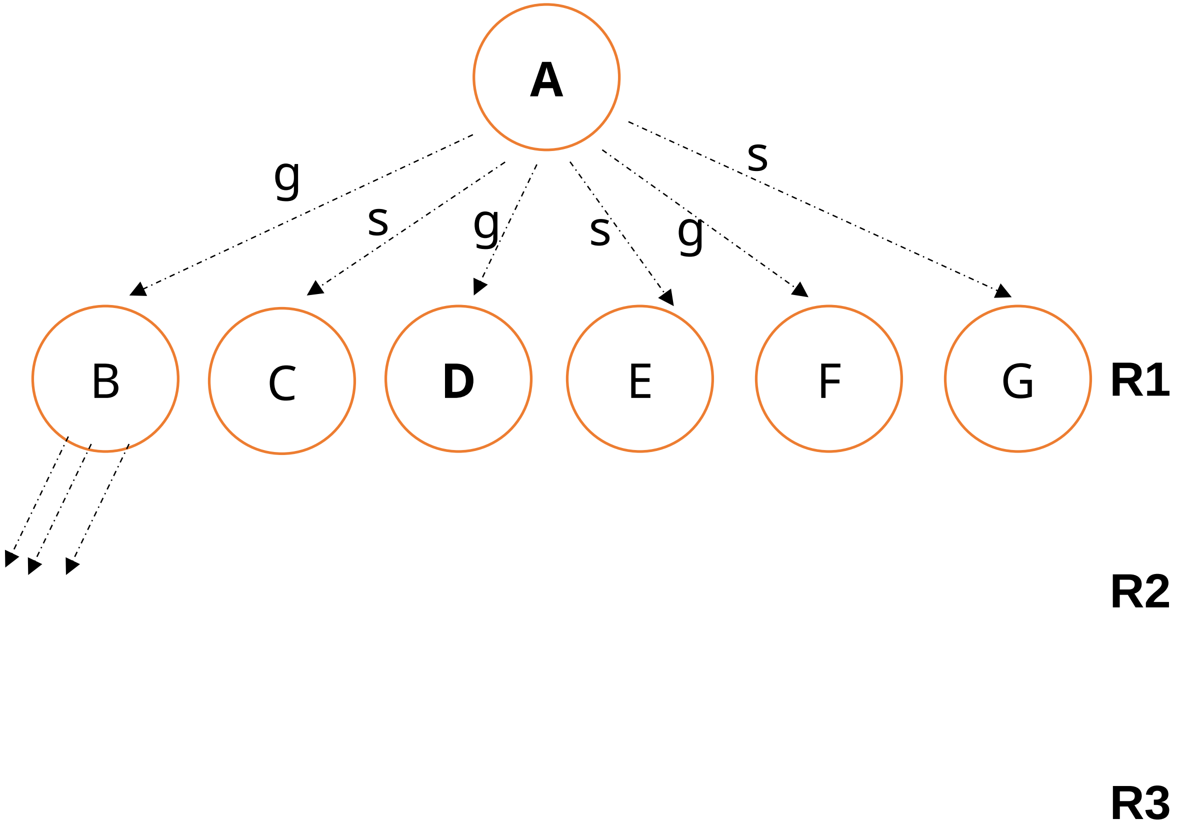
C: $A_s, B_g, D_s, E_s, F_g, G_s$

D: $A_g, B_g, C_s, E_s, F_g, G_s$

E: $A_s, B_g, C_s, D_g, F_g, G_s$

F: $A_g, B_g, C_s, D_s, E_s, G_s$

G: $A_s, B_g, C_s, D_g, E_s, F_g$



Imagine node D sends alternate g/s to every other node; now we have a problem! (our majority at this point is conflicting)

*We use **n+1** rounds of communication to share everything that everyone knows, allowing us to isolate bad actors*

A and **D** are malicious; there are 3n + 1 = 7 nodes in total

[1] Reaching Agreement in the Presence of Faults, Marshall, Shostak, Lamport, 1980

Byzantine failures

B: A_g, C_s, D_g, E_s, F_g, G_s

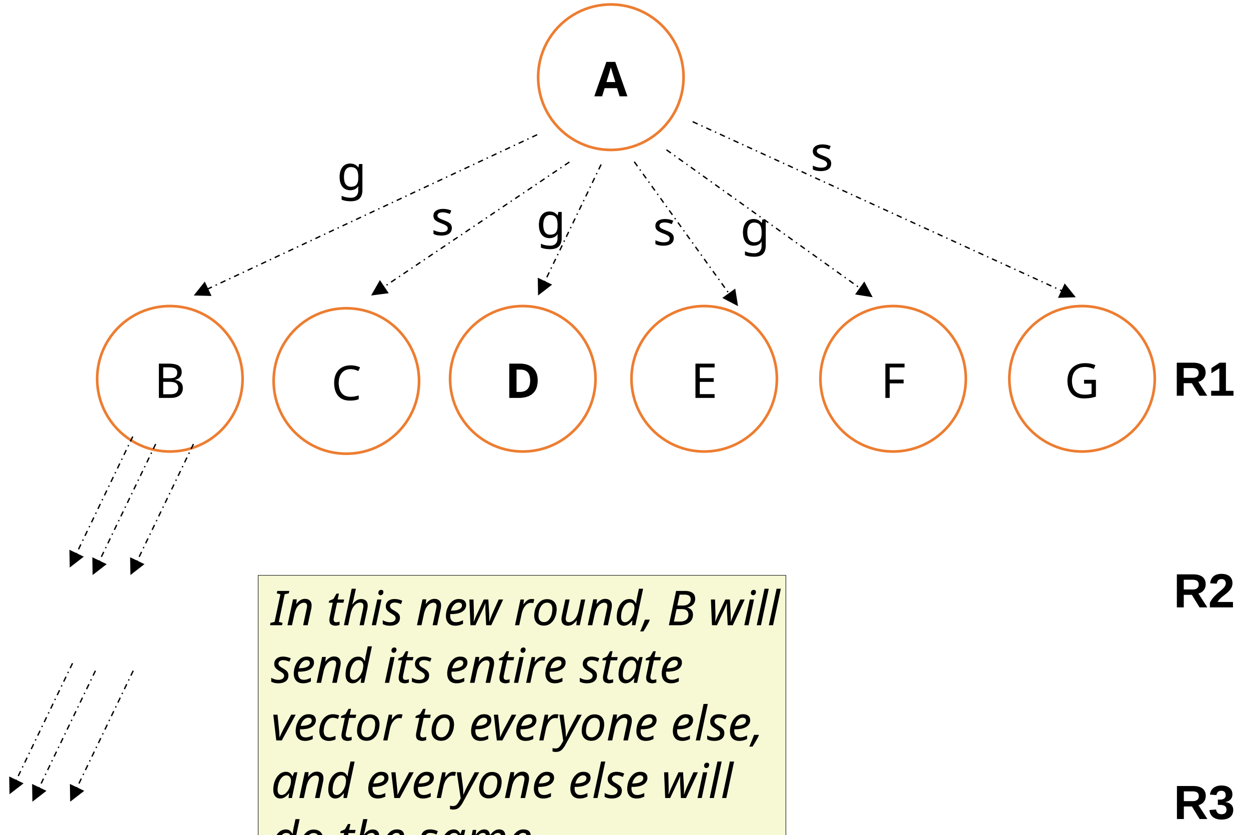
C: A_s, B_g, D_s, E_s, F_g, G_s

D: A_g, B_g, C_s, E_s, F_g, G_s

E: A_s, B_g, C_s, D_g, F_g, G_s

F: A_g, B_g, C_s, D_s, E_s, G_s

G: A_s, B_g, C_s, D_g, E_s, F_g



In this new round, B will send its entire state vector to everyone else, and everyone else will do the same

Imagine node D sends alternate g/s to every other node; now we have a problem! (our majority at this point is conflicting)

*We use **n+1** rounds of communication to share everything that everyone knows, allowing us to isolate bad actors*

A and **D** are malicious; there are 3n + 1 = 7 nodes in total

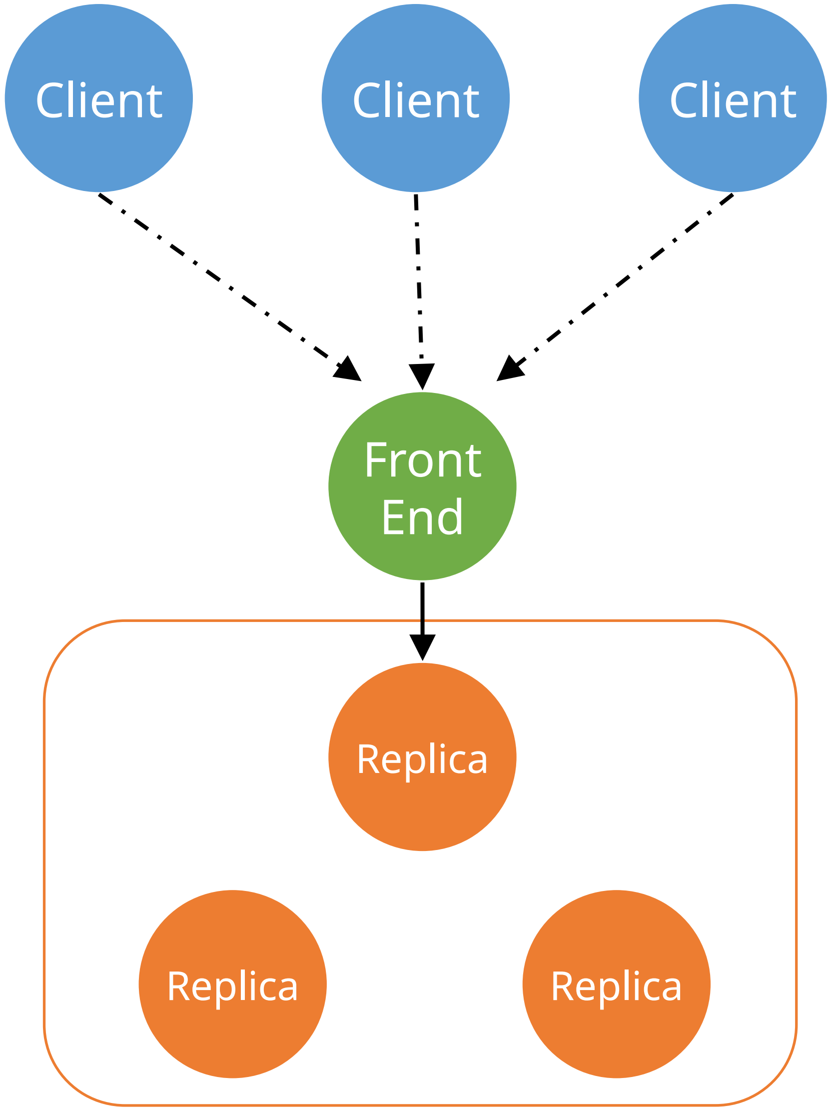
[1] Reaching Agreement in the Presence of Faults, Marshall, Shostak, Lamport, 1980

Replication

- We can use replication for either **fault tolerance** or **performance**
  - For fault tolerance, passive and active replication are the most common styles
  - **Passive replication** uses a primary replica to process all requests, and one or more backup replicas kept up to date by the primary
  - **Active replication** uses a group of replicas while all process every request – combined with a strategy to keep all members of the group up to date

Replication // General

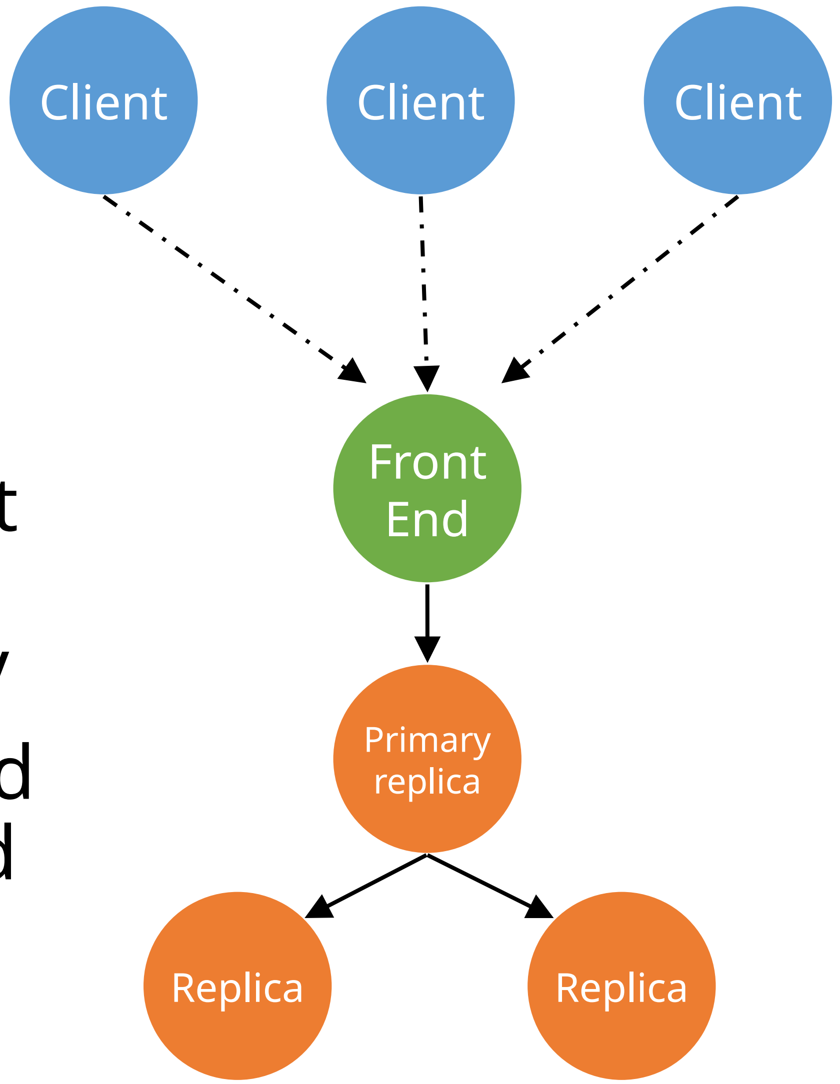
1. **Issue Request:** client request sent to one or more replicas
2. **Coordination:** replicas agree on request ordering and yes/no to execute
3. **Execution:** replicas perform request (assuming agreement to do so)
4. **Agreement:** replicas reach consensus on request outcome
5. **Response:** replica(s) reply to client request



Passive Replication

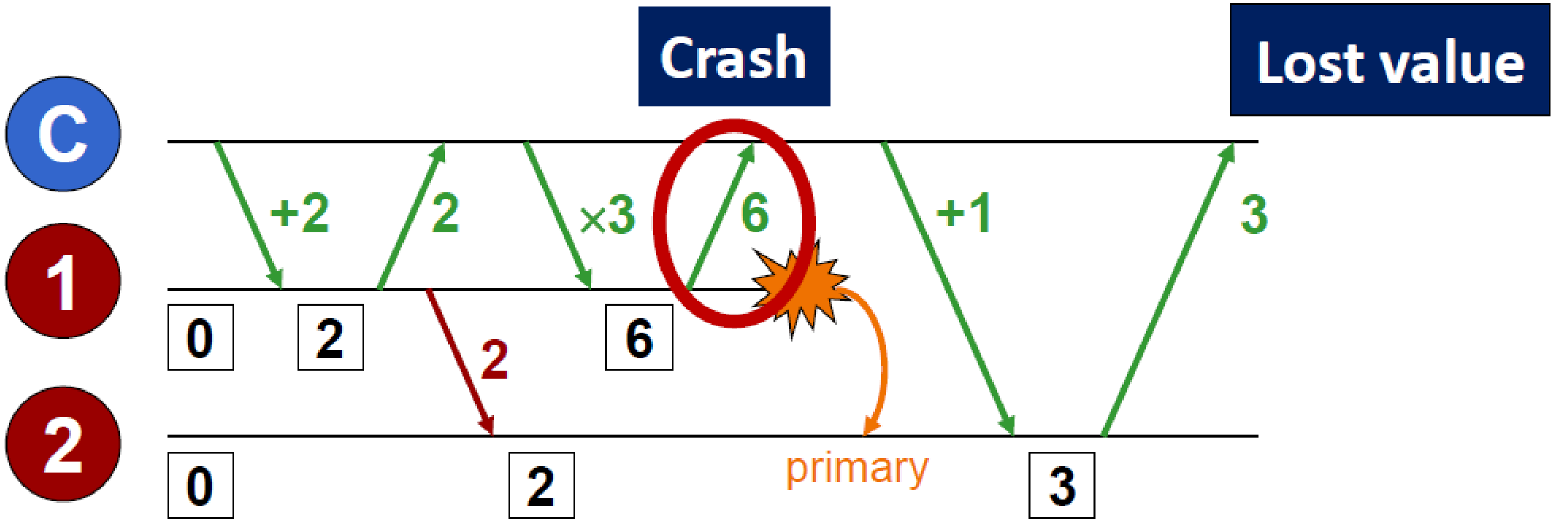
- We use a "front end" node as the contact point for clients
- If the primary replica fails, the front end will direct requests to one of the other replicas as a new primary
- The "coordination", "execution", and "agreement" phases are all decided by the primary replica alone

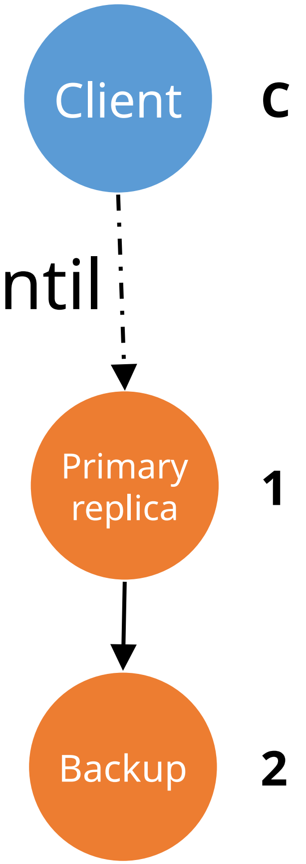
Supports crash failures



Passive Replication

- In passive replication we never show the client a response until we are sure that the state of our replica set is consistent

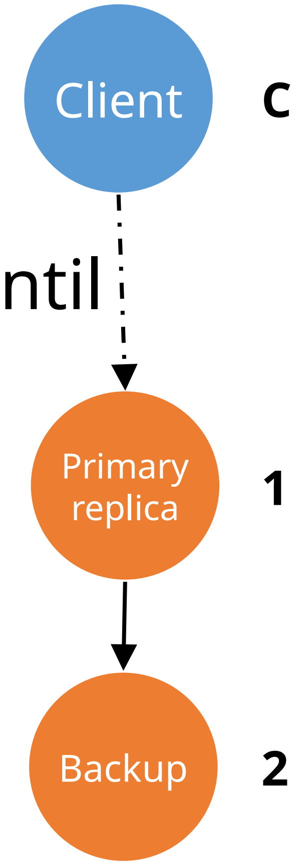


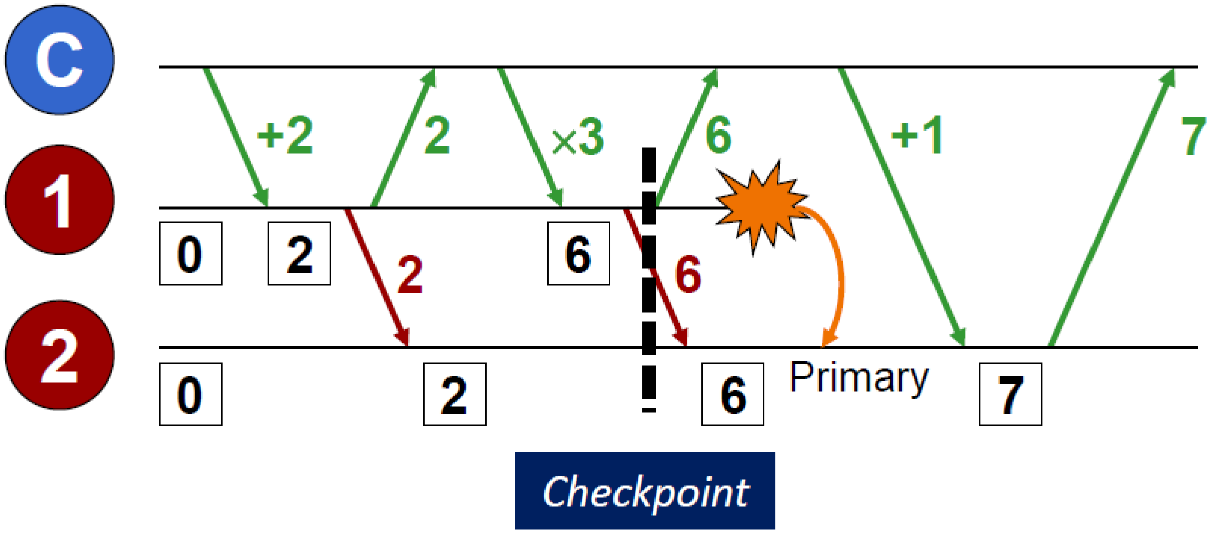


In this example, the client sees a state which is inconsistent with what it was last told

Passive Replication

- In passive replication we never show the client a response until we are sure that the state of our replica set is consistent



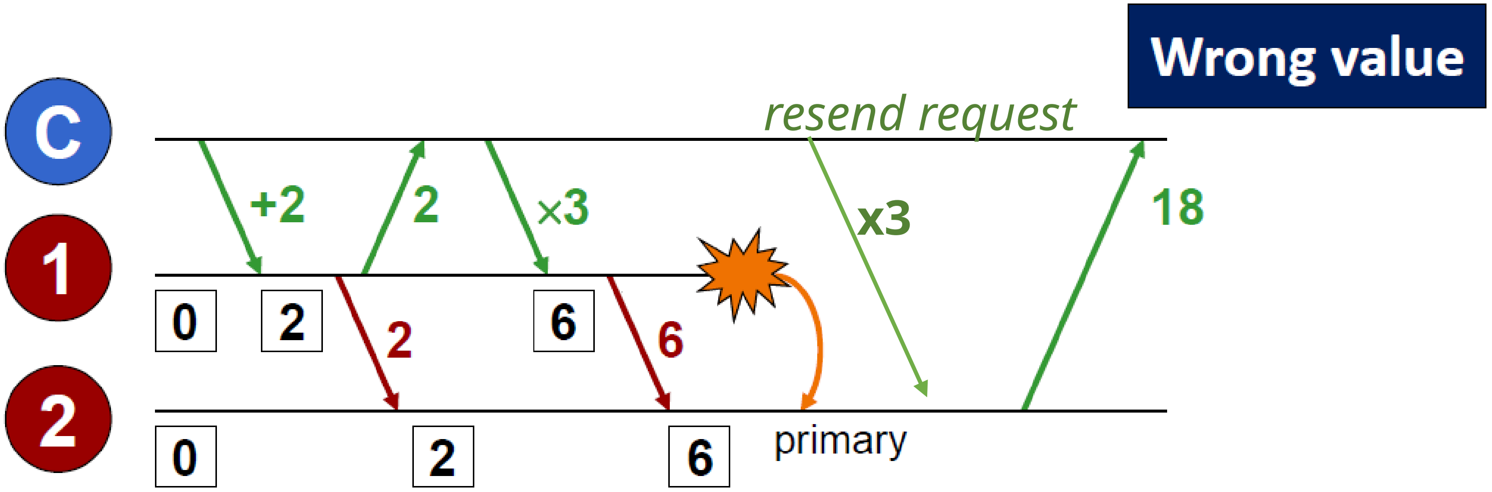


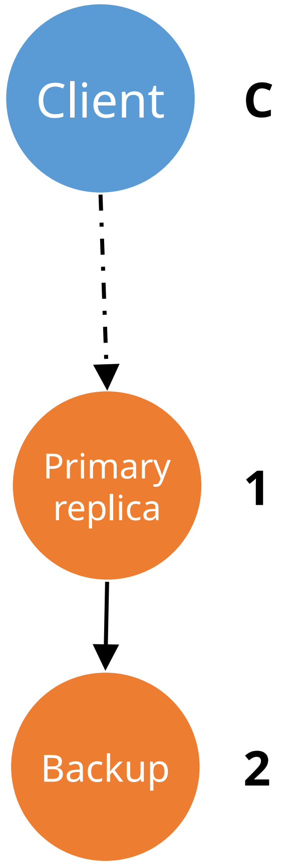
Here we make sure the backup replicas are up to date *before* sending a response; slower, but gives consistent view

Notice we're **not** sending the request to each replica; we're sending **state updates** within the replica group

Passive Replication

- What if the primary crashes before responding to the client?

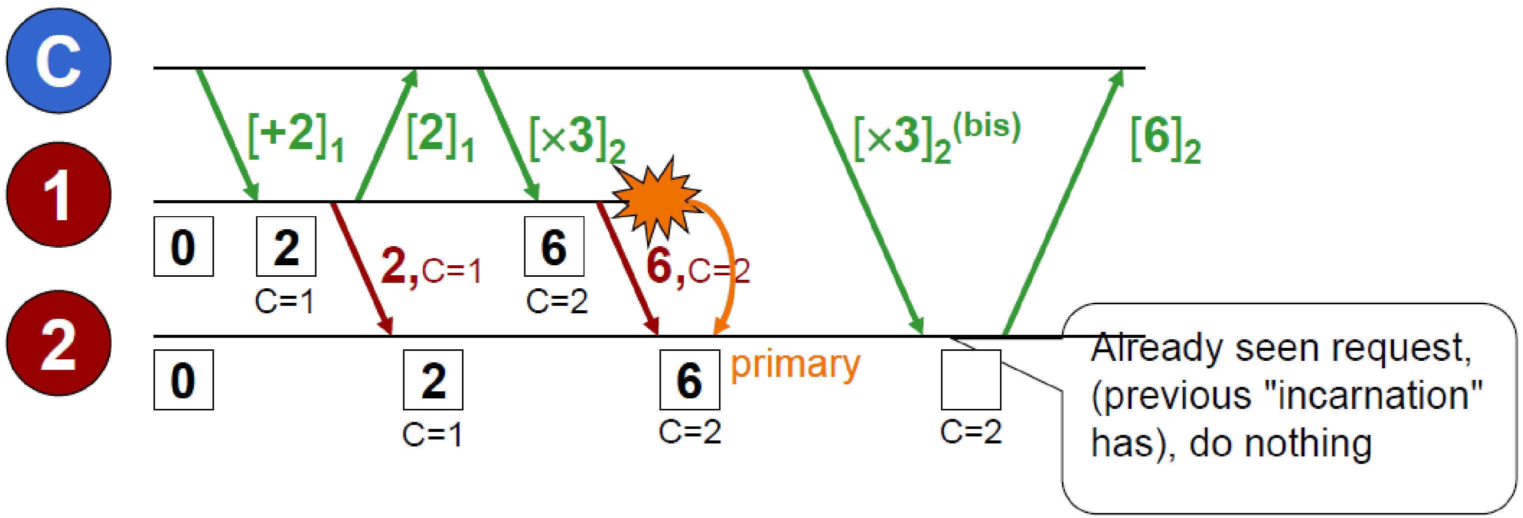


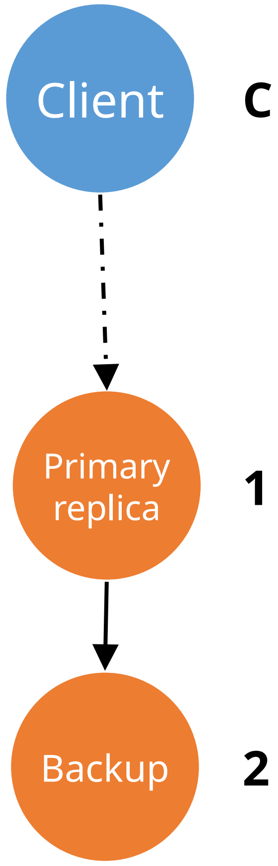


Client may re-send request, believing the first one failed; again this causes an inconsistent client/server view

Passive Replication

- What if the primary crashes before responding to the client?
  - Using monotonically increasing request IDs lets us check…

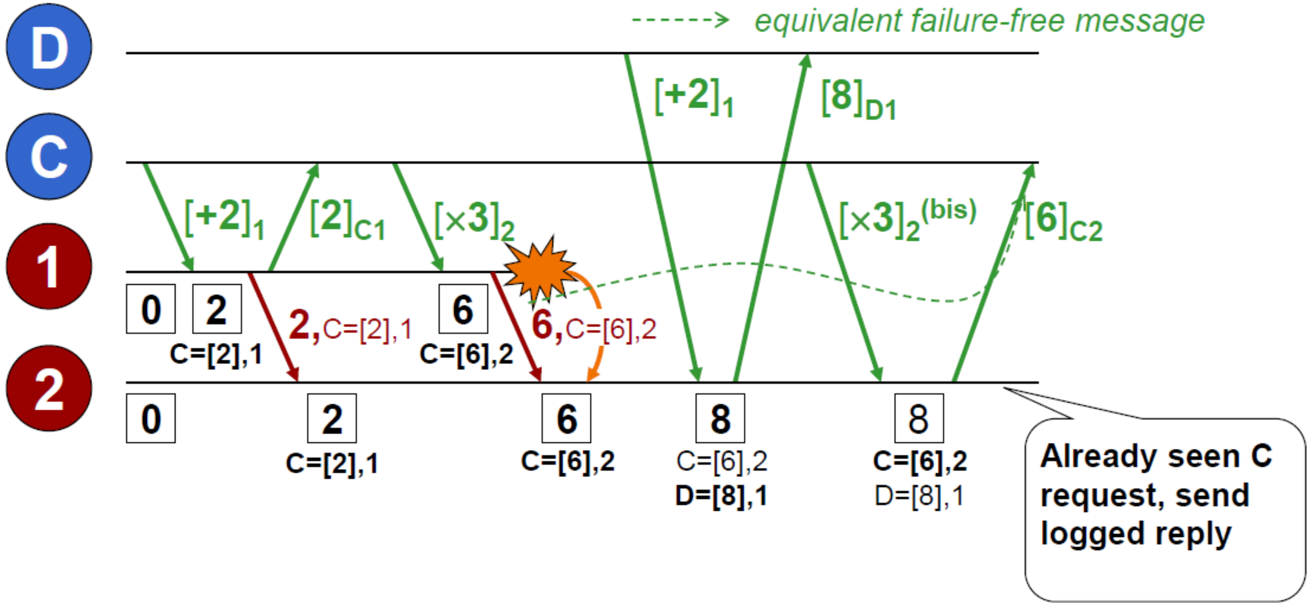


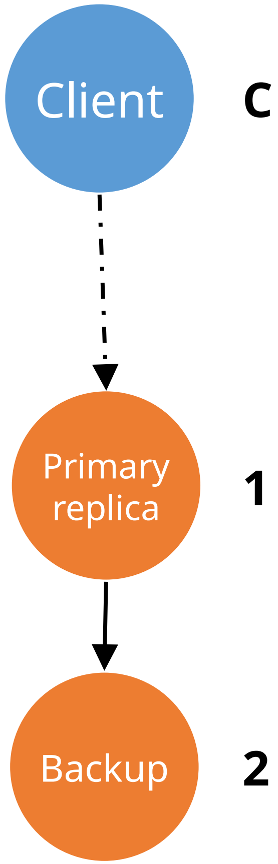


Now the client and server views remain consistent…but what about multiple clients?

Passive Replication

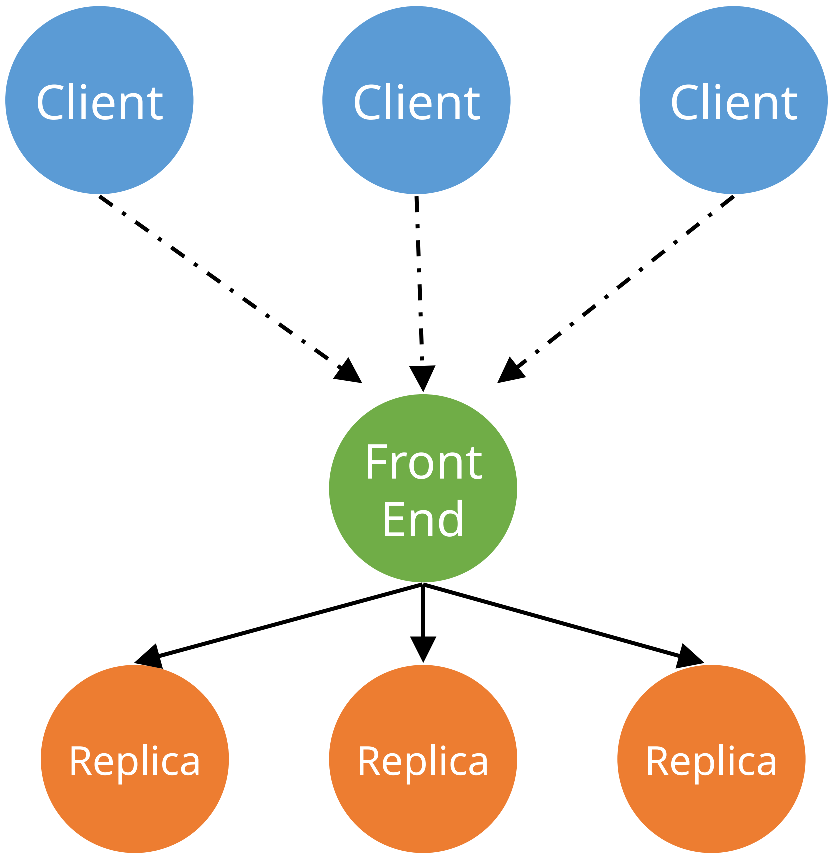
- What if the primary crashes before responding to the client?
  - Using monotonically increasing request IDs lets us check…
  - …and using unique client IDs allows us to differentiate clients





Active Replication

- The front end service sends each client request **to all replicas**
- This requires a reliable group communication service to ensure that each request actually reaches every replica *and in correct order*
- Front end checks for *agreement* between all replicas on response value
- We can use this approach to vote on replica replies to check for errors

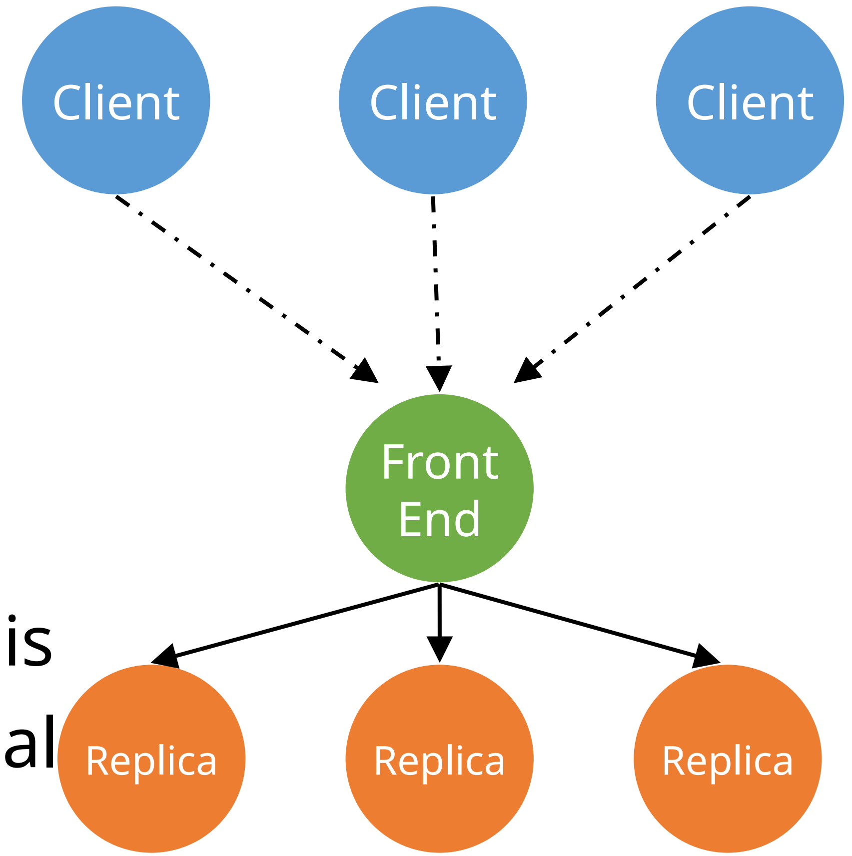


Supports crash failures

Supports Byzantine failures

Active Replication

- With multiple clients, we need to ensure that requests arrive at each replica in the same order
- We use a *total ordering* group communication scheme to achieve this
- All replicas should produce an identical response to reach request; if not, something has gone wrong and can take a majority vote



Supports crash failures

Supports Byzantine failures

Replication comparison

	Passive	Active
Communication overhead	**Low**	**High**
Processing overhead	**Low**	**High**
Recovery overhead	**High**	**Low**
Fault model	**Crash fault only**	**Byzantine faults**

Less expensive
Less complex

More expensive
More complex

Summary

- We've covered an introduction to fault tolerance concepts, from fault propagation to failure detection

- Discussed Byzantine fault detection in detail, and two different replication schemes used for fault tolerance

Further reading

- Section 7.5 (Distributed Commit) and 7.6 (Recovery) of Tanenbaum & van Steen; Sections 16 & 17 of Coulouris & al

- Chapter 7. Fault Tolerance of Tanenbaum & van Steen; Chapter 18 Coulouris & et. al