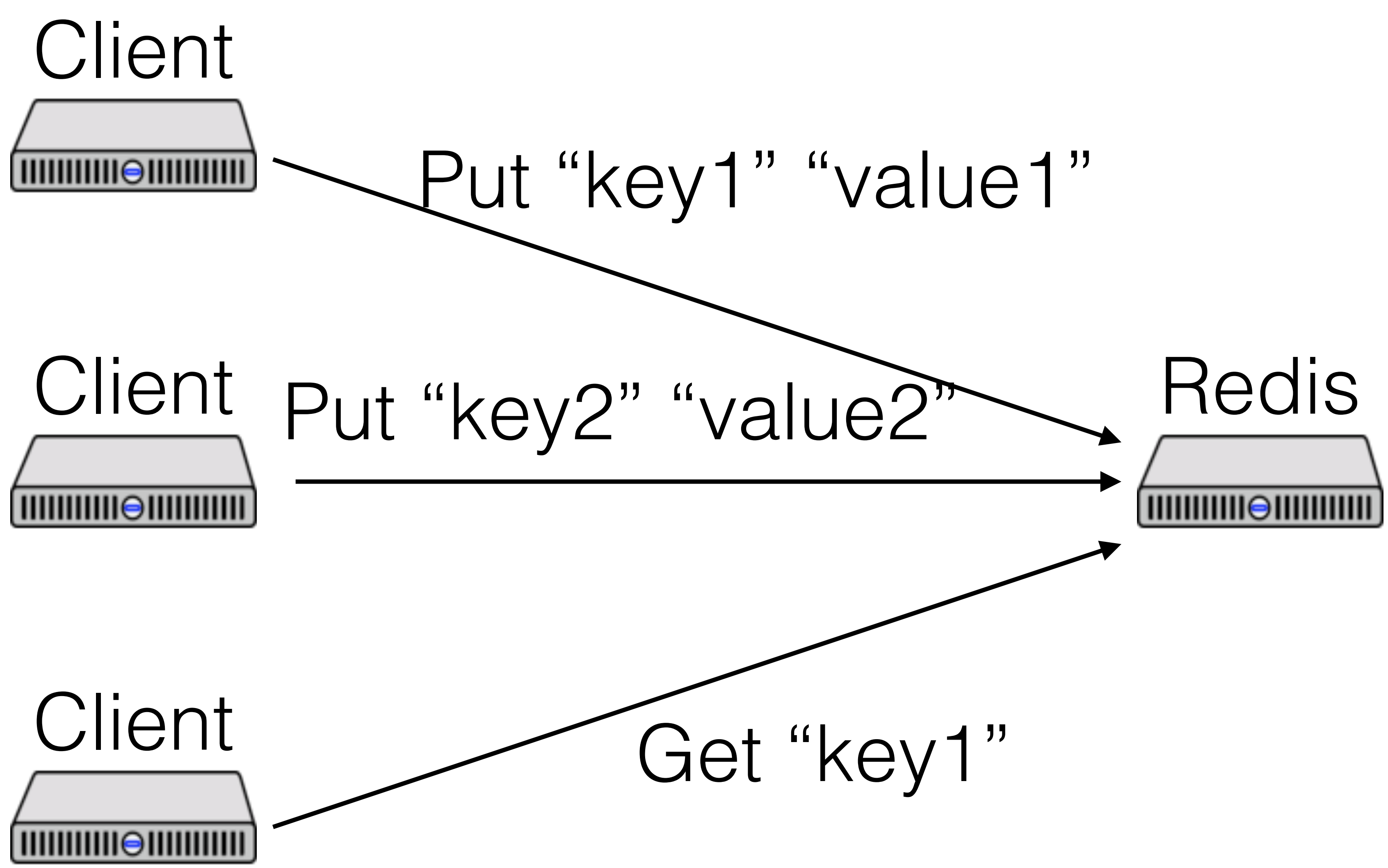
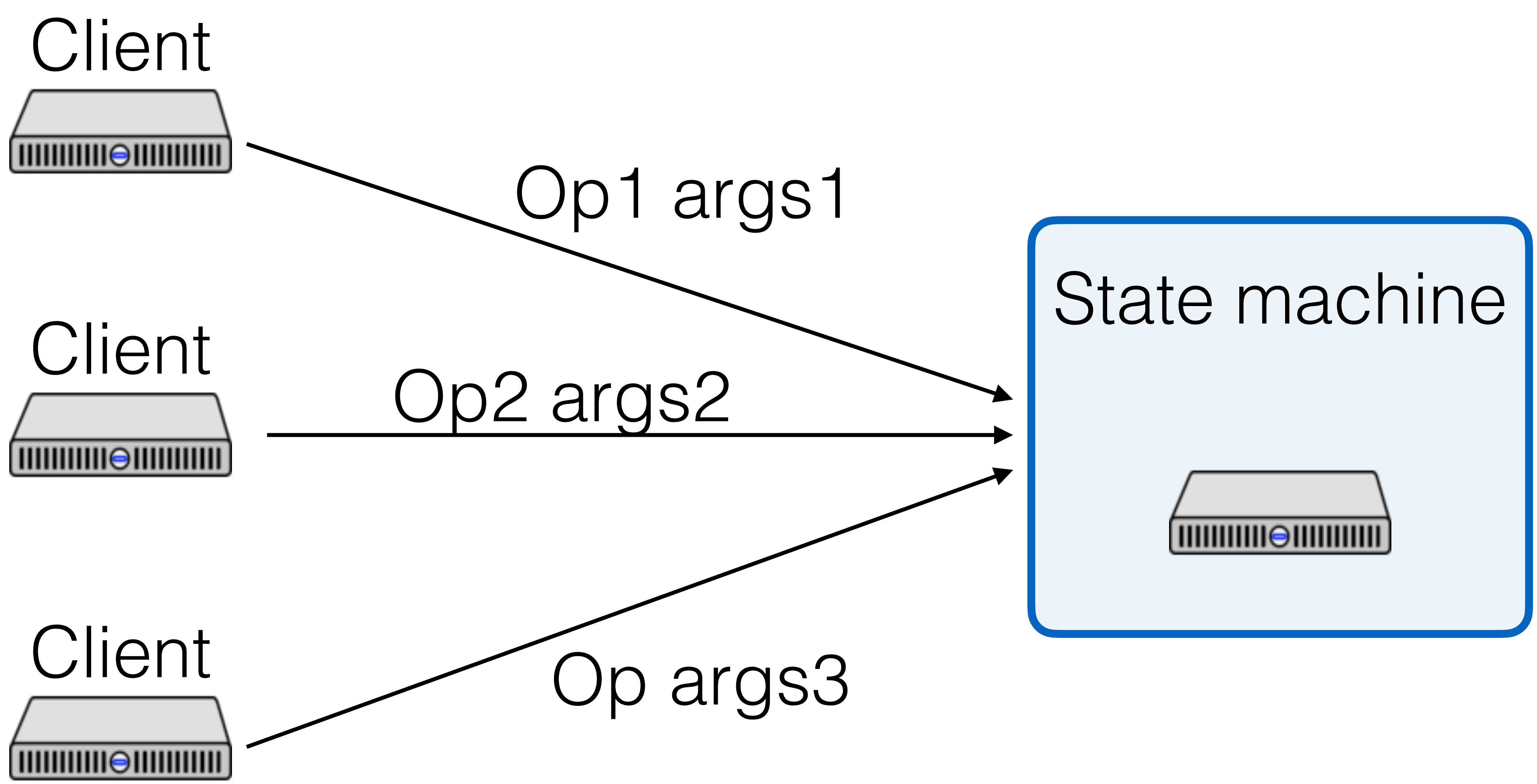
Primary/Backup

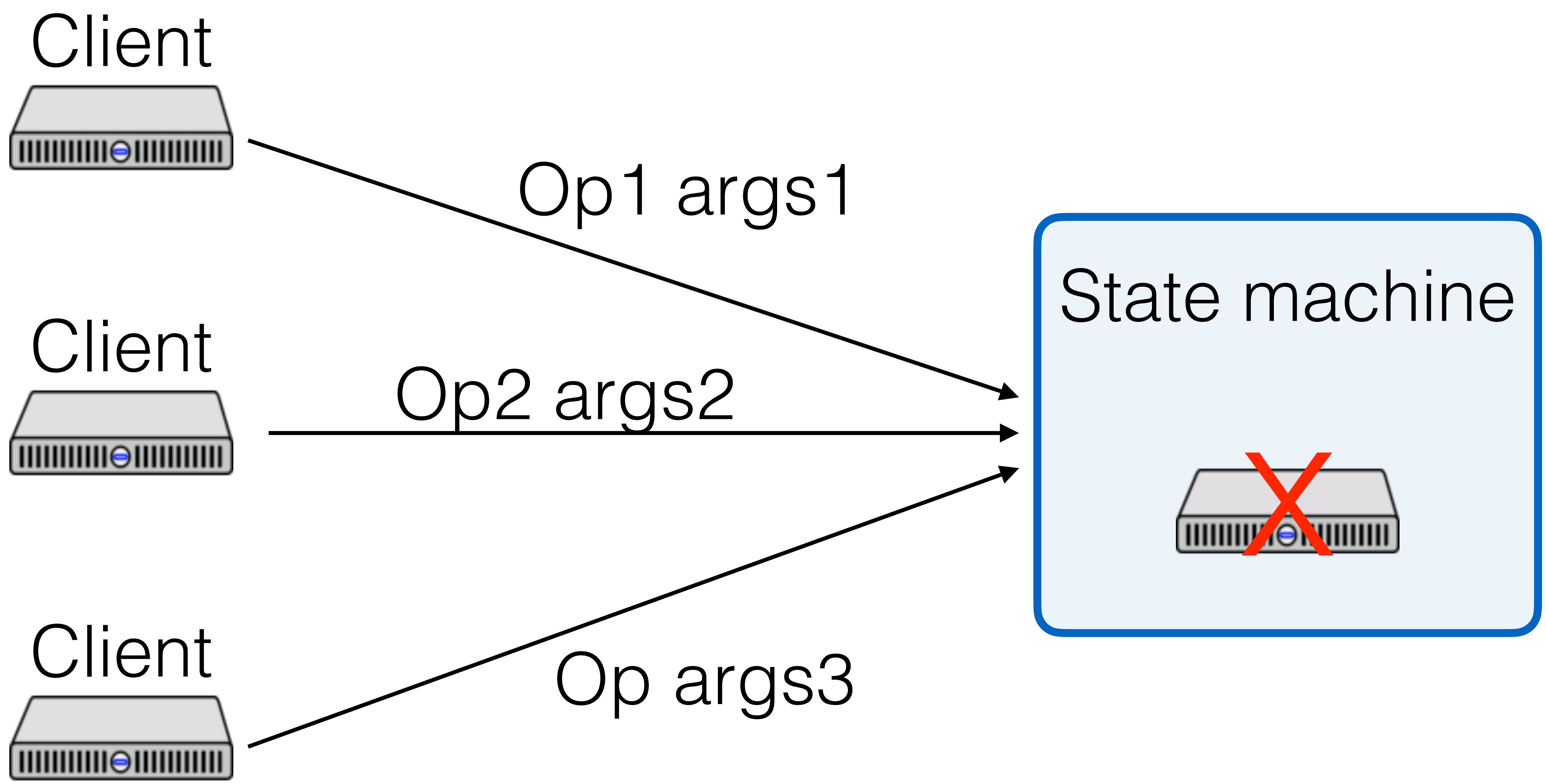
Single-node key/value store



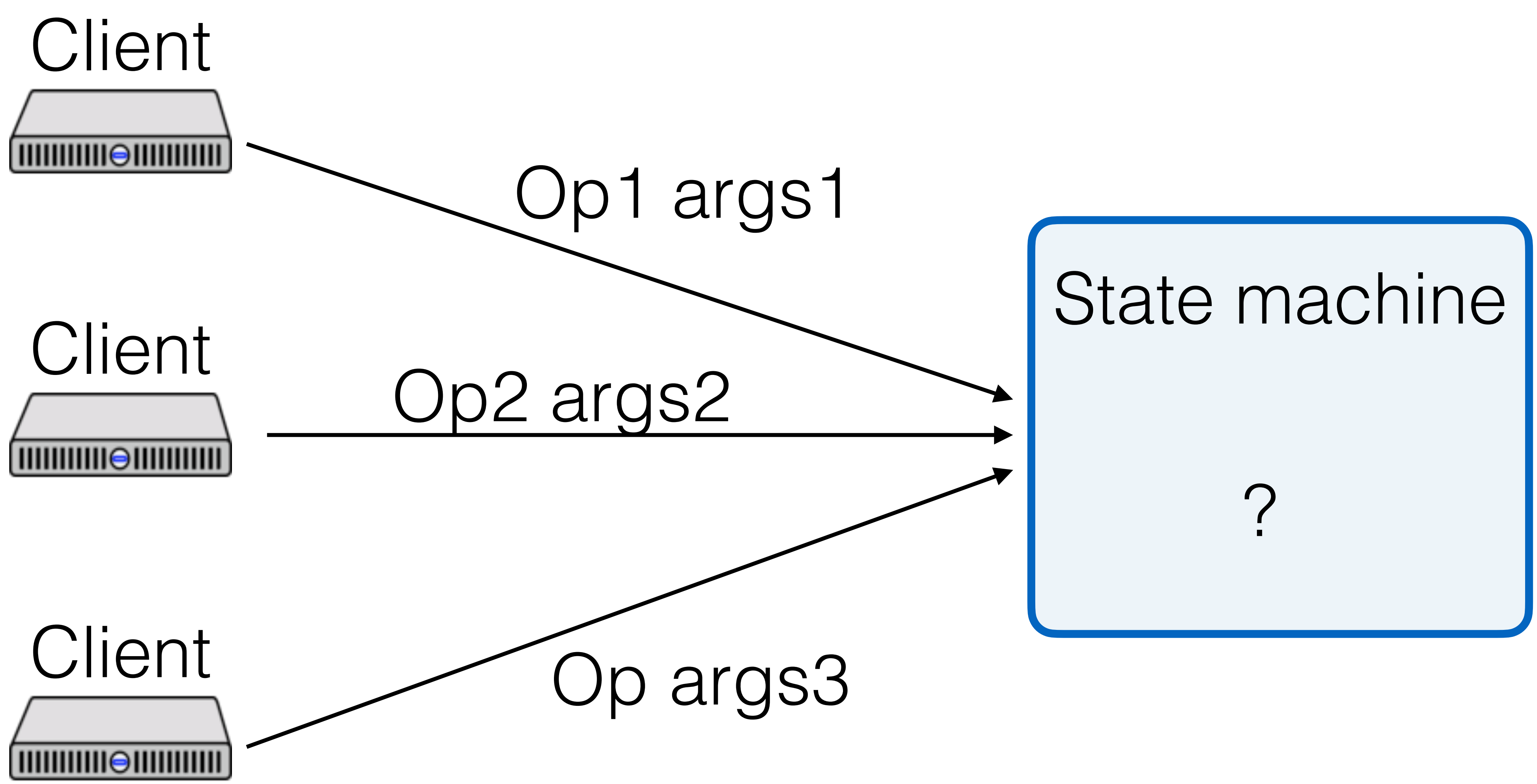
Single-node state machine



Single-node state machine



Single-node state machine



State machine replication

Replicate the state machine across multiple servers

Clients can view all servers as one state machine

What’s the simplest form of replication?

Two servers!

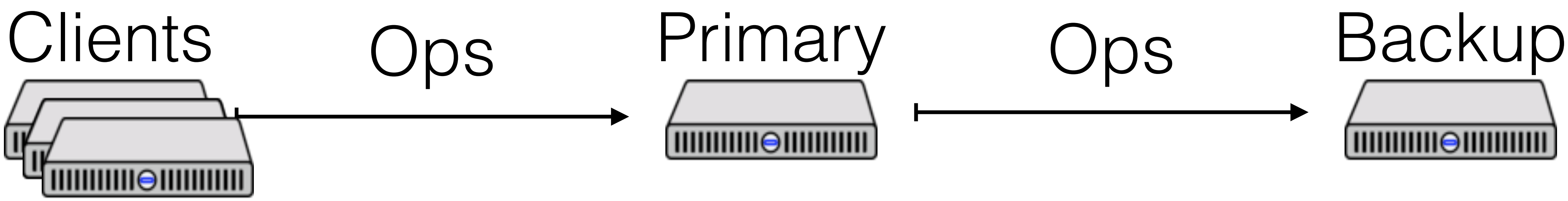
At a given time:

- Clients talk to one server, the primary
- Data are replicated on primary and backup
- If the primary fails, the backup becomes primary

Goals:

- Correct and available (as if a single highly available server)
- Despite *some* failures

Basic operation



Clients send operations (Put, Get) to primary

Primary decides which order to do operations

Primary forwards sequence of operations to backup

Backup performs ops in same order (hot standby)

- Or just saves the log of operations (cold standby)

After backup has saved sequence of operations, primary replies to clients

Challenges

Non-deterministic operations

Dropped messages

- client -> primary, or primary -> backup

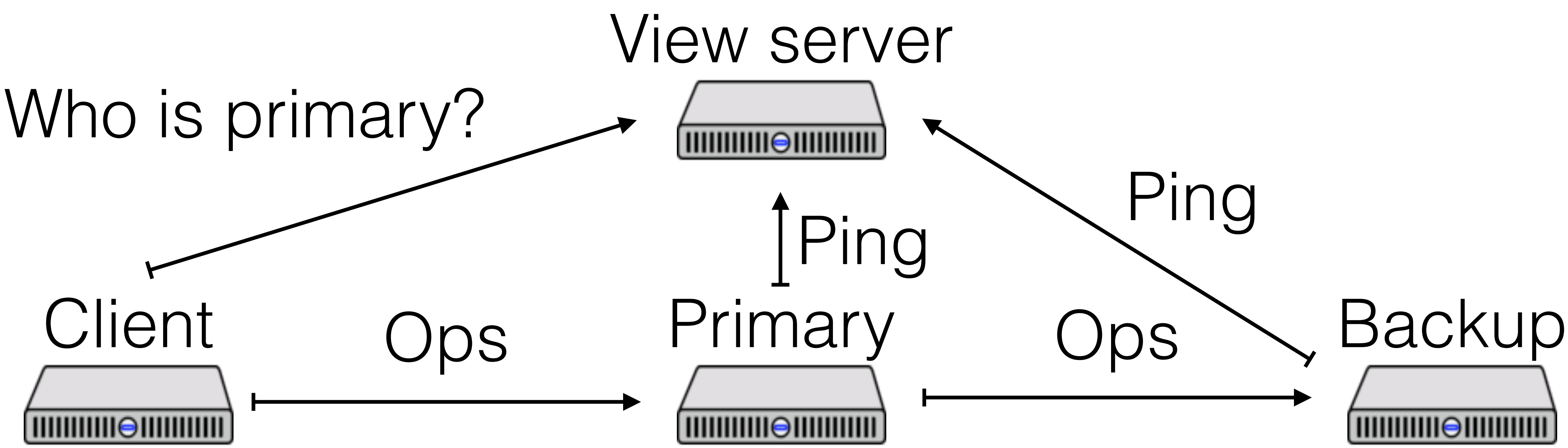
State transfer between primary and backup

- Log of operations? Contents of memory?

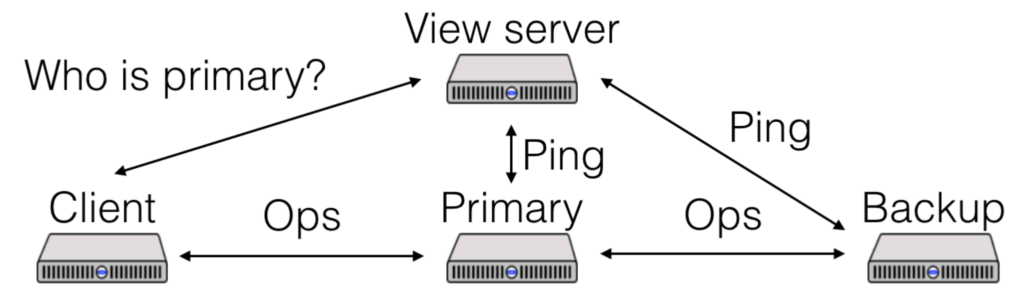
There can be only one primary at a time

- Clients, primary and backup need to agree

The View Service



The View service

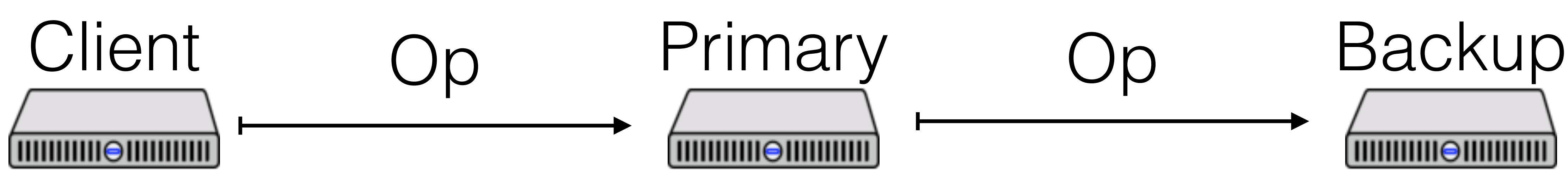


View server decides who is primary and backup

- Clients and servers depend on view server
- View server is a single point of failure (fix in Lab 3)

The hard part:

- Must be only one primary at a time
- Do **not** communicate with view server on every request



On failure

Primary fails

View server declares a new “view”, moves backup to primary

View server promotes an idle server as new backup

Primary initializes new backup’s state

Now ready to process ops, OK if primary fails

“Views”

A view is a statement about the current roles in the system

Views form a sequence in time

View 1	View 2	View 3
Primary = A	Primary = B	Primary = C
Backup = B	Backup = C	Backup = D

Detecting failure

Each server periodically pings (Ping RPC) view server

To the viewserver, a node is

- “dead” if missed *n* Pings
- “live” after a single Ping

Can a server ever be up but declared dead?

Managing servers

Any number of servers can send Pings

- If more than two servers are live, extras are “idle”
- Idle servers can be promoted to backup

If primary dies

- New view with old backup as primary, idle as backup

If backup dies

- New view with idle server as backup

OK to have a view with a primary and no backup

- Why?

Question

How to ensure new primary has up-to-date state?

- Only promote the backup -> primary
- Idle server can become primary at startup (why?)

What if the backup hasn't gotten the state yet?

- And new primary dies?
- First thing: transfer state to backup

View 1
Primary = A
Backup = B

A stops pinging

View 2
Primary = B
Backup = C

B *immediately* stops pinging

View 3
Primary = C
Backup = _

Can't move to View 3 until C gets state
How does view server know C has state?

Viewserver waits for primary ack

Track whether primary has acked (with ping) current view

MUST stay with current view until ack

Even if primary seems to have failed

This is another weakness of this protocol

Question

Can more than one server think it is the primary at the same time?

Split brain

1:A,B

A is still up, but can’t reach view server
(or is unlucky and pings get dropped)

2:B,_

B learns it is promoted to primary
A still thinks it is primary

Split brain

Can more than one server *act* as primary?

- Act as = respond to clients

Rules

1. Primary in view *i+1* must have been backup or primary in view *i*
2. Primary must wait for backup to accept/execute each op before doing op and replying to client
3. Backup must accept forwarded requests only if view is correct
4. Non-primary must reject client requests
5. Every operation must be before or after state transfer

Rules

1. Primary in view *i+1* must have been backup or primary in view *i*
2. Primary must wait for backup to accept/execute each op before doing op and replying to client
3. Backup must accept forwarded requests only if view is correct
4. Non-primary must reject client requests
5. Every operation must be before or after state transfer

Split brain (and !state)

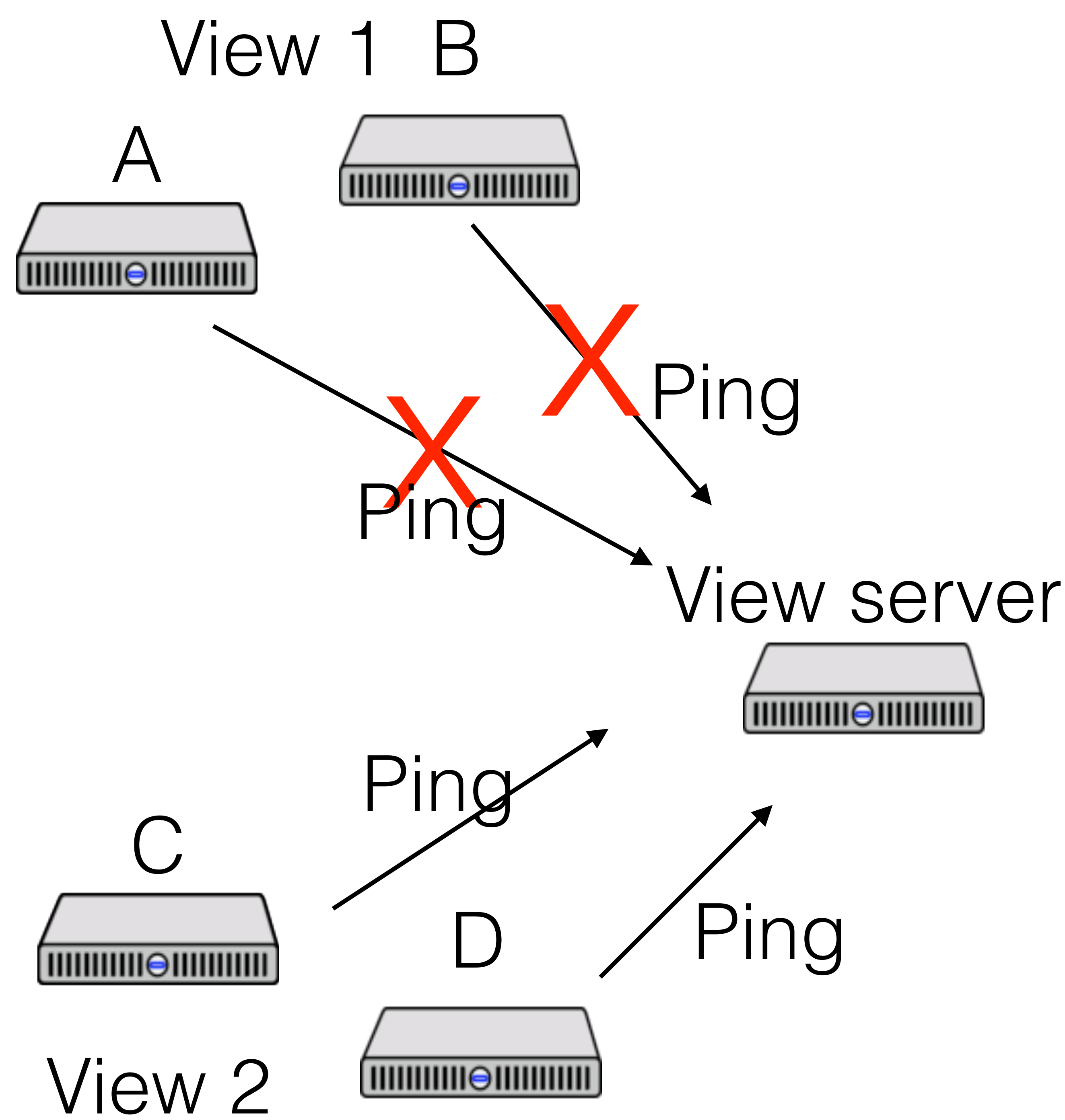
1:A,B

A is still up, but can't reach view server

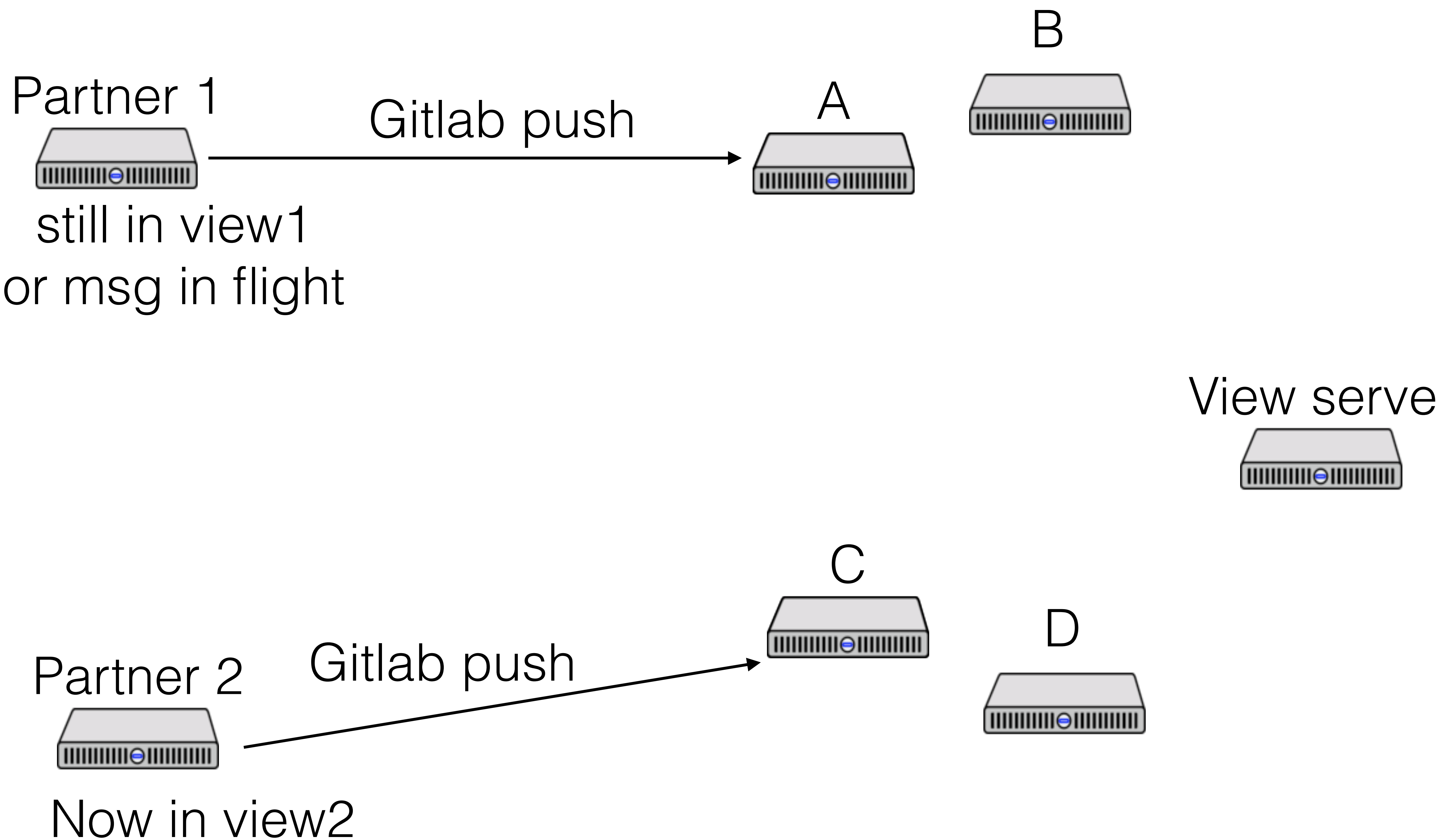
2:C,D

C learns it is promoted to primary
A still thinks it is primary
C doesn't know previous state

Split Brain In Action



Split Brain In Action



Rules

1. Primary in view *i+1* must have been backup or primary in view *i*
2. Primary must wait for backup to accept/execute each op before doing op and replying to client
3. Backup must accept forwarded requests only if view is correct
4. Non-primary must reject client requests
5. Every operation must be before or after state transfer

1. Missing writes

1:A,B

Client writes to A, receives response
A crashes before writing to B

2:B,C

Client reads from B
Write is missing

2. “Fast” Reads?

Does the primary need to forward reads to the backup?

(This is a common “optimization”)

Stale reads

1:A,B

A is still up, but can’t reach view server

2:B,C

Client 1 writes to B
Client 2 reads from A
A thinks it is primary, returns outdated value

Reads vs. writes

Reads treated as state machine operations too

But: can be executed more than once

RPC library can handle them differently

Rules

1. Primary in view *i+1* must have been backup or primary in view *i*
2. Primary must wait for backup to accept/execute each op before doing op and replying to client
3. Backup must accept forwarded requests only if view is correct
4. Non-primary must reject client requests
5. Every operation must be before or after state transfer

Partially split brain

1:A,B

A forwards a request…

2:B,C

Which arrives here

3:C,D

But by then maybe C is the new primary

Old messages

1:A,B

A forwards a request…

2:B,C

3:C,A

4:A,B

Which arrives here

Rules

1. Primary in view *i+1* must have been backup or primary in view *i*

2. Primary must wait for backup to accept/execute each op before doing op and replying to client

3. Backup must accept forwarded requests only if view is correct

4. Non-primary must reject client requests

5. Every operation must be before or after state transfer

Inconsistencies

1:A,B

2:B,C

3:B,A

Outdated client sends request to A
A shouldn't respond!

What about old messages to primary?

1:A,B

2:B,C

3:B,A

4:A,D

Outdated client sends request to A

Rules

1. Primary in view *i+1* must have been backup or primary in view *i*
2. Primary must wait for backup to accept/execute each op before doing op and replying to client
3. Backup must accept forwarded requests only if view is correct
4. Non-primary must reject client requests
5. Every operation must be before or after state transfer

Inconsistencies

1:A,B

A starts sending state to B
Client writes to A
A forwards op to B
A sends rest of state to B

Rules

1. Primary in view *i+1* must have been backup or primary in view *i*
2. Primary must wait for backup to accept/execute each op before doing op and replying to client
3. Backup must accept forwarded requests only if view is correct
4. Non-primary must reject client requests
5. Every operation must be before or after state transfer

Progress

Are there cases when the system can’t make further progress (i.e. process new client requests)?

Progress

- View server fails
- Network fails entirely (hard to get around this one)
- Client can’t reach primary but it can ping VS
- No backup and primary fails
- Primary fails before completing state transfer

State transfer and RPCs

State transfer must include RPC data

Duplicate writes

1:A,B

Client writes to A
A forwards to B
A replies to client
Reply is dropped

2:B,C

B transfers state to C, crashes

3:C,D

Client resends write. Duplicated!

One more corner case

1:A,B

View server stops hearing from A
A and B, and clients, can still communicate

2:B,C

B hasn’t heard from view server
Client in view 1 sends a request to A
What should happen?
Client in view 2 sends a request to B
What should happen?

Primary Backup: Why its hard

Primary may fail

Backup may fail

Communication may fail partially or temporarily

Participants may lag decisions made at:

- view server (has view changed?)
- primary (did it fail? reply to client message?)
- backup (did it fail? has it learned of new view? has state transfer completed?)
- client (has the view changed?)

Rules

1. Primary in view *i+1* must have been backup or primary in view *i*
2. Primary must wait for backup to accept/execute each op before doing op and replying to client
3. Backup must accept forwarded requests only if view is correct
4. Non-primary must reject client requests
5. Every operation must be before or after state transfer

Primary Backup In Practice

What state to replicate?

How does the backup get state?

Apply changes to backup, or just log?

When do we cut over to the backup?

Are anomalies visible at the cut over?

How do we repair/re-integrate?

Replicated Virtual Machines

Whole system replication

Completely transparent to applications and clients

High availability for any existing software

Challenge: Need state at backup to exactly mirror primary

Restricted to uniprocessor VMs

Deterministic Replay

Key idea: state of VM depends only on its input

- Content of all input/output
- Precise instruction of every interrupt
- Only a few exceptions (e.g., timestamp instruction)

Record all hardware events into a log

- Modern processors have instruction counters and can interrupt after (precisely) x instructions
- Trap and emulate any non-deterministic instructions

Replicated Virtual Machines

Replay I/O, interrupts, etc. at the backup

- Backup executes events at primary with a lag
- Backup stalls until it knows timing of next event
- Backup does not perform external events

Primary stalls until it knows backup has copy of every event up to (and incl.) output event

- Then it is safe to perform output

Example

Primary receives network interrupt

- hypervisor forwards interrupt plus data to backup
- hypervisor delivers network interrupt to OS kernel
- OS kernel runs, kernel delivers packet to server
- server/kernel write response to network card
- hypervisor delays sending network packet to client until backup acks

Backup receives log entries

- backup delivers network interrupt
- …
- hypervisor does *not* put response on the wire
- hypervisor ignores local clock interrupts

Questions

Why send output events to backup and delay output at primary until backup has acked?

What happens when primary fails after receiving network input but before sending log entry to backup?

Can the same output be produced twice?