

Acknowledgements

These DSLabs slides were developed as a collaborative effort among the TA's for UW CSE 452/552 over a period of several years. We thank all who contributed.

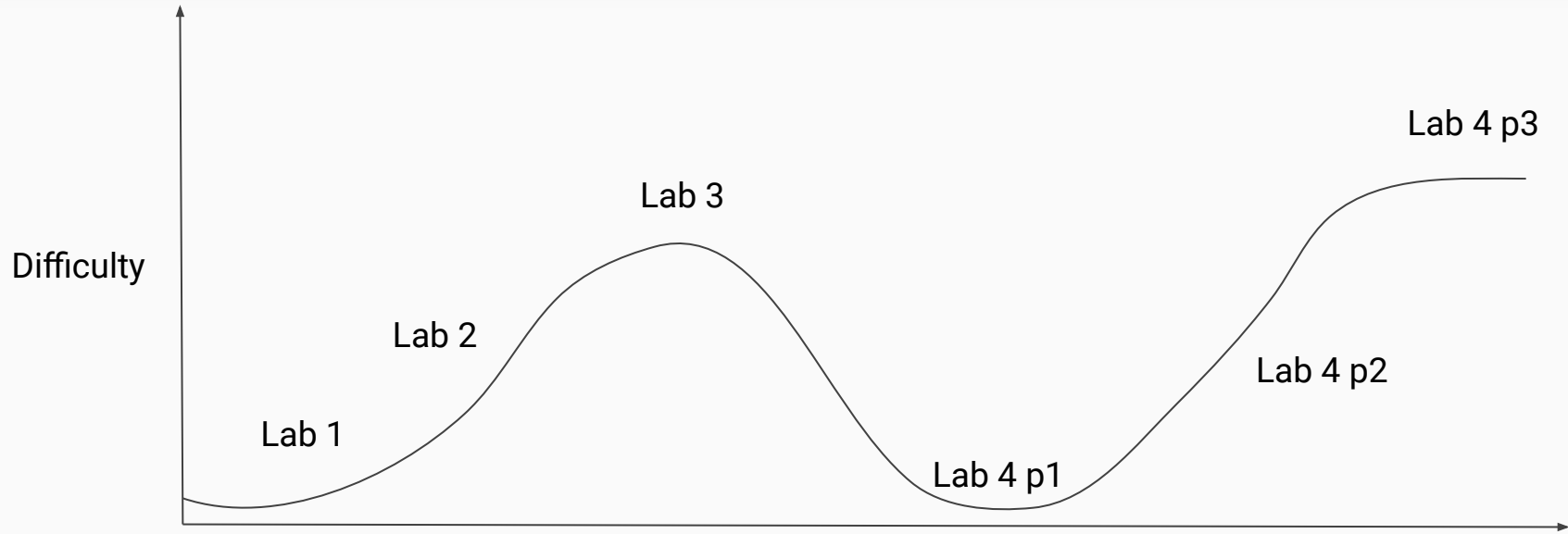
Adnan Ahmad, Nick Anderson, Anirban (Anir) Biswas, Anirudh Canumalla, Riley Chang, Tianyi Cui, Colin Evans, Yael Goldin, Divye Jain, Kushal Jhunjunwalla, Justin Johnson, Sarang Joshi, Lukas Joswiak, Arthur Liang, Jiuru Li, Boyan Li, CJ Lin, Wei Lin, Travis McGaha, Ellis Michael, Samantha Miller, David Porter, Raden (Roy) Pradana, Guramrit Singh, Luxi Wang, Andrew Wei, Zachariah Wolfe, Doug Woos, Lanhao Wu, Yuchong (Yvonna) Xiang, Shibin (Jack) Xu, Paul Yau, Dao Yi, Michael Yu, Sarah Yu, Aileen Zeng, Kaiyuan Zhang, Leiyi Zhang, Eddy Zhou

Section 1: Intro to Labs 0 + 1

CSE 452 Spring 2021



Labs (in terms of difficulty from what students have told us in the past)



Recommendations/General Notes

- **Start early!**
- Ask questions in section/lecture/on ed
- Labs get much much much harder
- Know what you're trying to implement before coding
- Read the spec and reread the spec
- A lot of time in later labs might be spent rewriting the code (usually cleaner or to trim unnecessary implementation details), but before you do that, you should probably talk to a TA or instructor to get a second opinion

Framework

- Collection of abstract classes and interfaces that you will extend/implement
- **Read the documentation!**
 - In `/doc/dslabs/framework` directory
- **Read the spec!**

Node

Read the documentation!

- Abstract class that you will subclass
- Basic unit of computation (aka one machine)
- Notable methods:
 - `set(Timer timer, int timerLengthMillis)`
 - `send(Message message, Address to)`
 - message and timer handlers (naming is important!)
 - Automatically triggers when an XXX message is received/XXX timer goes off
 - `handleXXX` where XXX is name of message (e.g. `handlePongReply`)
 - `onXXX` where XXX is name of timer (e.g. `onPingTimer`)
 - This is called [reflection](#)
- Handlers should be
 - Deterministic
 - Idempotent whenever possible

Client/Server

- Types of Nodes
- Client
 - interface that client Nodes should implement
 - `sendCommand()`, `hasResult()`, `getResult()`
 - Called by testing framework
- Server
 - No interface for server nodes
 - Will generally call `execute()` on an Application

Application

- Backing data structures + logic used by nodes (i.e. key/value store, shopping cart, etc.)
- Processes **Commands** and produces **Results**
 - Each application will define its own set of Commands and Results
 - Examples of commands are GETs and PUTs on a server
- **Interface**
 - `Result execute(Command c)`

Message

- `send(Message message, Address to)`
- Encapsulates data passed between Nodes
- Messages can contain **Commands** and **Results**
- Interface with no methods, but extends `Serializable`
- Messages are marshalled/unmarshalled for you
 - Objects are serialized when sent in messages and deserialized when received

Timer

- `set(Timer timer, int timerLengthMillis)`
- Delivered to Nodes via their timeout handlers
- Will not be automatically delivered! Need to set these manually in Nodes.
 - If a Node resets a Timer and calls a method to set the same Timer, and this goes on for a while, it may lead to the testing framework halting and you might see the framework make no progress (which is bad).
 - Additionally, to reduce the number of states, when a timer goes off:
if you want to reset the timer, make sure to **call set with the same Timer object**, instead of passing a new Timer object
- Can contain other data and call methods
 - Recommended: Reset timers at the end of the method/if-statement so that the timer isn't ticking while still processing the method

Synchronize

- You will see many methods that are synchronized, ex:

```
public synchronized Result getResult();
```

- Only one thread can execute a synchronized method on an object at a time (all other threads executing synchronized methods on that object will block until the first thread releases the lock)
 - We use method synchronization, so the entire object gets locked when a synchronized method gets called, but `wait()` releases the lock.
- Consider `PingClient` with `getResult()` and `handlePongReply()` without the `synchronized` keyword

Wait & Notify

- Think Condition Variable

```
while (!checkCondition()) wait();
```

- Re-evaluate condition when notified - notify as a “hint”
- Ex: `PingClient.getResult()` waits `while (pong == null)`, where `pong` is set by `handlePongReply!`

Lombok

It's really just for Boilerplate

- `@EqualsAndHashCode`
 - Generates equals and hashCode methods for you
- `@Data`
 - “A shortcut for `@ToString`, `@EqualsAndHashCode`, `@Getter` on all fields, `@Setter` on all non-final fields, and `@RequiredArgsConstructor`!” - <https://projectlombok.org/features/Data>
- **All messages and timers should have @Data**, will lead to an explosion in state space if you don't and you may fail some tests

If you use IntelliJ, install the Lombok Plugin:

<https://projectlombok.org/setup/intellij>

run-tests.py

- `./run-tests.py --lab 0 --debug 1 1 "Hello World,Goodbye World"`
- Options
 - `--debug <# servers> <# clients> <comma-separated list of client arguments>`: start the visual debugger with the given arguments
 - `--test-num, --lab`
 - `--no-run, --no-search`: execute only the runtime tests or the graph search tests
 - `-g FINEST`: logs every message
- **Use Python 3, Java 11 :)**

Type of Tests

Regular (Run)

- Runs the Nodes, usually in parallel
 - You can use `./run-tests.py --single-threaded`` to help debug
 - If it doesn't work in single threaded, it probably won't work when running in parallel
- “UNRELIABLE” tests means that messages could get dropped

Search

- Checks correctness and liveness
- State search to look for state violations and/or a goal state (BFS, DFS)
- **Do not use logging.** The logging messages won't make any sense because of how states are explored, i.e. you might observe they go back on each other.

Common run commands

Running a single test

- `./run-test.py -l LAB_NUM -n TEST_NUM`

Running multiple tests

- `./run-test.py -l LAB_NUM -n TEST_NUM_1,TEST_NUM_2,TEST_NUM_3`

Common run commands

Running with Logging [prints out message receives/timer handles]

- `./run-test.py -l LAB_NUM -n TEST_NUM -g FINER`

Running with Logging [prints out message sends & receives/timer sets and handles]

- `./run-test.py -l LAB_NUM -n TEST_NUM -g FINEST`

Running with Logging and writing to a file (**Don't do this for search tests**)

- `./run-test.py -l LAB_NUM -n TEST_NUM -g FINEST &> output.txt`

Common run commands -- [SEARCH] tests

Open up the visualizer on a non-search test with a custom workload

- `./run-tests.py -l LAB_NUM --debug NUM_SERVERS NUM_CLIENTS WORKLOAD`
 - WORKLOAD will look like "PUT:foo:bar,APPEND:foo:baz,GET:foo"

Open up the visualizer on a failed search test with an invariant violation

[ONLY STARTS VISUALIZER FOR FAILED SEARCH TESTS DUE TO INVARIANT VIOLATIONS]

(You'll want to click "Debug system with trace")

- `./run-test.py -l LAB_NUM -n SEARCH_TEST_NUM --start-viz`

Equals and Hashcode/Idempotency checks (really stupid reasons for failing search tests)

- `./run-test.py -l LAB_NUM -n SEARCH_TEST_NUM --checks`

Note: `--checks` doesn't work on run tests.

Saving SEARCH test traces

```
./run-test.py -l LAB_NUM -n SEARCH_TEST_NUM --save-trace
```

- Saves a copy of the trace for the search test if there's an invariant violation
- Helps with debugging some null pointer exceptions

```
./run-test.py --check-trace
```

- Runs the trace again to see if you get the same error
 - (You might need to manage your traces)
- Works with `--start-viz`

Tour of the Visualization Tool

```
./run-tests.py --lab 0 --debug 1 1 "hi,bye"
```

Idempotency and Determinism

Idempotent: When a unique message comes, the actions (e.g. changing state or creating timers) taken by it are only applied once and won't be applied again for duplicate messages

- Problems come up usually when people create new timers for a duplicate message, which expands the state space and makes exploration towards a goal harder

Determinism: Outcome is only a function of state and message/timer handlers

- This means no timestamps, no random numbers, no UUIDs

Lab 1 help/hints

- Do not store StringBuilder for the key in the KVStore map
- Make sure that alreadyExecuted uses the right comparison operators
- “Do not use any static, mutable fields in your classes (constants are fine). Your nodes (i.e., instances of the Node classes) should communicate only through message passing”
 - E.g. the **Map** in **KVStore**
- In Part 3, be sure to integrate the AMO package into SimpleClient and SimpleServer. In SimpleServer, make sure you are constructing an AMOApplication
- Do not use CompletableFuture
- While the labs recommend IntelliJ, it isn't necessary to use it. Feel free to use your favorite IDE/text editor. (vim)
 - There might be some weird saving issues with IntelliJ where it says it saved some change, but the version saved on the machine doesn't actually reflect that change, so watch out for that. Try checking with git status/git diff.
- If you have import errors on your IDE, add paths to the JARs provided

Lab 1 more help/hints

- Don't use a Hashtable (see framework docs for Node)
- Read the framework docs under docs
- You can follow the structure for Lab 0
- If you see effects from previous tests, make sure that you don't have static variables.

Section 2: Lab 2 Intro (ViewService)

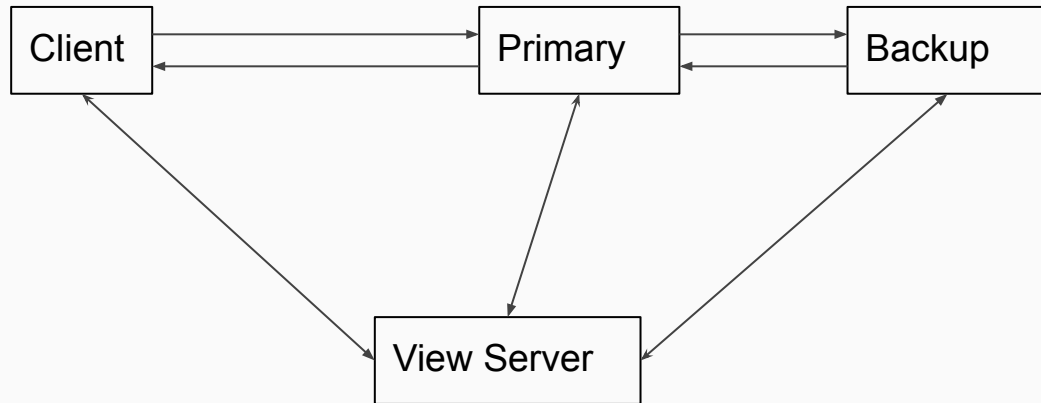
CSE 452 Spring 2021



Who are the players?

- ViewServer
- Primary
- Backup
- Other servers waiting in reserve...
- Clients

General Flow



What's a view?

View 1

Primary = A

Backup = B

View 2

Primary = B

Backup = C

View 3

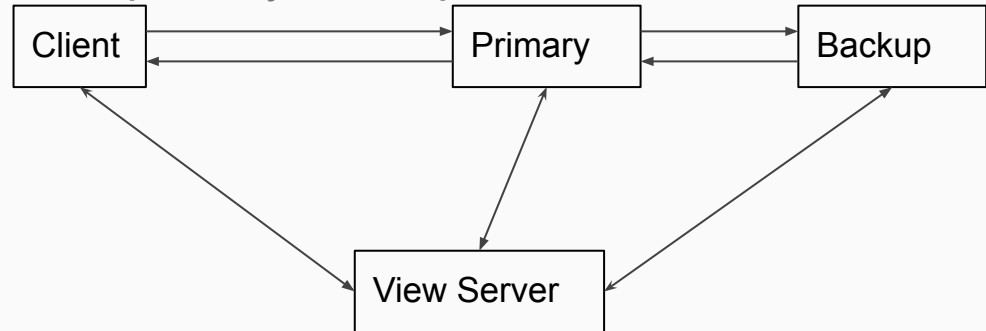
Primary = C

Backup = D

```
@Data
class View implements Serializable {
    private final int viewNum;
    private final Address primary, backup;
}
```

View Server

- Controls who is primary, who is backup, called a View
- System goes through sequence of views (increasing view number)
- Servers and clients contact the view server to learn identity of primary and backup
- Transitions between views ONLY when the current primary has acknowledged the current view (avoid split-brain)
- View server needs to “know” when primary/backup fail
- Single point of failure :(



Pings and Server failure

Heartbeat messages : Ping RPCs. Informs the View Server:

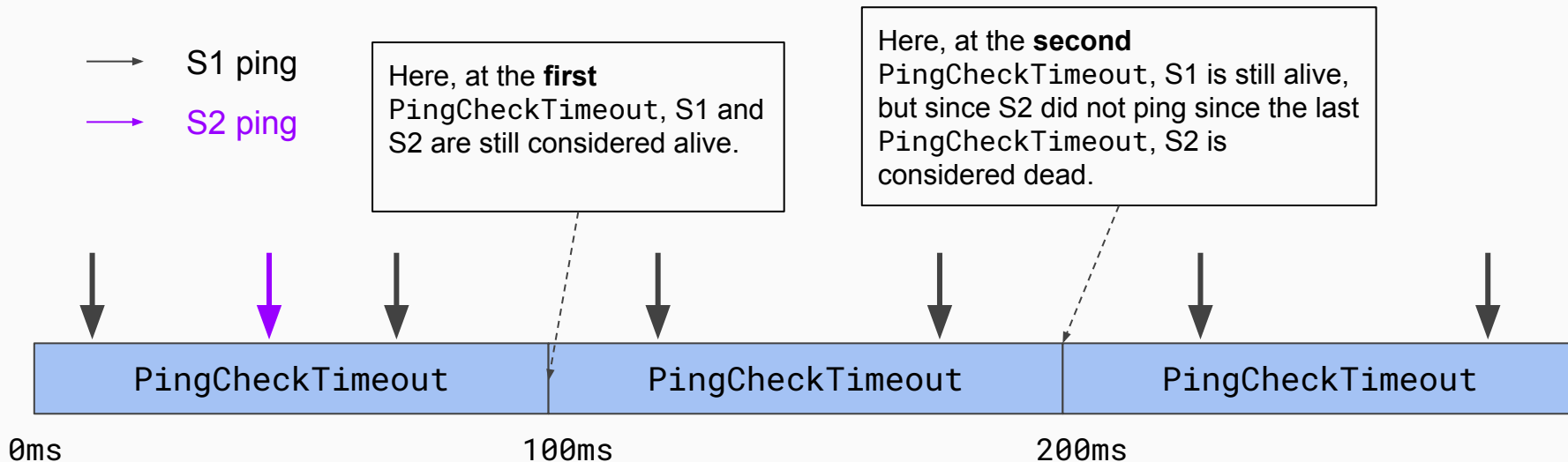
- 1) confirmation that the server is alive
- 2) the most recent view the server knows (why?)

Also lets the view server inform the server of the current view

Detect failure by absence of pings:

- Not received ping from server for some amount of time
 - For lab 2: Do **NOT** store timestamps on the view server! Needs to be deterministic for search tests.

Explaining PingCheckTimeout



"If the ViewServer doesn't receive a Ping from a server in-between two consecutive PingCheckTimers, it should consider the server to be dead"

The first interval in which it's alive counts as one of the two!

When can a view transition occur? (1)

- First view is (STARTUP_VIEWNUM): {null, null}
- The first ping of some server (server A) should result in transition of startup view to INITIAL_VIEWNUM (should be {primary=A, null})
- View INITIAL_VIEWNUM+1 should be {primary=A, backup=B} if there is a backup (server B) available
- Primary acknowledges first non-null view (INITIAL_VIEWNUM) with its own ping
 - What if a server has pinged since?
 - Should be added as backup when the primary acknowledges (In other words, transition to a new view)

When can a view transition occur? (2)

From start-up view (special case):

- Can transition to initial view

Any other view (general case):

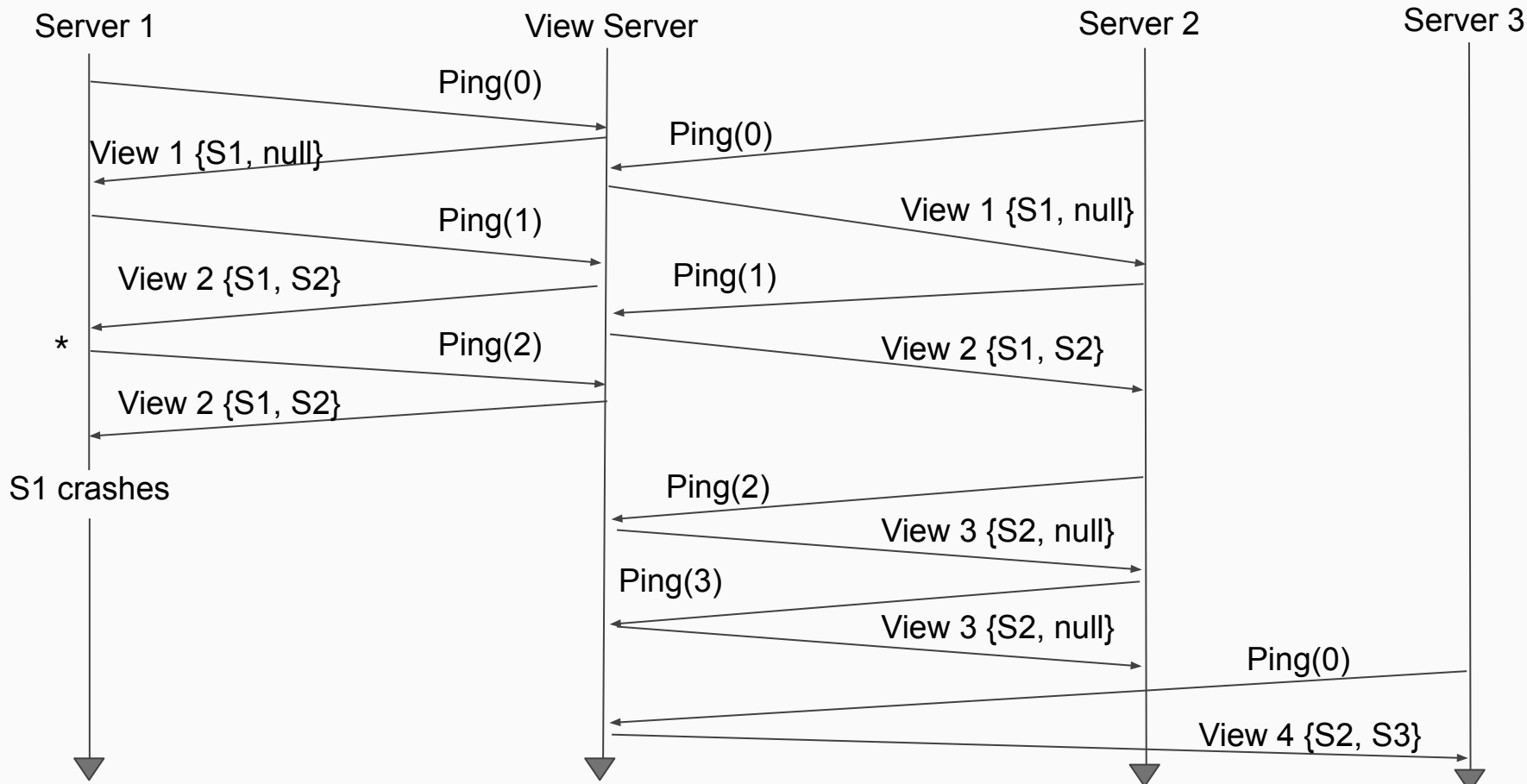
- The current view must have been acknowledged by the primary before the ViewServer can transition to the next view
 - If the primary has not acknowledged the current view yet, the ViewServer may not move onto another View

View Transition Timeouts

- Only move to a new view (i + 1) if the primary of view (i) has acknowledged view (i)!
- What happens if the primary fails? (assume current view has been ack'd)
 - Backup becomes new primary and try to get another backup if you can in View i+1
- What happens if backup fails?
 - New server should become backup or null backup in View i+1
- What happens if the primary fails with no backup?
 - Just do nothing - hope that it comes back
- What happens if both primary and backup fail?
 - Cry :(-> Also do nothing

Example Call Flow Diagram

* S1 sends application state to S2 and gets an ack back before Ping(2), acknowledging the view



Primary and Backup

- Client asks the View server for the latest view.
- Only the Primary responds to the client.
- Can a non-primary server get a client request? What should it do?
- Primary should pass requests to the backup and receive an ack before executing and responding to the client. What are some issues that can happen?
- What needs to be done when primary has a new backup?
 - Transfer state to backup
 - TIP: Send entire Application in a new message type
 - Ignore requests until state transfer complete

Remember the 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

Section 3: Lab 2B

CSE 452 Spring 2021



Things to design

- Messages
 - Request / Reply
 - “Copy”/Forward message from primary to backup
 - State transfer
 - Acks?
 - Others?
- Timer handlers
 - Ensure proper checks on timer handler, avoid calling set() if response to the message the timer was set for was received successfully
- State needed to keep for PBClient/PBServer
 - AMOApplication (only server)
 - Sequence number (on client)
 - Current View

General tips

- Primary should send commands to backup, backup should process command; send response to primary; primary processes the command; then the primary responds to client
 - What should primary do if it gets a request for a new command before it receives an acknowledgement for the previous command from the backup?
 - Dropping requests is simplest option
 - We recommend just processing one request at a time
 - Ensure primary and backup agree on current view
- **Send view with every message** and check viewNum on each receive
- Non primary should reject any requests!
- Simplify states as much as possible, and don't use more timers than necessary

State Transfer - Things to keep in mind

- If primary receives a new view with a backup - need to do state transfer
- Include all data (gets, puts, appends) and all RPC history
 - Sending entire AMOApplication in a message should be sufficient
- Can't process any requests while state transfer in progress (why?)
 - Primary should drop requests while state transfer in progress; client will retry
- Backup can receive duplicated/late state transfer messages (i.e. if state transfer message is duplicated/come later).
 - **Ensure that state on backup is only overwritten once per view change**
 - **What happens if the ack gets dropped?**
- Usual retry logic applies
 - State transfer messages and acks can be dropped
- Pings from primary during state transfer should reflect old view.
 - So the primary only moves to new view once state transfer is complete.

Additional Lab 2 hints/help

- Follow tips on slides
- Make sure that the view server can only move to a new view after the current primary acks the current view
- Add timers only when necessary, i.e. if things need to be retried
- You might want to run the test suite multiple times in case there are some infrequent errors
- You might need to tune your timers, since it takes time to process messages, e.g. 10ms is generally unreasonable

Even more Lab 2 hints

- If you fail search tests due to not being able to find the initial view:
 - Try running the visual debugger and making sure that you can find the initial view by taking some series of steps
 - Make sure that your View isn't static
- If you fail search tests due to not finding all client workloads finished:
 - If you have already executed a request, is it necessary to forward the request to the backup and wait for an ack?
 - Try only getting a new view if the server hasn't responded in a few timeouts on the client
- Things that can increase the number of states to explore:
 - Every state transition: setting new timers, sending new messages
 - Unnecessary state information, e.g. a retry counter that just keeps incrementing

Some debugging tips

- Don't use logging (-g FINEST) for the search tests
 - The search tests do a BFS/DFS through search states, so you may see inconsistent messages
- To debug things taking too long, you can run it with logging (-g FINER or -g FINEST) and write it to a file using &>
 - Look for stretches of text where the system makes no progress, e.g. repeated timers/handlers and no replies
- To debug could not find all client workloads finished on search tests
 - Run a single test with --checks and fix any errors that show up
 - Optimize your code, make sure that your implementation is minimal and clean
 - Reduce the number of timers / messages you're using to generate less state
 - Each message handler/timer handler transitions to a new state
- **To debug can't find view in a search test or otherwise**
 - Try walking through the visualizer to find the view the test is looking for.
 - `./run-tests.py -l 2 --debug 2 1 "PUT:foo:bar,APPEND:foo:baz,GET:foo"`

Other common issues

- Think about what should happen when you get duplicated/dropped messages:
 - State Transfers
 - State Transfer acks
 - Forward Requests
 - Forward Replies
 - View Replies
- Just because you pass the ViewServer tests, it doesn't mean it's 100% right
 - You shouldn't be going from (primary, null) -> (null, null)

Pro Tips

1. If you're failing test 19 and only test 19, try removing maps in the view server if you have any
2. don't hold a 'curr view' and 'next view' in the PBServer. You will make it much harder for yourself than it needs to be. Consider just pinging with the old view number or not pinging at all during a state transfer.
 - in `handleViewReply`, we don't care what the node WAS, we only care about what it is becoming in the new view, so it's unnecessary to store more than 1 view

Some other tips for search tests

- This means you're generating too many states
- Might be due to not having a clone/hashCode>equals
 - Try running --checks
 - Fix any equals or hashCode errors
 - Sometimes it's okay for things to be idempotent. It's better if they're not, but sometimes it's okay.
- Possibly due to receiving multiple copies of a message producing multiple messages/timers
 - E.g. getting the same handleViewReply produces multiple state transfer timers/messages
- Servers should ping the ViewServer to get a view
- Clients should send a GetView if they haven't received a response from the Primary within some number of tries. This logic should be contained in onClientTimer



Idk if these are helpful at this point

If you get not “all clients results are the same”

- Make sure that state transfers only happen once per view
- Make sure that you only handle a single request at a time
 - You should check the forward replies and make sure that it's for the same command as the one you're waiting for



If you're timing out on test 14/15

Try putting print statements on view changes and when you handle results on the clients as a point of departure for debugging

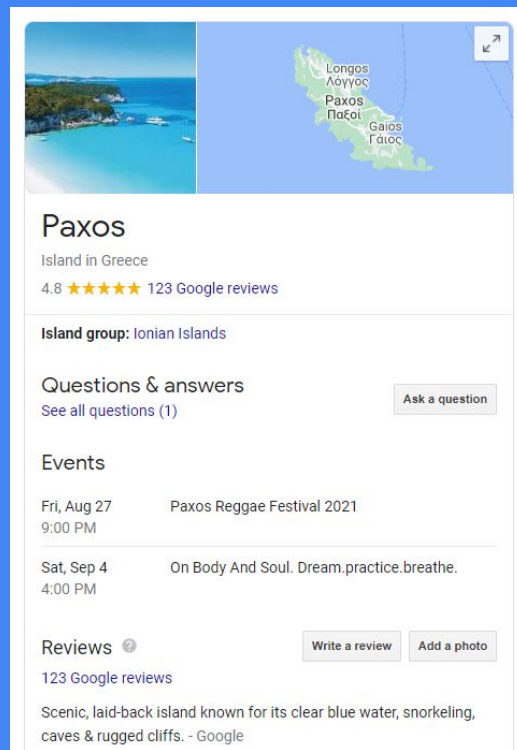
You can reply right away to alreadyExecuted commands, so you don't need to forward it to the backup and get an ack back

(Single Instance)

v

Section 5: Paxos

CSE 452 Spring 2021



The screenshot shows a Google search result for 'Paxos'. At the top, there is a landscape photo of a beach and a map of the Ionian Islands with Paxos highlighted. Below the images, the title 'Paxos' is followed by 'Island in Greece' and a rating of 4.8 stars from 123 Google reviews. The 'Island group' is listed as 'Ionian Islands'. There are sections for 'Questions & answers' (with 1 question), 'Events' (listing 'Paxos Reggae Festival 2021' and 'On Body And Soul'), and 'Reviews' (with 123 reviews and a snippet: 'Scenic, laid-back island known for its clear blue water, snorkeling, caves & rugged cliffs. - Google').

Paxos
Island in Greece
4.8 ★★★★★ 123 Google reviews

Island group: Ionian Islands

Questions & answers
See all questions (1) [Ask a question](#)

Events

Fri, Aug 27 9:00 PM	Paxos Reggae Festival 2021
Sat, Sep 4 4:00 PM	On Body And Soul. Dream.practice.breathe.

Reviews ⓘ [Write a review](#) [Add a photo](#)
123 Google reviews
Scenic, laid-back island known for its clear blue water, snorkeling, caves & rugged cliffs. - Google

Paxos Made Somewhat Simpler
But Not Really That Much Simpler
Because It's Still Complicated

On a high level, Paxos works with majorities (more than half) and promises

- What is in a majority gets propagated
- Promise to not accept anything lower is maintained

Motivation

- Consensus: want to decide on a single value among multiple distributed nodes
 - Each node is a replica of a state machine (in our case a key/value store)
- Want to be able to do this even if some nodes fail
 - Should be able to continue operation if a majority are alive
- Want to be able to do this **asynchronously**
- All servers must execute all client requests in the same order [Linearizability]

Today...

We're talking about a single instance of Paxos:

- How to decide on a single value

Next week, Multi(-instance) Paxos [What you're doing for Lab 3!]:

- How to decide on multiple values

Proposer Protocol [proposer -> acceptors]

A proposer chooses a new proposal number **n** and sends a request to each member of some (majority) set of acceptors, asking it to respond with:

- a. a promise never again to accept a proposal numbered less than **n**, and [acceptor won't accept things with lower proposal numbers]
- b. the accepted value associated with the highest proposal number less than **n** *if any*. [Basically what it has currently accepted]

...call this a **prepare request** with number **n**

Proposer Protocol [proposer \leftarrow acceptors]

If the proposer receives a positive response from a majority of acceptors, then it can issue a proposal with proposal number, n , and value, v , where v is

- a. the value associated with the highest proposal-numbered response among the responses, or
- b. is any value selected by the proposer if all responders in the majority had not accepted any previous values

A proposer issues a proposal by sending, to some set of acceptors, a request that the proposal be accepted called an **accept request**.

Acceptor Protocol [acceptor \leftarrow proposer]

An acceptor receives prepare and accept requests from proposers. It can ignore these without affecting safety.

- It can always respond to a prepare request
- It can respond to an accept request, accepting the proposal, iff it has not promised not to, viz.:

P1a: An acceptor can accept a proposal numbered n iff it has not responded to a prepare request having proposal number greater than n

Paxos in 25 lines

```
--- Paxos Proposer ---
```

```
1 proposer(v):
2   while not decided:
3     choose n, unique and higher than any n seen so far
4     send prepare(n) to all servers including self
5     if prepare_ok(n, na, va) from majority:
6       v' = va with highest na; choose own v otherwise
7       send accept(n, v') to all
8       if accept_ok(n) from majority:
9         send decided(v') to all
```

```
--- Paxos Acceptor ---
```

```
9 acceptor state on each node (persistent):
10 np      --- highest prepare seen
11 na, va --- highest accept seen

12 acceptor's prepare(n) handler:
13   if n > np
14     np = n
15     reply prepare_ok(n, na, va)
16   else
17     reply prepare_reject

18 acceptor's accept(n, v) handler:
19   if n >= np
20     np = n
21     na = n
22     va = v
23     reply accept_ok(n)
24   else
25     reply accept_reject
```

Example Key

Messages

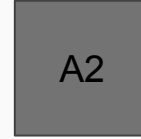
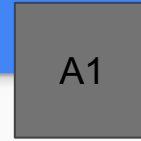
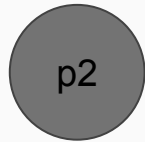
- $P(\mathbf{n})$: prepare request with proposal number \mathbf{n}
- $PR(\mathbf{n}, \mathbf{v})$: prepare reply with \mathbf{n} as the highest proposal number accepted and \mathbf{v} as the corresponding accepted value
- $A(\mathbf{n}, \mathbf{v})$: accept request with proposal number \mathbf{n} and value \mathbf{v}
- $AR(\mathbf{ok})$: accept reply with either "accept" or "reject"

Acceptor State

- HP#: Highest Prepare Proposal Number Seen
- HA#: Highest Accepted Proposal Number
- HAV: Highest Accepted Value

P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



HP#, HA#, HAV

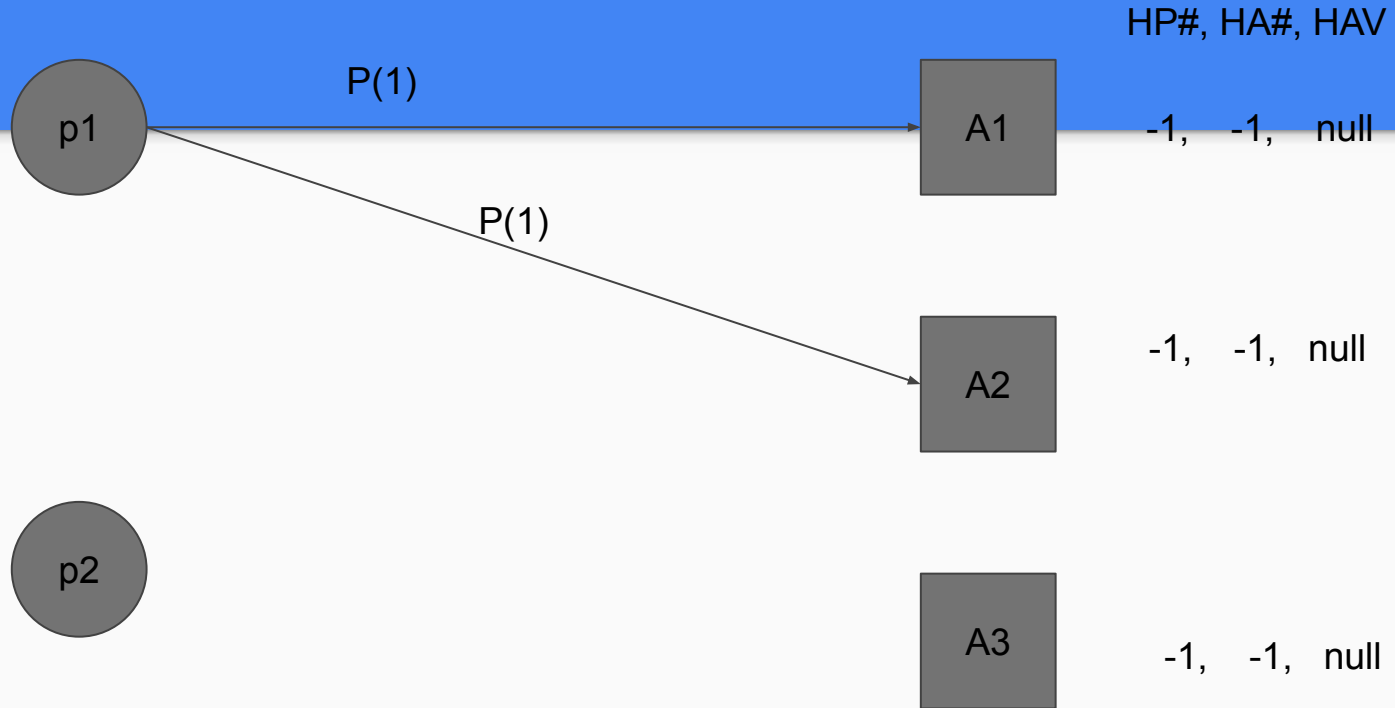
-1, -1, null

-1, -1, null

-1, -1, null

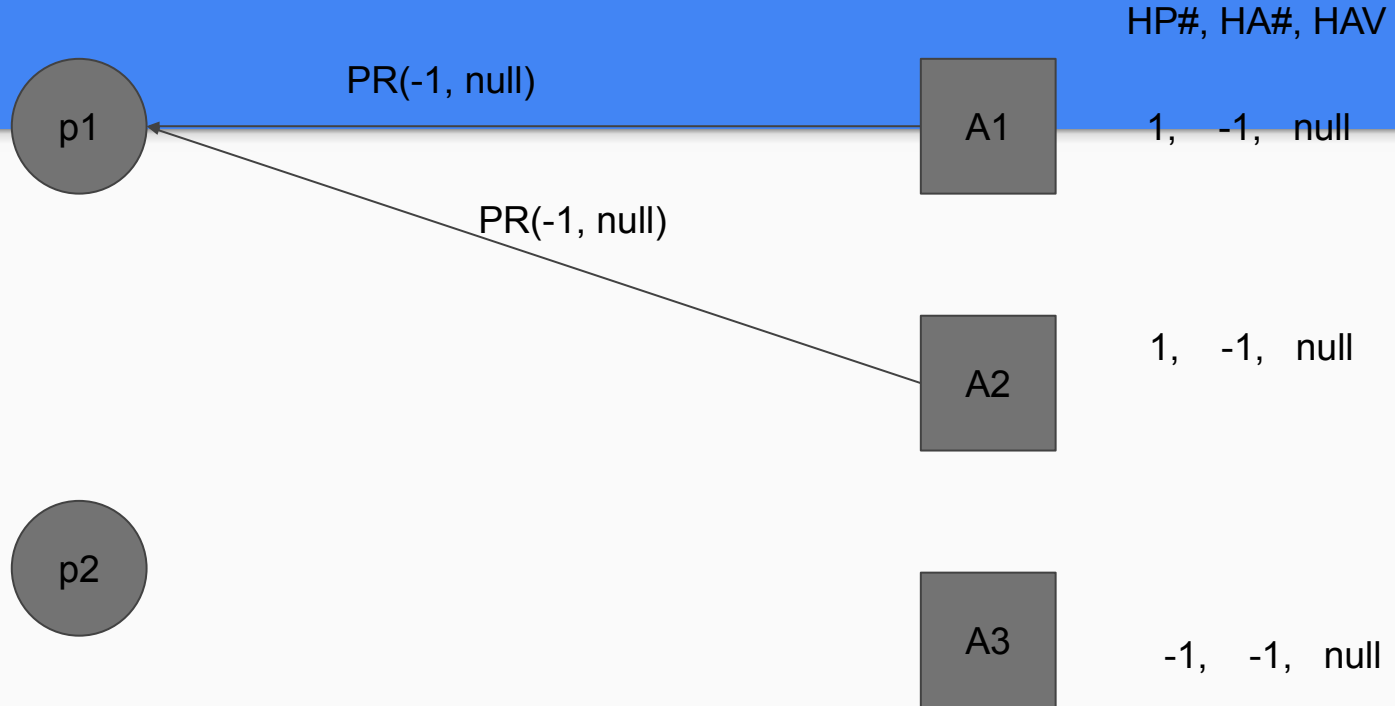
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



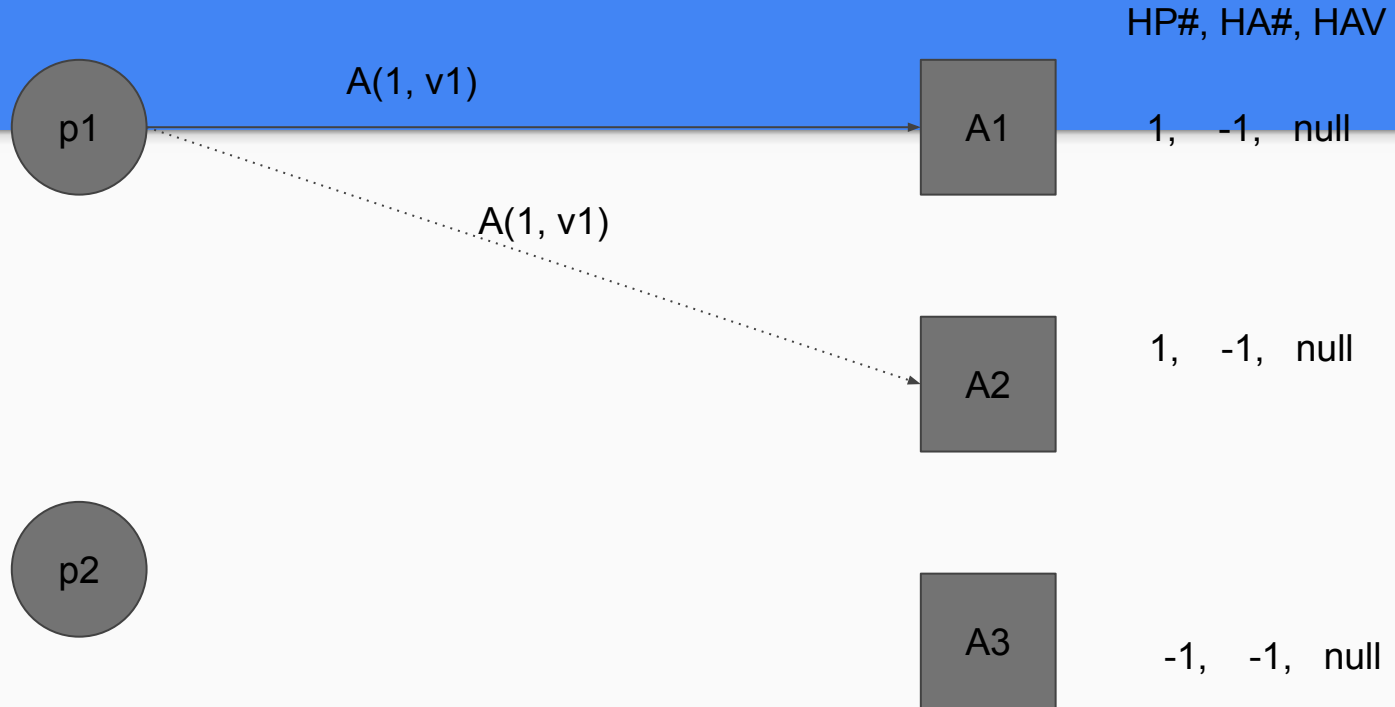
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



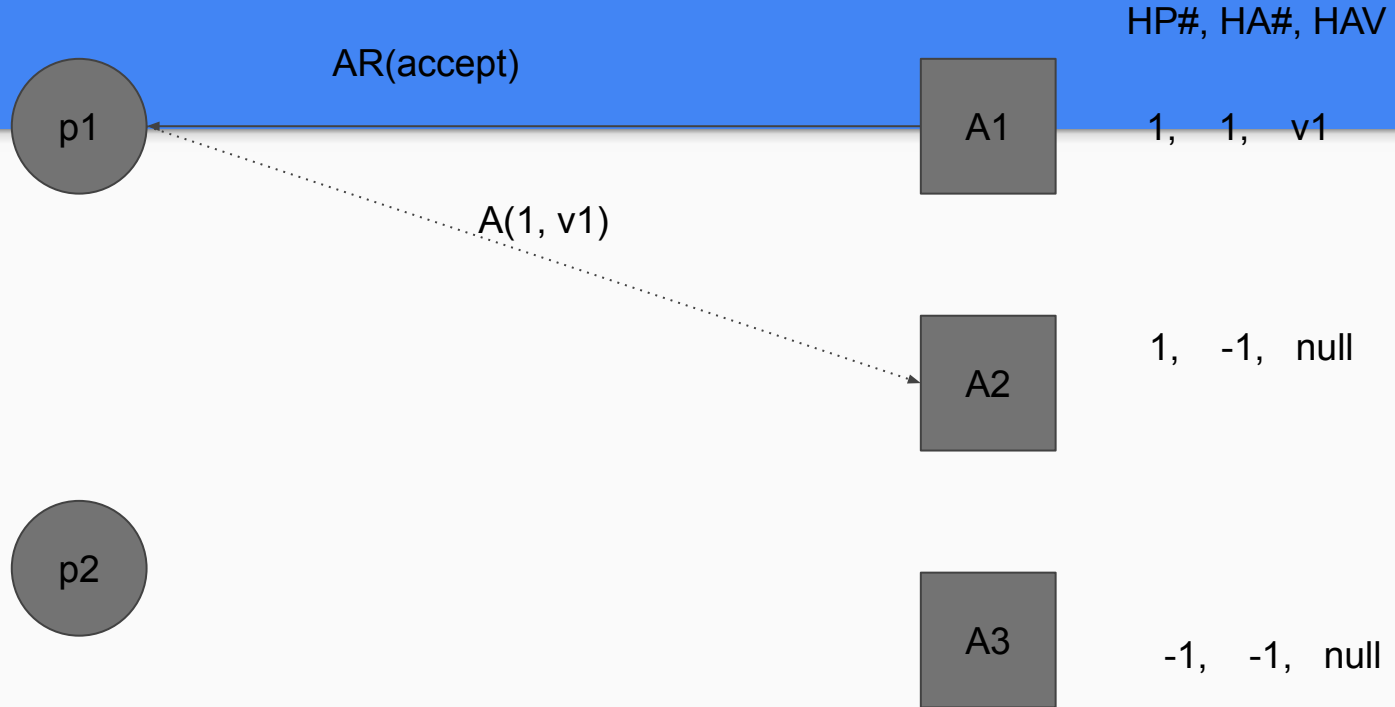
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



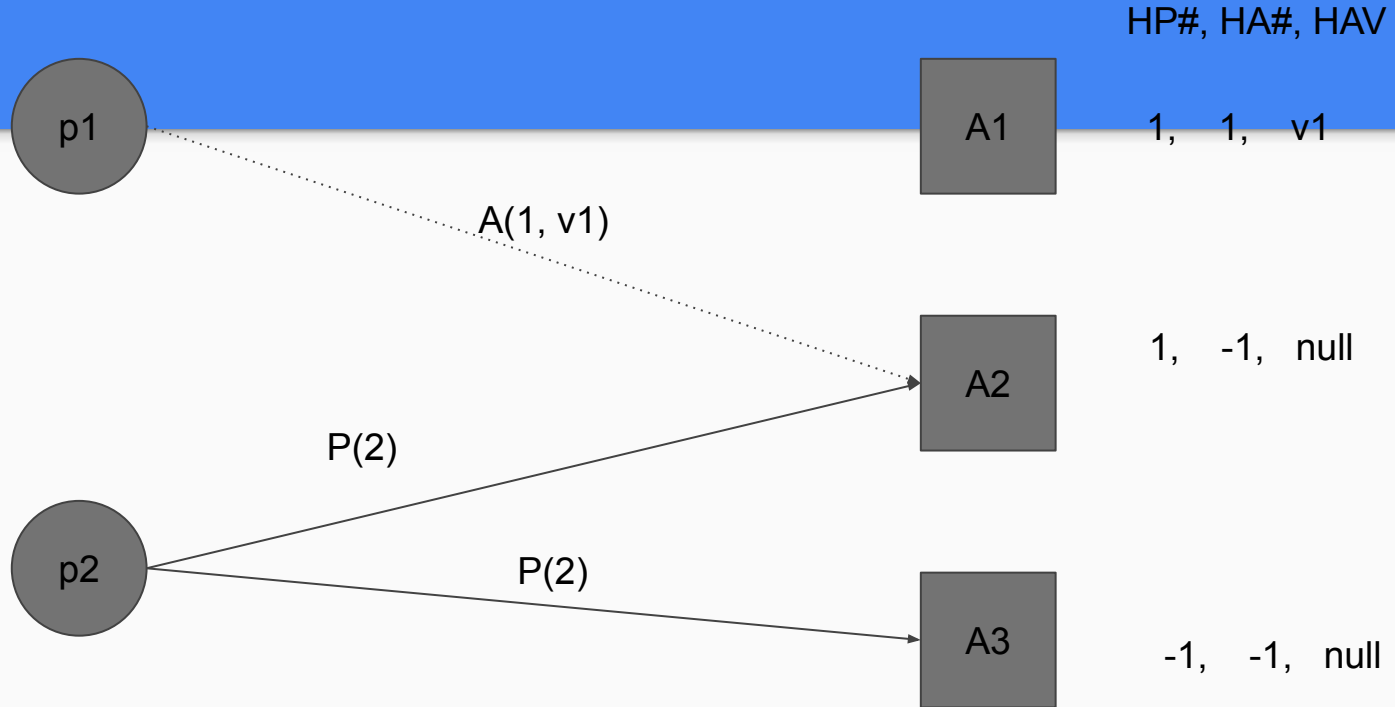
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



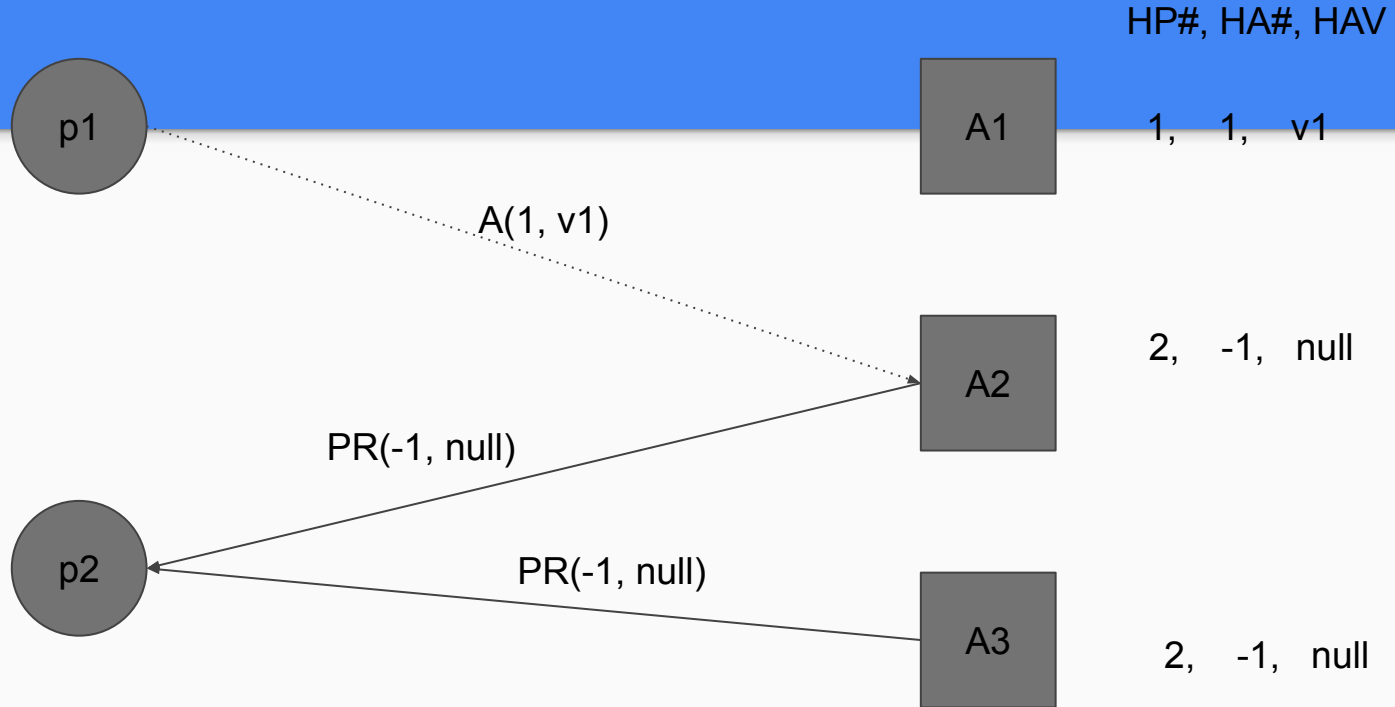
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



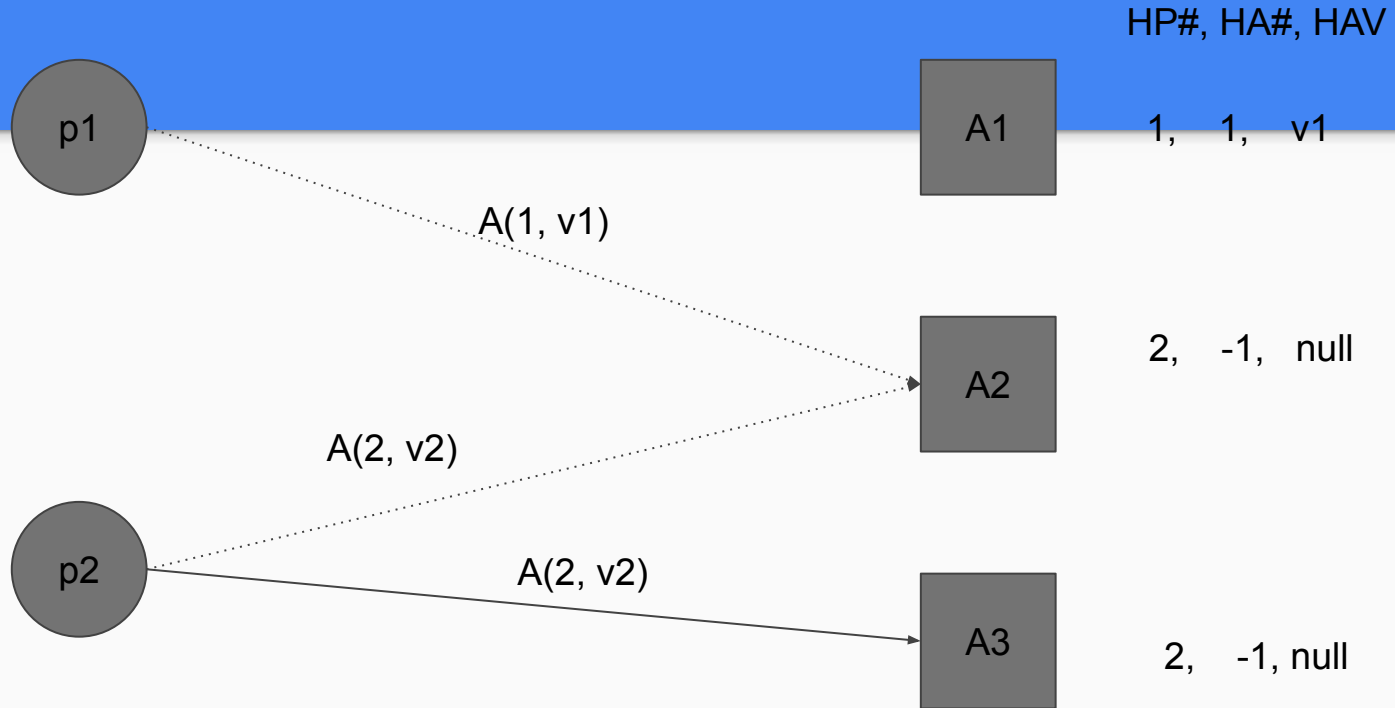
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



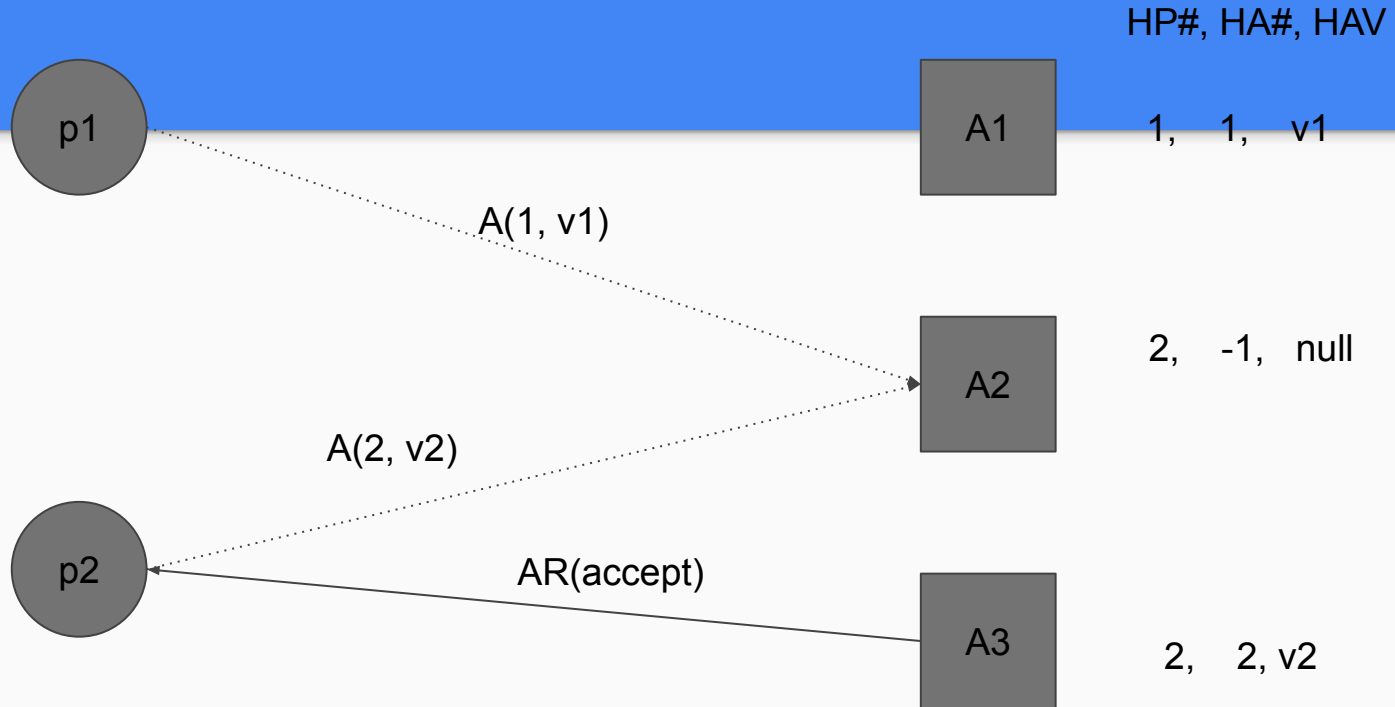
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



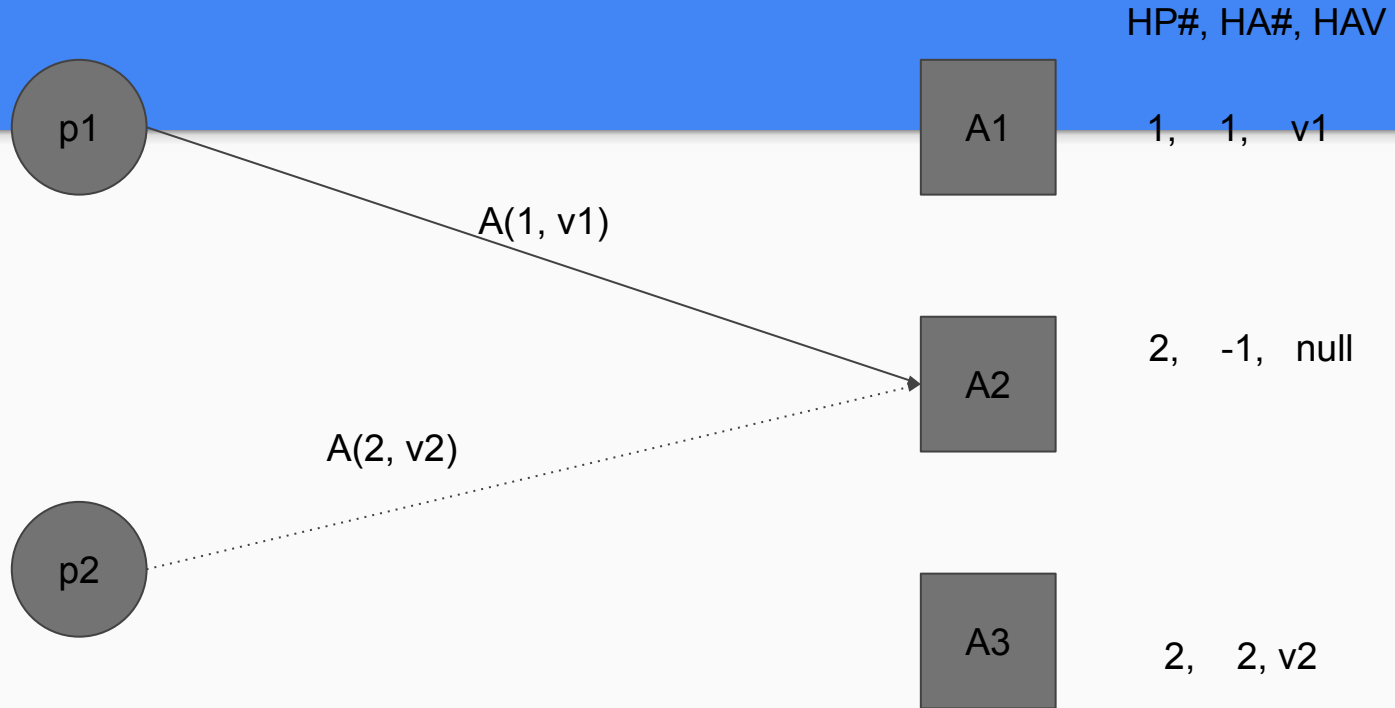
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



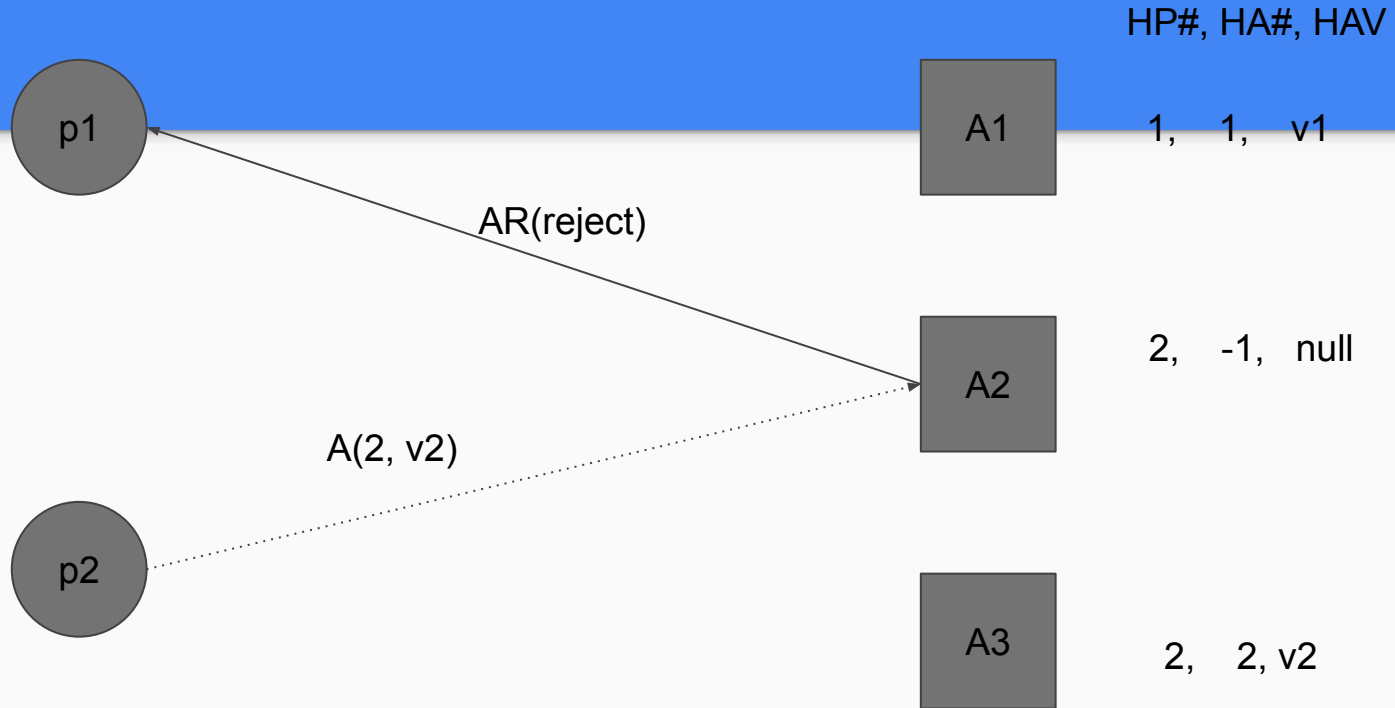
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



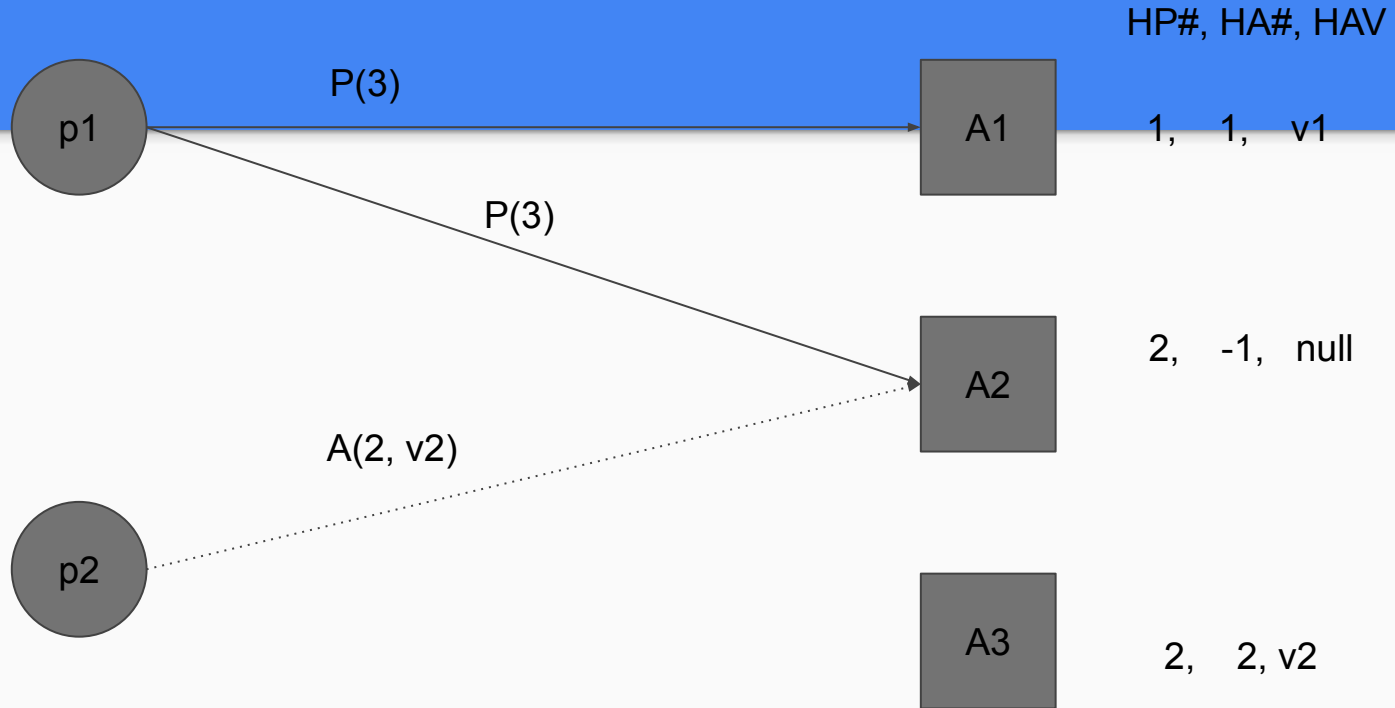
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



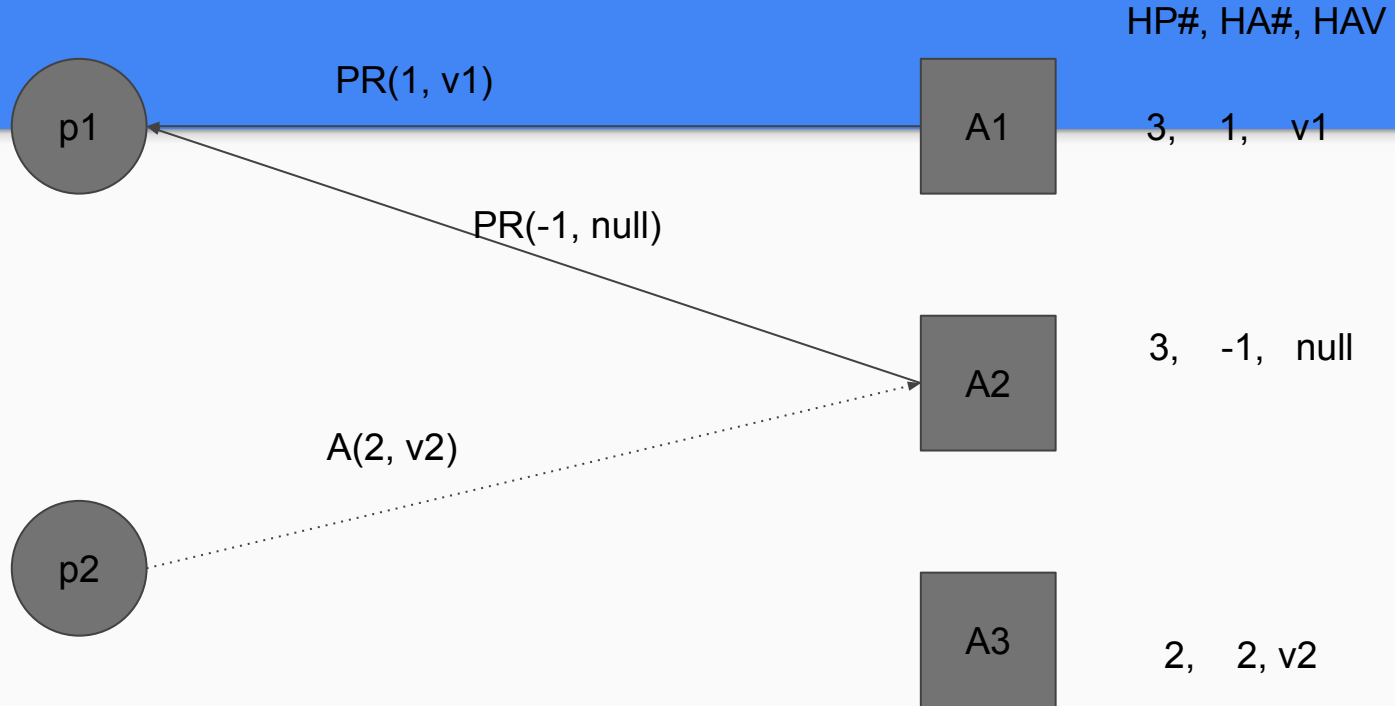
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



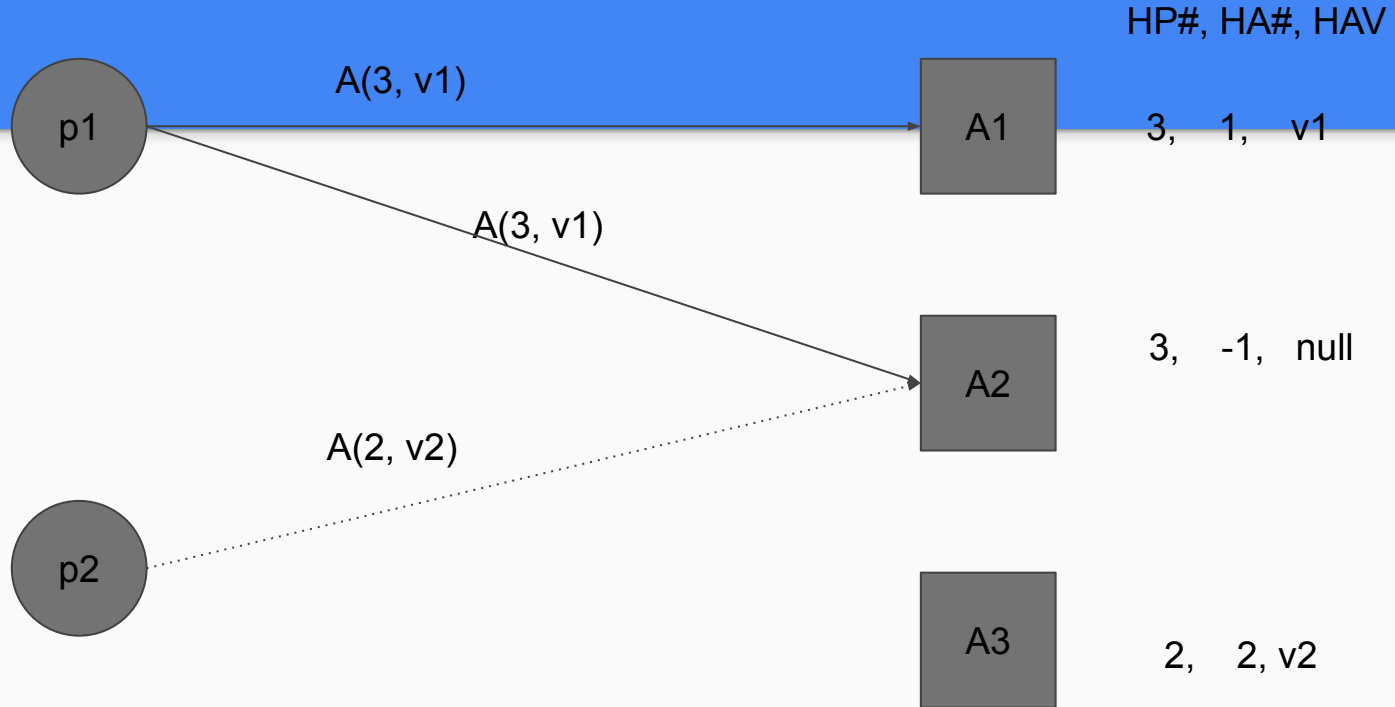
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



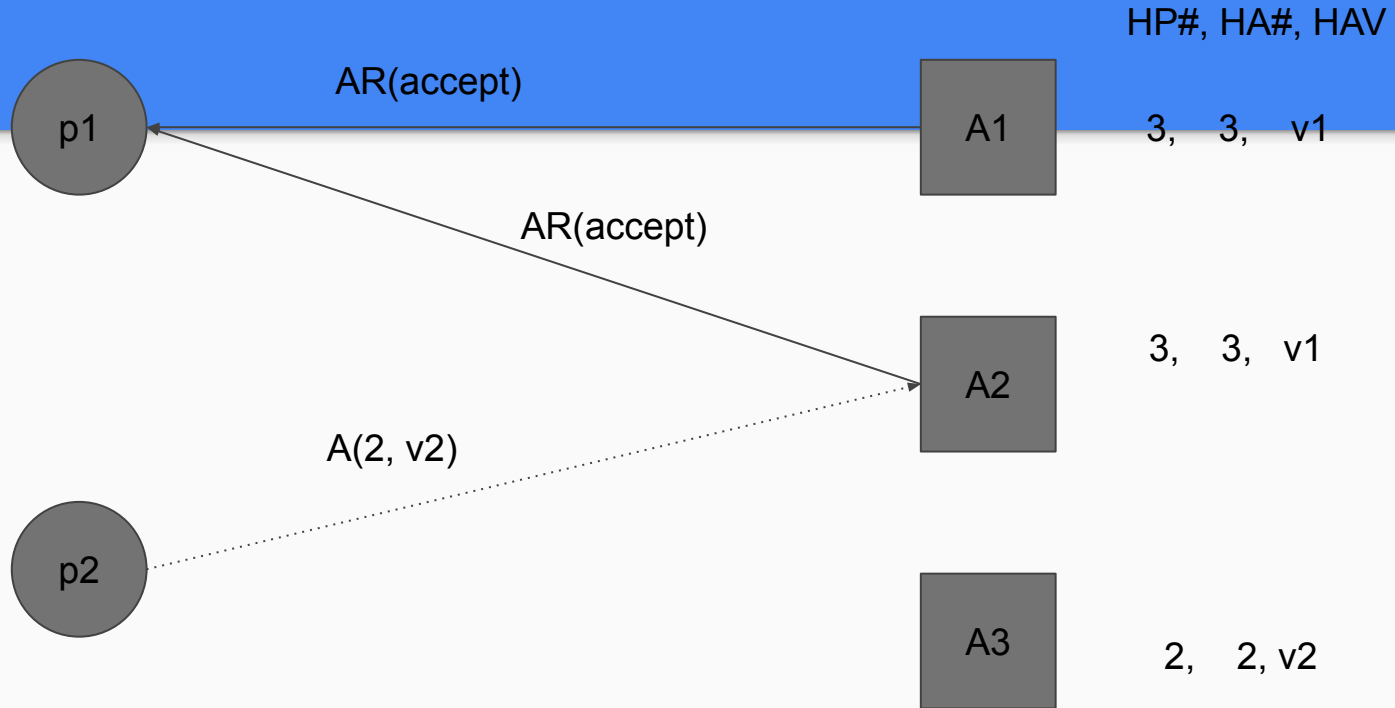
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



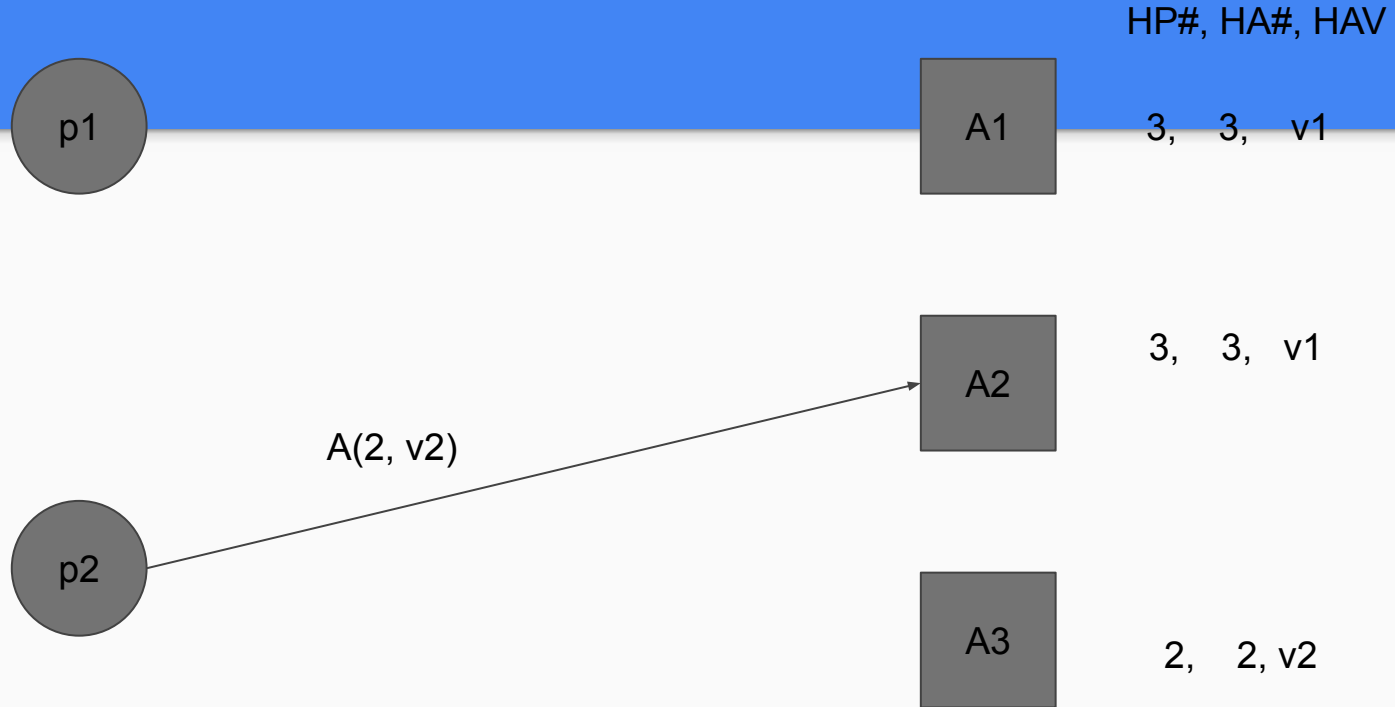
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



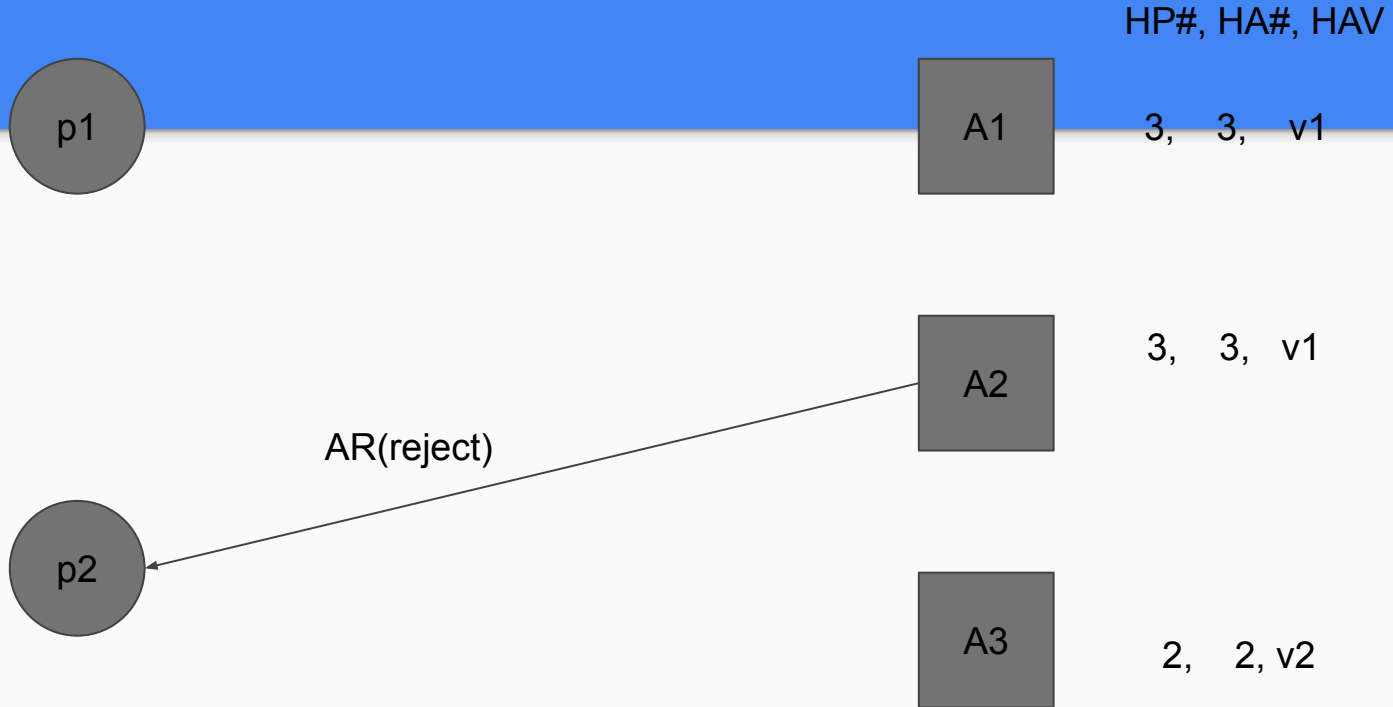
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



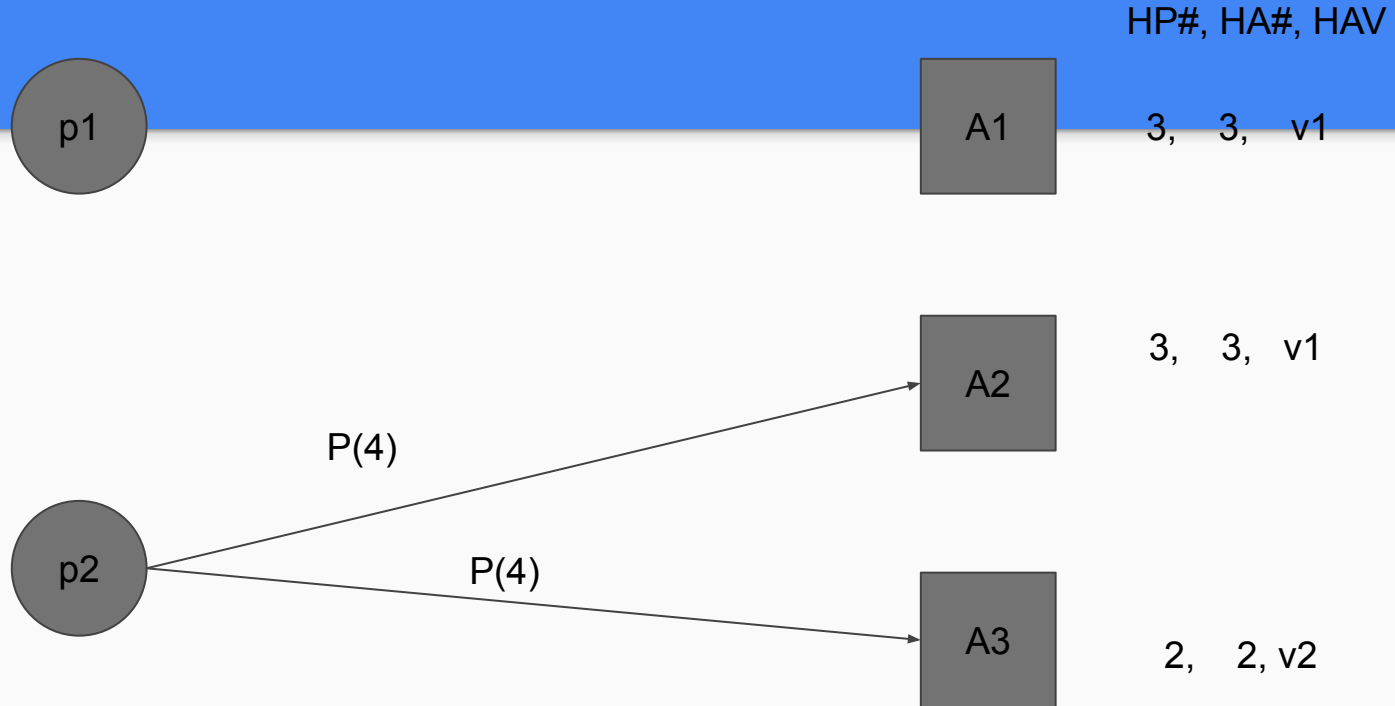
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



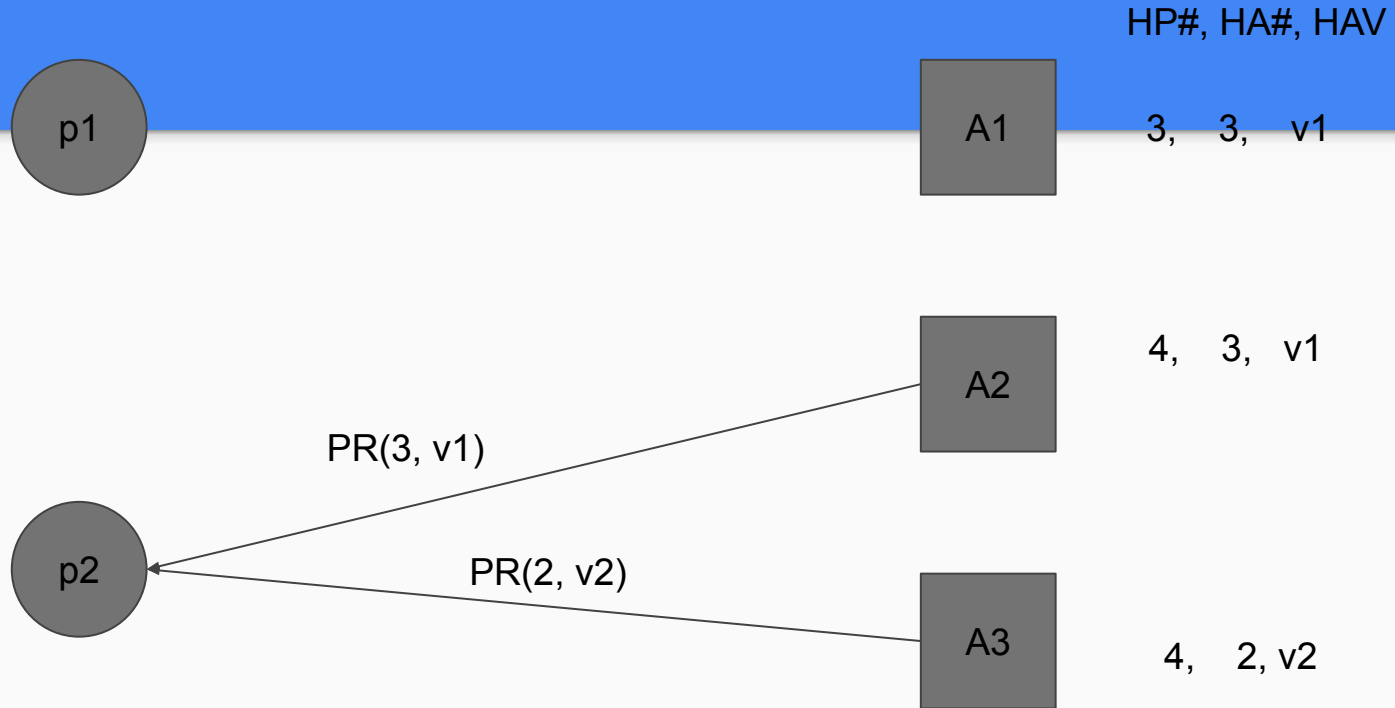
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



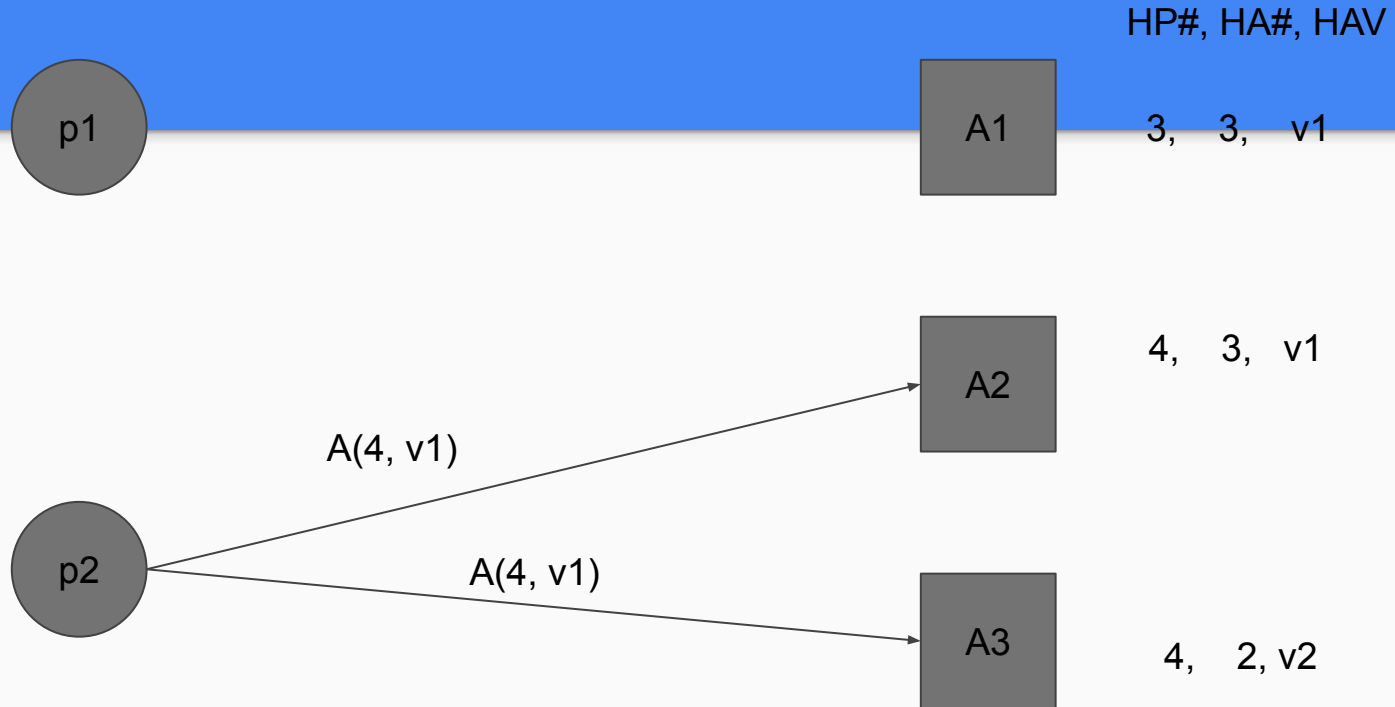
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



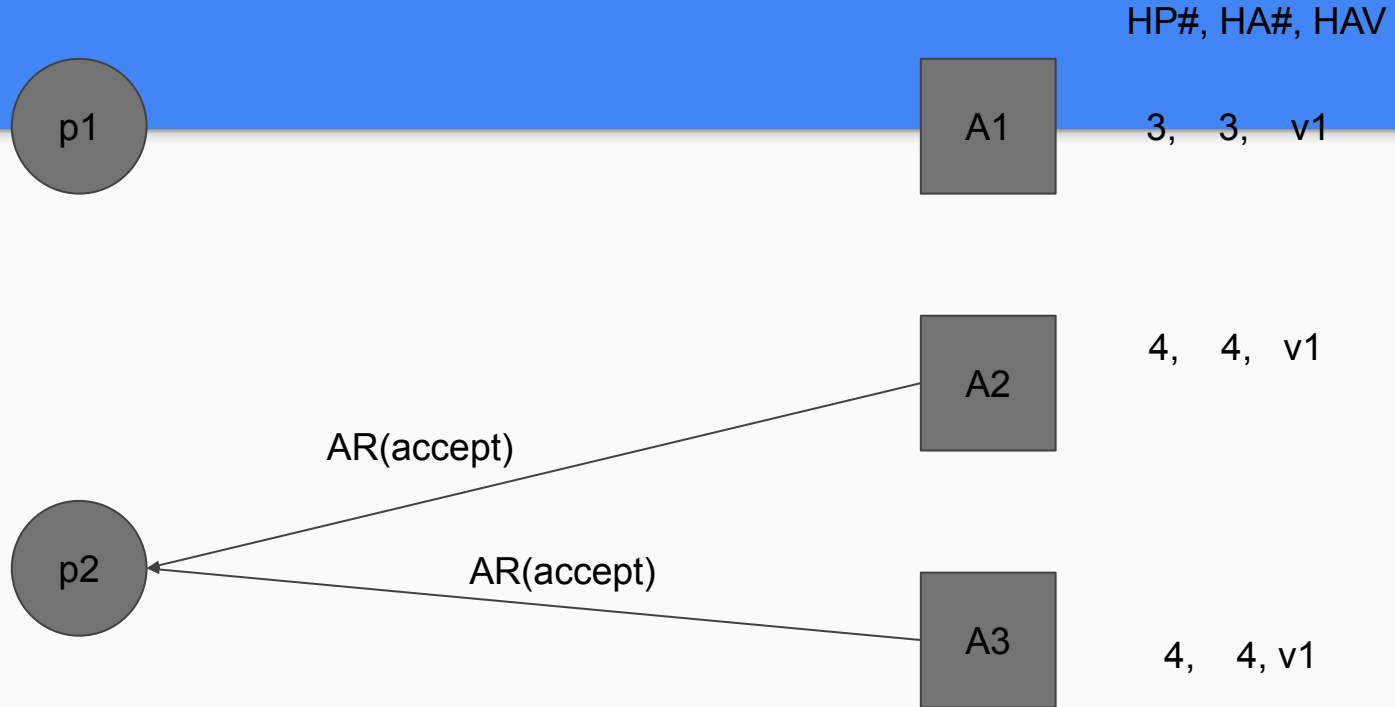
P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



P: Prepare Request
PR: Prepare Reply
A: Accept Request
AR: Accept Reply

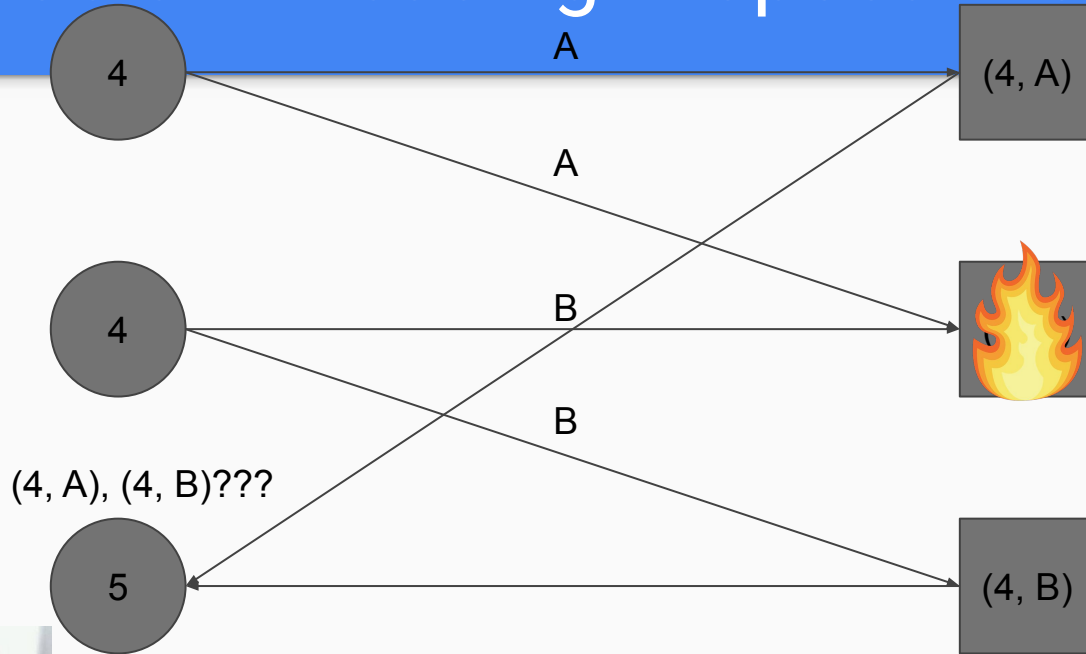
HP#: Highest Prepare Proposal Number
HA#: Highest Accepted Proposal Number
HAV: Highest Accepted Value



Consensus Value: v1



A Problem: Reusing Proposal Numbers



Guaranteeing Unique Proposal Numbers

- Proposers only use strictly increasing proposal numbers
- Each proposer is assigned a “slice” of all numbers
- Example:
 - P1: 1, 4, 7 ...
 - P2: 2, 5, 8 ...
 - P3: 3, 6, 9 ...
- Alternatively, make the proposal number unique by combining with the proposer's address
 - Server1: 1.1, 2.1, 3.1, 4.1, ...
 - Server2: 1.2, 2.2, 3.2, 4.2, ...
 - Server3: 1.3, 2.3, 3.3, 4.3, ...

Leader Election

- Allows the system to skip phase 1 (prepare) when there is a stable leader
- 2 possible designs (maybe more?)
 - Proposers immediately start first phase of Paxos when they are “preempted” (Design outlined in PMMC, but we don’t recommend doing this)
 - Can be inefficient - no bound on the number of preemptions that can occur
 - Case when proposers compete with each other with higher proposal number
 - Alternative: assume that the preempting node (higher proposal number) is the new “leader”. Wait for that node to drive the Paxos instance to a chosen value (or timeout).
 - Still need to handle multiple simultaneous requests!
 - Alternative 2: read the lab3 spec

More Examples

<https://web.archive.org/web/20160722025351/https://ramcloud.stanford.edu/~ongaro/userstudy/paxos.pdf>

Lab 3 Tips

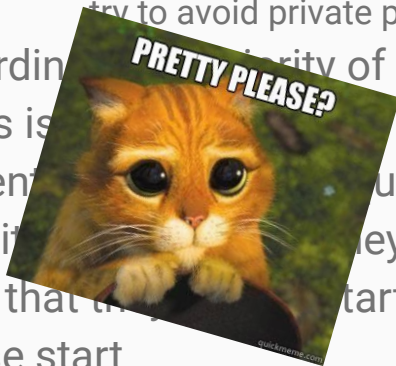
- We recommend using PMMC as a starting point, but...
 - Don't code up PMMC exactly (lab3 spec provides guidance)
 - Try to figure out what each part/role does and why it's there
- Every PaxosServer will be playing acceptor and learner/replica roles
 - If a server believes itself to be leader (or is trying to become leader), it will also play the (distinguished) proposer role
 - With regards to PMMC and implementing something similar, try to avoid making inner classes/subnodes for roles like Scout or Commander, since that may blow up the state space if you aren't careful
- Read the lab3 spec about “sending” a message from a server to itself
- If you receive a message with a higher ballot number than yours, stop being leader (or stop trying to become leader)
 - Not required for correctness, but makes it easier to decide on a value

General Tips/What previous students said

- Start early (applies to life too!)
 - Consider writing a design doc first!
 - But you might not consider/understand everything the first time around, so expect that you'll need to go over it a few times before it's right. Ask staff for clarifications! Also please try to avoid private posts because people might have similar issues/problems/questions.
- According to a majority of students in previous offerings/quarters:
"Paxos is non-trivial"
- Students from previous quarters also say to expect this to take more time than you think it would and that they wish they started earlier and that we emphasized to them that they should start earlier, hence this slide
- Please start

General Tips/What previous students said

- Start early (applies to life too!)
 - Consider writing a design doc first
 - But you might not completely understand everything the first time around, so expect that you'll need to go over it a few times before it's right. Ask staff for clarifications! Also please try to avoid private posts because people might have similar issues/problems/questions.
- According to the majority of students in previous offerings/quarters:
Paxos is
- Students in previous quarters also say to expect this to take more time than you think it will. They wish they started earlier and that we emphasized to them that they should start earlier, hence this slide
- Please start



Please Start

Other Resources

- This Google tech talk: https://youtu.be/d7nAGI_NZPk
- Stanford lecture: <https://youtu.be/JEpsBq0A06o>
- More about Paxos in a website: <http://paxos.systems/>
- Alternatively, you can do [Raft](#)
 - Basically Paxos, but more stuff is done on heartbeats and what you do for leader election is slightly different
 - Some people said it was easier to understand, but they likely spent as much time understanding it as they would've spent for Paxos
 - It's an option, but most of the staff implemented Paxos, so if you do do Raft, you likely won't get as much help

Section 6: Lab 3 (contd.)

CSE 452 Spring 2021



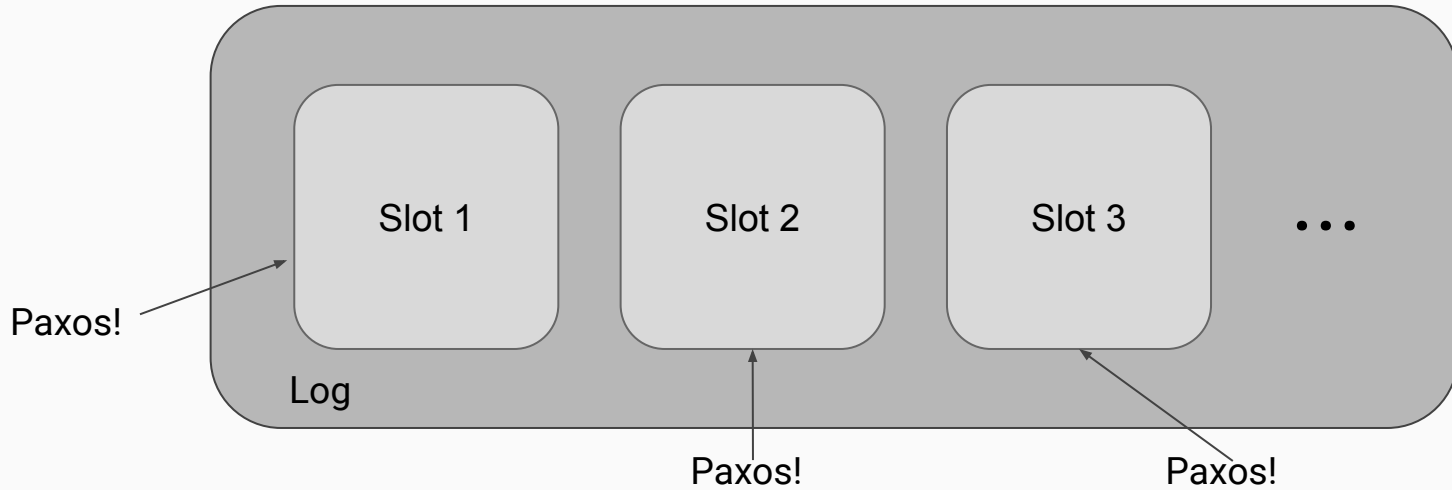
Basic Paxos -> Multi-Paxos

Last week:

- We know how to reach consensus on a single value (basic Paxos!)

This week (and what we're doing in lab 3):

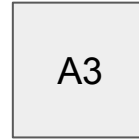
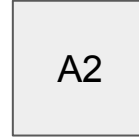
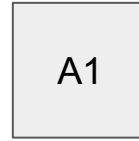
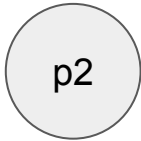
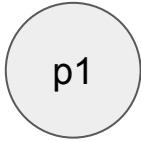
- Do Paxos, but instead of doing it once, do it multiple times; once for each index (or "slot") in a log
- Once a value (command) has become chosen we can try to execute it.
 - **Important:** Need to execute log **in order**, i.e. moving from slot to slot in order,
 - E.g. execute slot 1 first, then 2, then 3, ...
 - You can start executing from where you left off last time



Problem: Contention

If many proposers are proposing values for the same slot at the same time, problems can occur

- Acceptors may reject accept requests if proposal number isn't high enough
 - Proposers may retry with a higher proposal number
 - Proposers keep proposing with higher and higher proposal numbers
 - No progress is made in the system

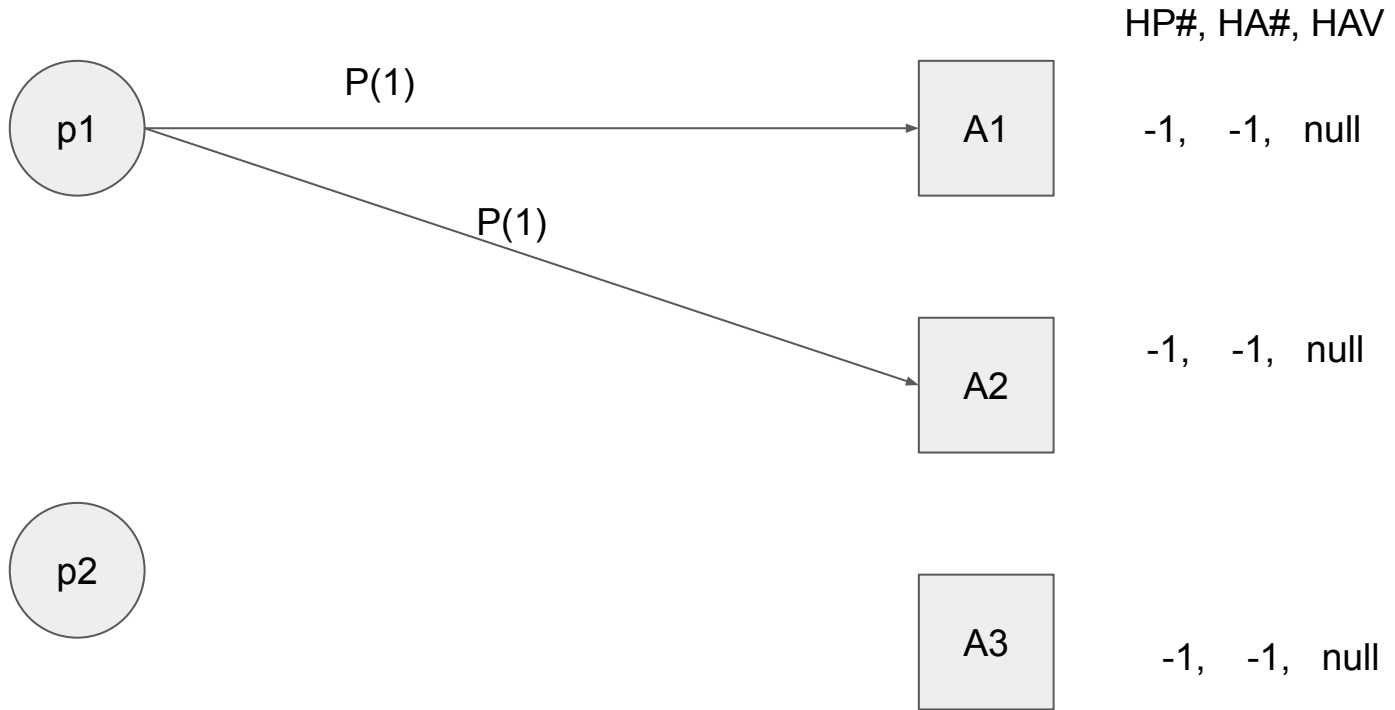


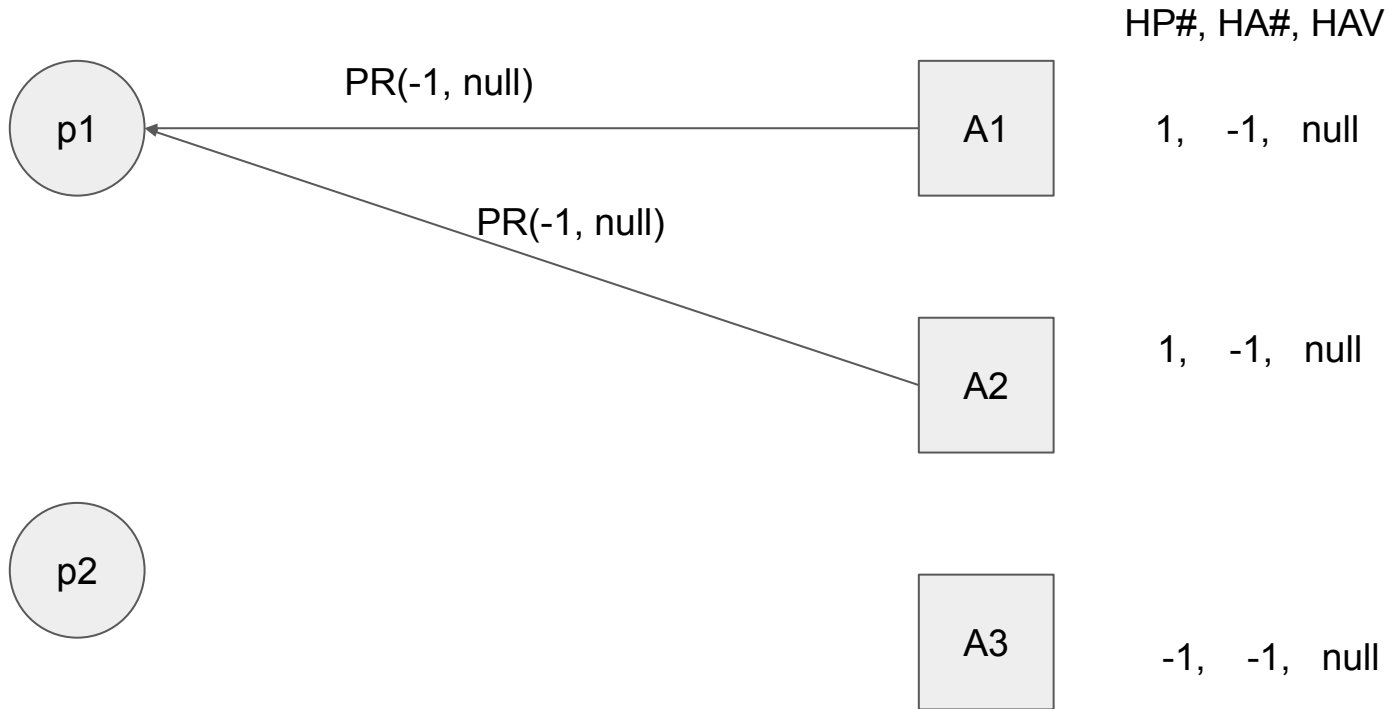
HP#, HA#, HAV

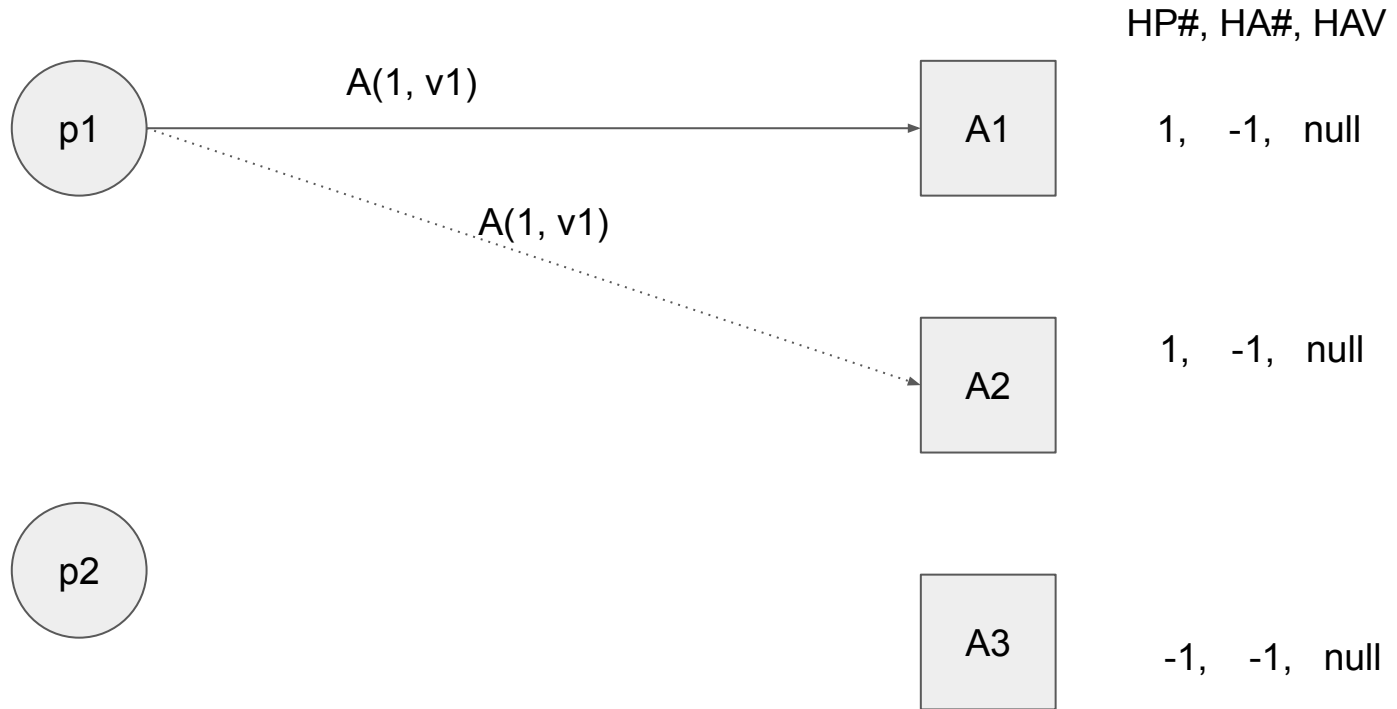
-1, -1, null

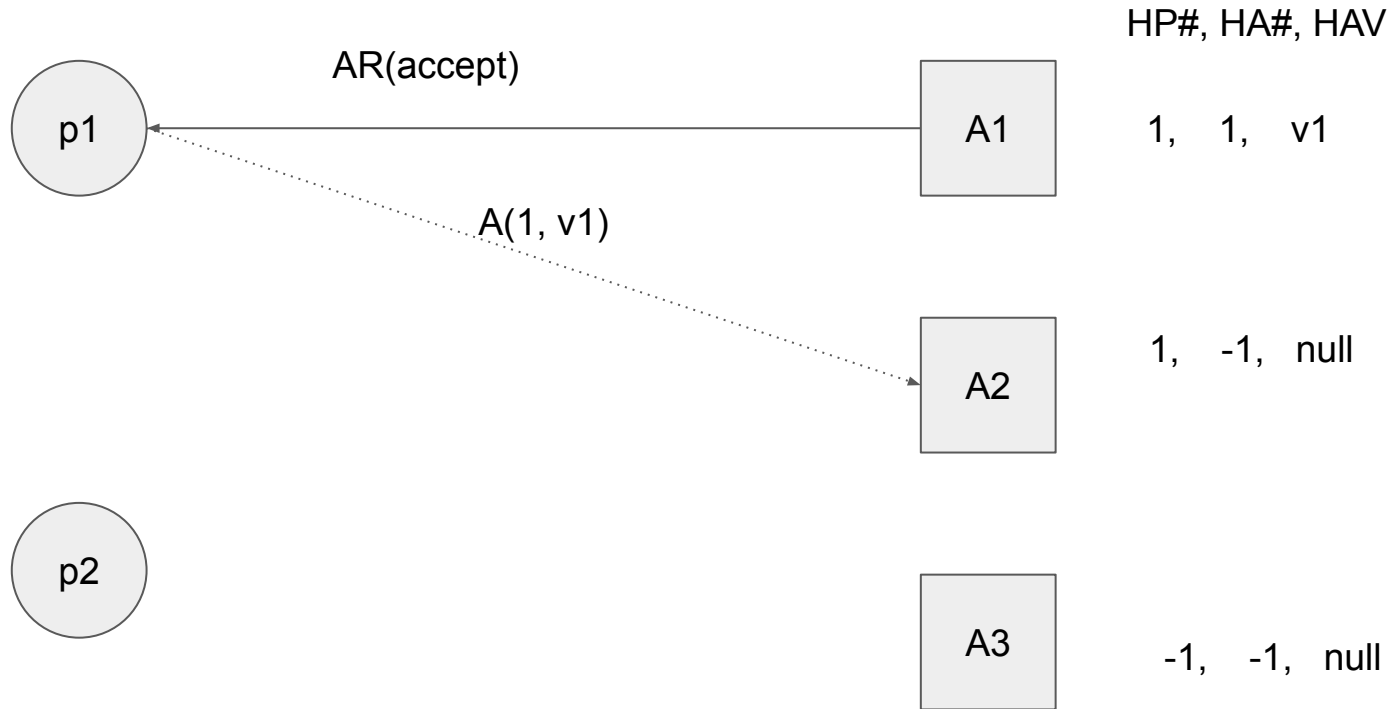
-1, -1, null

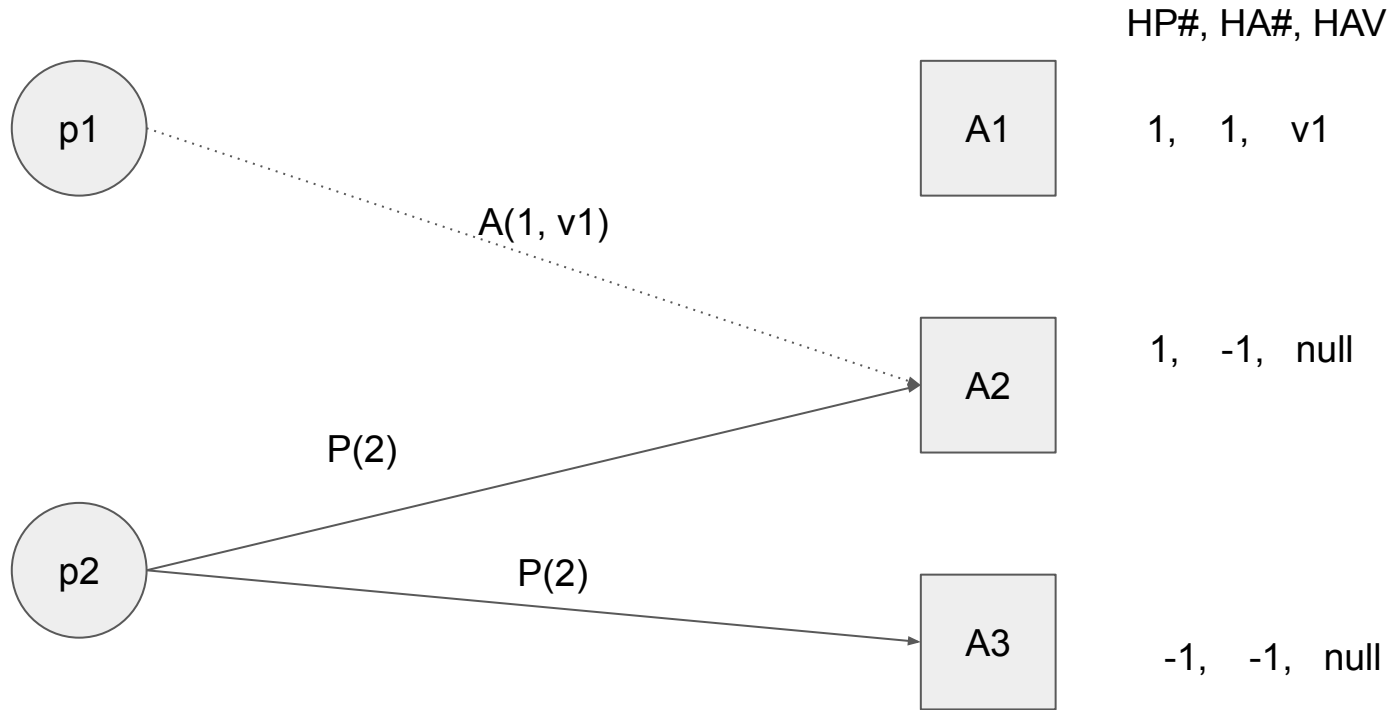
-1, -1, null

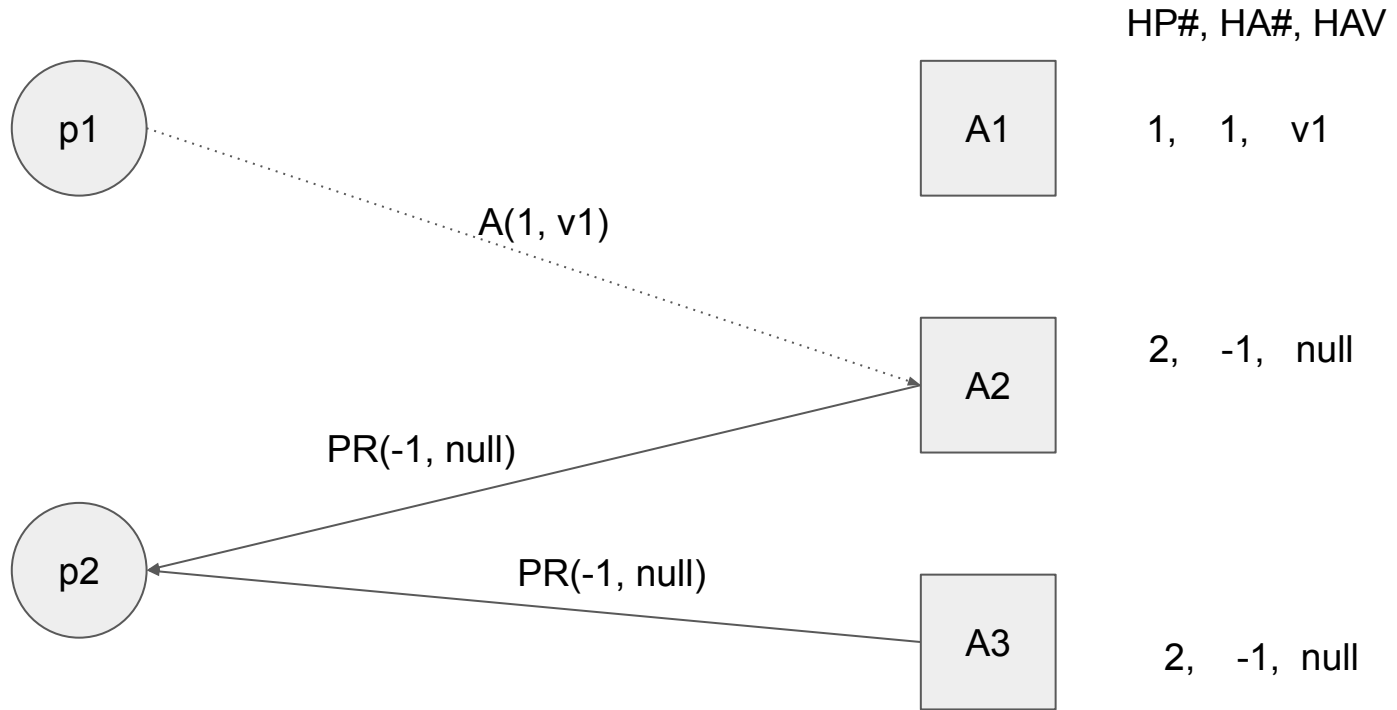


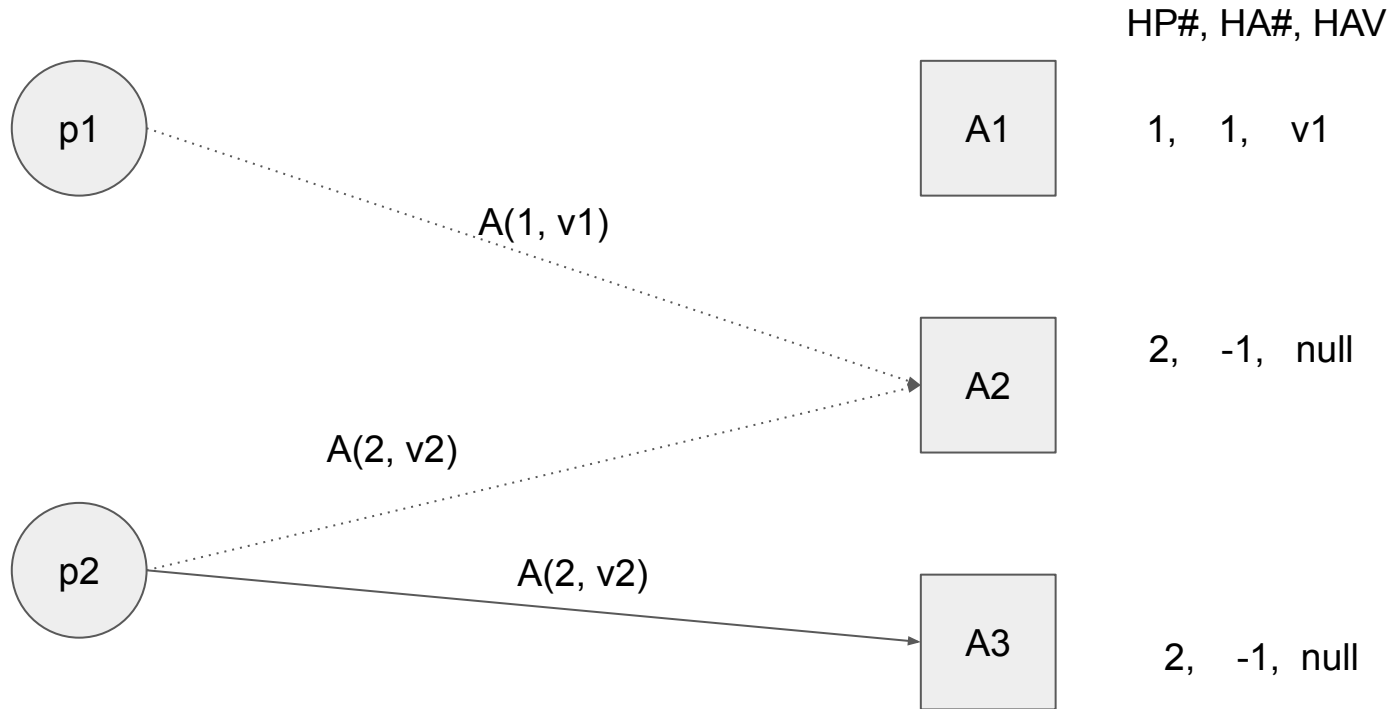


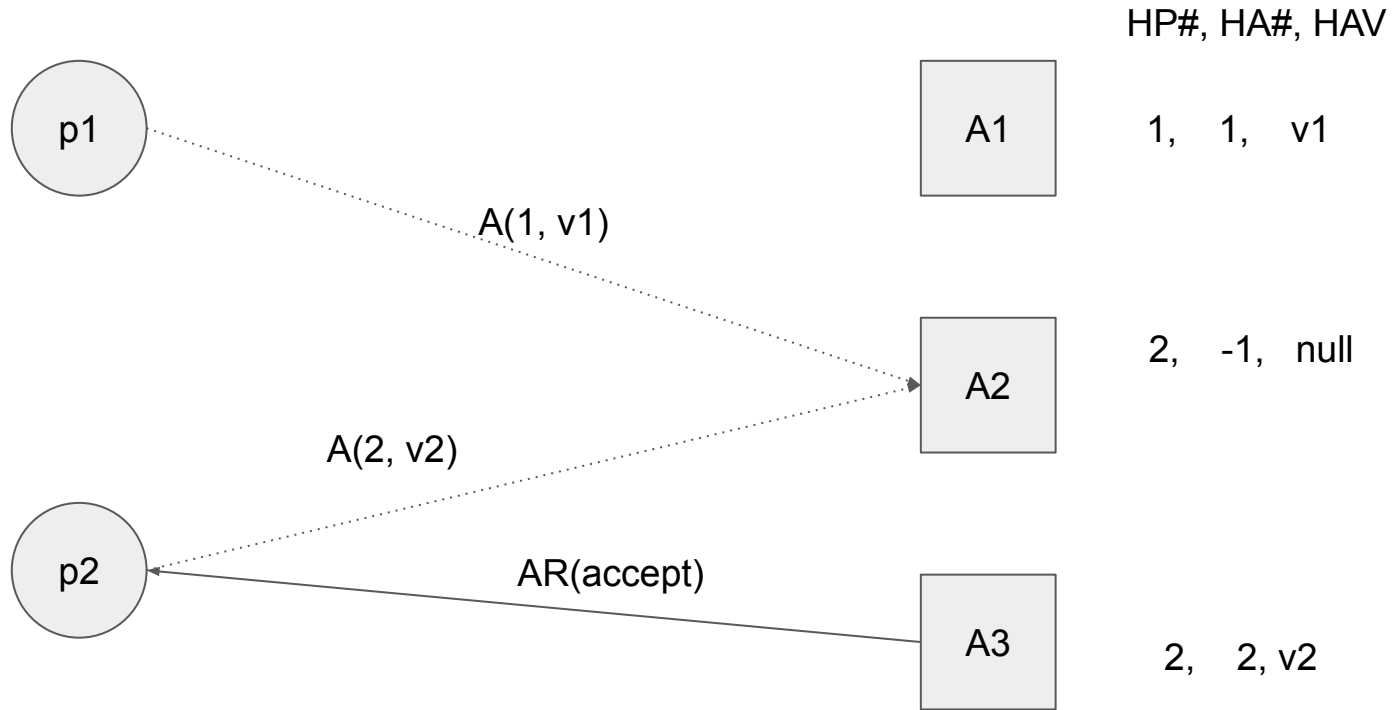


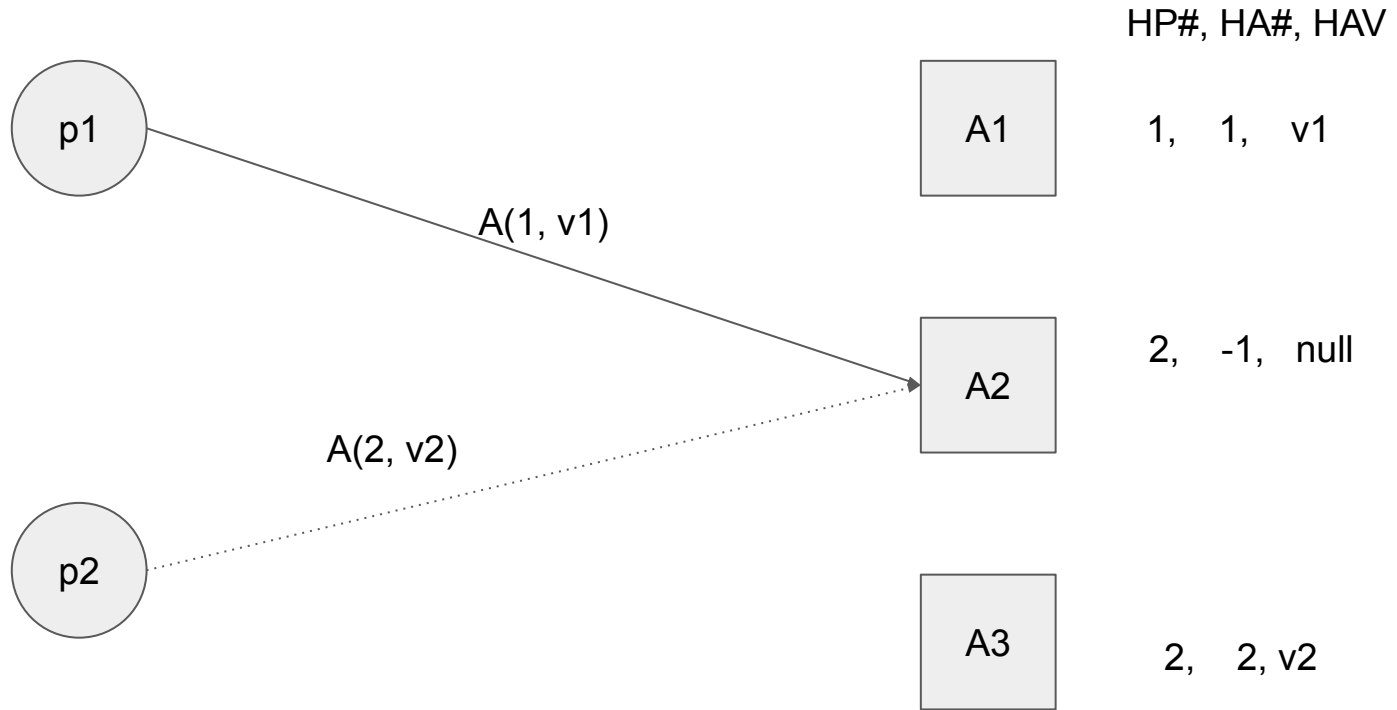


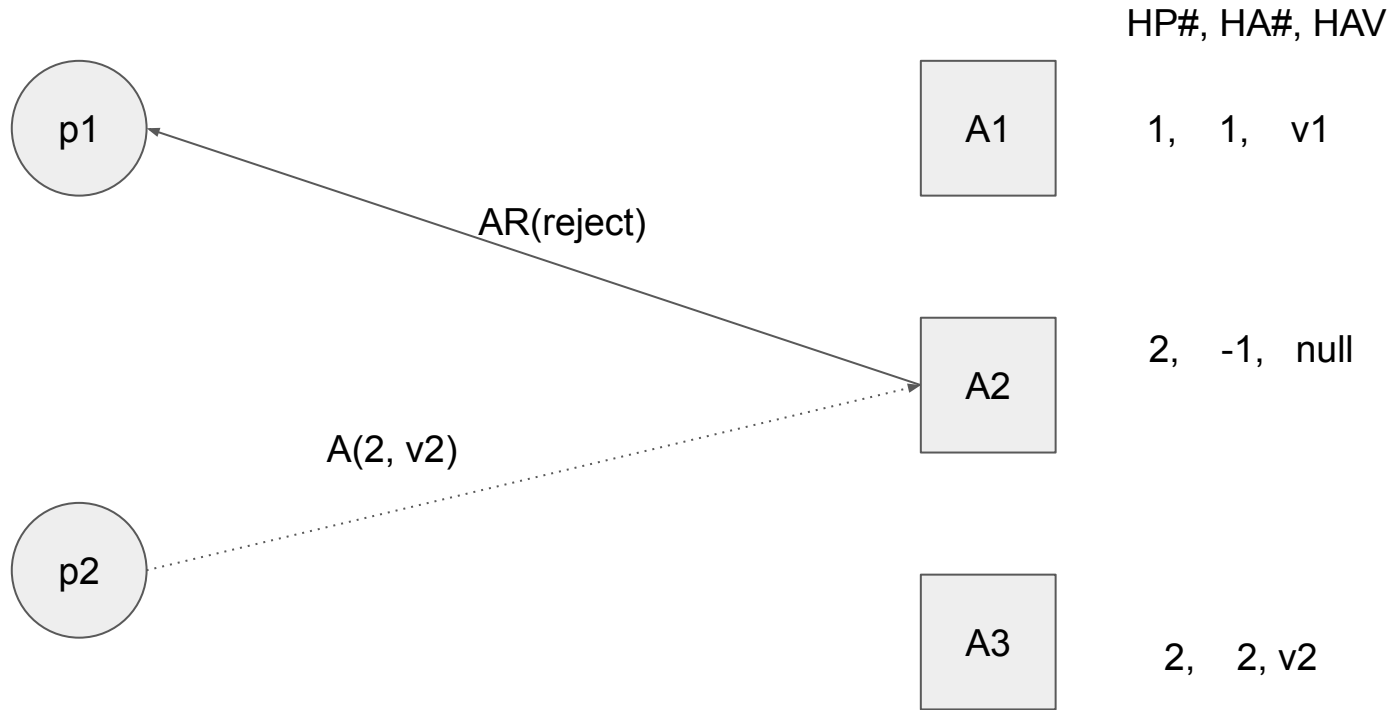


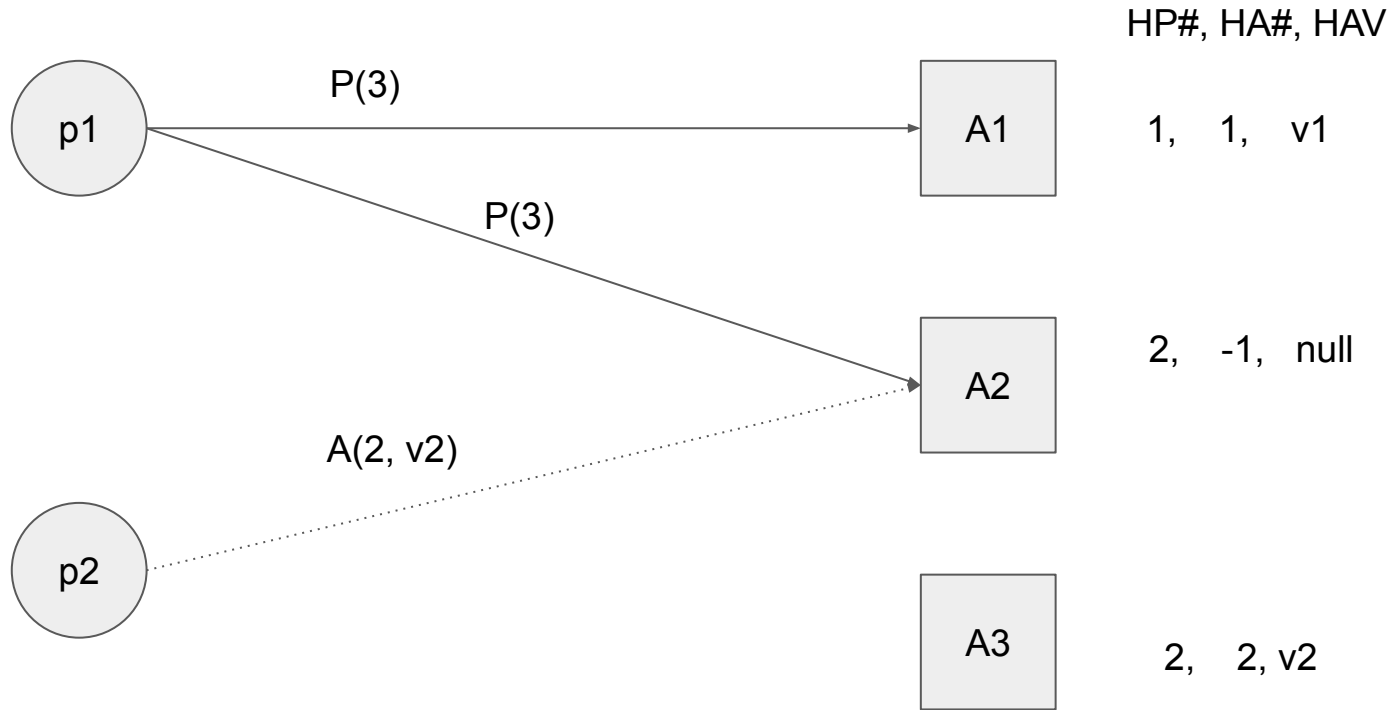


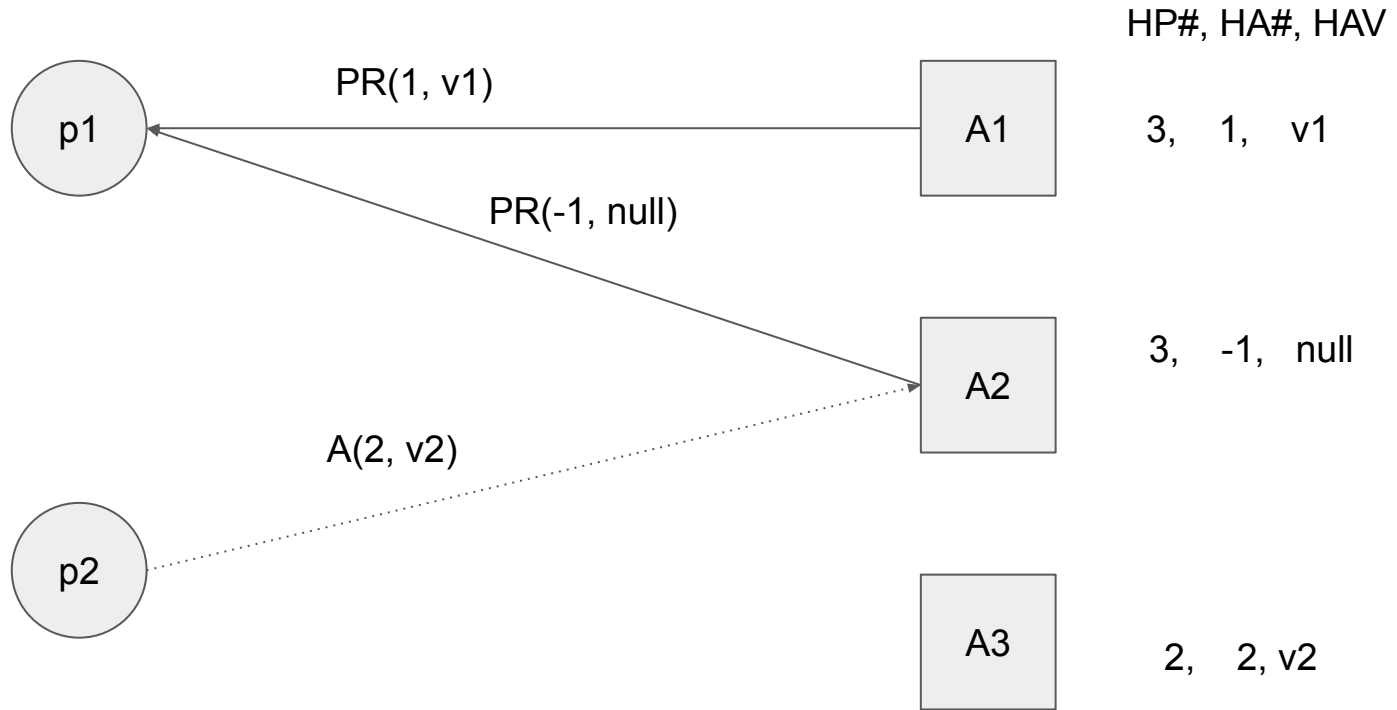


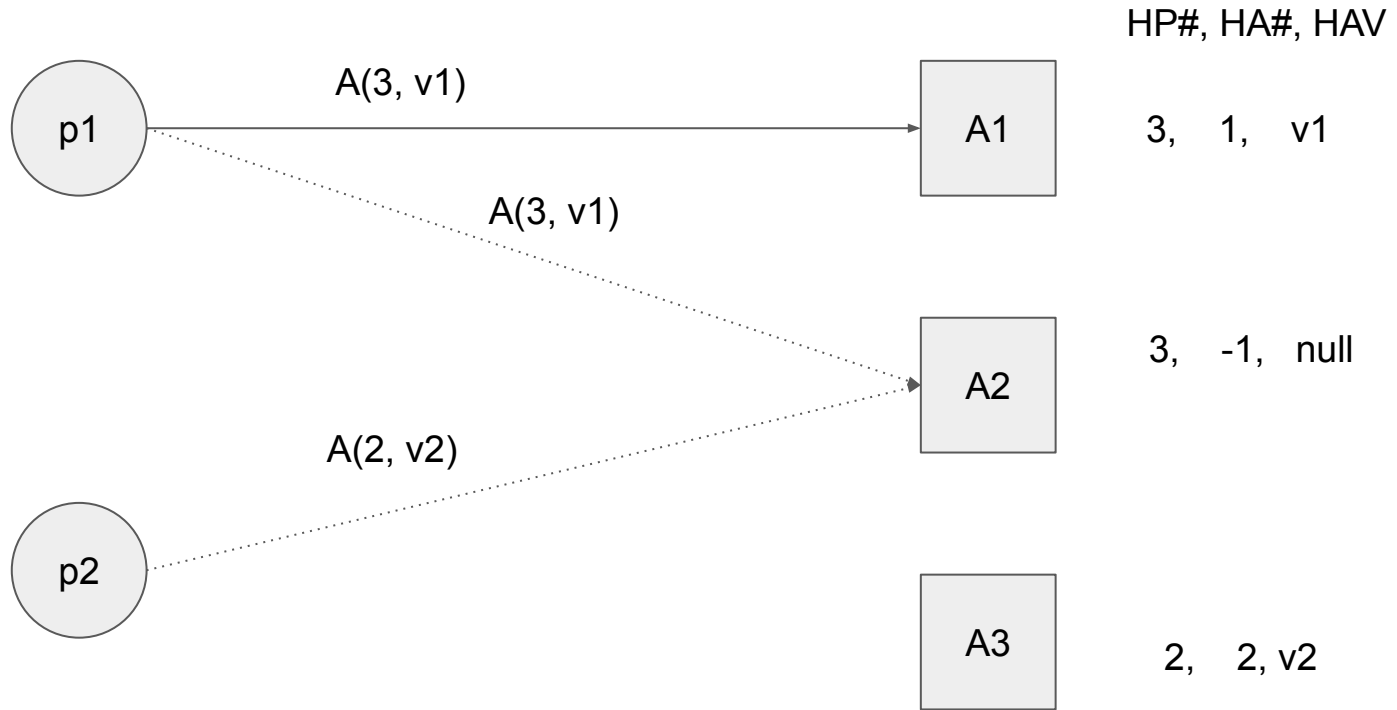


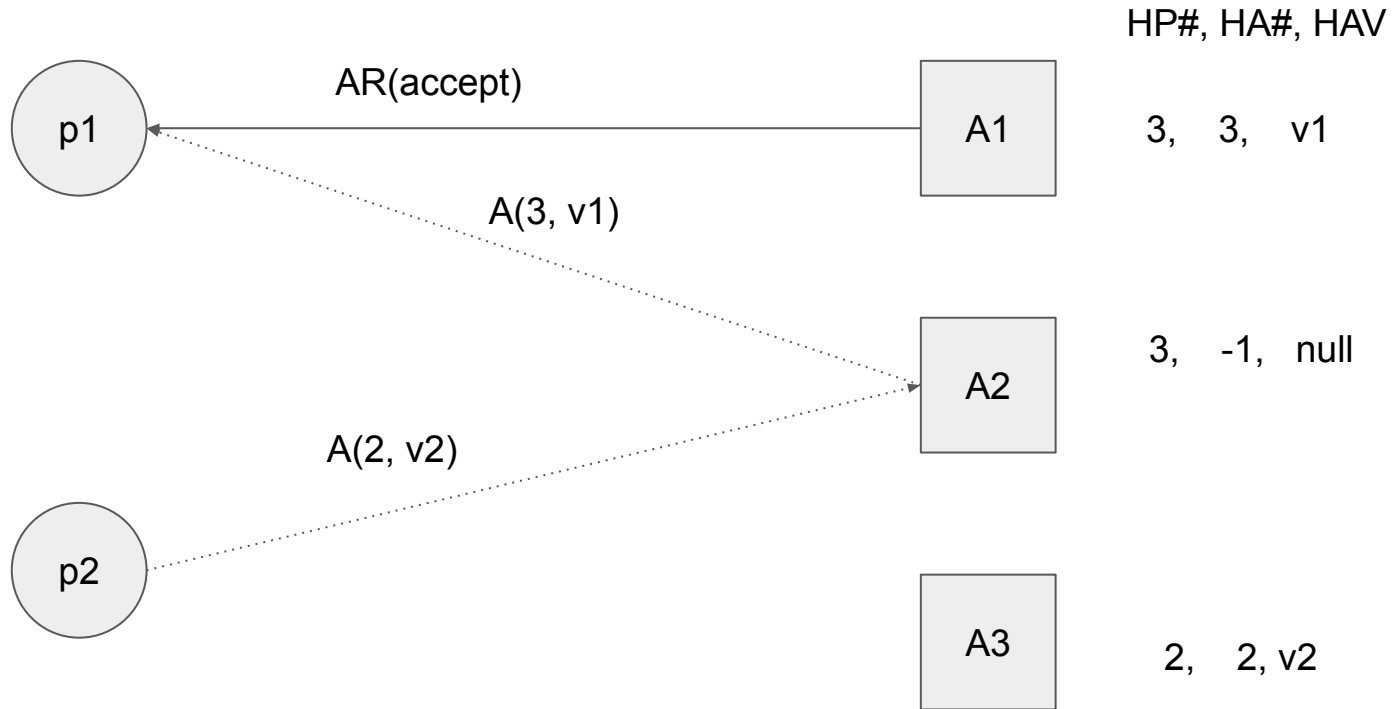


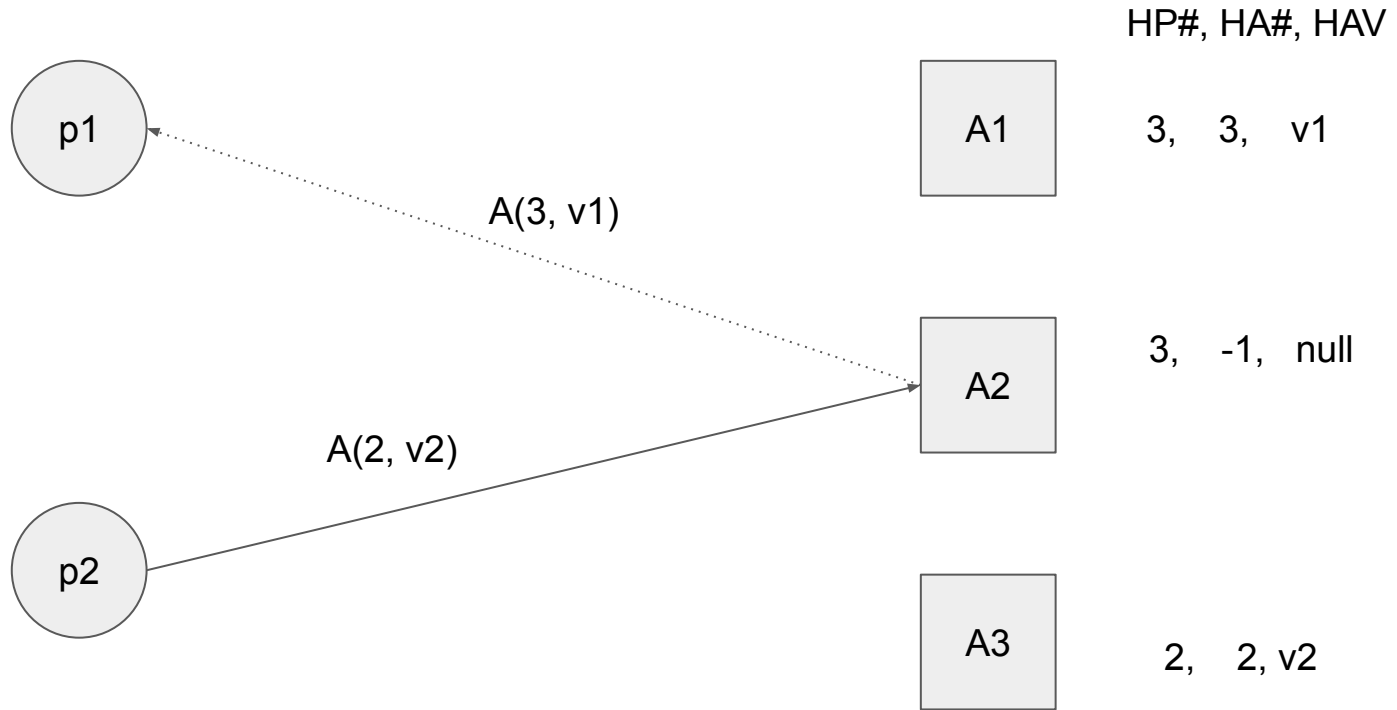


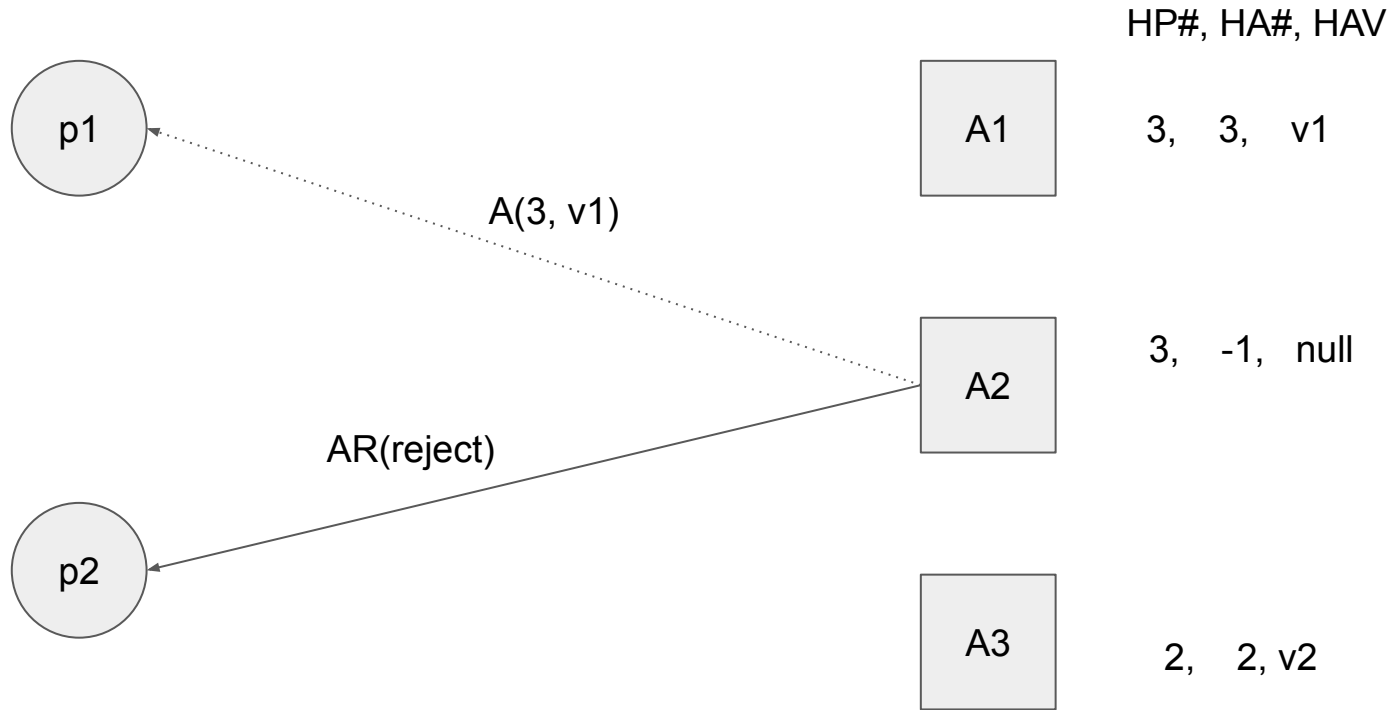












Problem: Contention

- If many proposers are sending values for the same slot at the same time, problems can occur
 - Acceptors may reject accept requests if proposal number isn't high enough
- Don't want to deal with contention for every log slot!
- Solutions:
 - Random timeouts
 - Exponential backoff
 - **Stable leader (Recommended)** aka distinguished proposer

Stable Leader - Multi-Paxos (PMMC)

- Why?
 - Prevents contention (generally)
 - Enable 1 round trip instead of 2 round trips for most requests
- How?
 - Elect a single server as leader
 - Use prepare phase to become leader!
 - Once leader, skip prepare phase, only run accept phase
- Issues:
 - Multiple servers may think they are leader
 - Can still have contention among servers trying to become leader
- PMMC describes an implementation of Multi-Paxos

Terminology (Basic Paxos -> PMMC)

- Proposal number -> ballot/ballot number
 - Ballot: (round number, address)
- Prepare request/response -> phase 1 (P1A, P1B, scouts)
- Accept request/response -> phase 2 (P2A, P2B, commanders)

Recommendation: Ballots

If you define the Ballot as an inner class, remember to make it **static**! If not, it might lead to failed garbage collection or lead to inefficiencies in testing and cause you to fail tests you would otherwise pass. Also make sure it's serializable and has equals and hashCode!

Even though doing something like having ballots being 1.1, 1.2, 1.3, 2.1, ... would work, we recommend making ballot a class like:

- sequence/round number
- Address

E.g. (3, server1)

Then making that comparable and defining a compareTo.

We recommend using a single ballot on each server (not for each role).

PMMC

vs.

Lab 3

- Split into roles (replicas, acceptors, leaders, etc...)
 - Multi-threaded
 - Handles reconfiguration
 - Retries leader election when preempted
- One class (PaxosServer)
 - Handles all the roles
 - **Recommendation**: Don't use a separate object for each role*
 - Single-threaded on each Node
 - Ignores reconfiguration
 - Uses heartbeat messages to determine when to begin leader election

Phase 1: Leader election

- Goal: We want to only have one proposer at a time, will call this a distinguished proposer or the leader (prevents contention, 1 RTT instead of 2)
- Need way to pick a leader (many different solutions).
 - E.g. largest ballot number (using compareTo) out of those pinged tries to become leader (starts phase 1)
 - Once a leader is elected, it can send a heartbeat message to every other server every T ms
 - If a server hasn't received heartbeat from the leader for an entire HeartbeatCheckTimer interval (similar to PingCheckTimer), it decides to try to become the new leader

Phase 1: How to get elected

- Multiple nodes may attempt to become leader at once, can cause contention for the position
- To become leader, send phase 1 (P1A) messages to all acceptors with your ballot number
- If the ballot is higher than any ballot the acceptor has seen before, the acceptor should update its ballot number, then respond with a message containing accepted values **for all slots**
 - Don't try to become the leader again right away
 - Can include chosen values to avoid needing to go through consensus again
- If the sender receives a majority of responses from acceptors with its ballot, it is now the leader
 - The sender can merge each log as it receives them to avoid keeping them around (Why?)
- What happens if the sender doesn't receive a majority of responses?
 - Keep trying until you get a message with a higher ballot number (or you get elected)

Phase 1: I'm elected, now what?

- To preserve linearizability, newly elected leader must update its state with accepted values from acceptors
 - For each slot, take the value with the highest ballot
- Send out phase 2 (P2A) messages for all values that have been accepted already
- Can begin to send out phase 2 (P2A) messages as client requests arrive

Server1

Ballot: 1.2

Accepted: {1->(A, 1.2)}

Server2

Ballot: 1.2

Accepted: {1->(A, 1.2)}

Server3

Ballot: 1.3

Accepted: {}

Server1

Ballot: 1.2

Accepted: {1->(A, 1.2)}

Server2

Ballot: 1.2

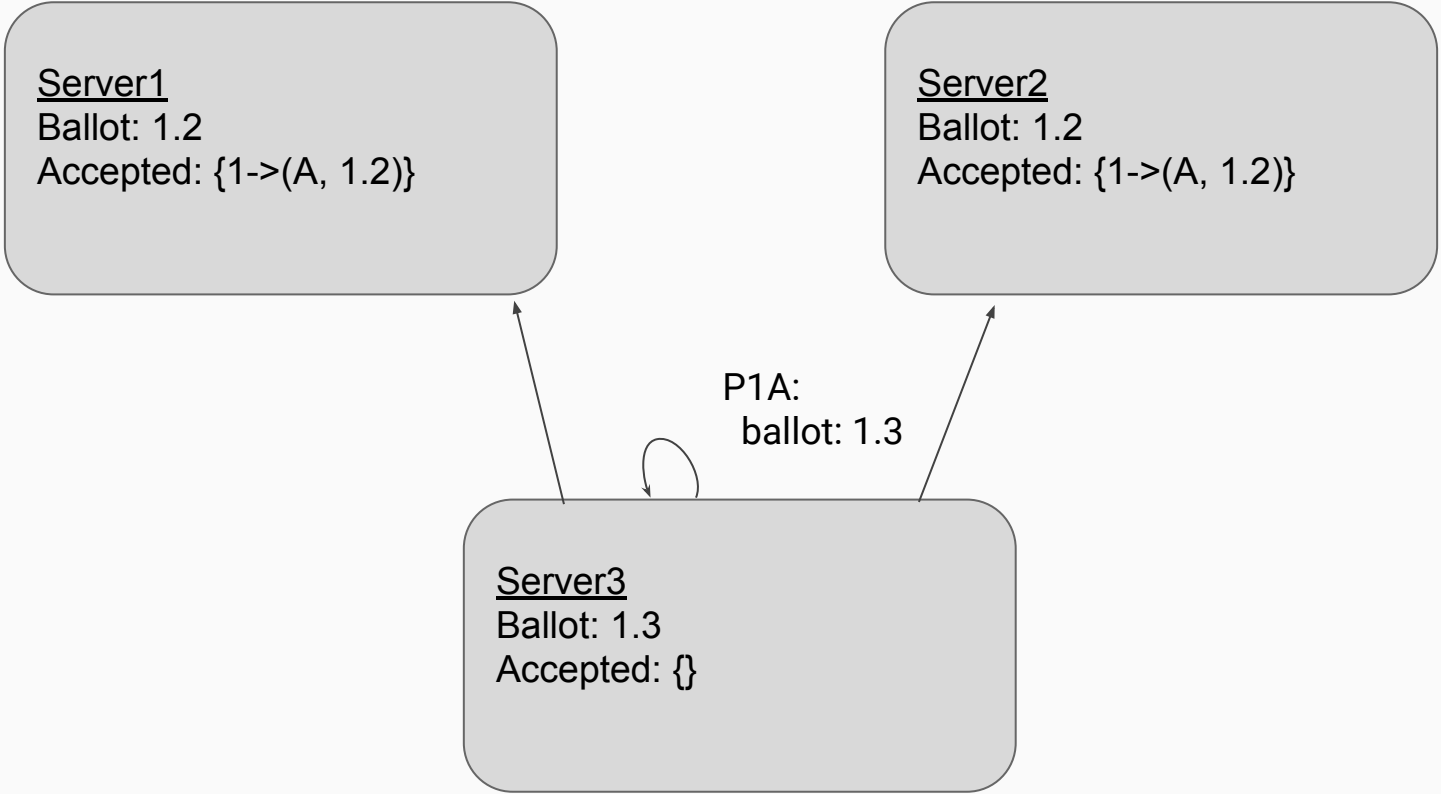
Accepted: {1->(A, 1.2)}

Server3

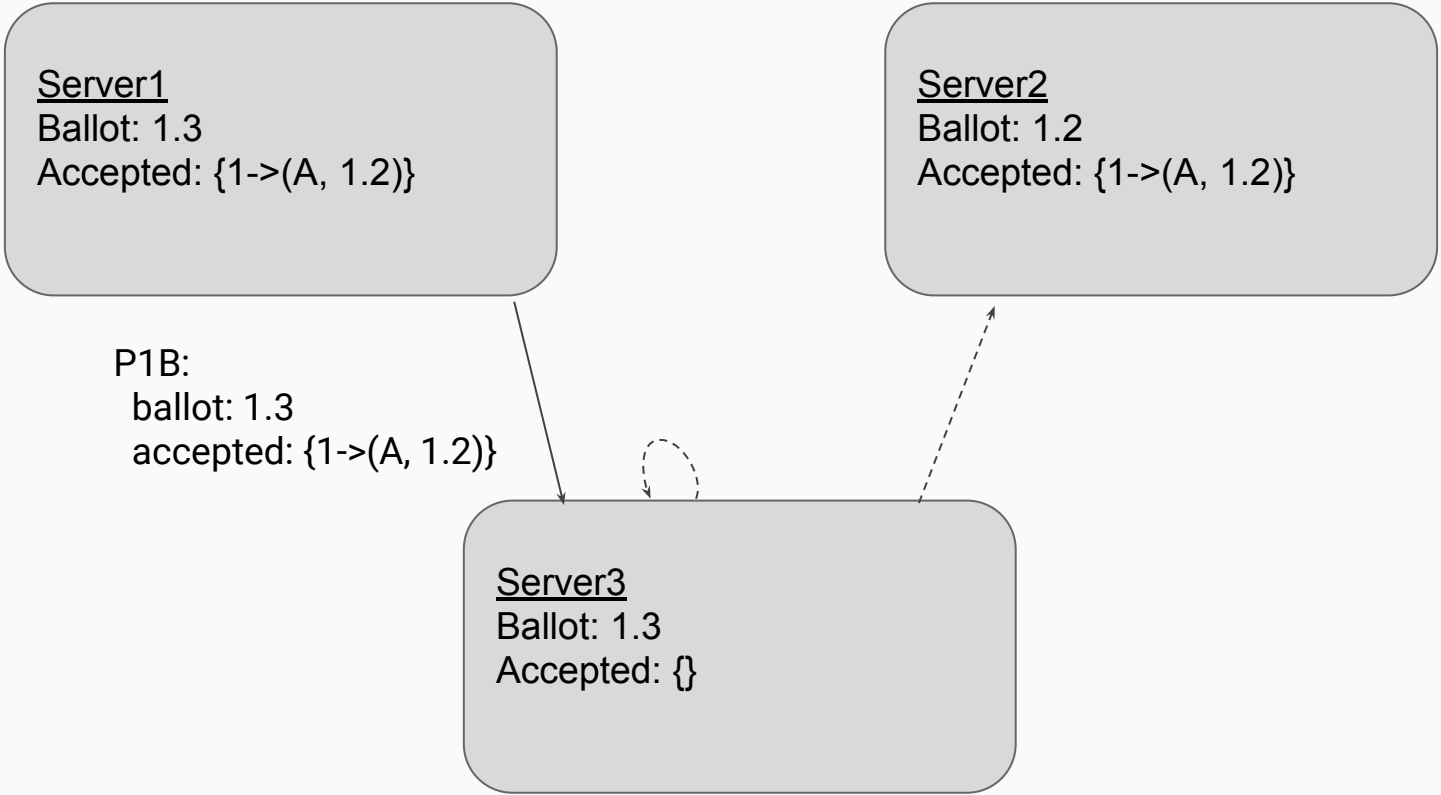
Ballot: 1.3

Accepted: {}

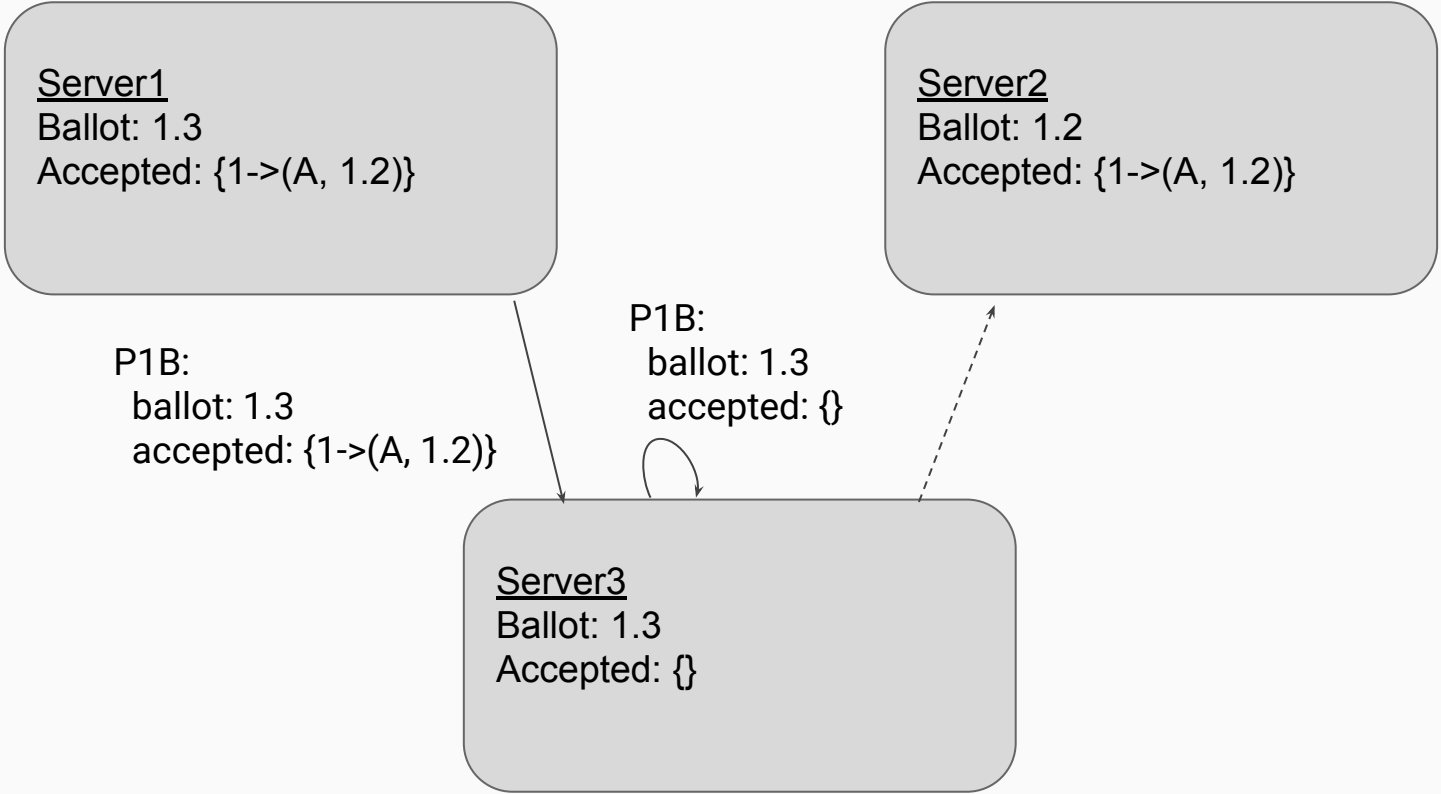
Server 3 wants to become leader



Server 3 wants to become leader



Server 3 wants to become leader



Server 3 wants to become leader

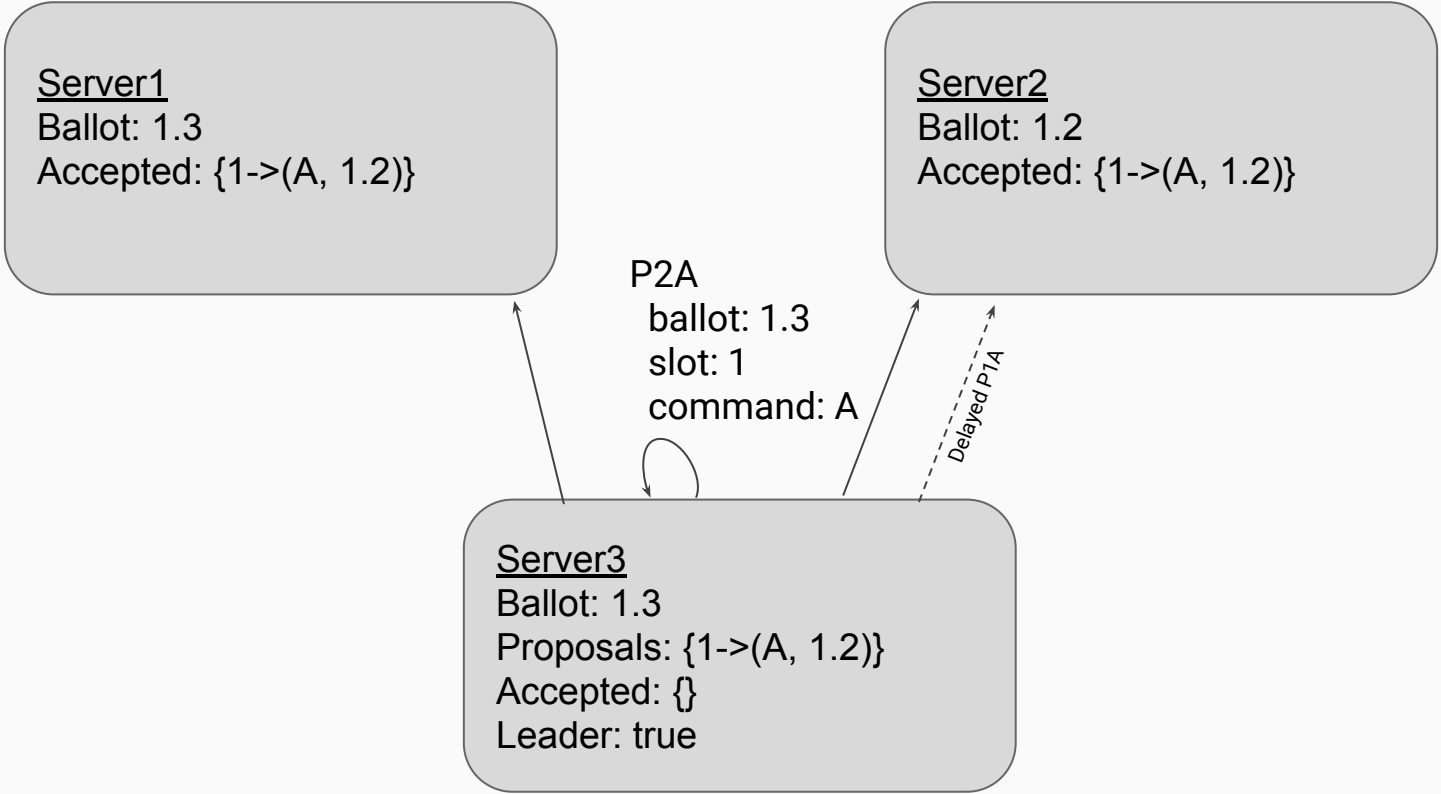
Server1
Ballot: 1.3
Accepted: {1->(A, 1.2)}

Server2
Ballot: 1.2
Accepted: {1->(A, 1.2)}

Server3
Ballot: 1.3
Proposals: {1->(A, 1.2)}
Accepted: {}
Leader: true

Delayed P/A

Server 3 is now leader



Server 3 is now leader

Phase 2: Leader

- When a server that thinks it is leader receives a request, it can skip phase 1 and immediately send out phase 2 (P2A) messages
 - P2A message should contain (**ballot, slot number, command**)
- In PMMC, upon receiving a P2A, an acceptor should only accept it if the ballot in the message matches the acceptor's ballot
 - This means the acceptor considers the sender to be the current leader
 - Prevents split brain problems
 - You can always reply with a P2B, as long as you don't update the state if ballots don't match!
 - But you might not need to respond
- Need a way to sync logs between leader and all the acceptors to learn about missed decisions.
 - They can also sync up with heartbeat messages.
 - Alternatively, when the leader sends an accept message, attach the leader's log with the request. When acceptor replies, attach the acceptor's log with the response. Then update/merge each side's log. (Although you should get normal p2a working first.) Lots of different possible protocols and implementations to consider!

Recommendation: Roles

- As the leader:
 - Act as a proposer and acceptor from Basic Paxos
 - Propose requests from clients
 - First, check if the command has already been proposed, decided, or executed
 - Keeps replicas up to date
 - Send heartbeats to servers so they know the leader is alive
 - Can include garbage collection information in these messages
- As anyone else (not the leader):
 - Drop client requests
 - Act only as an acceptor, not a proposer
 - That is, until the server believes the leader died. Then it should start phase 1 (scout phase)

Garbage Collection

- Problem: log (and thus memory usage) can grow indefinitely
- Solution: garbage collect (delete) log entries that are no longer needed
- Can discard log slots that **everyone** has already executed
- Need a way to determine what slots other nodes are done with
 - One solution: piggyback this information on heartbeat replies
 - Needs some state about what everyone knows

Garbage Collection: 3 servers in the system

Server1 (leader):
Latest executed slot: 4

Server2:
Latest executed slot: 4

Server3:
Latest executed slot: 3
Hasn't heard about slot 4

Can garbage collect slots 1-3

Garbage Collection: 5 servers in the system

Server1 (leader):
Latest executed slot: 4

Server4:
Latest executed slot: N/A

Server2:
Latest executed slot: 4

Server5:
Latest executed slot: N/A

Server3:
Latest executed slot: 3
Hasn't heard about slot 4

Cannot garbage collect anything until
all servers have executed a slot

Recommendation: `slot_in` and `slot_out`

Keep track of `slot_in` and `slot_out` from PMMC:

- `slot_in`: Where the leader/distinguished proposer puts new proposals
 - Make sure that you don't repropose things that have:
 - Already been executed [`app.alreadyExecuted`]
 - Already been proposed [already exists in the log]
 - Acceptors/Replicas: Updated when learning about greater slots
- `slot_out`: first slot that hasn't been decided on/executed in the log
 - Should be monotonically increasing and shouldn't jump around
 - Execution of slots should happen in order
 - When executing, execute as much as possible until next unchosen slot

Lab 3 Questions, q1

Question: What's the difference between a ballot number and a log slot?

Answer:

- Ballots are tuples of (round number, leader) and are used during phase 1 of Multi-Paxos (analogous to proposal numbers from Basic Paxos).
- Log slots are more general – namely, each log slot corresponds to an instance of Paxos.
 - Basic Paxos has no notion of log slots because it is only used to decide on a single value
 - A single ballot can be used to decide multiple log slots
 - This means the leader did not change

Lab 3 Questions, q2

Question: When are you done with a log slot? How can you make sure that they're garbage-collected efficiently?

Answer: You are done when all servers in the Paxos group have executed the request. You can piggyback messages to let everyone know not only what you are done with, but what you think everyone else is done with.

- **Note:** Some log slots may have not been decided because the proposer may have died, stopping the push to consensus on that log slot. You can resolve this by proposing a value, which will either drive any already accepted value to be chosen for that log slot, or will propose a no-op for that log slot.

Dealing with Log “Holes”/Proposing no-ops

If we have in our log:

put(a, 1), _____, _____, append(a, 2)

Do we know those empty log slots are decided? How can we push these log slots to completion and execute? Why do we need no-ops?

Your implementation needs to be able to handle "holes" in the Paxos log. That is, at certain points, a server might see agreement being reached on a slot but not previous slots. Your implementation should still make progress in this case.

*Log only contains chosen slots in this example

Server1
Ballot: 1.2
Accepted: {1->(A, 1.2),
2->(B, 1.2)}

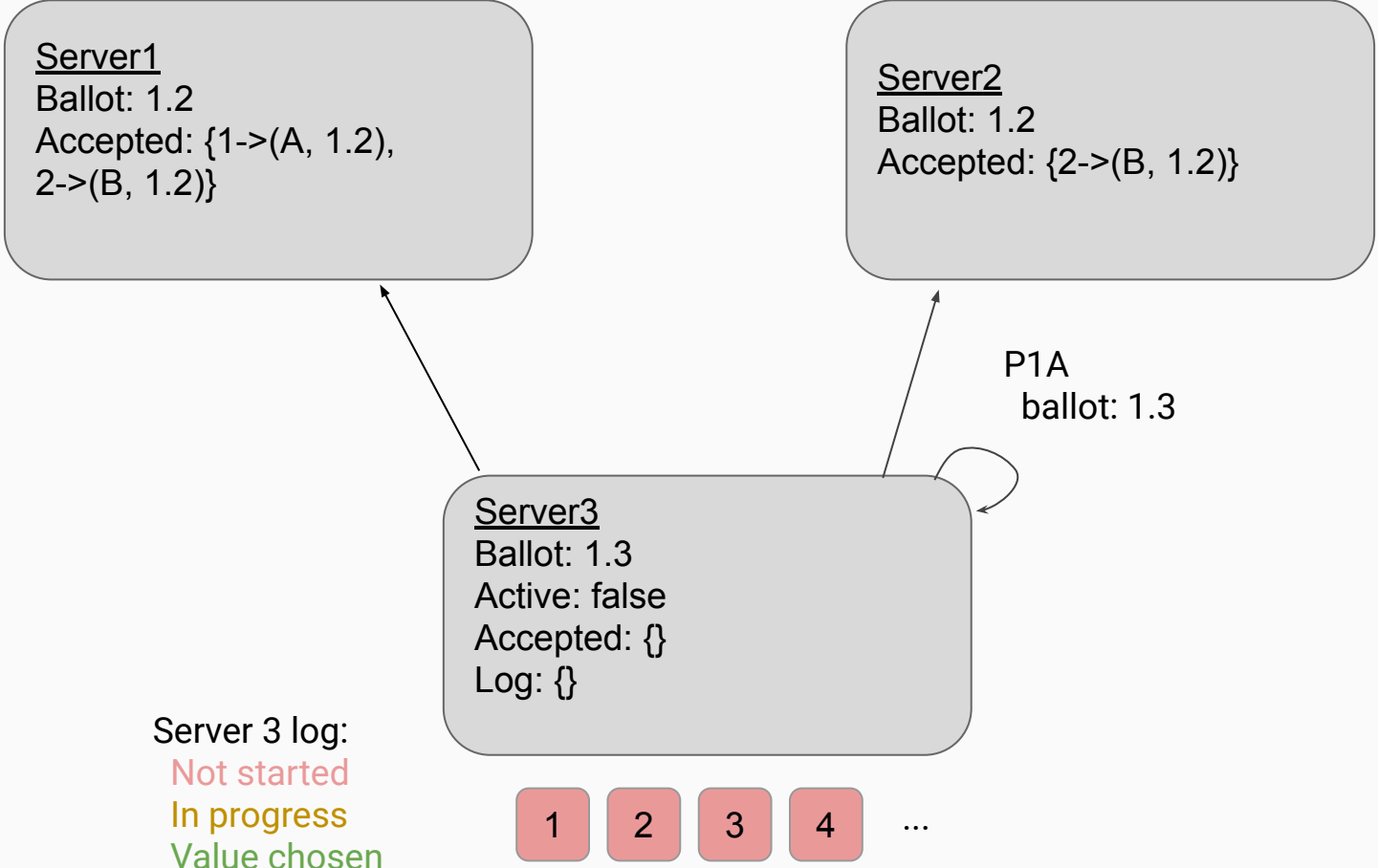
Server2
Ballot: 1.2
Accepted: {2->(B, 1.2)}

Server3
Ballot: 1.3
Active: false
Accepted: {}
Log: {}

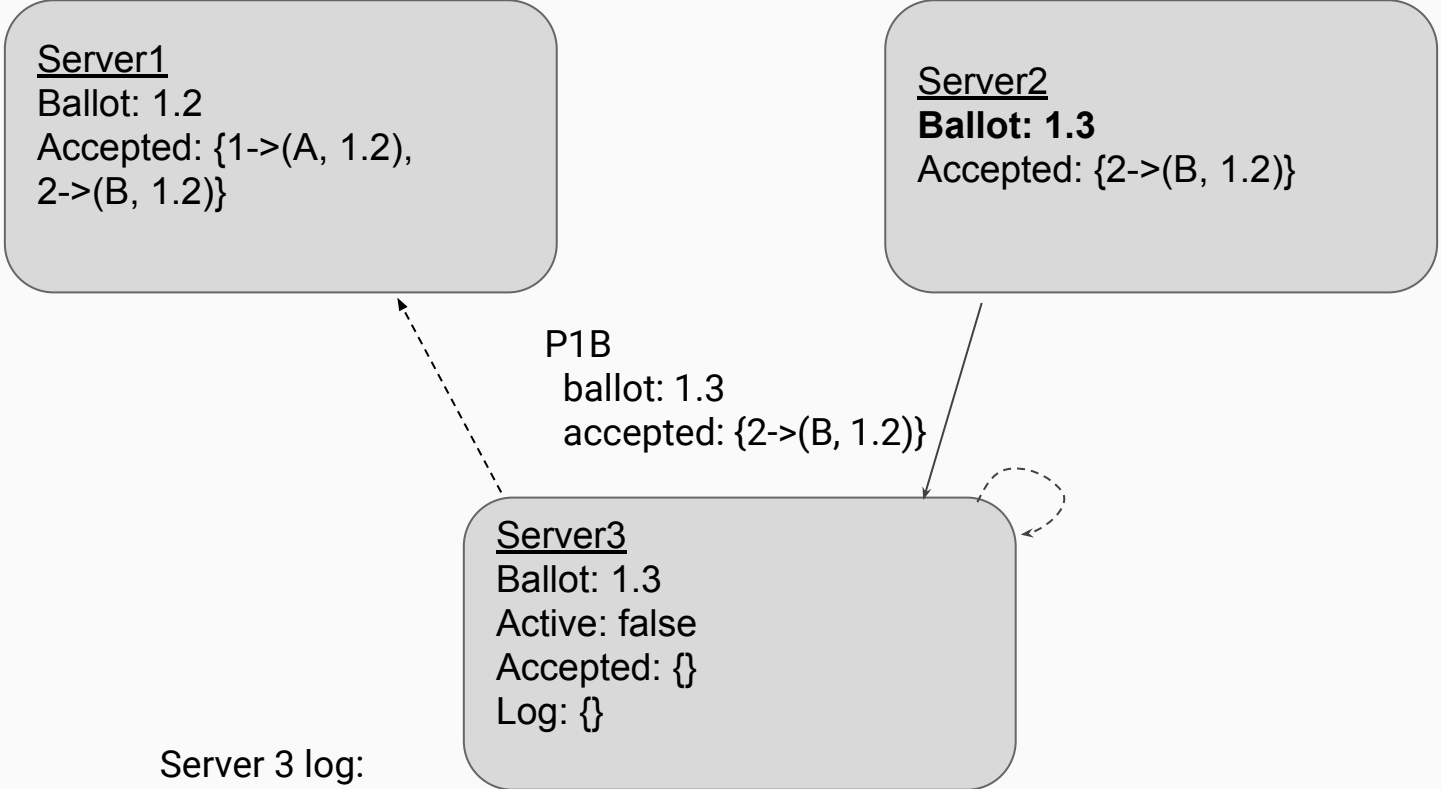
Server 3 log:
Not started
In progress
Value chosen



*Log only contains chosen slots in this example



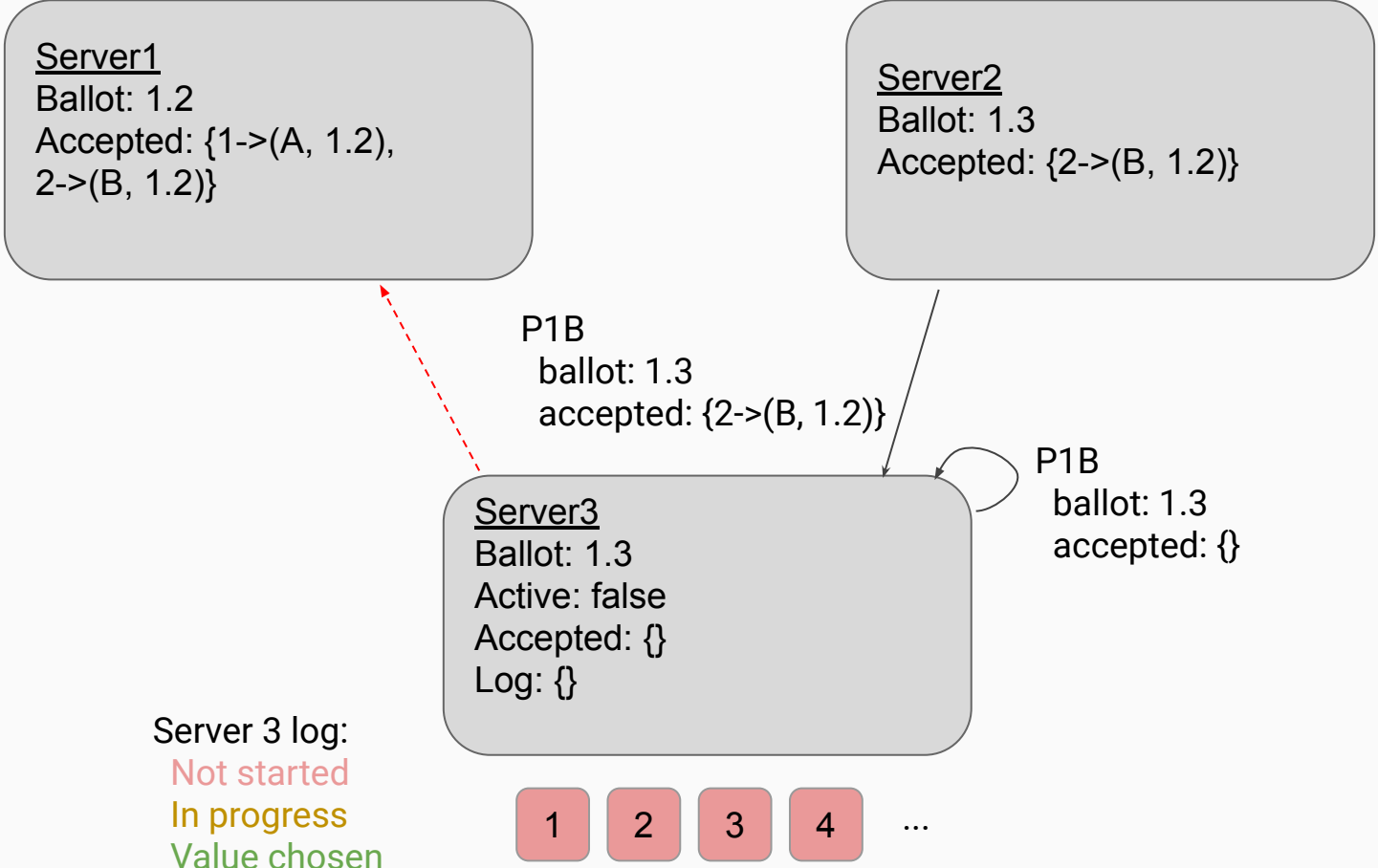
*Log only contains chosen slots in this example



Server 3 log:
Not started
In progress
Value chosen



*Log only contains chosen slots in this example



*Log only contains chosen slots in this example

Server1
Ballot: 1.2
Accepted: {1->(A, 1.2),
2->(B, 1.2)}

Server2
Ballot: 1.3
Accepted: {2->(B, 1.2)}

Server3
Ballot: 1.3
Active: true
Proposals: {2->(B, 1.2)}
Log: {}

Server 3 log:
Not started
In progress
Value chosen



*Log only contains chosen slots in this example

Server1
Ballot: 1.2
Accepted: {1->(A, 1.2),
2->(B, 1.2)}

Server2
Ballot: 1.3
Accepted: {2->(B, 1.2)}

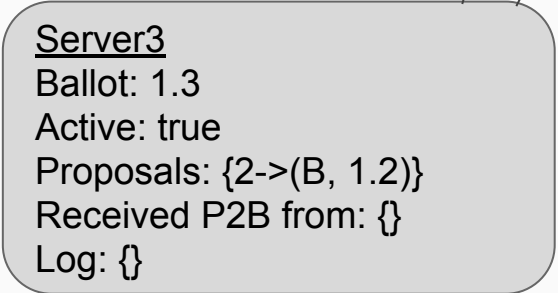
Server3
Ballot: 1.3
Active: true
Proposals: {2->(B, 1.2)}
Received P2B from: {}
Log: {}

P2A
ballot: 1.3
slot number: 1
command: no-op

Server 3 log:
Not started
In progress
Value chosen



*Log only contains chosen slots in this example

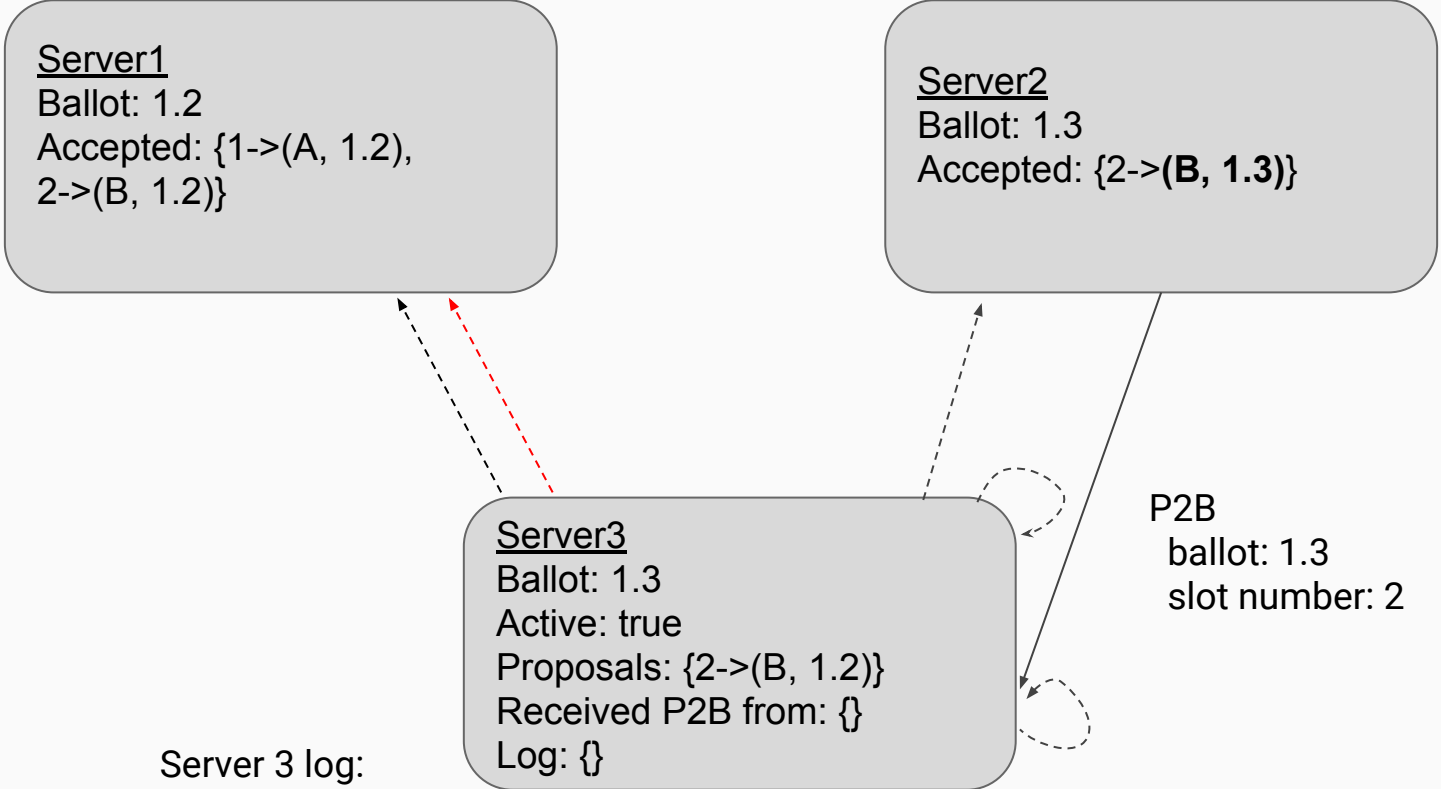


P2A
ballot: 1.3
slot number: 2
command: B

Server 3 log:
Not started
In progress
Value chosen



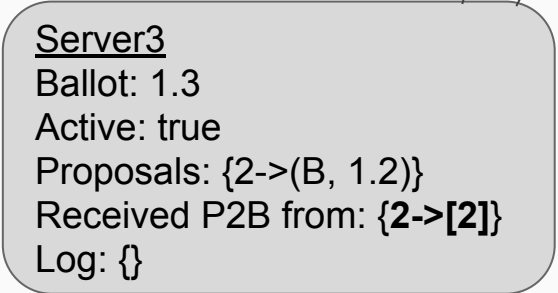
*Log only contains chosen slots in this example



Server 3 log:
Not started
In progress
Value chosen



*Log only contains chosen slots in this example



Server 3 log:
Not started
In progress
Value chosen



*Log only contains chosen slots in this example

Server1
 Ballot: 1.2
 Accepted: {1->(A, 1.2),
 2->(B, 1.2)}

Server2
 Ballot: 1.3
 Accepted: {2->(B, 1.3)}

Server3
 Ballot: 1.3
 Active: true
 Proposals: {2->(B, 1.2)}
 Received P2B from: {2->[2,3]}
 Log: {2->B}

P2A
 ballot: 1.3
 slot number: 1
 command: no-op

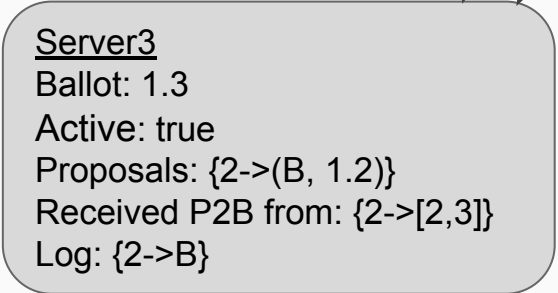
Server 3 log:
 Not started
 In progress
 Value chosen



*Log only contains chosen slots in this example



P2B
ballot: 1.3
slot number: 1



Server 3 log:
Not started
In progress
Value chosen



*Log only contains chosen slots in this example

Server1
 Ballot: 1.2
 Accepted: {1->(A, 1.2),
 2->(B, 1.2)}

Server2
 Ballot: 1.3
 Accepted: {1->(no-op, 1.3),
 2->(B, 1.3)}

Server3
 Ballot: 1.3
 Active: true
 Proposals: {2->(B, 1.2)}
 Received P2B from: {1->**[2]**,
 2->**[2,3]**}
 Log: {2->B}

P2B
 ballot: 1.3
 slot number: 1

Server 3 log:
 Not started
 In progress
 Value chosen



*Log only contains chosen slots in this example

Server1
 Ballot: 1.2
 Accepted: {1->(A, 1.2),
 2->(B, 1.2)}

Server2
 Ballot: 1.3
 Accepted: {1->(no-op, 1.3),
 2->(B, 1.3)}

Server3
 Ballot: 1.3
 Active: true
 Proposals: {2->(B, 1.2)}
 Received P2B from: {1->**[2, 3]**,
 2->**[2,3]**}
 Log: {1->**no-op**, 2->**B**}

Server 3 log:
 Not started
 In progress
 Value chosen



Example of merging logs (from P1Bs, with optimization of also sending chosen slots)
 Suppose there are 5 servers.
 (s4 trying to become leader with ballot number (5, s4).

([C]hosen/[A]ccepted, (Ballot), value)
Grey is executed

Slot #	1	2	3	4	5	6
Candidate (s4)	(C, _, v)	(A, (2, s4), h)		(A, (3, s2), x)		
Follower 1 (s3)		(A, (2, s5), y)		(C, _, x)	(A, (3, s5), b)	
Follower 2 (s1)	(C, _, v)	(C, _, y)		(A, (3, s5), x)	(A, (4, s2), g)	



Slot #	1	2	3	4	5	6
Leader (s4)	Chosen: v	Chosen: y	Propose: NO-OP with ballot (5, s4)	Chosen: x	Propose: g with ballot (5, s4)	

What was/is slot_in?
 What was/is slot_out?

What should the leader do next?
 How does s3 (+ other nodes) learn that slot 1 and 2 were chosen?

*=For people who follow PMMC REALLY closely

Cycling Leaders/Late Rejections*

Consider the case with 5 servers:

Server 4 is leader.

Server 4 sends P2A for value v in slot 1 with ballot number $\langle 1, 4 \rangle$ to every server. ← delayed

Server 5 sends P1A to every server with ballot $\langle 1, 5 \rangle$.

Servers 1 and 2 accept ballot $\langle 1, 5 \rangle$ and send P1B.

Server 5 then goes down.

Server 4 sees this and starts scouting sending P1A with ballot $\langle 2, 4 \rangle$.

Servers 1 and 2 accept ballot $\langle 2, 4 \rangle$ and send P1B.

Server 4 is now the active leader.

Server 4 sends P2A for value v in slot 1 with ballot $\langle 2, 4 \rangle$ to every server.

Server 2 then receives the old P2A with value v in slot 1 and ballot $\langle 1, 4 \rangle$.

Server 2 rejects this P2A and then sends back a P2B with the ballot $\langle 2, 4 \rangle$ without accepting the P2A.

Server 4 gets the P2B with ballot $\langle 2, 4 \rangle$ and believes server 2 has accepted the new P2A, when it has not.

Possible order for implementation of Paxos

1. Ballots
2. Log(s)
3. Accept messages/timers (P2As, P2Bs, Decisions)
 - a. You can start by hard-coding one of the servers as the leader
 - b. Make sure executions are happening in order (starting from slot_out and don't skip slots) and progress can be made for clients
4. Leader election/Prepares messages/timers (P1As, P1Bs)
 - a. Start leader election at the very start
 - b. Check if leader is alive
5. Heartbeats
 - a. Catch-up mechanism
6. Garbage collection

Some cases/questions to consider when writing your code

- Receiving a request that's already in your log as a proposer
- When do holes come up in your log? When should you propose no-ops?
- What happens when you receive a larger ballot in a message?
- How does a server know who the leader is?
- If you receive an accept message for a slot that you already know is chosen, what should you do?
- Why do we execute the log in order?
- What happens when a leader dies?

Misc. tips

- Don't use objects to separate the roles, as that might lead to unnecessary/extra states if not done carefully
 - If you're failing the search tests, run with `--checks` to make sure that you had `@Data` and stuff where it's necessary
- You don't need `WINDOW` because we aren't reconfiguring
- You don't need to re-propose alreadyExecuted commands
- <http://paxos.systems/> is easier to parse than PMMC
- Try to get Lab 3 to work because Lab 4 is built on top of Lab 3
- Try to understand how the changes you make impacts the system so you aren't scared about adding/changing/removing code you write
- Don't put things back in slots that have already been cleared

Tips for debugging

- If you finished prepares and accepts, you can debug them by using search test 20 and looking for invariant-violating states on the visualizer.
 - If you fail search tests due to too many states explored, try running the test with `--checks` and make sure that you have `@Data` or `@EqualsAndHashCode` for your classes
 - If you failed search tests due to not having a majority accept for a chosen command, but the visualizer says that you did get a majority accept, it's possible you had an error in your interface methods (`command(...)` and `status(...)`)
- For the random search tests, you can change the random depths to shorter depths so that you can get easier traces to step through.
 - In `./labs/lab3-paxos/tst/dslabs/paxos/PaxosTest.java`, change `maxDepth(1000)` to like `maxDepth(20)`
 - Make sure your interface methods work, since the tests use those to check things, e.g. `status`, `lastNonEmpty`, etc.
 - **Remember to change it back!**
- **Inner classes must be static**, otherwise it implicitly creates a copy of the parent class when serialized. Also applies to lab 4; might lead to random, weird timeouts/slowdowns.

If you fail test 8 and...

Client workload wasn't finished

- Phase 2 probably doesn't work right

It's due to a timeout

- Make sure that the interface methods are fast (avoid iterating over things if you can, but it doesn't matter if you have garbage collection)

If you fail test 10 and it takes too long...

- Make sure that you reply to the client right away when you execute and not waiting for the client to contact you again in order to reply

If you fail 16,17,18

- Try debugging by saving the logging output into a file and looking at regions where the responses for the client that it said waited too long actually waited too long
- Common issues:
 - Not doing no-ops
 - Not doing leader election properly,
 - e.g. not allowing servers who previously tried to get elected try again
 - Not reproposing/setting timers for proposing accepted slots
 - Followers trying to become leader too fast
- Maybe add print statements when you come to a decision and when you receive a response to get an idea of what's causing the slowdown

Some other tips for debugging 17,18

You can add print statements and run 17 and 18 (not writing it to a file) to see where it gets stuck.

Suggested positions:

- When leaders get elected
- When a command becomes chosen
- When clients get the correct result back

If you fail 23 because...

Search space exhausted:

- Make sure you're properly resetting timers for things like leader elections, heartbeats, or p2as

Could not find state matching [...]:

- Reduce the number of timers that you set/messages that you send

If you see servers that are supposed to be dead with timers triggering...

- Check to make sure that you're not exponentially setting timers, e.g. one timer calling a method that sets a timer and resetting the original timer

If you see that a handler isn't idempotent

You can run the visualizer by itself and try to get to a state in which you send the message with the handler that it says isn't idempotent, then right-click the message and click "duplicate". Then process those messages in order. Note that the network model uses a set of messages, so handling a message again shouldn't create a brand new message/timer.

To run the visualizer by itself, do something like this:

- `./run-tests.py -l 3 --debug 3 2 "PUT:foo:bar,APPEND:foo:baz,GET:foo"`

Miscellaneous design things we saw that we thought were good ideas

- Keeping track of `slot_in` and `slot_out` on followers for `lastNonEmpty` and other interface methods
- Just merge the logs as you receive them when you're trying to become the leader so that you don't have to store logs around

Section 7: Lab 4 - ShardMaster

CSE 452 Spring 2021



Lab 3 Wrap up

- Lab 4 relies heavily on lab 3
- However, even if your lab 3 implementation isn't perfect, you can still work on lab 4
 - But definitely work on finishing up lab 3 first! Especially any correctness bugs.
 - Some liveness issues may not trigger problems in lab 4.
 - ShardMaster (lab 4.1) doesn't depend on Paxos
 - Some 4.2/4.3 tests use Paxos groups of size 1, so you can still pass those without a perfect Paxos implementation
 - Make sure you are passing test 26 from lab3
 - You can also debug the rest of the 4.2/4.3 tests by changing them to use Paxos groups of size 1, to test the actual lab 4 logic separately from the Paxos logic

Lab 4 Overview

Goal: Build a “linearizable, **sharded** key-value store with **multi-key updates** and **dynamic load balancing**, similar in functionality to Amazon's DynamoDB or Google's Spanner”.

Lab 4 Overview

- Paxos increases **reliability**, sharding increases **performance** and **scalability**
- Part 1 implements the ShardMaster application (handles load balancing)
- Part 2 implements a Sharded KV store and handles moving shards
- Part 3 adds multi-key transaction support
 - This means a single request can update multiple keys in different shards, while maintaining linearizability
 - Implemented using **two-phase commit**
 - More on this in section in two weeks

Sharding: what is it?

- Divides keyspace (the **K** in **K/V**) into multiple groups, called **shards**
 - Can shard keys on many things (alphabetically, random/hashes, load-balanced etc.)
- Each shard will be handled by a group of servers. Each group:
 - Runs Paxos from lab 3. So we can assume a group will not fail :)
 - Stores all key/value pairs in the database that correspond to its shard
 - Accepts/responds to client requests that correspond to its shard
- Since different sharding groups can run in parallel without communicating, performance is increased proportional to the number of shards
- Lab 4: You won't have to change what keys go into which shards. The number of shards will stay the same

Terminology

- Shard Master
 - “Application” replicated by Paxos
 - Service that responds to changes in configuration (new Paxos groups being added, removed, etc)
- Configuration:
 - Similar to a view in primary/backup lab 2
 - Specifies which groups are responsible for which shards
 - Has configuration number (monotonically increasing)
- Paxos Replica Group
 - Group of servers performing Paxos agreement with each other - just like Lab 3
 - Handles key/value storage for assigned shards
- Shard
 - In charge of a subset of key/value pairs,
 - e.g. shard that stores all keys starting with “a” or that stores all keys that start from “a-g”
 - Shards are numbered 1....numShards

Lab 4

Sharding/partitioning

Put(A, 0)



Get(B)



Append(C, 123)



Peer
0

Peer
1

Peer
2

Peer
0

Peer
1

Peer
2

Peer
0

Peer
1

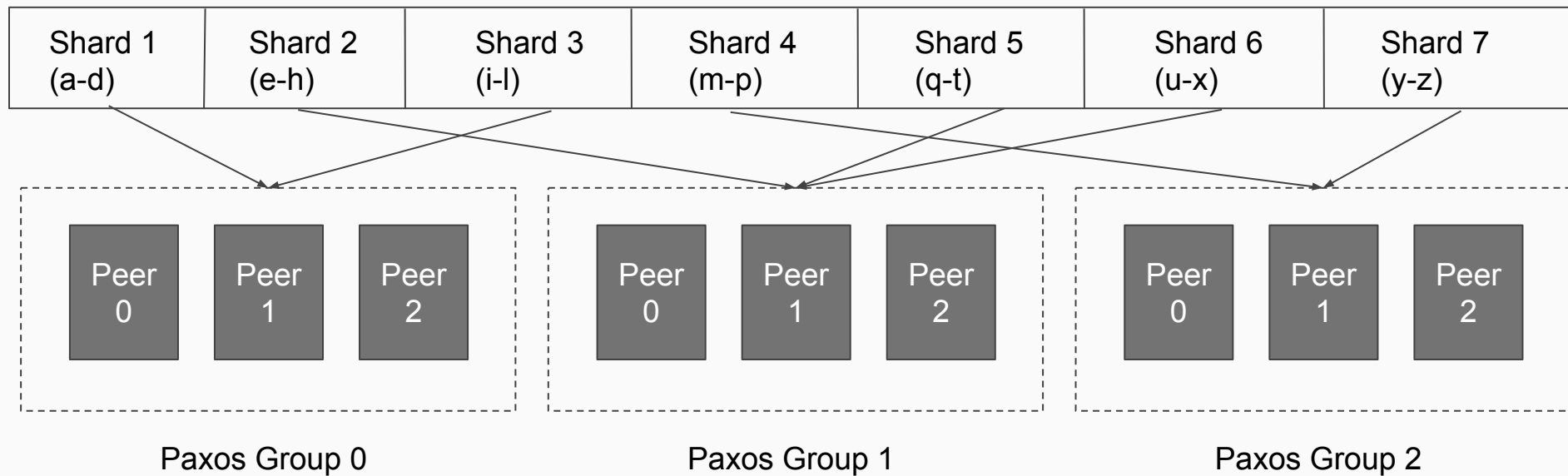
Peer
2

Paxos Group 0

Paxos Group 1

Paxos Group 2

Shards vs Groups



ShardMaster

- A service to keep track of which groups serve which shards
- Necessary because:
 - Clients need to be able to figure out what group to send requests to (i.e. which replica group is responsible for a given key)
 - We might want to reconfigure the system (inducing redistribution of shards)
 - Add/Remove Paxos Replica Group
 - Move a shard to another group (testing or, in practice, load balancing popular keys)
- Conceptually similar to the View Server in primary/backup

ShardMaster continued

- Keeps track of a current configuration object (ShardConfig):
 - `private final int configNum;`
 - `private final Map<Integer, Pair<Set<Address>, Set<Integer>>> groupInfo;`
 - `Integer`: group id
 - `Set<Address>`: addresses of all members in that group
 - `Set<Integer>`: all the shard numbers the group is responsible for
- Also remembers all old configurations
 - Does not need to be garbage collected
 - Query can ask for any past configurations (see slide 15)
 - For every configuration number, want to store a configuration object like above

ShardMaster Application

- ShardMaster class is an Application
- Accepts 4 command types:
 - Join
 - Leave
 - Move
 - Query
- Responds with 3 reply types:
 - Ok
 - Error
 - ShardConfig
- You'll only need to call Query commands for the other parts of the lab (test code calls others for you)

Join

- The way that new replica groups are added to the system
 - First Join's config num should be INITIAL_CONFIG_NUM (0)
- Join commands contain:
 - Integer for replica group ID (Must be unique, or ERROR is the result)
 - Set of server addresses that should be in the group
- ShardMaster responds by creating a new configuration
 - New config includes new shard group
 - Redistributes the shards among the updated set of groups.
 - Should move as **few shards as possible** ← **this may be a little bit tricky.**
 - Returns Ok result

Leave

- Command contains: Group Id that should “leave”
- Opposite of Join: “deletes” a group from the system
- ShardMaster must redistribute the group’s shards to other groups
- Should still move as few shards as possible
- OK on success, ERROR when
 - the current config does not contain group or
 - the final group is trying to leave (not actually tested)

Move

- Command contains:
 - Shard number
 - Replica Group id (which group the shard should be moved **to**)
- Moves a shard from one Paxos Replica Group to another Paxos Replica Group
- Practically, helpful for load balancing in the real world - operations on really hot keys perhaps should be more isolated than other keys!
- Returns OK on successful move, ERROR otherwise (e.g. when the current config does not contain the group or if it already has the shard)

Query

- Command contains: configuration number
- Should reply with ShardConfig
- Returns configuration for a specific configuration number
 - e.g. a server is outdated and needs to catch up on all the missed configurations
- If number is -1 **or** larger than largest known configuration number, the ShardMaster should reply with the latest configuration.

Join Example

Configuration: -1 [should return Error if queried]

Shard	1	2	3	4	5	6	7	8	9	10
Group	null	null	null	null	null	null	null	null	null	null

Join(1)

Join Example

Configuration: 0

Shard	1	2	3	4	5	6	7	8	9	10
Group	1	1	1	1	1	1	1	1	1	1

Join(2)

Join Example

Configuration: 1

Shard	1	2	3	4	5	6	7	8	9	10
Group	1	1	1	1	1	2	2	2	2	2

Join(5)

Join Example

Configuration: 2

Shard	1	2	3	4	5	6	7	8	9	10
Group	1	1	1	5	5	2	2	2	2	5

Some series of transitions
occur...

Leave Example

Configuration: 9

Shard	1	2	3	4	5	6	7	8	9	10
Group	1	1	5	2	2	7	7	4	6	6

Leave(1)

Leave Example

Configuration: 10

Shard	1	2	3	4	5	6	7	8	9	10
Group	5	4	5	2	2	7	7	4	6	6

Some tips

- Remember to make deep copies of configurations if you're going to edit them to create the next configurations

Section 8: Lab 4 Part 2

CSE 452 Spring 2021



Read the spec!

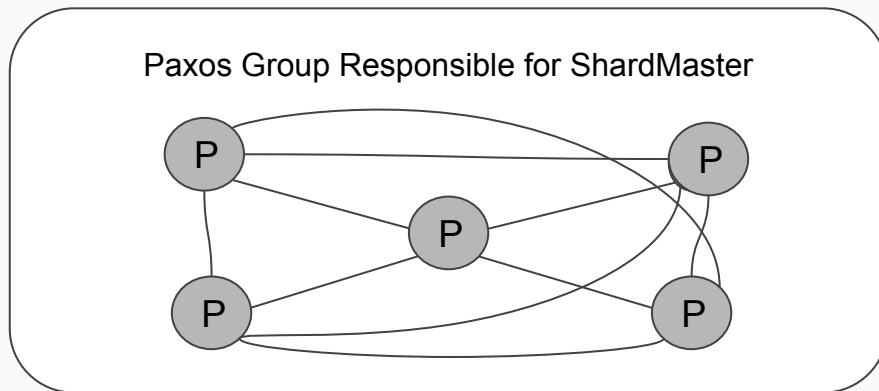
Part 2 in some bullet points

- Replicate messages
- Move shards around to transition into new configs
- Handle requests

As a note, just because you pass the part 1 tests, it doesn't mean that the ShardMaster is 100% correct.

100% code coverage is hard.

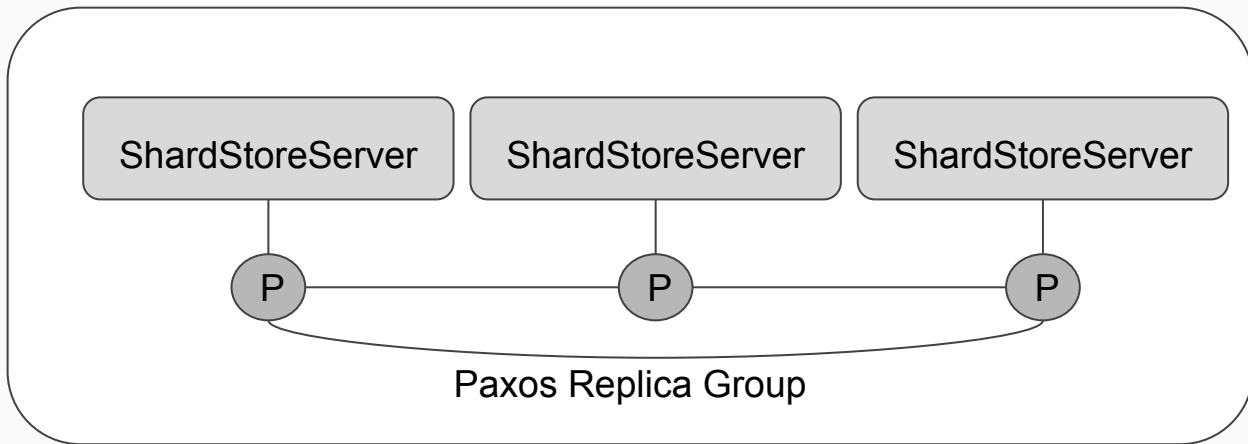
Overview: Shardmaster



ShardMaster is an Application that is running within a PaxosServer:

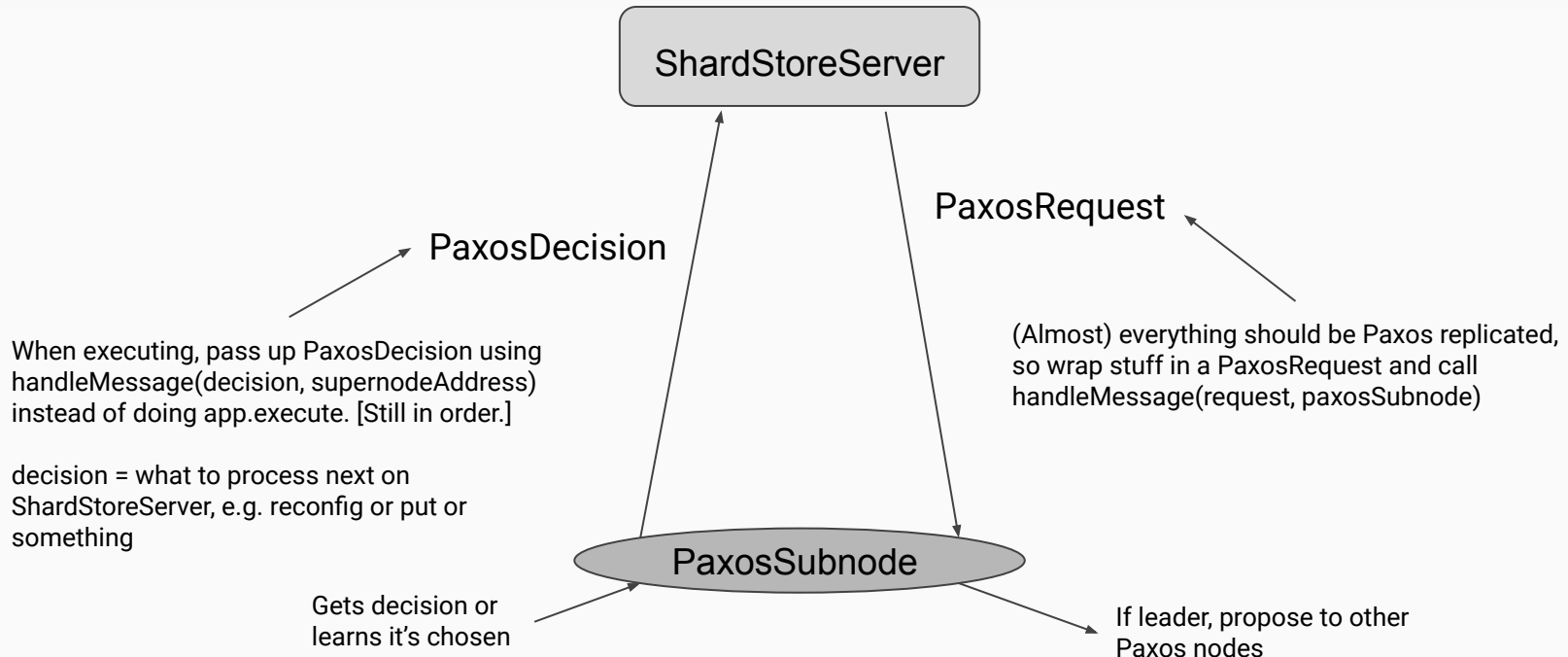
```
new PaxosServer(address, shardMasters.clone(), new ShardMaster(numShards));
```

Overview: Paxos Replica Group



A zoomed in version of a Paxos Replica Group, where P stands for Paxos Subnode. Lines between `ShardStoreServer` and its P subnode indicate message passing between the two.

Message Passing between ShardStoreServer and PaxosSubnode



Overview: Paxos Replica Group's Subnode

```
public void init() {  
    // Setup Paxos  
    paxosAddress = Address.subAddress(address(), PAXOS_ADDRESS_ID);  
  
    Address[] paxosAddresses = new Address[group.length];  
    for (int i = 0; i < paxosAddresses.length; i++) {  
        paxosAddresses[i] = Address.subAddress(group[i], PAXOS_ADDRESS_ID);  
    }  
  
    PaxosServer paxosServer =  
        new PaxosServer(paxosAddress, paxosAddresses, address());  
    addSubNode(paxosServer);  
    paxosServer.init();  
    ...  
}
```

Init for ShardStoreServer
Code is also in spec

This won't work until you
add the constructor, see
next slide

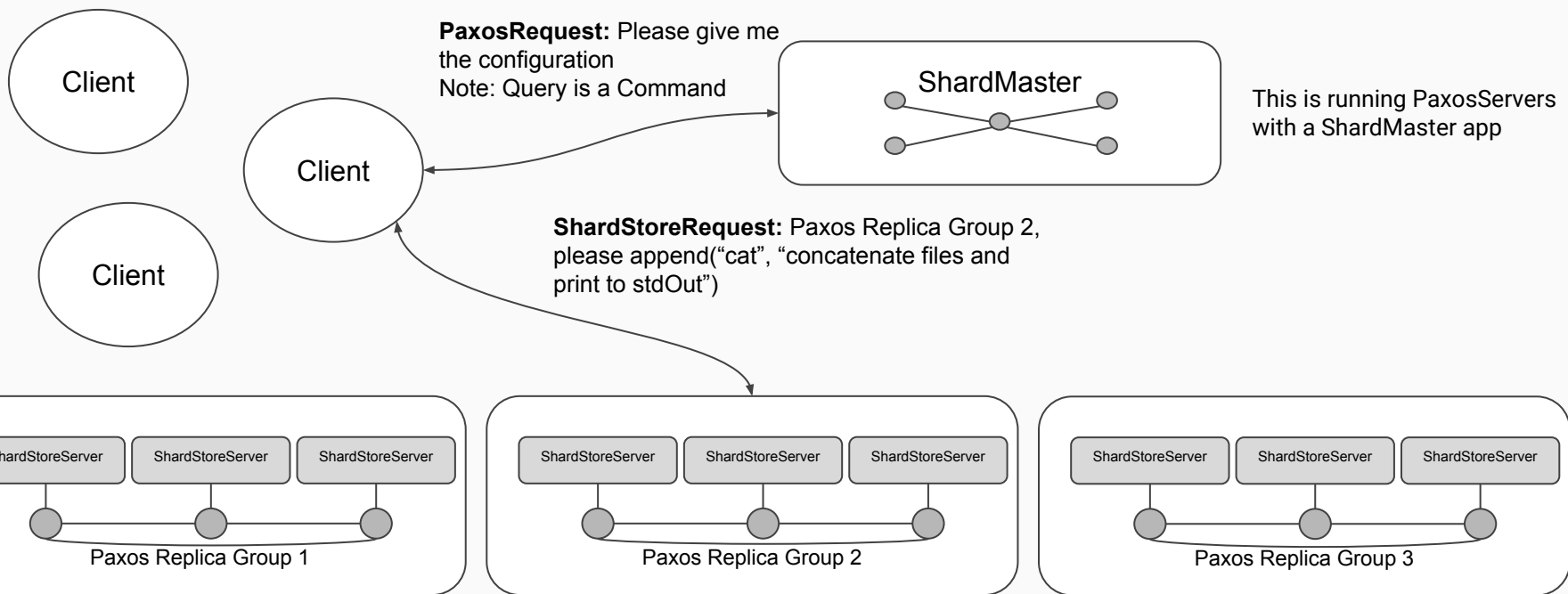
Constructing the Paxos Sub Node

```
public PaxosServer(Address address, Address[] servers, Address parentAddress) {  
    super(address); // 'address' is the address of this node  
    this.servers = servers;  
    this.parentAddress = parentAddress;  
    // 'parentAddress' is the address of the 'parent' ShardStoreServer  
    // Again, just call handleMessage(decision, this.parentAddress);  
    // Note: There is no app.  
}
```


Changes for PaxosServer

- Add the constructor
- Basically add `handleMessage(PaxosDecision, parentAddress)` everywhere that executes
- You might want to be able to repropose Query commands (or you can perform stale reads)

Overview: System



ShardStoreClient

- Sends requests to `ShardStoreServers`
- Similar to your client from lab2 (Primary/Backup/Viewserver)
- Needs to get configuration from `ShardMaster` to be able to know to where to send requests
 - Tip: Use the existing `broadcastToShardMasters(...)` defined in `ShardStoreNode` to broadcast a `Query` to get current configuration
- Response from shard masters will be a `PaxosReply` → need a `handlePaxosReply` in client to get updated `ShardConfig`
- If client times out or receives an error message → client out of date, need to get updated config
- For individual `KVStoreCommands` [`Puts`, `Gets`, `Appends`], wrap them in a `ShardStoreRequest` and broadcast to all servers in the correct replica group (maybe define a `getGroupIdForShard(...)` method in the config and a `keyToShard(...)*` method is provided for you to figure out which servers to broadcast too)

Messages

- In `ShardStoreServer`, you receive:
 - **PaxosReplies** from the `ShardMaster`, informing you about new configs.
 - You will be pinged the `ShardMaster` every so often for new configurations (via `Query`)
 - **ShardStoreRequests** from clients asking you to perform a `SingleKeyCommand` (The commands you are used to in Lab 2/Lab 3) or `Transaction` (Lab 4 part 3)
 - **ShardMoves/ShardMoveAcks*** from other `ShardStoreServers`
 - **PaxosDecisions** from the `Paxos` subnode - the replicated messages
- When a `ShardStoreServer` has received one of these messages, it **CANNOT** act on them until they are replicated (unless app has executed already, in which case it's still safe to reply, as before)

Suggestion for implementation

- Commands received by a `ShardStoreServer` will usually need to be sent to Paxos to get a consensus
 - Basic structure will look something like this:

```
handleShardStoreRequest(Request m) {  
    // Do some validation (e.g. is the request being sent to the correct group, etc...)  
    // If valid, call process (coming in a slide or two) with replicated as false  
}
```
- Since this can get a bit difficult to keep straight, we recommend the following approach instead...

Suggestion for implementation (continued)

- Have a single generic “process” method that takes (Command `c`, boolean `replicated`)
 - Process method does “instanceof” on the command and calls a specific `processXX` where `XX` is the specific command type (e.g. `AMOCommand`, `ShardMove`, etc...)
 - When you receive a Paxos decision, just send it into this switch table, which will send it to the appropriate `processXX` method. See the next slide.
- Each message handler will only call process method
 - E.g. `handleShardStoreRequest(ShardStoreRequest r, ...)` calls process with the message’s command and `false` for `replicated` (since command has not been replicated by Paxos)
 - On `PaxosDecision` handler, also call process with command but `replicated` would be `true` instead
- Then in the individual `processXX` methods you can do something like...

Suggestion for implementation (continued)

```
private void process(Command command, boolean replicated) {
    if (command instanceof ShardMove) {
        processShardMove((ShardMove) command, replicated);
    } else if (command instanceof ShardMoveAck) {
        processShardMoveAck((ShardMoveAck) command, replicated);
    } else if (command instanceof NewConfig) {
        processNewConfig((NewConfig) command, replicated);
    } else if (command instanceof AMOCommand) {
        processAMOCommand(command, replicated);
    }
    // Add cases for Lab 4 Part 3
    else {
        LOG.severe("Got unknown command: " + command);
    }
}
```

```
private void processShardMove(ShardMove m, boolean replicated) {
    // some checks to see if this is valid command
    if (!replicated) {
        // propose via paxos. process will call this method again
        // with replicated=true from handlePaxosDecision
        // Propose m to Paxos subnode (Some wrapping may be required)
        return;
    }
    // the message was replicated can now act upon it
    // do stuff here
}

private void handleShardMoveMessage(ShardMoveMessage m, Address sender) {
    process(m.command(), false /* It's not replicated yet... */ );
}

private void handlePaxosDecision(PaxosDecision p, Address sender) {
    process(p.command(), true /*It's replicated (from Paxos) */ );
}
```

Suggestion for implementation (continued)

```
processXX(Message m, Address a, boolean replicated) {  
    // ...  
    // What if this command has been executed?  
    // What if this is not an active Config?  
    // ...  
    if (!replicated) {  
        paxosPropose(m.command());  
        return;  
    }  
    // Yay, it's replicated! Now we can act upon the message, here...  
}
```

Idea: For a given message, you will go into the processXX method twice (once before the message is replicated (in which case you will propose it via Paxos and then return) and the second time after the message has already been replicated via Paxos

When are shards created?

Shards are created on the first reconfiguration after it has been replicated by Paxos

The first group should create all the shards necessary for the first configuration, since no shards currently exist in the system, so it can't receive them from somewhere else

Reconfiguration

- You don't want to move on when you are transitioning between configurations. Think back to `hasPrimaryAked` in `ViewServer`
 - This is also true about reconfigurations, you don't want to process any key/transaction commands until you have an active configuration!
- You don't want a single `ShardStoreServer` to be able to send a Shard or receive (acknowledge) a Shard by itself - it must perform Paxos consensus (look back to slide 10)

What does a configuration need to move?

- **From the spec:**

Be careful about when guaranteeing at-most-once semantics for key-value operations. When a server sends shards to another, the server needs to send AMOApplication state as well. Think about how the receiver of the shards should send its AMOApplication state.

- Our KVStore AMOApplication is more than just a KVStore - it also needs <Client, SeqNum> mappings so that it doesn't repeat operations!
- **Solution 1:** In ShardMove, send your Map<Client, SeqNum> mappings and keys associated with the moved shards. In handleShardMove, just update your reply mappings for that client to the maximum SeqNum.
- **Solution 2:** Store one AMOApplication per shard to make moving shards easier in a Map<Integer, AMOApplication>

Reconfiguration: A Potential Workflow

1. New config received from a `PaxosReply` from pinging the `ShardMaster` group
2. Replicate this into the paxos log (`if !replicated { propose; }`), and then continue when this operation is replicated
 - a. You will have to create a new command to wrap the config in to pass it to Paxos
3. Send shards to the Paxos Replica Group responsible for the Shard in the new configuration
 - a. Retry sending these shards until you receive an `Ack` (also need to replicate `Ack` received in paxos log). Don't resend to groups for which you have received an `Ack`
 - b. The shard group receiving the `ShardMove` request should put it in its log to replicate it. Then it can respond with an `Ack`.^{*} What if the shard comes from the future, i.e. with a higher `configNum` than what is currently known?
4. Receive Shards from the Paxos Replica Group responsible for your new Shards in the new configuration
5. Wait until `ShardAcksNeeded` and `ShardMovesNeeded` are both empty/0
 - a. Helpful utility method: Given a `groupId`, an old `Config`, and a new `Config`, what shards need to be moved? This can be used for both finding out what `ShardMoves` and `ShardMoveAcks` you need to wait to receive
6. Apply the `ShardMove` operations you have received when you are moving to a new config.
 - a. `KVStore` data updates and `<Client, SeqNum>` `AMOApp` updates

Note that messages need to be replicated for the state to be consistent among the servers in the groups.

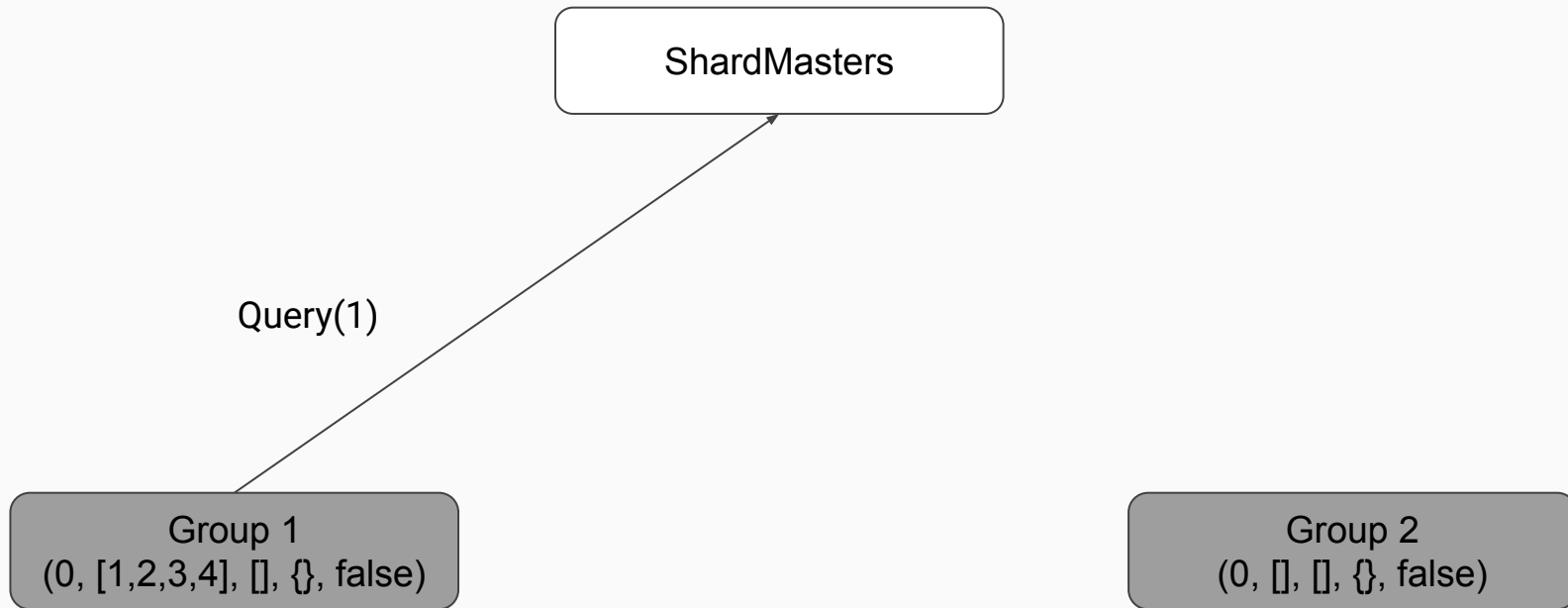
ShardMasters

Group 1
(0, [1,2,3,4], [], {}, false)

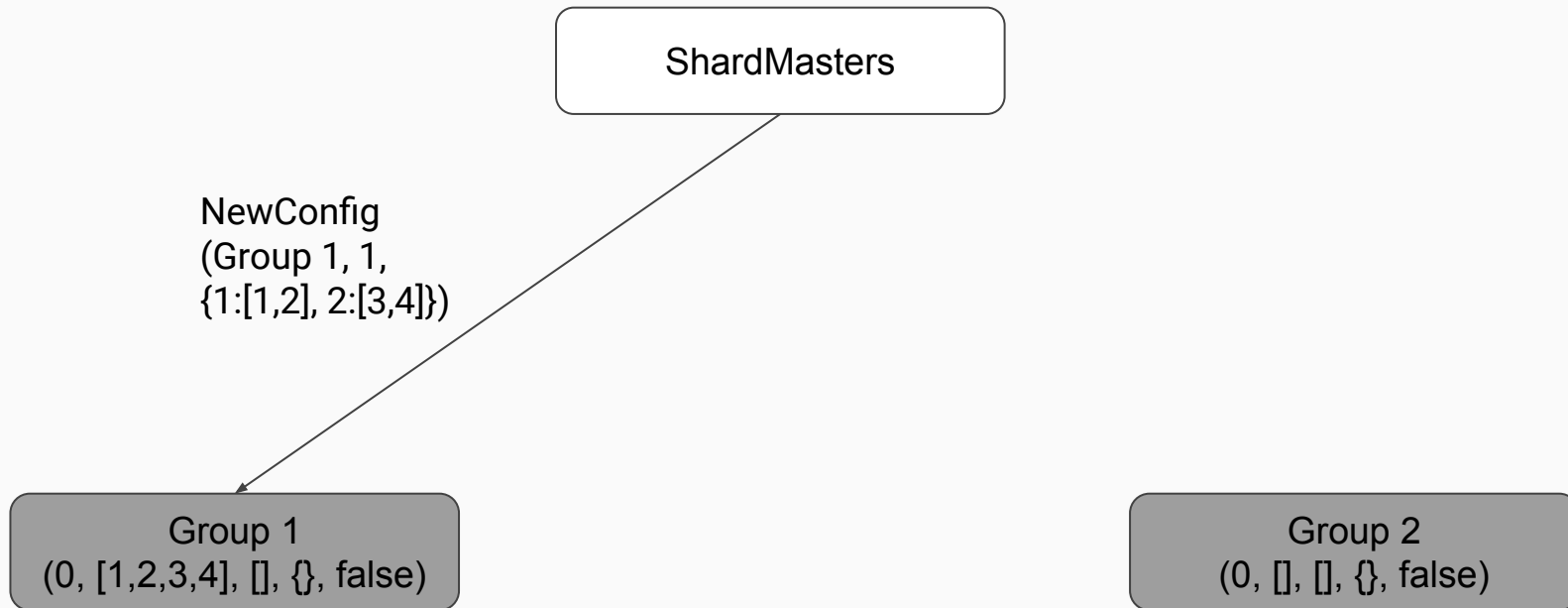
Group 2
(0, [], [], {}, false)

Note: Shards created in first config

(config num, shards owned, shards needed, shards to move, inReconfig)



(config num, shards owned, shards needed, shards to move, inReconfig)



(config num, shards owned, shards needed, shards to move, inReconfig)

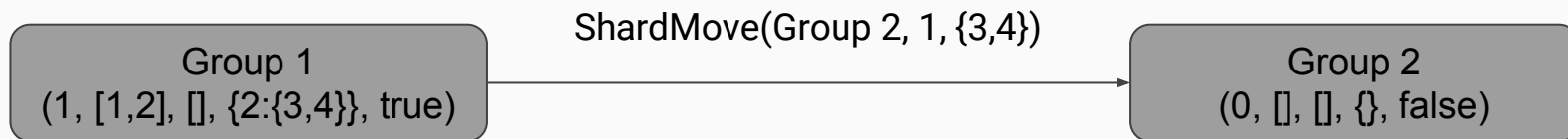
ShardMasters

Group 1
(1, [1,2], [], {2:{3,4}}, true)

Group 2
(0, [], [], {}, false)

(config num, shards owned, shards needed, shards to move, inReconfig)

ShardMasters



(config num, shards owned, shards needed, shards to move, inReconfig)

ShardMasters

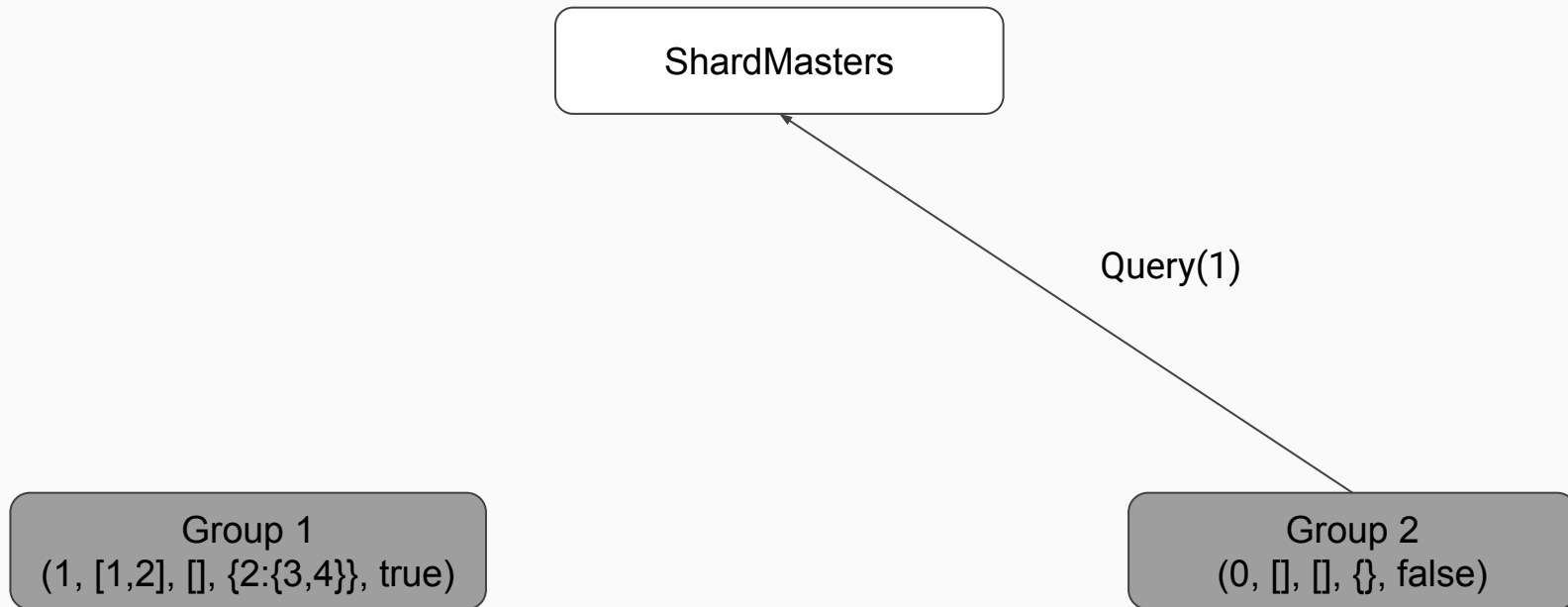
Group 1

(1, [1,2], [], {2:{3,4}}, true)

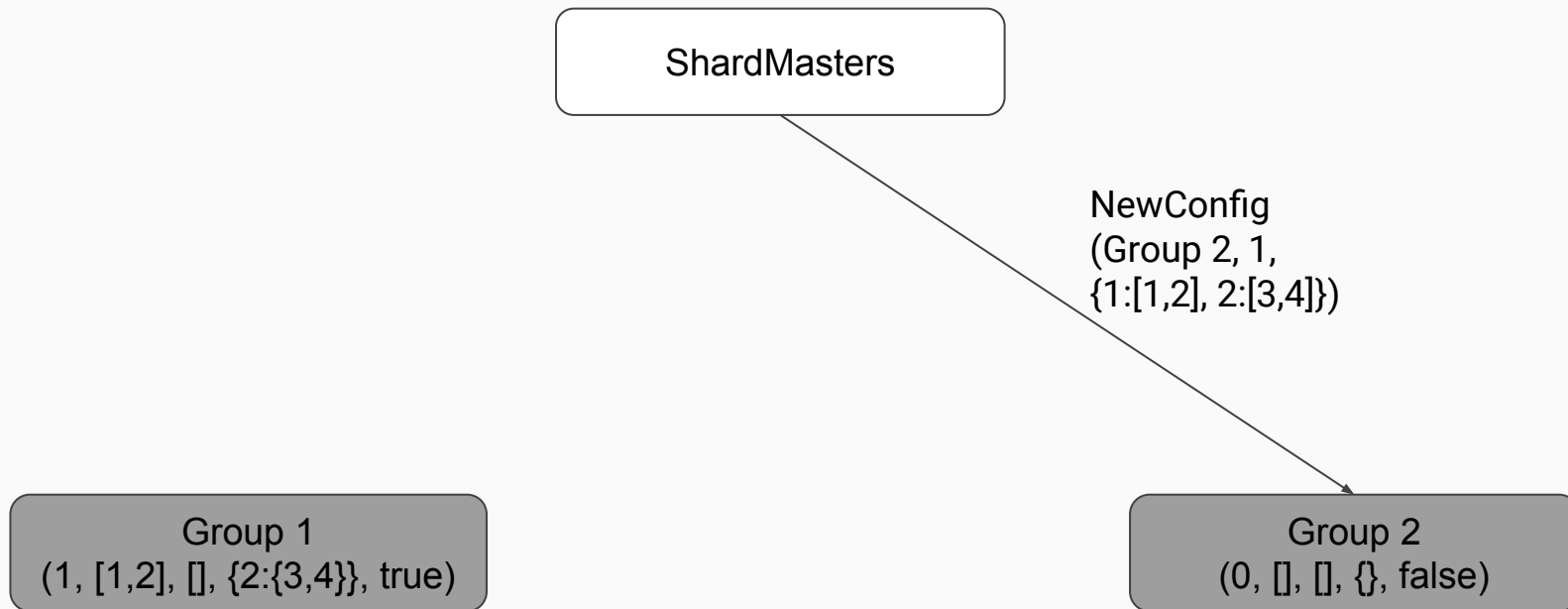
Group 2

(0, [], [], {}, false)

(config num, shards owned, shards needed, shards to move, inReconfig)



(config num, shards owned, shards needed, shards to move, inReconfig)



(config num, shards owned, shards needed, shards to move, inReconfig)

ShardMasters

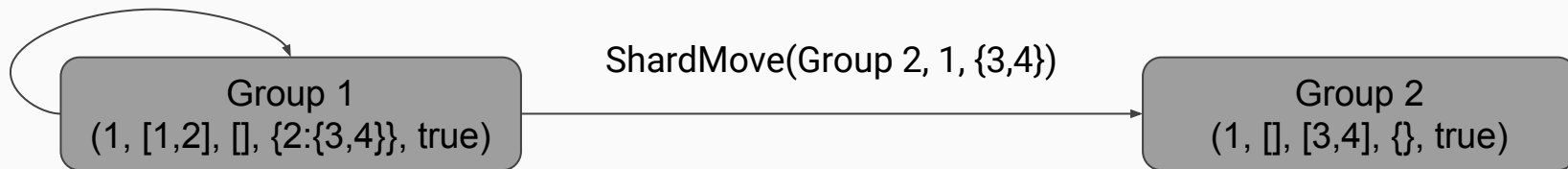
Group 1
(1, [1,2], [], {2:{3,4}}, true)

Group 2
(1, [], [3,4], {}, true)

(config num, shards owned, shards needed, shards to move, inReconfig)

ShardMasters

onShardMoveTimer(1, {2:{3,4}})



(config num, shards owned, shards needed, shards to move, inReconfig)

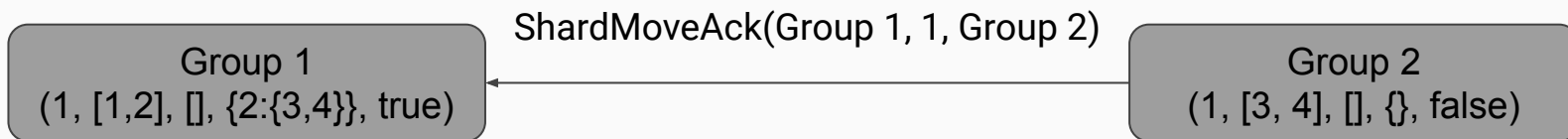
ShardMasters

Group 1
(1, [1,2], [], {2:{3,4}}, true)

Group 2
(1, [3, 4], [], {}, false)

(config num, shards owned, shards needed, shards to move, inReconfig)

ShardMasters



(config num, shards owned, shards needed, shards to move, inReconfig)

ShardMasters

Group 1
(1, [1,2], [], {}, false)

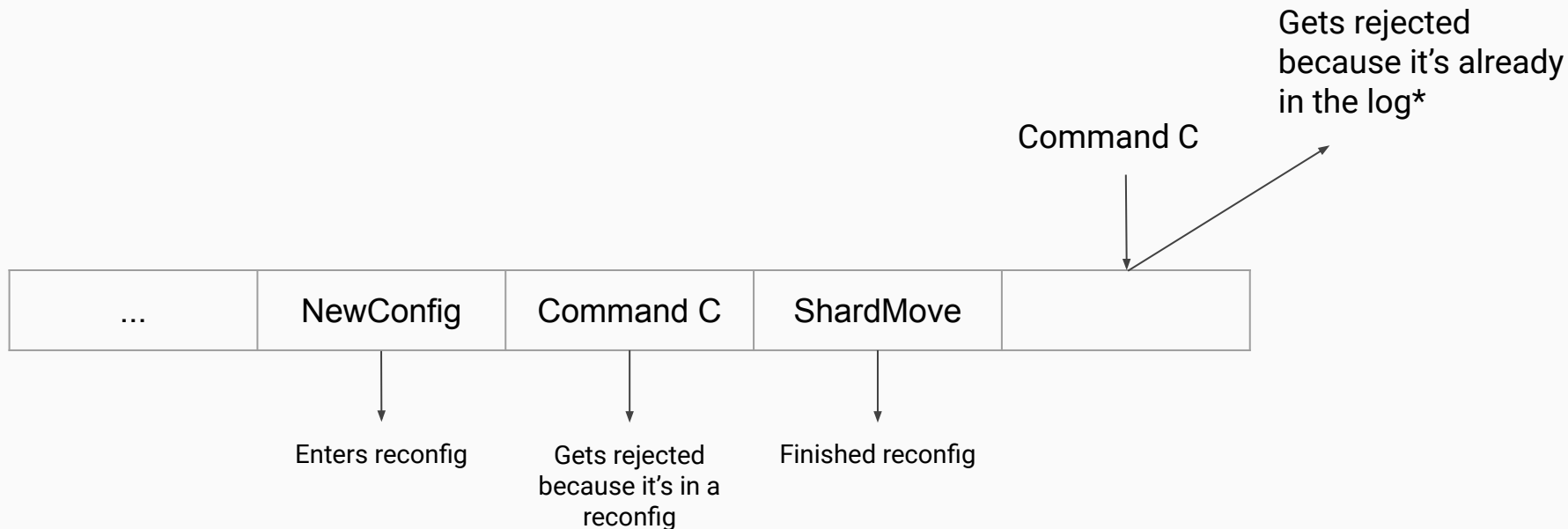
Group 2
(1, [3, 4], [], {}, false)

(config num, shards owned, shards needed, shards to move, inReconfig)

Maybe: Storing Heard Requests during a Reconfig

*=would be reproposed after garbage collection

Consider the case when



Maybe try to store replicated Commands you receive while you're doing a reconfig and iterating over them when you're done with the reconfig. (Make sure they get processed in the same order.)

Sending messages between groups

- You can tag shard moves and shard move acks with groupid information or the set of addresses to figure out who to send the result to, makes it easier if you get a lot of messages from older configurations

Misc. Tips

- If you don't pass 16,17,18 from Lab 3, you might want to consider going back to fix it if you're failing tests 6-9 on Lab 4
- Don't change state on a timer (e.g. enter reconfig), since it might lead to inconsistent state on the servers. Each of the ShardStoreServers in a group can start the timer at a different time, so if you change state based on a timer, then the ShardStoreServers might diverge in terms of behavior
- You need to define PaxosDecision [in its own file](#) so that it can be accessed by other packages

A note on read-only commands

- **IMPORTANT:** Check if `command.readOnly()` and use `app.executeReadOnly`
 - Otherwise the results of the Query commands will be cached and it'll just keep returning the same configuration if you don't use an increasing sequence number. If you aren't careful about using an increasing sequence number for the Query, the states may explode and you may fail search tests. Don't just short-circuit and respond with a stale read, you might get inconsistent responses.
- You can reply to read-only commands without taking a log slot if you do it carefully: see [Section 4.5](#) in PMMC
 - Issue: how to make sure the read sees an up-to-date chosen value, as of when the client issued the request (or later) but before the reply to the client; higher than any in progress op
 - Solutions:
 - Poll the acceptors for the slot with highest accepted value; do the read after that slot is decided
 - Another: hold the read until a new write request can be filled, and do the read before that write - since that confirms that you are still the (only) leader after the read operation starts
 - But we suggest not doing this, because it's tricky to get right

Why Can't This Happen?

Config 10

Group 1: shards 1,2,3

Group 2: shards 4,5,6

Group 3: shards 7,8,9,10

Config 11

Group 1: shards 2,3,4 (sent shard 1, received shard 4)

Group 2: shards 5,6 (sent shard 4)

Group 3: shards 8,9,10 (sent shard 7)

Group 4: shards 1,7 (received shards 1 and 7)

It doesn't make sense to have a group sending and receiving a shard. We had 3 shards moved, but the optimal solution should have only moved 2 shards from group 3 to group 4

You should only be waiting for other groups to ack the shards you're giving them (ShardAcksNeeded) or waiting for other groups to give you shards (ShardMovesNeeded)

Clients and Servers should only accept increasing configuration numbers

Clients want the most up-to-date state

- You'll probably want checks to make sure that the state is

Servers want the state for the next configuration that they can transition to, if there is one

- Query for the next configuration

Potential Workflow for part 2

1. Constructor for Paxos subnodes and communication between subnode and parent node
2. Process/replication of commands outline
3. Timer for getting new configurations
 - a. Be able to process the first configuration and create shards
4. Logic for processing Client requests [Should pass test 1 for part 2]
5. Logic for processing reconfigurations
 - a. inReconfig, shardsNeeded, ShardMoves, ShardMoveAcks, ShardMoveTimers, acknowledging duplicates in case of drops
 - b. Client requests during reconfiguration

Using the visualizer

See the bottom of the lab 4 spec for more details

```
./run-tests.py -l 4 -d NUM_GROUPS NUM_SERVERS_PER_GROUP  
NUM_SHARDMASTERS NUM_CLIENTS CLIENT_WORKLOAD [CONFIG_WORKLOAD]
```

Additional notes [not mentioned in spec]

- configController which handles all of the configuration changes like Joins/Leaves/Moves is a PaxosClient

Lab 4 Part 3: Two-phase commit

- Keys partitioned over different hosts; one coordinator per transaction
- Acquire locks on all data read/written; release after commit
- To commit, coordinator first sends prepare messages to all involved shard-holders; they respond prepare_ok (acquiring locks) or abort.
 - if prepare_ok, they must be able to commit transaction
 - Paxos is used to replicate the prepare log entry (including locks) as well as commit and abort messages
 - These messages should be replicated
- If all prepare_ok, coordinator sends commit to all; they write commit record and release locks

Lab 4 Part 3: Hints

All `ShardStoreServer` nodes tag their transaction-handling messages with their configuration number.

Servers reject any prepare requests coming from different configurations, causing the transaction to abort.

Servers delay reconfigurations when there are outstanding locks for keys (i.e., there are transactions pending in the previous configuration).

Section 9: Lab 4 Part 3

CSE 452 Spring 2021



Goal of Lab 4 Part 3

- Support transactions across multiple keys potentially located across different replica groups
 - Transactions can be:
 - Series of reads
 - Series of writes
 - Swaps - new one :)
- Use Two-Phase Commit to acquire locks during transaction.
- Why can't we just use Paxos? Because each group is responsible for a different set of shards and we want to support transactions across shards/replica groups.

Two-Phase Commit

1. Allows all participants to arrive at same conclusion.
2. Pick a coordinator (i.e leader of the transaction)
3. Coordinator sends out prepares to all replica groups responsible for a given key (participants)*
4. Participants check if the key of the transaction is locked
 - a. If so, reply with an abort
 - b. Otherwise, reply `PrepareOK` and lock the key to prevent reads and writes on it
5. If Coordinator received `PrepareOKs` from all groups, send out commits
6. Commits actually perform read/write, and reply back to coordinator
7. Coordinator replies back to the client once all `CommitOks` are received and Results are combined

- **Transaction interface**
 - `Set<String> readSet();`
 - `Set<String> writeSet();`
 - `Set<String> keySet() // union of readSet and writeSet`
- **Implementations of Transaction**
 - MultiGet
 - `Set<String> keys`
 - **Read one or more keys across replica groups**
 - MultiPut
 - `Map<String, String> values`
 - **Write one or more keys across replica groups**
 - Swap
 - `String key1, key2`
 - **Given two keys, swap their values**
- **Your goal: implement support for these 3 types of Transactions**

Transactions: The Normal / Happy Path from Coordinator's perspective

- Suggestion: Transaction coordinator is the replica group with the largest group ID in the transaction
- Client sends the Transaction to the coordinator
- Coordinator replicates Transaction through its Paxos subnodes
 - In general, incoming messages to replica groups should be replicated (just like in part 2).
 - Use the same design of `process(Command c, boolean replicated)`
- The coordinator will run 2 phase commit to execute the transaction
 - Prepare
 - Coordinator sends out prepares to all replica groups involved in the transaction
 - Once coordinator receives `prepareOKs` from every replica group (not majority anymore, this ain't Paxos), can proceed to commit stage
 - Commit
 - Coordinator sends out commit message to all replica groups involved in transaction
 - Participants can unlock keys after receiving commit
 - Coordinator responds to client after all `CommitOKs` received from participants

Transactions: The Normal / Happy Path from Participant's perspective

- Upon receiving prepare:
 - Validation: check config nums match
 - Participants check if key locked
 - If locked, tell Coordinator to abort
 - Coordinator should send aborts, wait for client to retry the entire transaction
 - If not locked, lock keys and send `PrepareOK`
- Once coordinator receives `PrepareOK` from every replica group (NOT just majority anymore, this still isn't Paxos), the coordinator will send commit messages to participants
- Upon receiving commit:
 - Replicate in Paxos
 - Verify that commit is for the correct transaction, config num matches, and attempt num matches
 - If write commit
 - Perform write and update application state
 - Update client address -> highest sequence number seen mapping
 - Might want some state to store results of transactions somewhere
 - Unlock keys
 - Respond with `CommitOK`

keys={b,c}



keys={a}

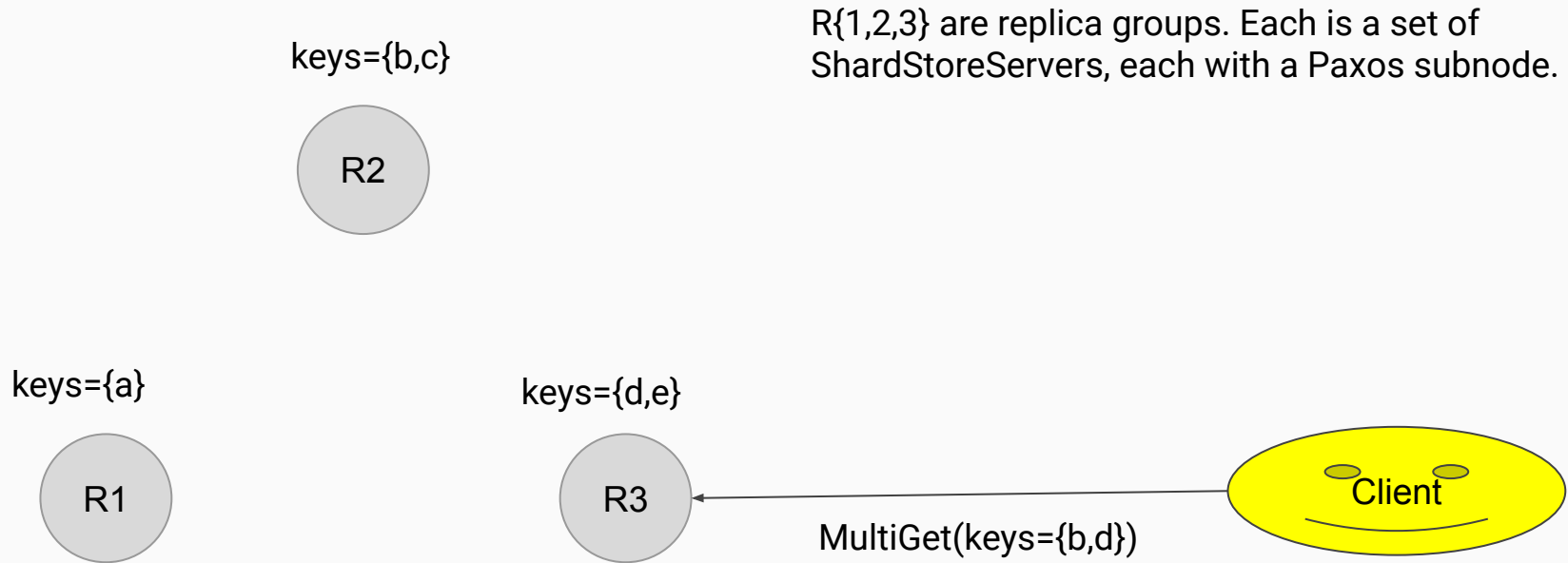


keys={d,e}



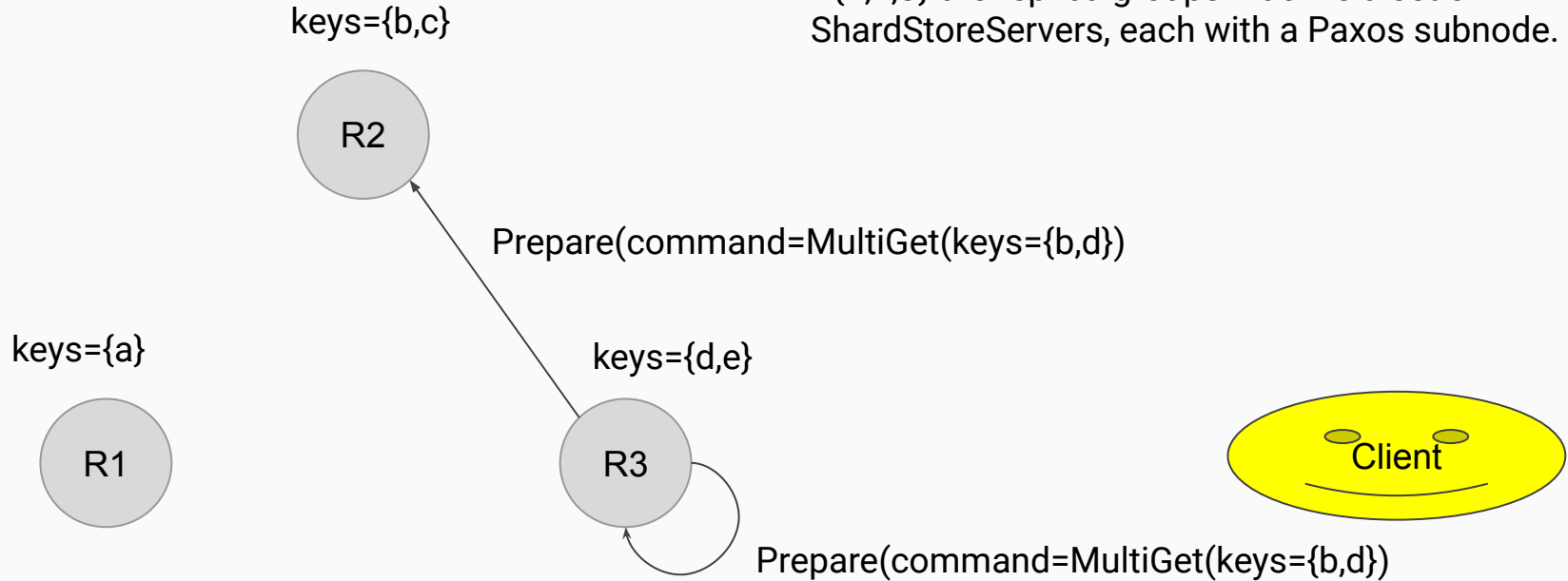
R{1,2,3} are replica groups. Each is a set of ShardStoreServers, each with a Paxos subnode.





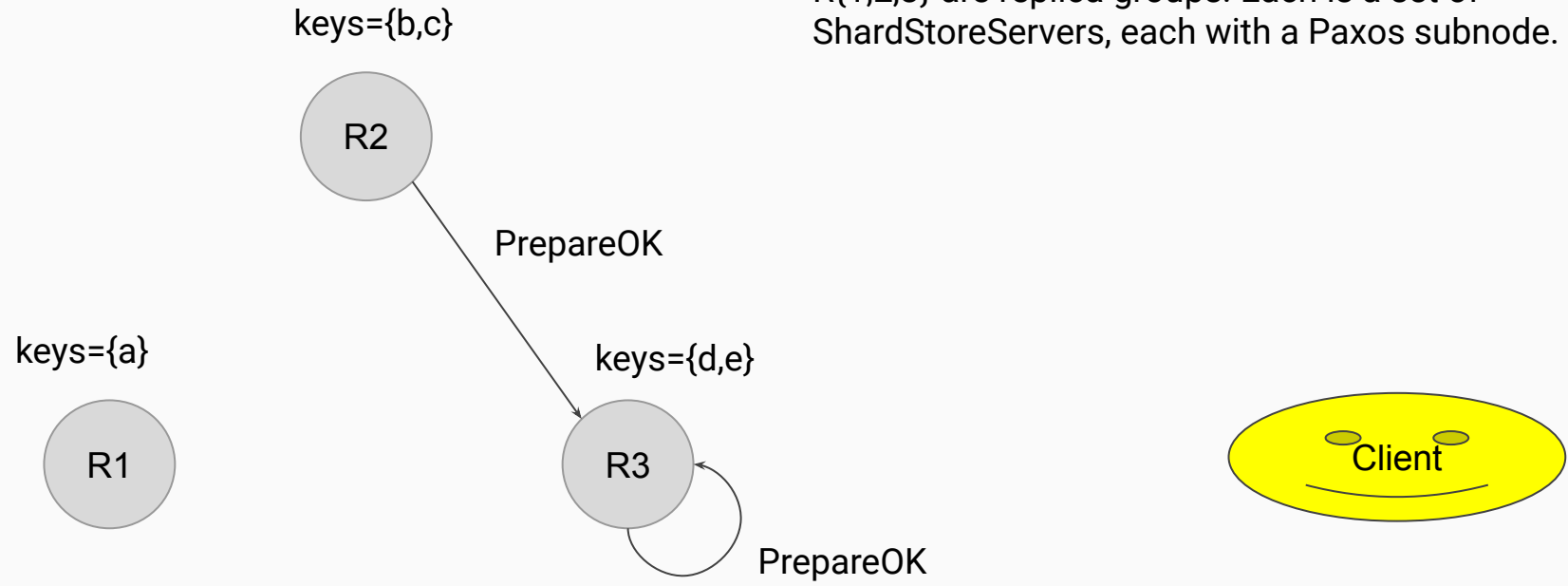
Client sends a request to the Coordinator. Coordinator is the $\max(\{\text{replica group ids involved in this transaction}\})$. In this case, R3 (for key=d) and R2 (key=b) are involved, $\max(3,2) = 3$. So replica group 3 is coordinator.

R{1,2,3} are replica groups. Each is a set of ShardStoreServers, each with a Paxos subnode.



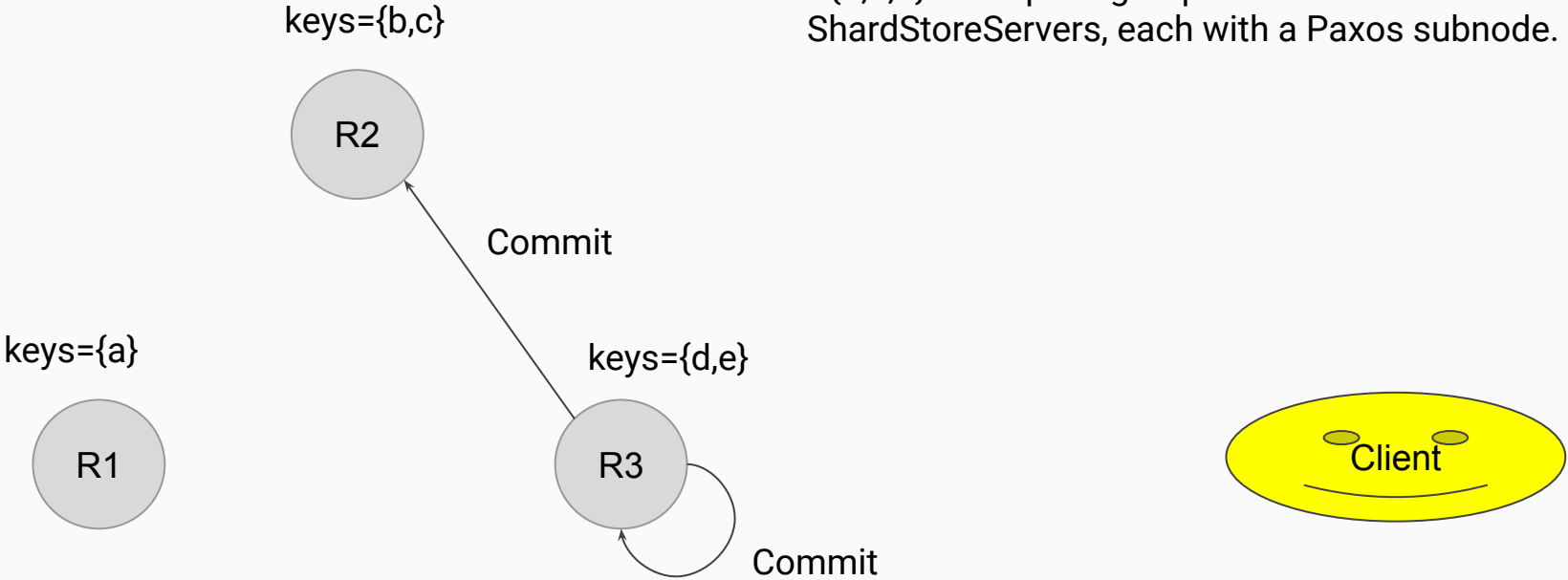
Coordinator sends Prepare messages to all participants in the transaction. Participant should lock keys involved in transaction.

R{1,2,3} are replica groups. Each is a set of ShardStoreServers, each with a Paxos subnode.

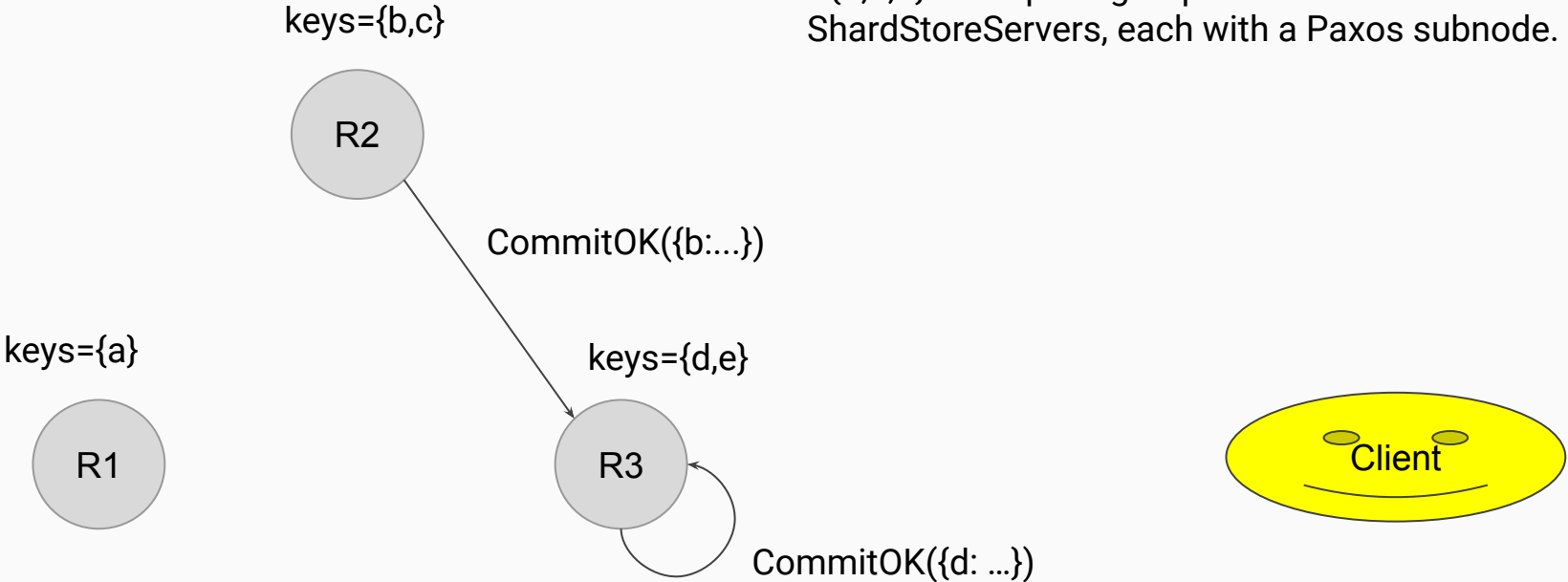


Each replica group responds with PrepareOK (including the information on what it's responding to and maybe some values depending on what command is being performed [swap])

R{1,2,3} are replica groups. Each is a set of ShardStoreServers, each with a Paxos subnode.



R{1,2,3} are replica groups. Each is a set of ShardStoreServers, each with a Paxos subnode.



R{1,2,3} are replica groups. Each is a set of ShardStoreServers, each with a Paxos subnode.

keys={b,c}



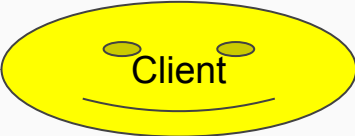
keys={a}



keys={d,e}



Reply({b: ..., d: ...})



Tips

- Groups can be in two roles: participant and coordinator. May need to handle sending messages to your own group (i.e. if a given group is both a participant and coordinator).
 - Similar to how in Paxos nodes can be both acceptors/proposers.
- Keep track of ongoing (active) transactions at each node (each transaction could keep track of a set of keys that are currently locked)
- What happens if coordinator receives an abort?
 - It may be the case that coordinator config is out of sync
 - Participants may reject prepare from coordinator.
 - If coordinator learns of a new config while receiving prepare responses, the coordinator should finish any outstanding transactions, send out aborts as a response to other transactions if needed and process the config change. Then client will retry and will start a new transaction.
 - This means each transaction should have an attempt / retry # associated with it. If a transaction needs to be retried, the coordinator needs to reach out to all participants again as part of a new attempt #. Attempt # needed so participants can differentiate different transaction attempts from a given coordinator. Needs to deal with attempts from different configs.

Aborts

- Make sure that the aborts you receive are valid (right transaction, config, and attempt number)
 - Valid aborts from the coordinator should unlock the locks
- Safe to send aborts as long as you haven't sent a `prepare_ok` for the current attempt
- Coordinator should accept `AbortOks` to end the transaction attempt
- Cases when aborts can happen:
 - Coordinator sends `prepare` for a transaction but key is locked on participant
 - Replicate abort on Paxos subnodes, send back `Abort` to coordinator
 - Coordinator should end up trying again with a new round of prepares
 - Participant has higher config number than coordinator's config number
 - What should happen when the participant is in the middle of a reconfiguration?

Potential Workflow for part 3

Tackling the entire two-phase commit process at once can be difficult, so one suggestion of what order to get things done is

1. SingleKeyCommands and Transactions that are all on one shard (no 2PC)
2. Transactions that span shards within one group (no 2PC). [Should pass test 1 for part 3]
3. 2-phase commit to everyone (involved), coordinator logic, transaction attempts:
 - a. Sending Prepares and locking keys, PrepareOks
 - b. Commits and CommitOks
 - c. Sending Aborts for different configuration, already locked keys, processing AbortOks
 - d. Retry transaction if necessary/possible
 - e. Adding logic for interaction with reconfigurations

Note: Sending to everyone (involved) would require getting AbortOks from everyone (involved) if an abort happens

Note: 3. is the bulk of the work because you need these messages and also timers to retry as necessary (for Prepares, Commits, Aborts) [can delay timers until unreliable tests]

Potential Workflow for part 3 [continued]

4. MultiGet and MultiPut through 2PC
5. Swap through 2PC (slightly more complicated)
 - a. Swap values are aggregated on prepares and written on commit
6. Sending prepares (grabbing locks) in order to avoid live-locking
 - a. Ordering for who to contact next
 - b. Sending out Aborts to people you contacted in case of an Abort and receiving AbortOks from them