6.1800 Spring 2025

Lecture #20: Replicated State Machines

high availability + single-copy consistency

Katrina LaCurts | lacurts@mit.edu | 6.1800 2025

6.1800 in the news

It Took a Century to Find This Colossal Squid

An expedition spotted a baby of the species in the South Sandwich Islands. This cephalopod can grow to more than 20 feet and has proved elusive in its deep-sea environs.

The footage was taken by a remotely operated submersible called SuBastian, which the Schmidt Ocean Institute uses to explore the deep sea. This particular dive was a partnership with the Nippon Foundation-Nekton Ocean Census', an initiative to discover unknown species. The submersible stopped for a few minutes on descent to film the small, transparent cephalopod.

https://www.nytimes.com/2025/04/15/science/squid-colossal-deep-sea-exploration.html

6.1800 in the news

The footage from the 4K camera is transmitted through six dedicated fiber-optic cables with lowest glass-to-glass latency, enabling fast transmission and enhancing ROV manipulation and response. Once transmitted, the footage is recorded and stored on RV *Falkor (too)*'s servers. On board *Falkor (too)*, a multimedia technician curates the

https://sea-technology.com/rov-subastian

6.1800 in the news

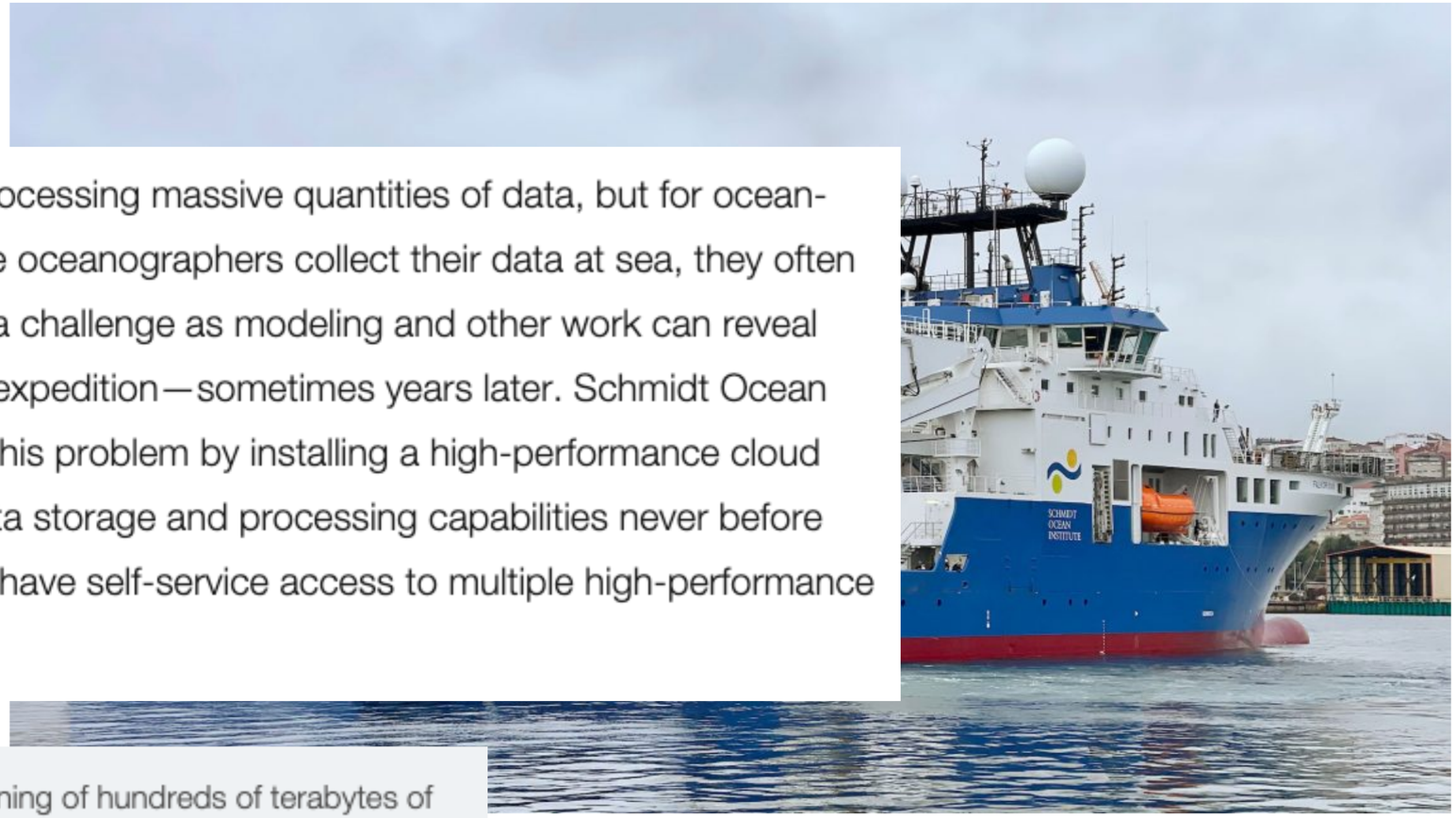
VSATs access satellites in geosynchronous orbit to relay data – this means the satellite orbits Earth at the same speed as the planet is turning, enabling the satellite to stay in place over a single location. Since a ship at sea shifts with the ocean's movement, the antenna needs to be stabilized. Using a combination of GPS sensors and gyroscopes, a motorized system controls the azimuth, elevation, and skew of the antenna, so that it will be constantly pointing at the satellite it uses to transmit and receive signals.

To distribute bandwidth, Schmidt Ocean Institute has set up over 15 Virtual Local Area Networks (VLAN) for different devices and users to access. A VLAN is a system that is partitioned and isolated in a computer network at the data link layer. This means there are different segments dedicated to different missions, each with allocations and protocols to interpret supply/demand of bandwidth. With a limited amount of bandwidth, management is incredibly important. The system is smart as well, giving priority not just based on which VLAN you are connected to but your credentials and where you are. When a video presentation occurs on board, the network orchestrator detects the location and knows to allocate bandwidth appropriately.

https://schmidtocean.org/education/falkor-too-qa/
https://schmidtocean.org/technology/internet-connectivity/

6.1800 in the news

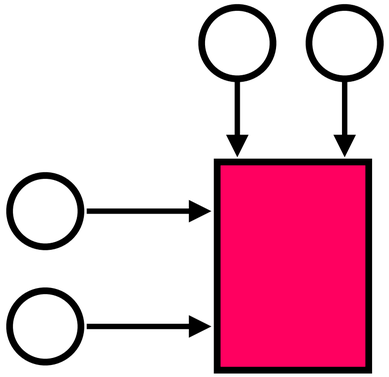
Most modern researchers face hurdles in effectively processing massive quantities of data, but for ocean-going scientists the challenge is especially acute. While oceanographers collect their data at sea, they often can not process it until returning to shore. This poses a challenge as modeling and other work can reveal gaps or trends that can not be explored until the next expedition—sometimes years later. Schmidt Ocean Institute (SOI) has taken a pioneering step to address this problem by installing a high-performance cloud computing system on R/V *Falkor (too)* that enables data storage and processing capabilities never before available to scientists at sea. Scientists on *Falkor (too)* have self-service access to multiple high-performance computer processors and storage.



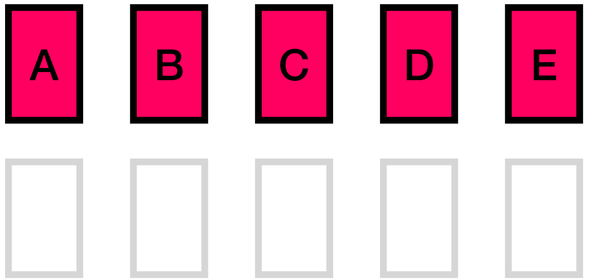
A high-performance computer can provide self-service provisioning of hundreds of terabytes of storage memory, and a hundred or more central processing units, or cores, to multiple on-board science teams. The computer will give scientists on *Falkor (too)* the combined power, speed and memory of 60 or more high end desktops acting in unison. This cluster of interconnected high-performance computers, make *Falkor (too)* the first research vessel with a supercomputing system available to scientists during research cruises.

https://schmidtocean.org/education/falkor-too-qa/
https://schmidtocean.org/technology/high-performance-computing/

our goal is to build **reliable systems from unreliable components**. we want to build systems that serve many clients, store a lot of data, perform well, all while keeping availability high



our lingering problem is that we aren't replicating data across multiple machines



transactions — which provide **atomicity** and **isolation** — make it easier for us to reason about failures

our job in lecture is to understand how a system *implements* these two abstractions.
how do our systems guarantee atomicity? how do they guarantee isolation?

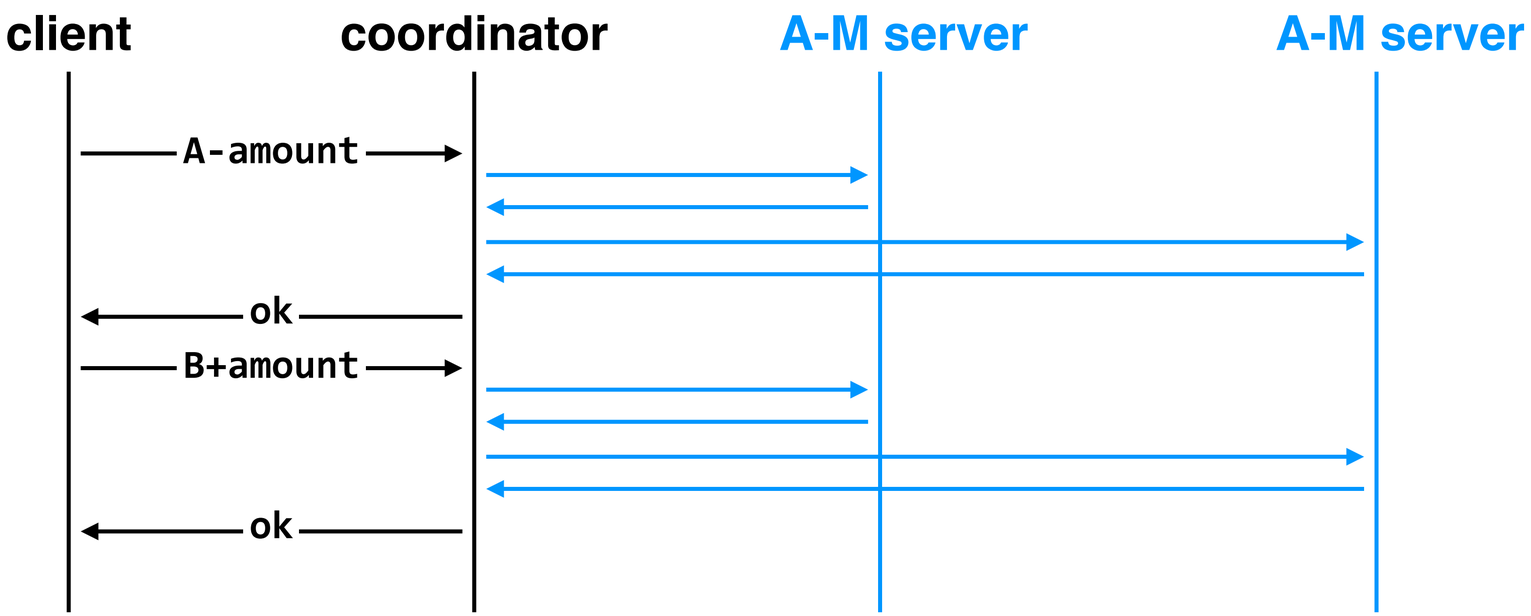
atomicity: provided by **logging**, which gives better performance than shadow copies* at the cost of some added complexity; **two-phase commit** gives us multi-site atomicity

isolation: provided by **two-phase locking**

* shadow copies *are* used in some systems

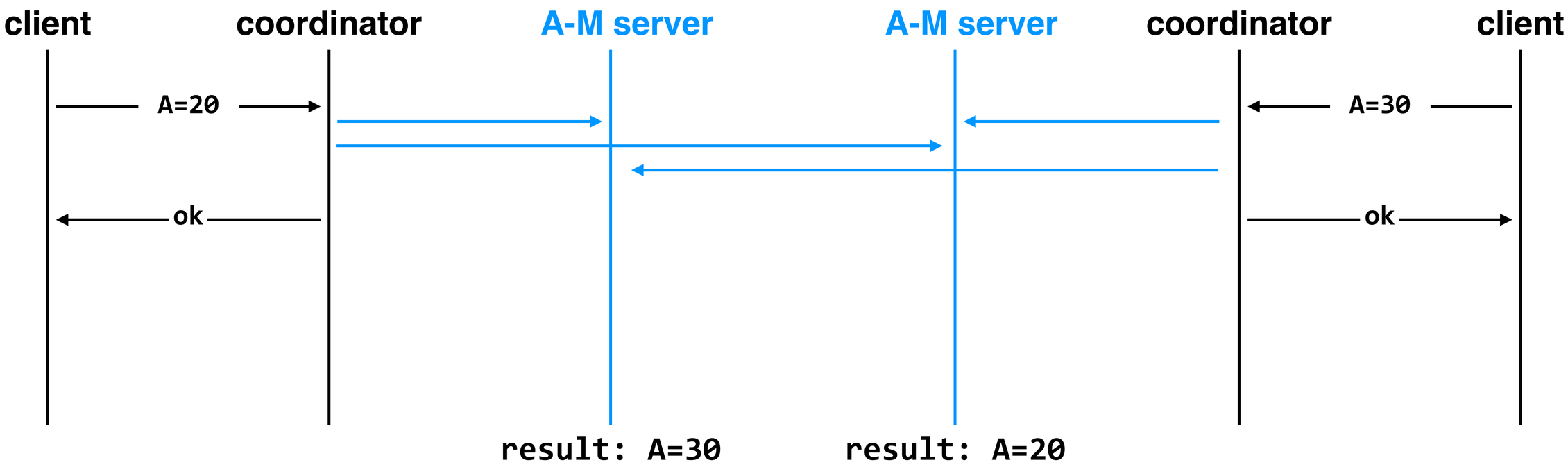
to increase availability, let's try replicating data on two servers

attempt 1: nothing special, just two copies of the data



to increase availability, let's try replicating data on two servers

attempt 1: nothing special, just two copies of the data



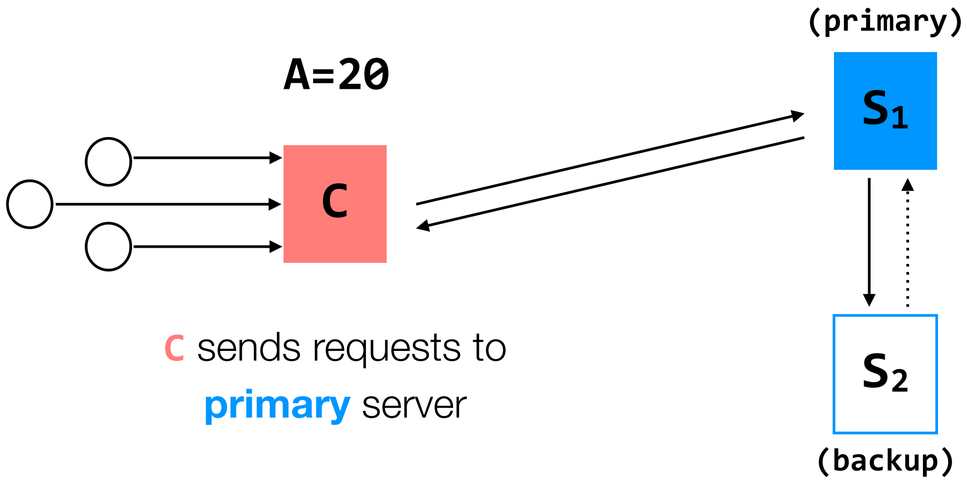
problem: order of messages can cause replicas to become inconsistent

(here, we're imagining a situation where we allow those two writes to be issued concurrently,
to illustrate how message-ordering can lead to inconsistencies)

to increase availability, let's try replicating data on two servers

attempt 2: make one replica the **primary** replica, and have **coordinators** in place to help manage failures

clients communicate only with C, not with replicas



C sends requests to **primary** server

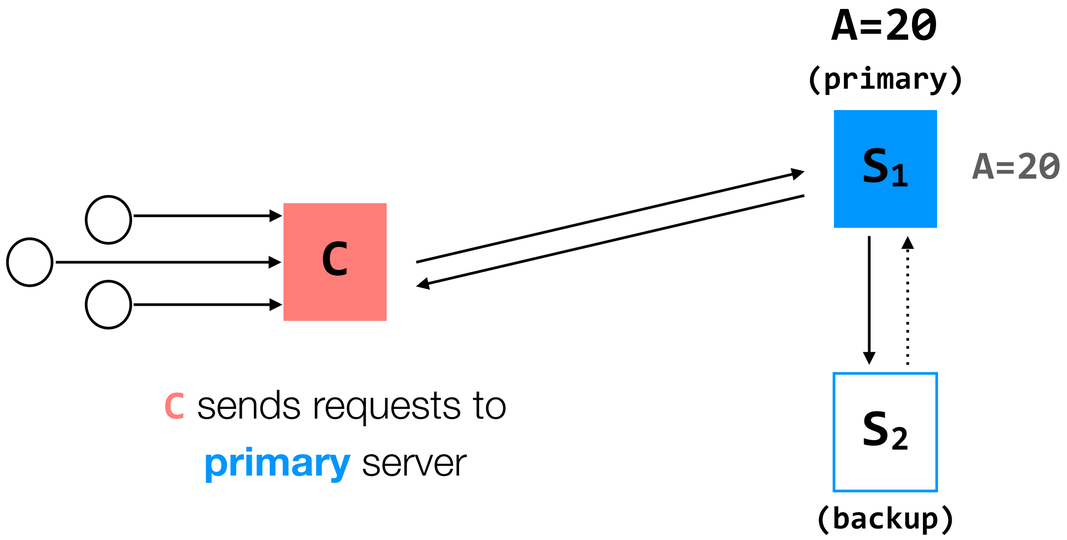
primary chooses order of operations, decides all non-deterministic values

primary ACKs **coordinator** only after it's sure that **backup** has all updates

to increase availability, let's try replicating data on two servers

attempt 2: make one replica the **primary** replica, and have **coordinators** in place to help manage failures

clients communicate only with C, not with replicas



C sends requests to **primary** server

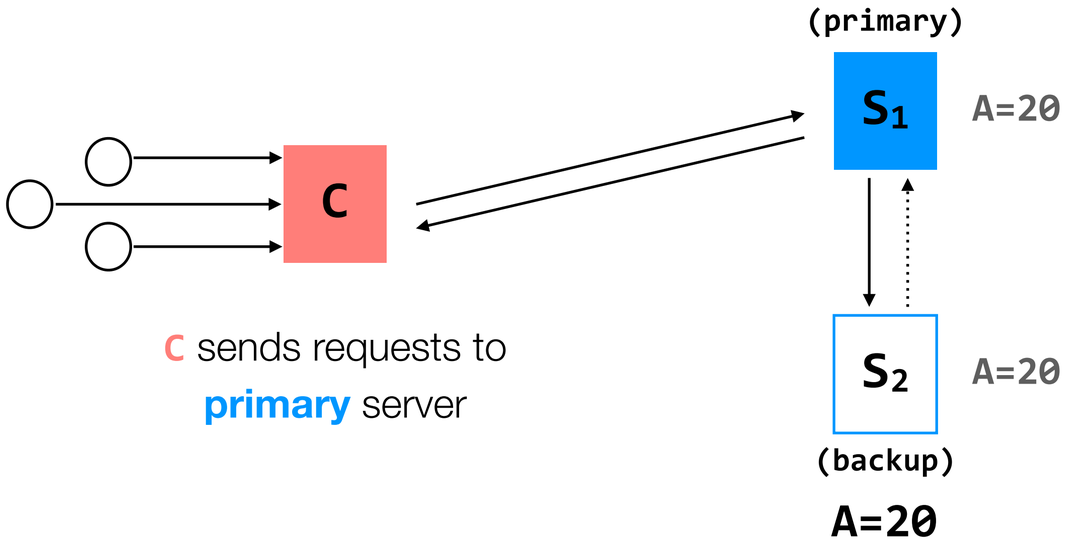
primary chooses order of operations, decides all non-deterministic values

primary ACKs **coordinator** only after it's sure that **backup** has all updates

to increase availability, let's try replicating data on two servers

attempt 2: make one replica the **primary** replica, and have **coordinators** in place to help manage failures

clients communicate only with C, not with replicas



C sends requests to **primary** server

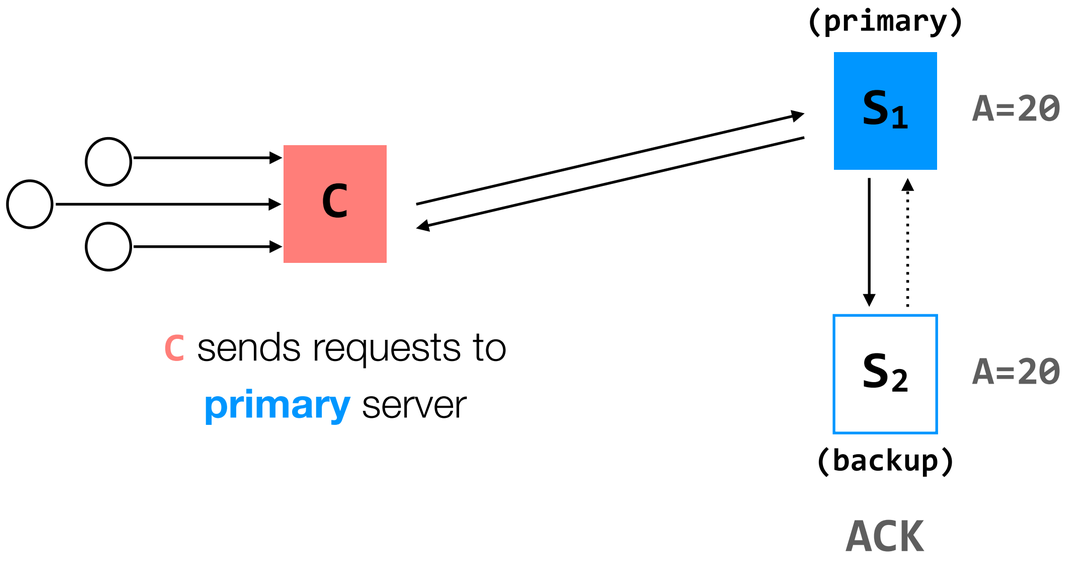
primary chooses order of operations, decides all non-deterministic values

primary ACKs **coordinator** only after it's sure that **backup** has all updates

to increase availability, let's try replicating data on two servers

attempt 2: make one replica the **primary** replica, and have **coordinators** in place to help manage failures

clients communicate only with C, not with replicas



C sends requests to **primary** server

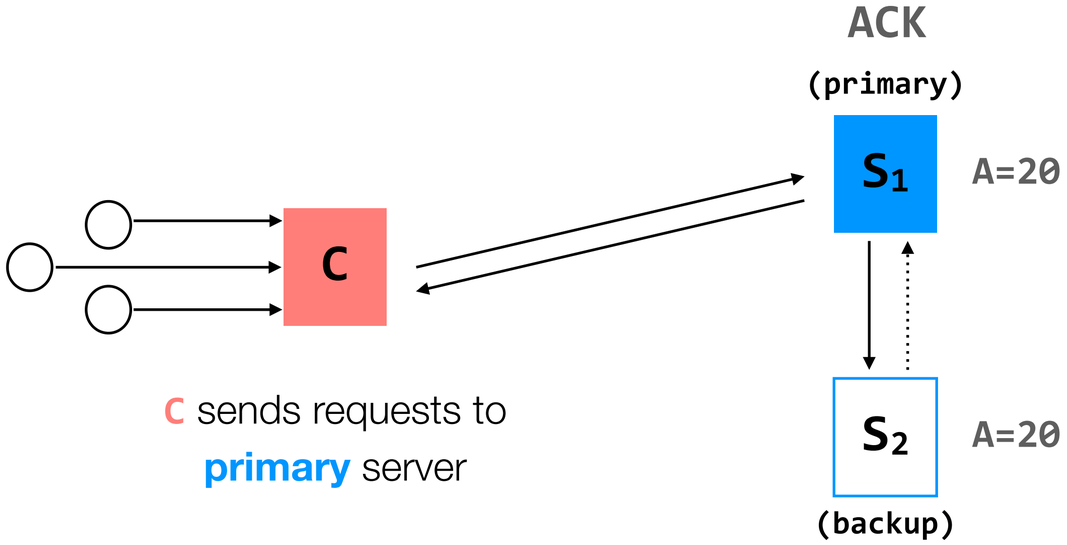
primary chooses order of operations, decides all non-deterministic values

primary ACKs **coordinator** only after it's sure that **backup** has all updates

to increase availability, let's try replicating data on two servers

attempt 2: make one replica the **primary** replica, and have **coordinators** in place to help manage failures

clients communicate only with C, not with replicas



C sends requests to **primary** server

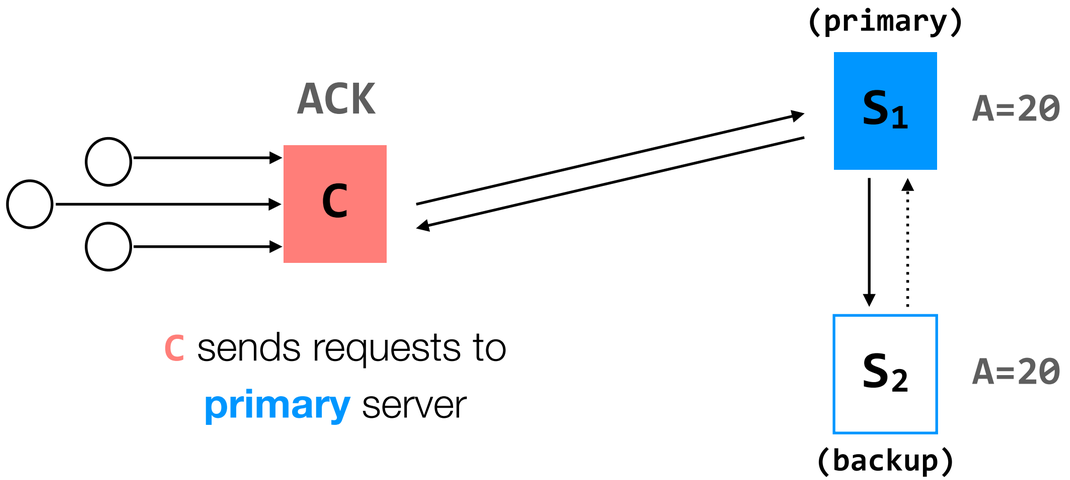
primary chooses order of operations, decides all non-deterministic values

primary ACKs **coordinator** only after it's sure that **backup** has all updates

to increase availability, let's try replicating data on two servers

attempt 2: make one replica the **primary** replica, and have **coordinators** in place to help manage failures

clients communicate only with C, not with replicas



C sends requests to **primary** server

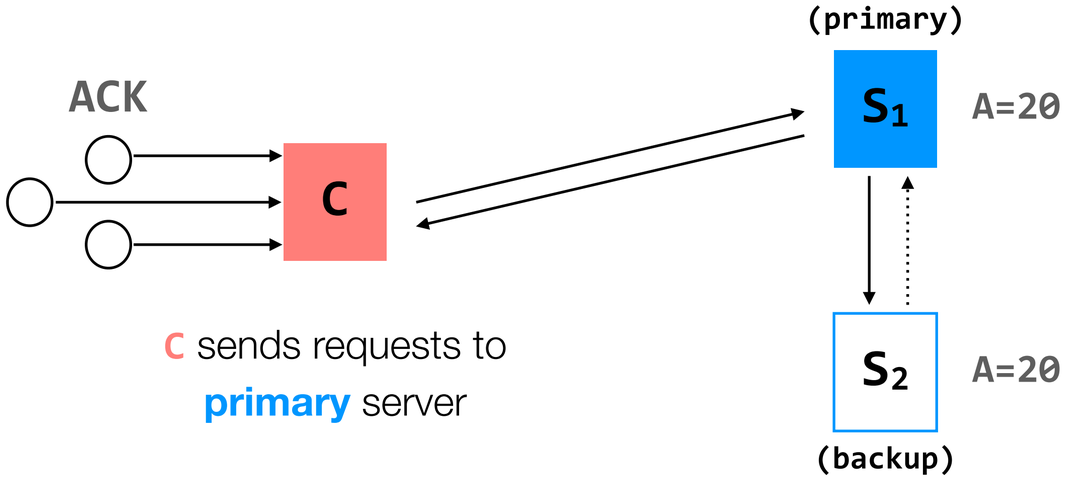
primary chooses order of operations, decides all non-deterministic values

primary ACKs **coordinator** only after it's sure that **backup** has all updates

to increase availability, let's try replicating data on two servers

attempt 2: make one replica the **primary** replica, and have **coordinators** in place to help manage failures

clients communicate only with C, not with replicas



C sends requests to **primary** server

primary chooses order of operations, decides all non-deterministic values

primary ACKs **coordinator** only after it's sure that **backup** has all updates

all coordinators send requests to the **primary** server, which avoids the problem we saw in our first attempt

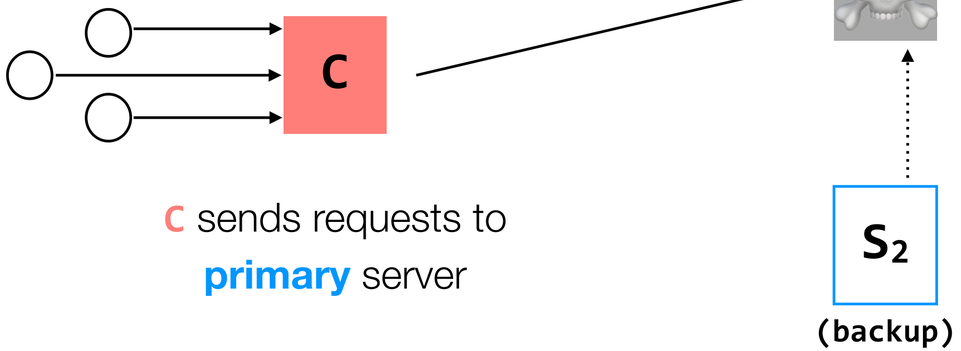
to increase availability, let's try replicating data on two servers

attempt 2: make one replica the **primary** replica, and have **coordinators** in place to help manage failures

if **primary** fails, C switches to **backup**

C knows how to contact backup server

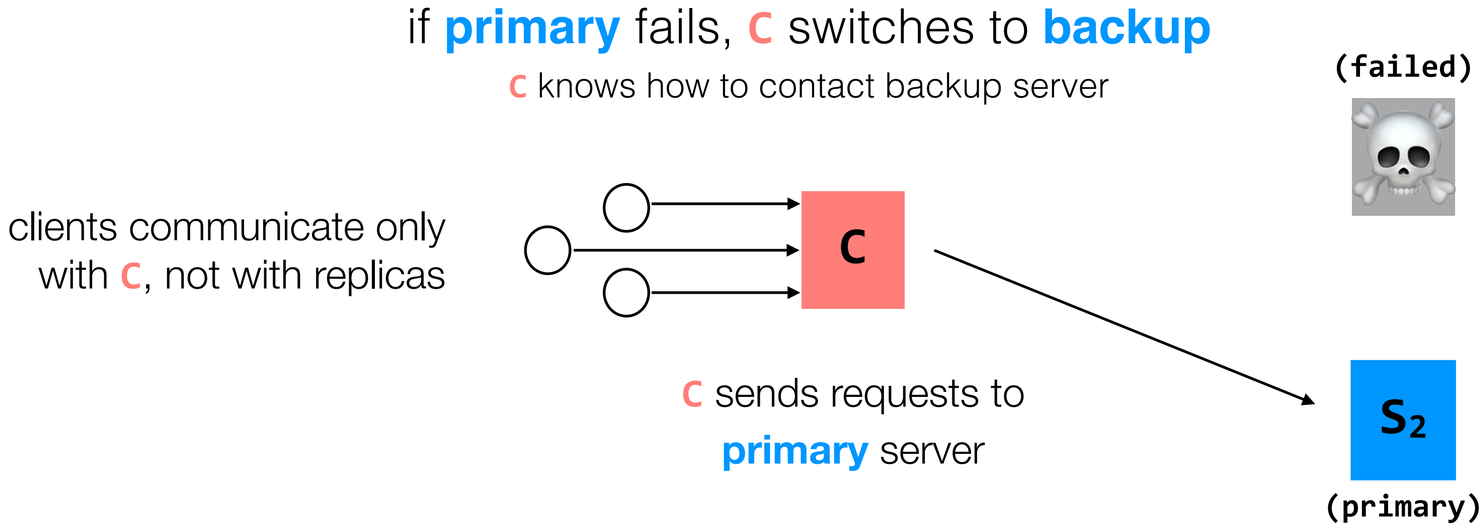
clients communicate only with C, not with replicas



C sends requests to **primary** server

to increase availability, let's try replicating data on two servers

attempt 2: make one replica the **primary** replica, and have **coordinators** in place to help manage failures

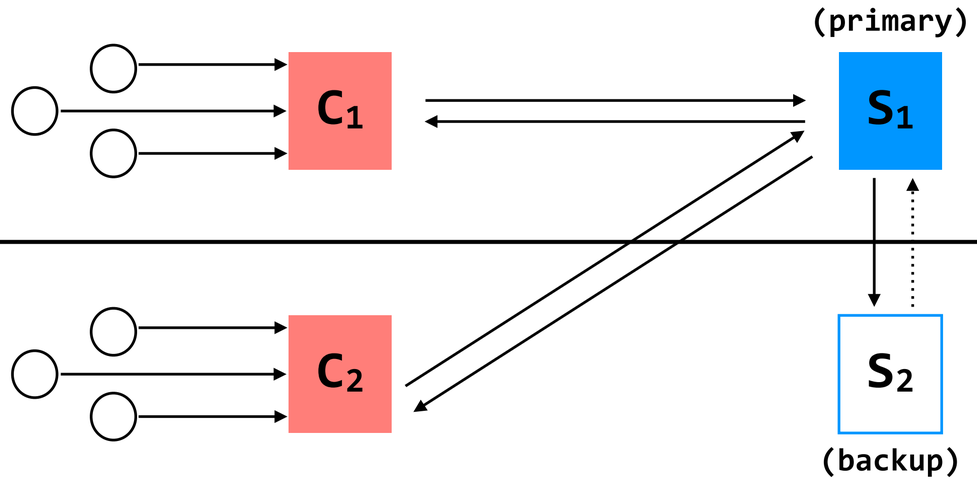


ideally, S_1 recovers at some point, or we get some other replacement machine, and we go back to having both a primary and a backup. but for the purposes of this example, we're just concerned about correctly switching over to the backup server

to increase availability, let's try replicating data on two servers

attempt 2: make one replica the **primary** replica, and have **coordinators** in place to help manage failures

for a single transaction, a client would communicate with a single `coordinator`



a network partition means that machines on the same side of this line can communicate with each other, but not with machines on the other side

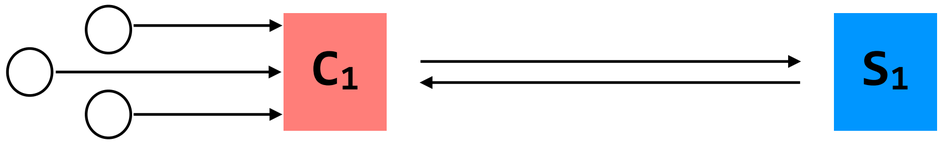
suppose that all machines remain up, but that there is a **network partition** that effectively splits this network in half

to increase availability, let's try replicating data on two servers

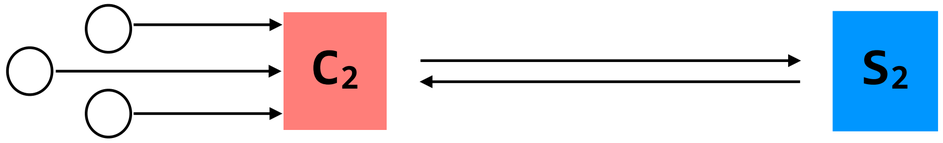
attempt 2: make one replica the **primary** replica, and have **coordinators** in place to help manage failures

for a single transaction, a client would communicate with a single `coordinator`

C_1 keeps using S_1 as primary, with no backup



a network partition means that machines on the same side of this line can communicate with each other, but not with machines on the other side

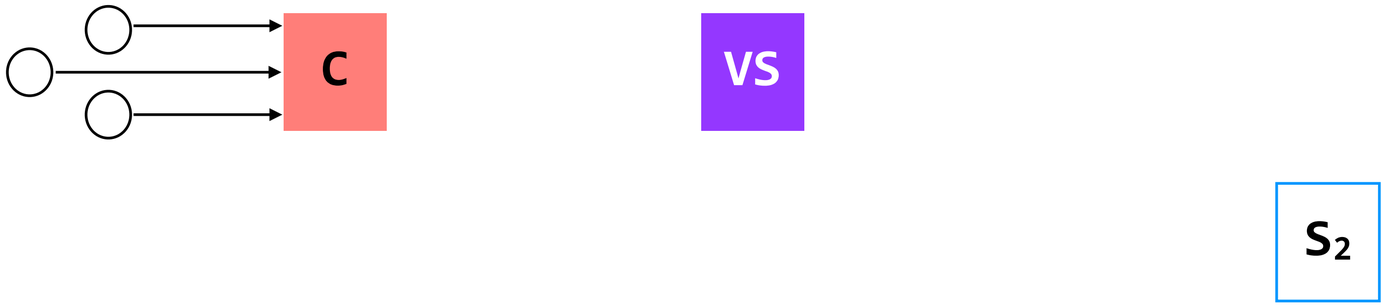


C_2 begins using S_2 as primary, with no backup

because two different replicas both think that they are the **primary** replica, our data can become **inconsistent**

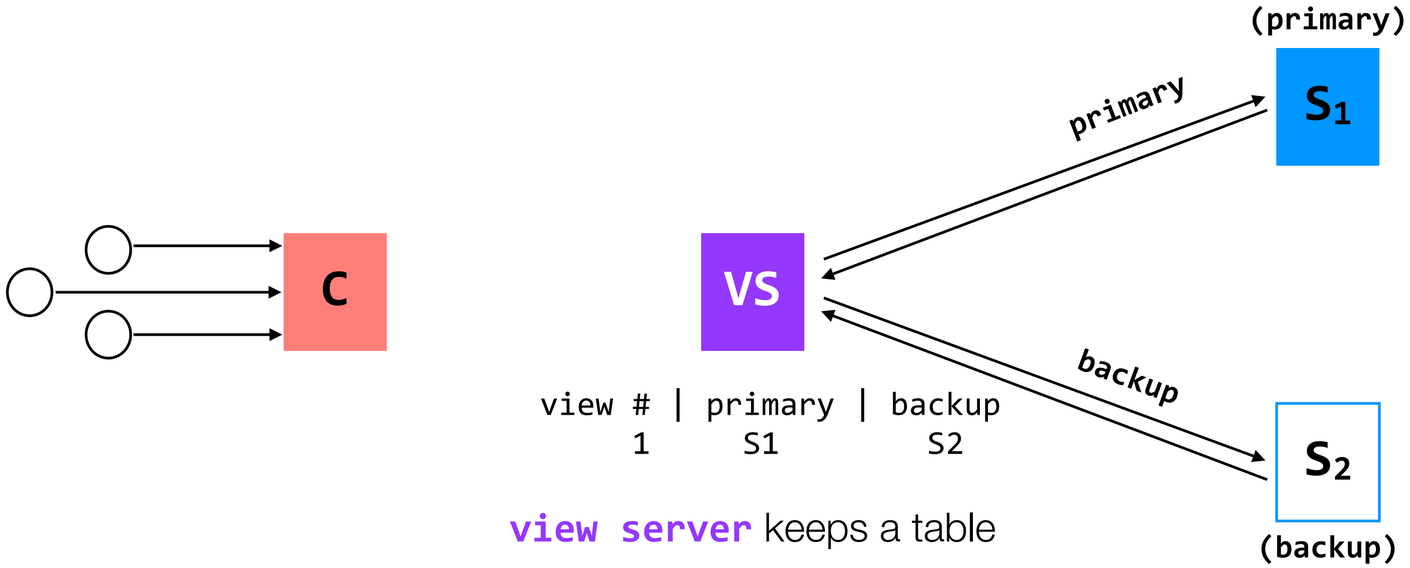
to increase availability, let's try replicating data on two servers

attempt 3: use a **view server** to determine which replica is primary, in hopes that we can deal with network partitions



to increase availability, let's try replicating data on two servers

attempt 3: use a **view server** to determine which replica is primary, in hopes that we can deal with network partitions



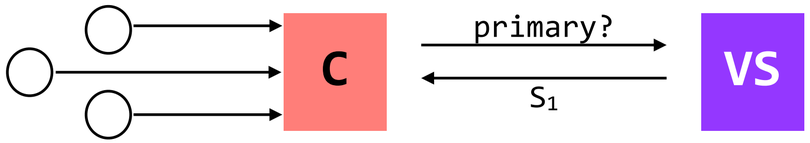
`view server` keeps a table that maintains a sequence of views

`view server` alerts `primary`/`backups` about their roles

to increase availability, let's try replicating data on two servers

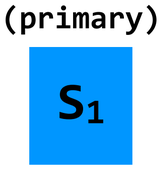
attempt 3: use a **view server** to determine which replica is primary, in hopes that we can deal with network partitions

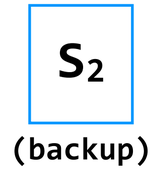
coordinators make requests to **view server** to find out which replica is **primary**




```
view # | primary | backup
     1 |     S1  |     S2
```

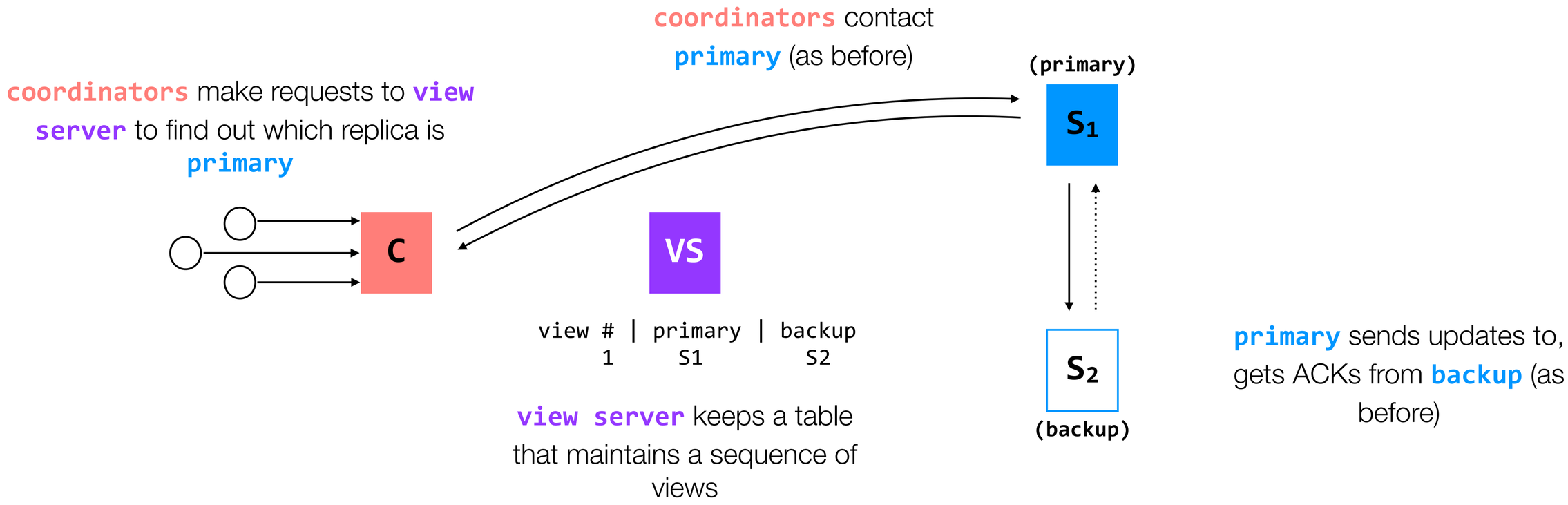
view server keeps a table that maintains a sequence of views





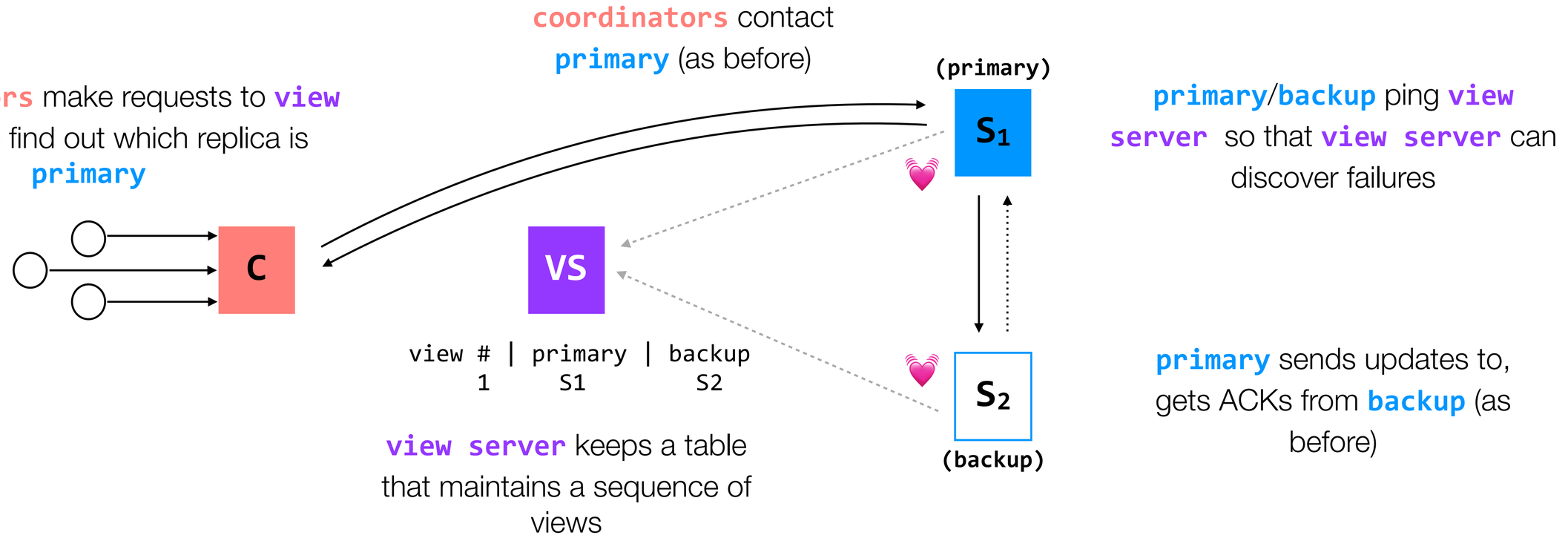
to increase availability, let's try replicating data on two servers

attempt 3: use a **view server** to determine which replica is primary, in hopes that we can deal with network partitions



to increase availability, let's try replicating data on two servers

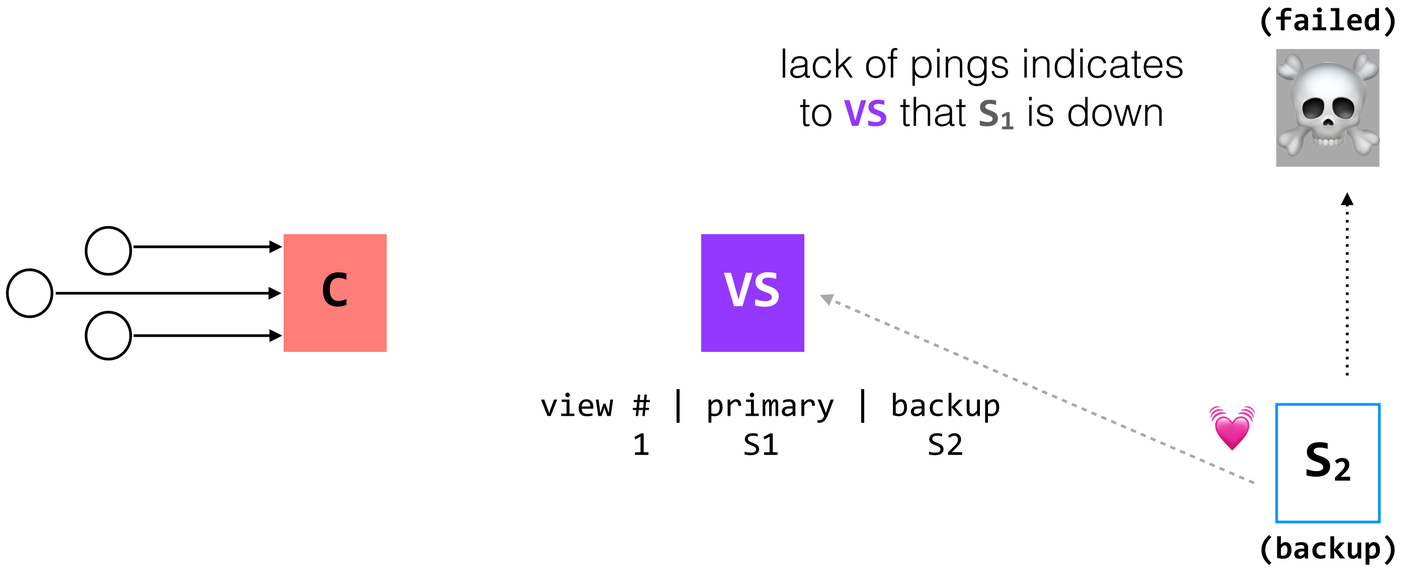
attempt 3: use a **view server** to determine which replica is primary, in hopes that we can deal with network partitions



question: in our set-up, there is **one view server** for this entire system, whereas there can be multiple **coordinators**. why might having a single view server help us when failures (such as the examples you've already seen) occur?

to increase availability, let's try replicating data on two servers

attempt 3: use a **view server** to determine which replica is primary, in hopes that we can deal with network partitions



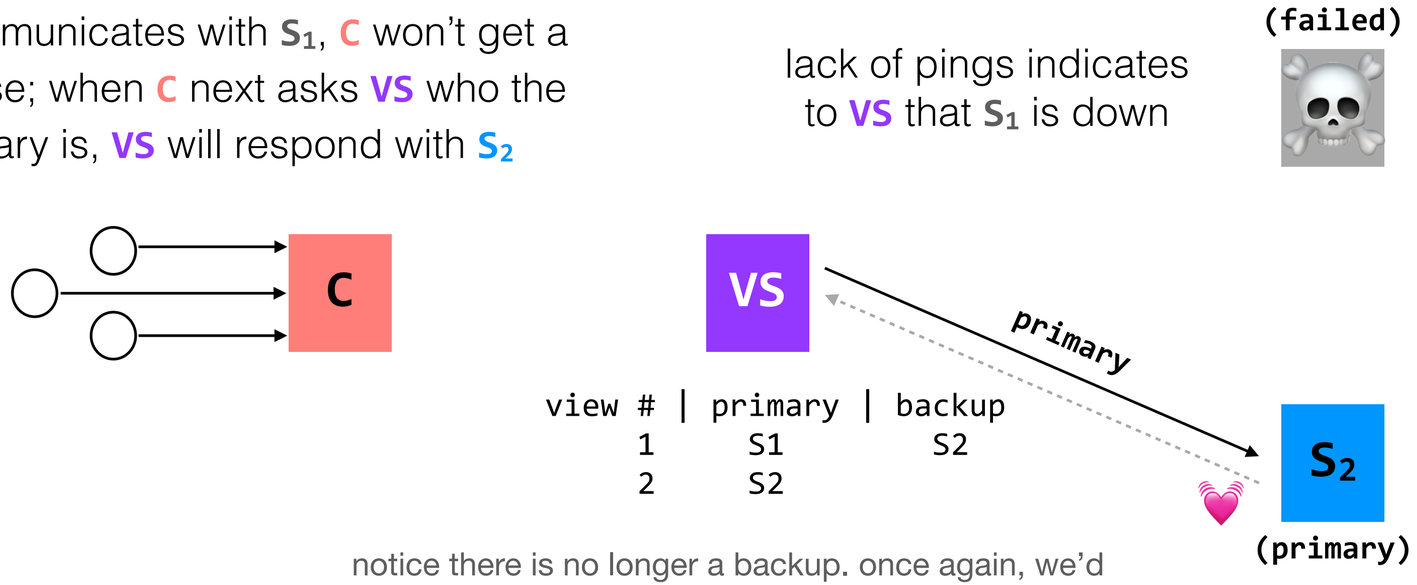
what happens if the primary replica fails?

to increase availability, let's try replicating data on two servers

attempt 3: use a **view server** to determine which replica is primary, in hopes that we can deal with network partitions

if C communicates with S_1, C won't get a response; when C next asks VS who the primary is, VS will respond with S_2

lack of pings indicates to VS that S_1 is down

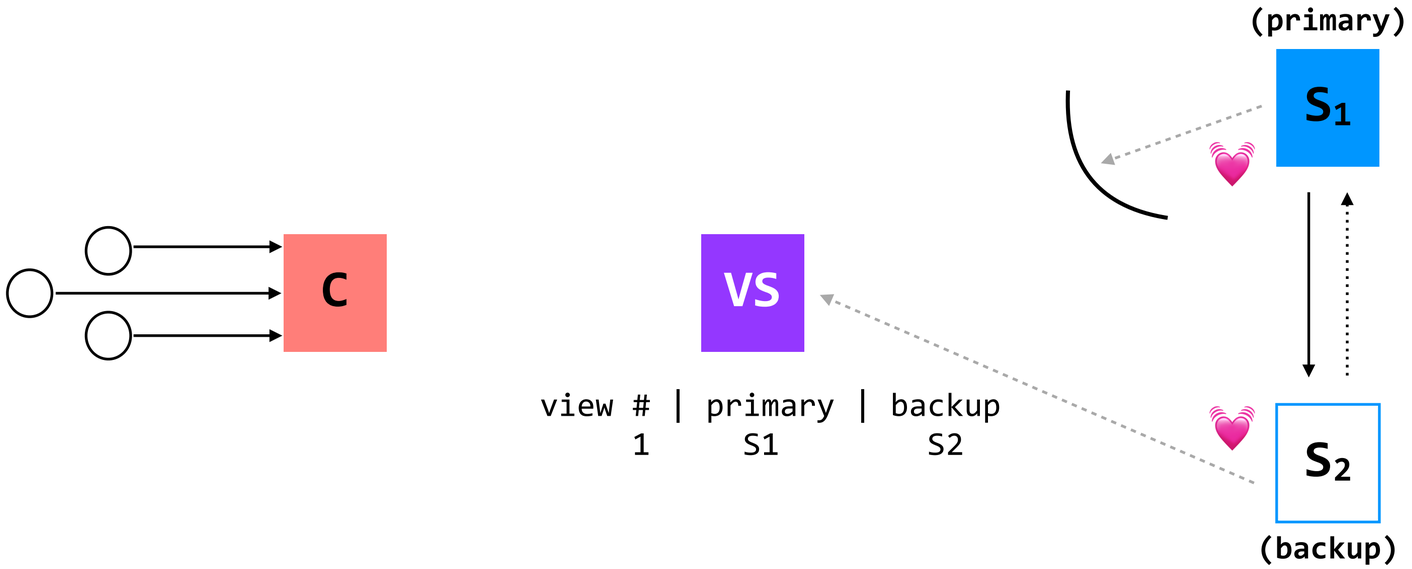


notice there is no longer a backup. once again, we'd hope to eventually bring S_1 back online, or find a new machine to act as a backup. but in this example, we're only interested in safely making S_2 the new primary.

what happens if the primary replica fails?

to increase availability, let's try replicating data on two servers

attempt 3: use a **view server** to determine which replica is primary, in hopes that we can deal with network partitions

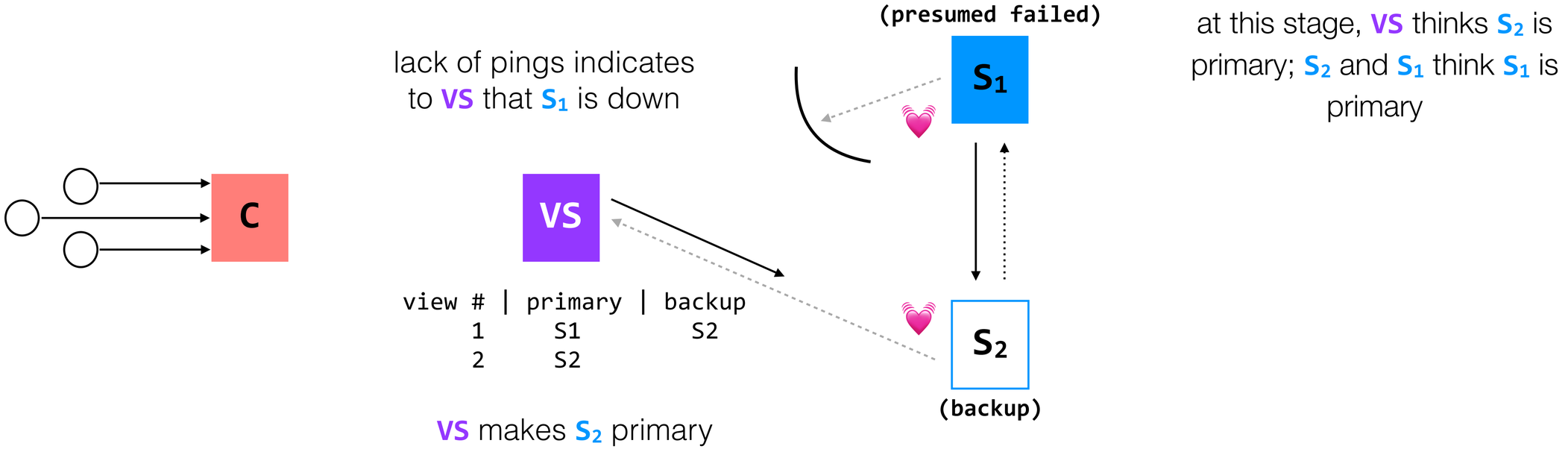


what happens if a network partition prevents S_1 from communicating with VS?

in a sense, this is the worst possible partition: **VS** is going to presume S_1 has failed (and so switch to using S_2 as a backup), while S_1 can still communicate with everyone *except* **VS**

to increase availability, let's try replicating data on two servers

attempt 3: use a **view server** to determine which replica is primary, in hopes that we can deal with network partitions



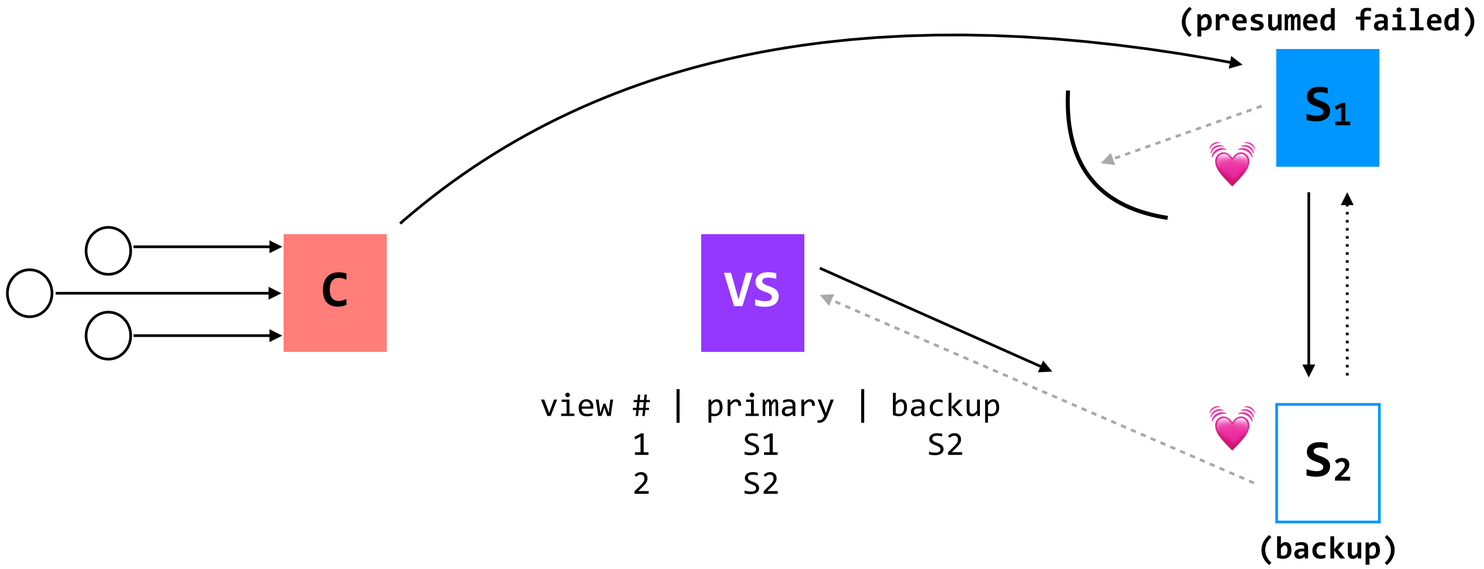
at this stage, VS thinks S_2 is primary; S_2 and S_1 think S_1 is primary

what happens if a network partition prevents S_1 from communicating with VS?

in a sense, this is the worst possible partition: VS is going to presume S_1 has failed (and so switch to using S_2 as a backup), while S_1 can still communicate with everyone *except* VS

to increase availability, let's try replicating data on two servers

attempt 3: use a **view server** to determine which replica is primary, in hopes that we can deal with network partitions



at this stage, VS thinks S_2 is primary; S_2 and S_1 think S_1 is primary

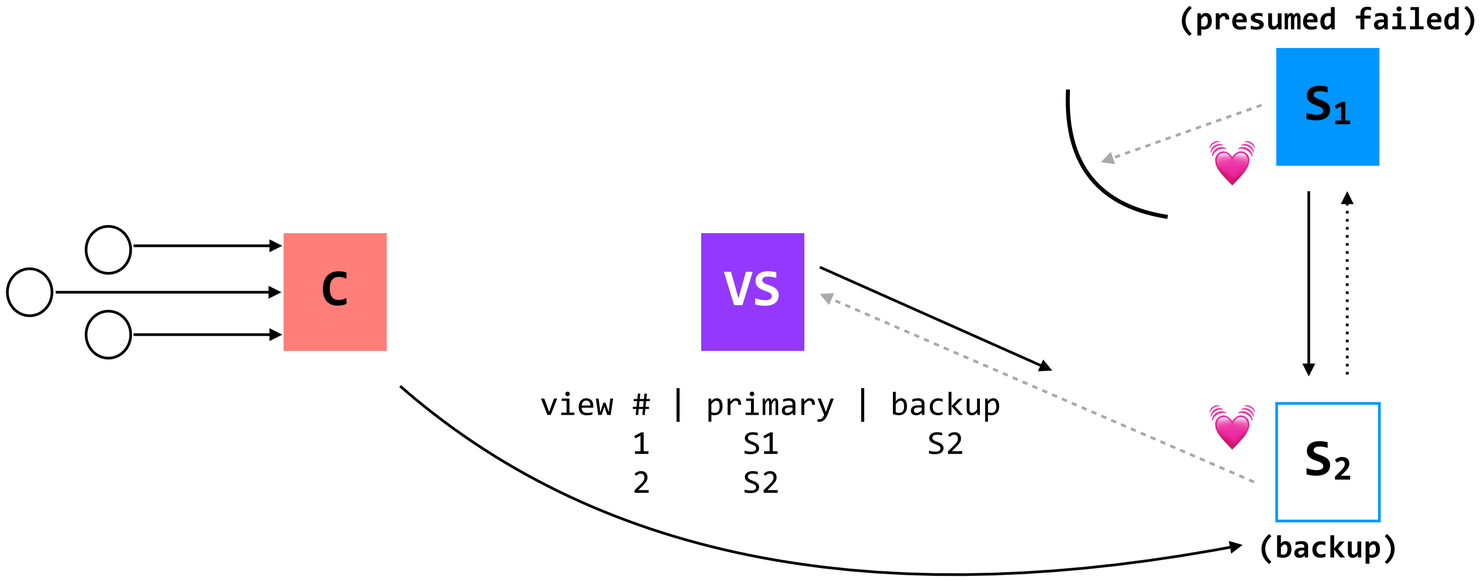
if S_1 receives any requests from C, it will behave as primary with S_2 as backup

what happens if a network partition prevents S_1 from communicating with VS?

in a sense, this is the worst possible partition: VS is going to presume S_1 has failed (and so switch to using S_2 as a backup), while S_1 can still communicate with everyone *except* VS

to increase availability, let's try replicating data on two servers

attempt 3: use a **view server** to determine which replica is primary, in hopes that we can deal with network partitions



at this stage, VS thinks S_2 is primary; S_2 and S_1 think S_1 is primary

if S_1 receives any requests from C, it will behave as primary with S_2 as backup

if S_2 receives any requests from C, it will reject them; it believes that it is the backup (and so does not communicate directly with C)

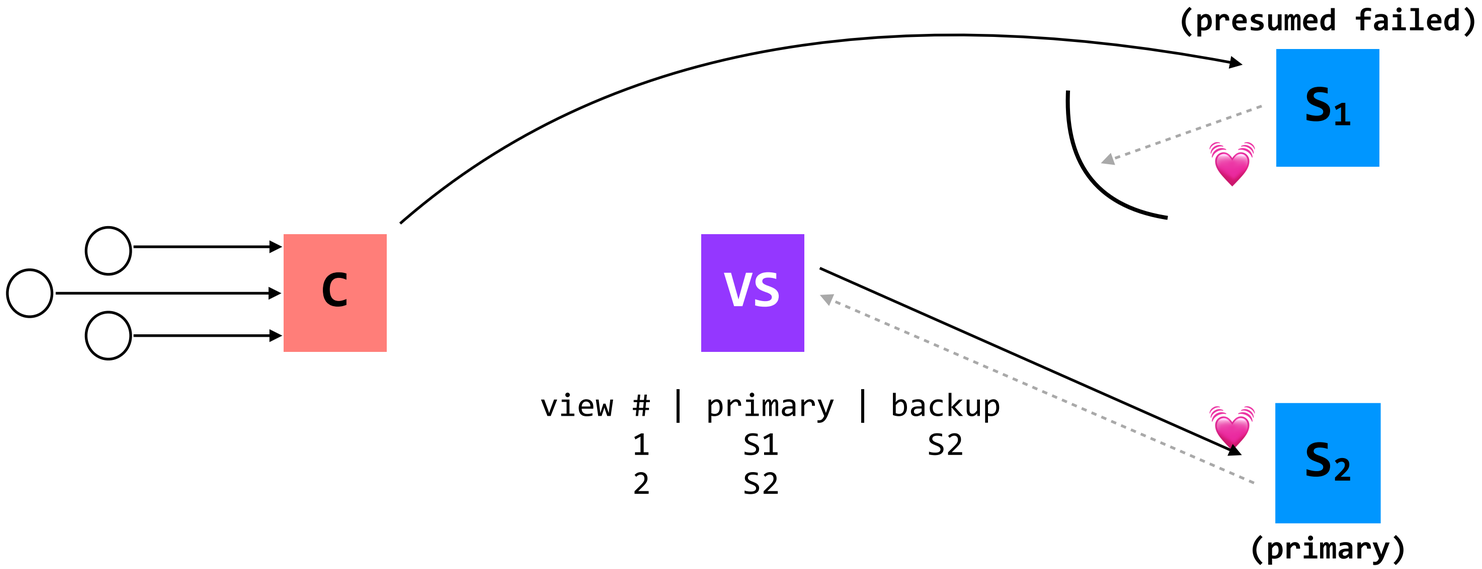
new detail: backups **reject** any requests from coordinators

what happens if a network partition prevents S_1 from communicating with VS?

in a sense, this is the worst possible partition: VS is going to presume S_1 has failed (and so switch to using S_2 as a backup), while S_1 can still communicate with everyone *except* VS

to increase availability, let's try replicating data on two servers

attempt 3: use a **view server** to determine which replica is primary, in hopes that we can deal with network partitions



at this stage, VS and S_2 think S_2 is primary; S_1 thinks S_1 is primary

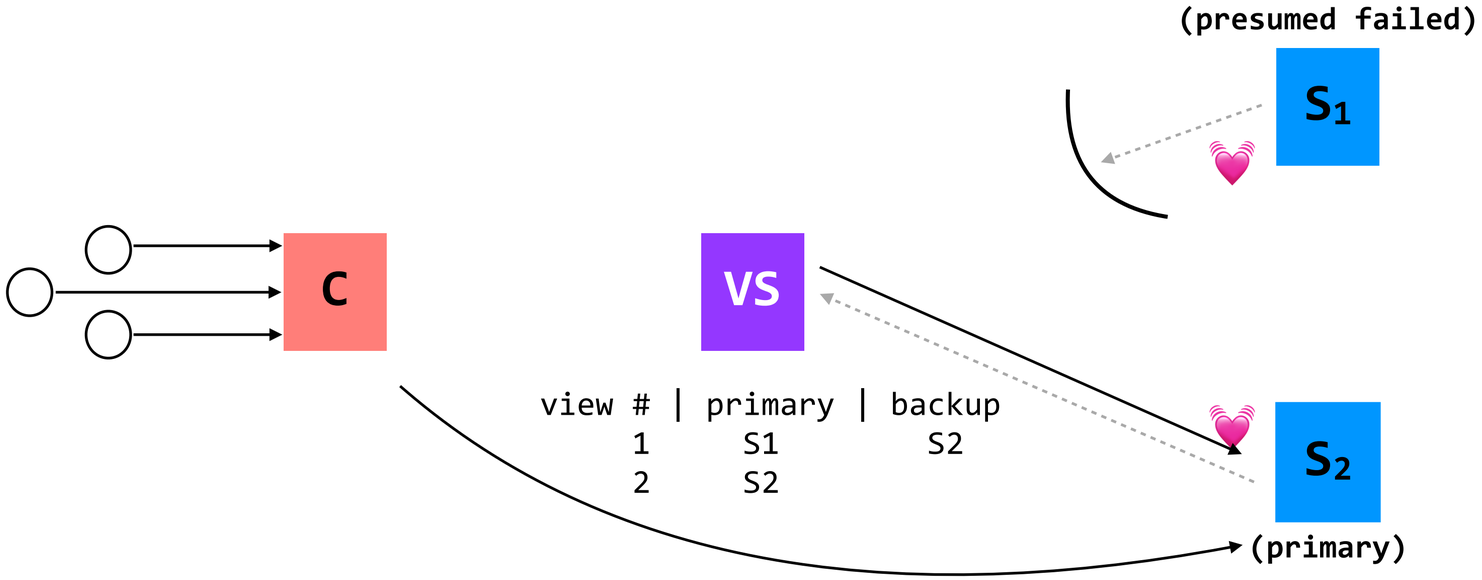
if S_1 receives any requests from C, it won't be able to get an ACK from S_2, and so will reject

what happens if a network partition prevents S_1 from communicating with VS?

in a sense, this is the worst possible partition: VS is going to presume S_1 has failed (and so switch to using S_2 as a backup), while S_1 can still communicate with everyone *except* VS

to increase availability, let's try replicating data on two servers

attempt 3: use a **view server** to determine which replica is primary, in hopes that we can deal with network partitions



at this stage, VS and S_2 think S_2 is primary; S_1 thinks S_1 is primary

if S_1 receives any requests from C, it won't be able to get an ACK from S_2, and so will reject

if S_2 receives any requests from C, it will respond as the primary (in line with what VS expects)

what happens if a network partition prevents S_1 from communicating with VS?

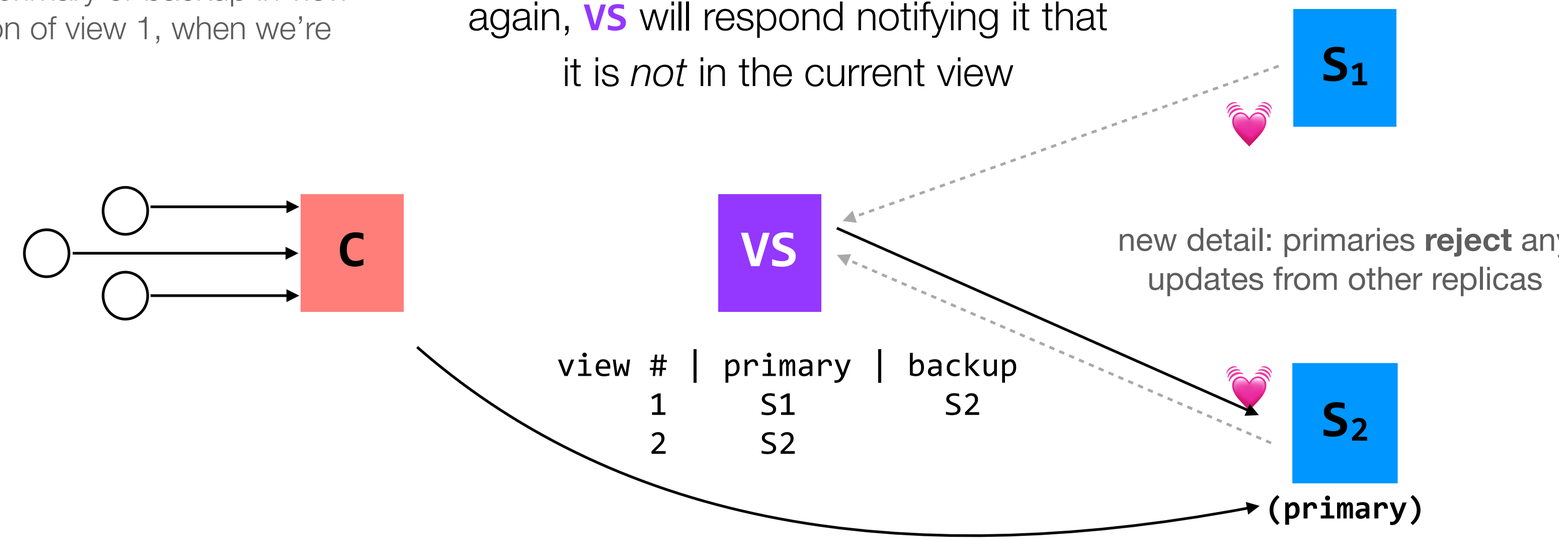
in a sense, this is the worst possible partition: VS is going to presume S_1 has failed (and so switch to using S_2 as a backup), while S_1 can still communicate with everyone *except* VS

to increase availability, let's try replicating data on two servers

attempt 3: use a **view server** to determine which replica is primary, in hopes that we can deal with network partitions

important rule: if a machine is primary in view n, it must have been primary or backup in view n-1 (with the exception of view 1, when we're just starting)

once S_1 can communicate with VS again, VS will respond notifying it that it is *not* in the current view



at this stage, VS and S_2 think S_2 is primary; S_1 thinks S_1 is primary

if S_1 receives any requests from C, it won't be able to get an ACK from S_2, and so will reject

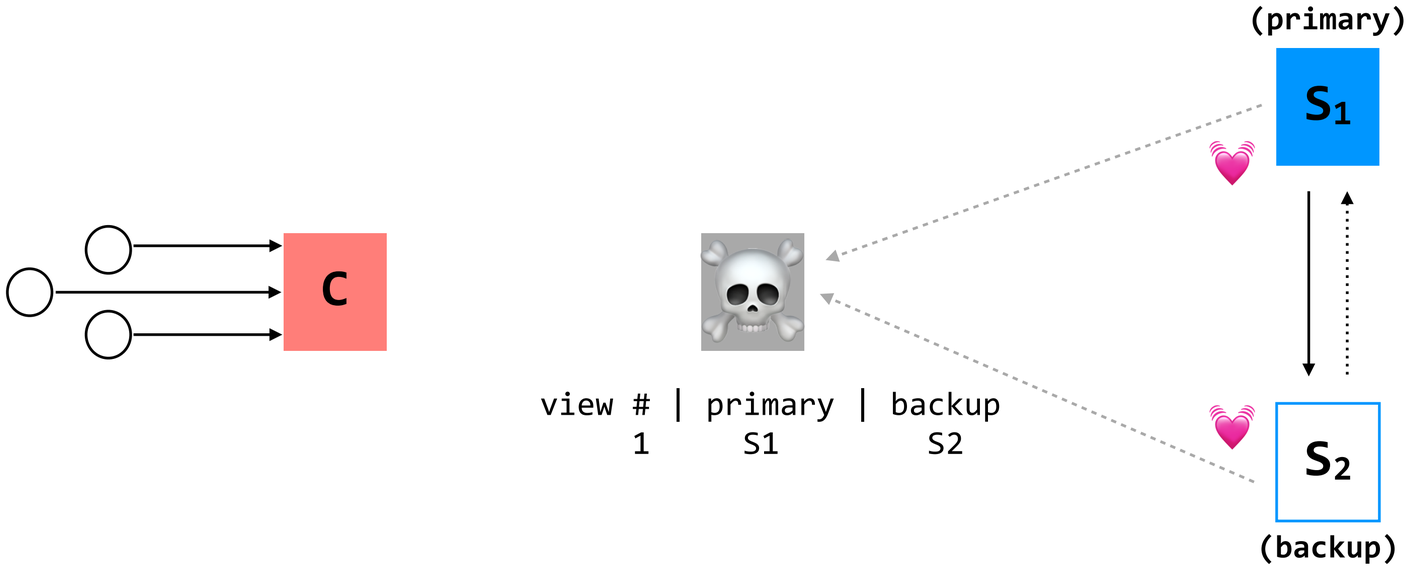
if S_2 receives any requests from C, it will respond as the primary (in line with what VS expects)

what happens if a network partition prevents S_1 from communicating with VS?

in a sense, this is the worst possible partition: VS is going to presume S_1 has failed (and so switch to using S_2 as a backup), while S_1 can still communicate with everyone *except* VS

to increase availability, let's try replicating data on two servers

attempt 3: use a **view server** to determine which replica is primary, in hopes that we can deal with network partitions



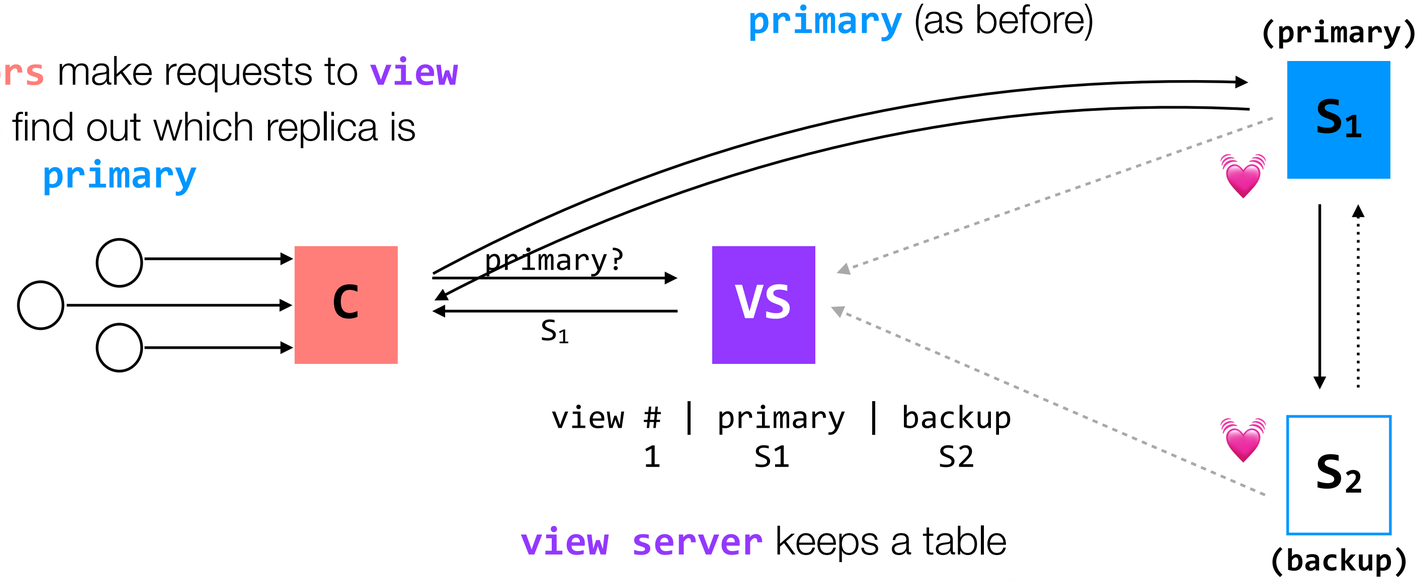
what happens if VS fails?

find out in Tuesday's recitation

replicated state machines

`coordinators` make requests to `view server` to find out which replica is `primary`

`coordinators` contact `primary` (as before)



`primary`/`backup` ping `view server` so that `view server` can discover failures

`view server` keeps a table that maintains a sequence of views

if a machine is primary in view n, it **must** have been primary or backup in view n-1 (with the exception of view 1, when we're just starting)

`primary` sends updates to, gets ACKs from `backup` (as before)

primary **must** get an ACK from its backups before completing the update

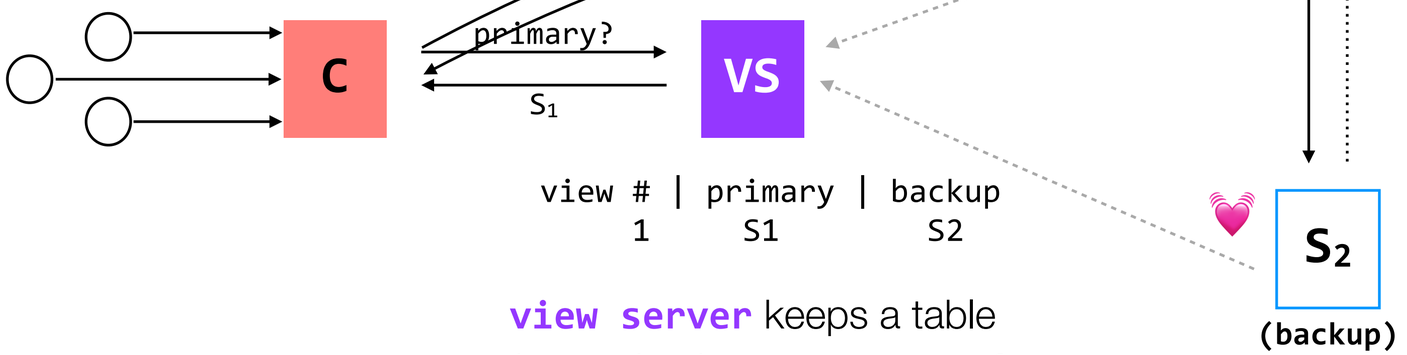
backups will **reject** any requests that they get directly from coordinators; primary will **reject** any update that comes from a backup

(both of these events can happen in the case of certain types of failures)

replicated state machines

`coordinators` contact `primary` (as before)

`coordinators` make requests to `view server` to find out which replica is `primary`



`primary`/`backup` ping `view server` so that `view server` can discover failures

view #	primary	backup
1	S1	S2

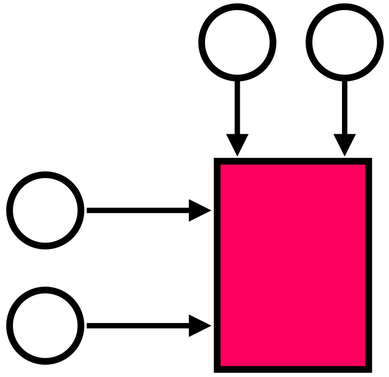
`primary` sends updates to, gets ACKs from `backup` (as before)

`view server` keeps a table that maintains a sequence of views

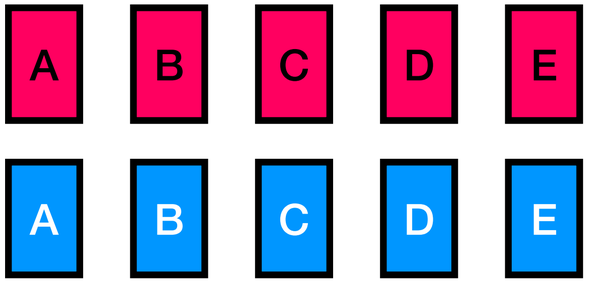
how would we extend this to a primary (S_1) + two backups (say S_2 and S_3)?

after a machine fails, how does the process for recruiting an idle server and turning it into a backup work?
for example, do we have to pause any transactions while we copy data onto the new machine?

our goal is to build **reliable systems from unreliable components**. we want to build systems that serve many clients, store a lot of data, perform well, all while keeping availability high



replicated state machines give us single-copy consistency even with replicated data



transactions — which provide **atomicity** and **isolation** — make it easier for us to reason about failures

our job in lecture is to understand how a system *implements* these two abstractions.
how do our systems guarantee atomicity? how do they guarantee isolation?

atomicity: provided by **logging**, which gives better performance than shadow copies* at the cost of some added complexity; **two-phase commit** gives us multi-site atomicity

isolation: provided by **two-phase locking**

* shadow copies *are* used in some systems

replicated state machines (RSMs) provide **single-copy consistency**: externally, it appears as if there is a single copy of the data, though internally there are replicas

RSMs use a **primary/backup** mechanism for replication. the **view server** ensures that only one replica acts as the primary, and can recruit new backups if servers fail

to extend this model to handle view-server failures, we need a mechanism to provide **distributed consensus**; see tomorrow's recitation