Primary/Backup

CS 452
Single-node key/value store

Client

Put “key1” “value1”

Client

Put “key2” “value2”

Redis

Get “key1”
Single-node state machine

Client → Op1 args1 → State machine
Client → Op2 args2 → State machine
Client → Op args3 → State machine
Single-node state machine

Client

Op1 args1

Op2 args2

Op args3

State machine
Single-node state machine

Client → Op1 args1
Client → Op2 args2
Client → Op args3

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
- Despite *some* failures
Clenet

Ops

Primary

Ops

Backup

Clients send operations (Put, Get) to primary
Primary decides on order of ops
Primary forwards sequence of ops to backup
Backup performs ops in same order (hot standby)
- Or just saves the log of operations (cold standby)
After backup has saved ops, primary replies to client
Challenges

Non-deterministic operations
Dropped messages
State transfer between primary and backup
  - Write log? Write state?
There can be only one primary at a time
  - Clients, primary and backup need to agree
The View Service

Who is primary?

Client → Ops → Primary → Backup

Ping:
- Client to Primary
- Primary to Ops (Backup)
- Backup to Primary

Ops:
- Client to Primary
- Primary to Backup
The View service

View server decides who is primary and backup
  - Clients and servers depend on view server

The hard part:
  - Must be only one primary at a time
  - Clients shouldn’t communicate with view server on every request
  - Careful protocol design

View server is a single point of failure (fixed in Lab 3)
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
A view is a statement about the current roles in the system.

Views form a sequence in time:

- **View 1**
  - Primary = A
  - Backup = B

- **View 2**
  - Primary = B
  - Backup = C

- **View 3**
  - Primary = C
  - Backup = D
Detecting failure

Each server periodically pings (Ping RPC) view server To the view server, 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

- But can’t process any meaningful ops
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?
Viewservlet 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
Incomplete 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
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 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
Old messages

1: A, B

2: B, C

3: C, A

4: A, B

A forwards a request…

Which arrives here
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
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?