Consensus & RAFT RAFTER

Tom Santero

I 

cats

newports

distributed systems

Andrew Stone

I 

cats

bread

distributed systems

@tsantero

@andew_j_stone

tsantero

andewjstone

tsantero

astone

@

basho.com

tsantero

@

basho.com

astone

(notice Andrew's contact keeps getting shorter?)

http://thinkdistributed.io

A Chris Meiklejohn Production

The Usual Suspects

“Strongly Consistent Datastores”

MongoDB

Redis

MySQL

others…

async { replication / disk persistence

Failure Detection

Problem?

Failure Mode 1

Single Node w/ async disc writes

Data is written to fs buffer, user is sent acknowledgement, power goes out

Data not yet written to disk is LOST

System is UNAVAILABLE

Single Disk Solutions: fsync, battery backup, prayer

Failure Mode 2

Master/Slave with asynchronous replication

Data is written by user and acknowledged

Data synced on Primary, but crashes

Consistent ? Available

Consistent ? Available

Primary Failed. Data not yet written to Secondary

Write already ack’d to Client

Consistent ? Available

```
if promote_secondary() == true;
  {
    stderr(“data loss”);
  }
else
  {
    stderr(“system unavailable”);
  }
```

(╯°□°)╯︵ ┻━┻

Synchronous Writes FTW?

PostgreSQL / Oracle

Master / Slave

Ack when Slave confirms Write

Problem?

Problem?

Failure Detection

Automated Failover

“split brain” partitions

Solution!

Solution!

Consensus protocols!
(Paxos, ZAB, Raft)

Safe Serializability

RYOW Consistency

What is Consensus?

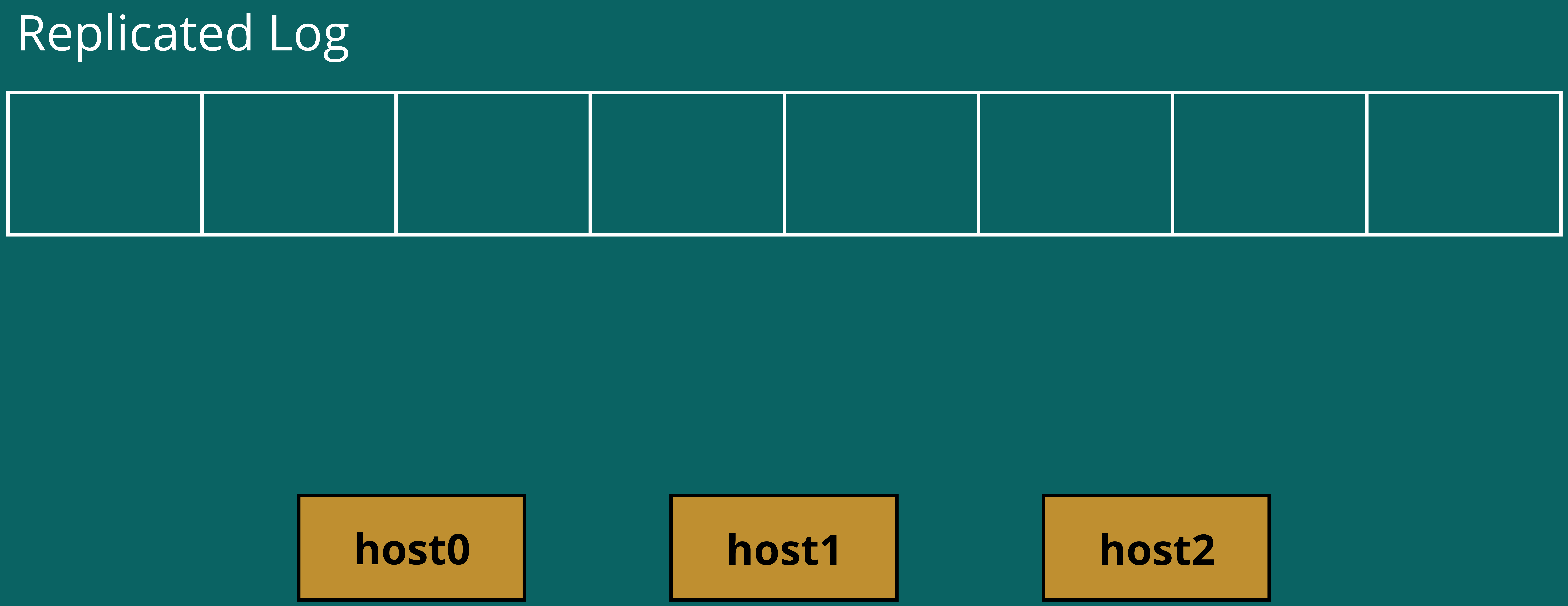
“The problem of reaching agreement among remote processes is one of the most fundamental problems in distributed computing and is at the core of many algorithms for distributed data processing, distributed file management, and fault-tolerant distributed applications.”

In a distributed system…

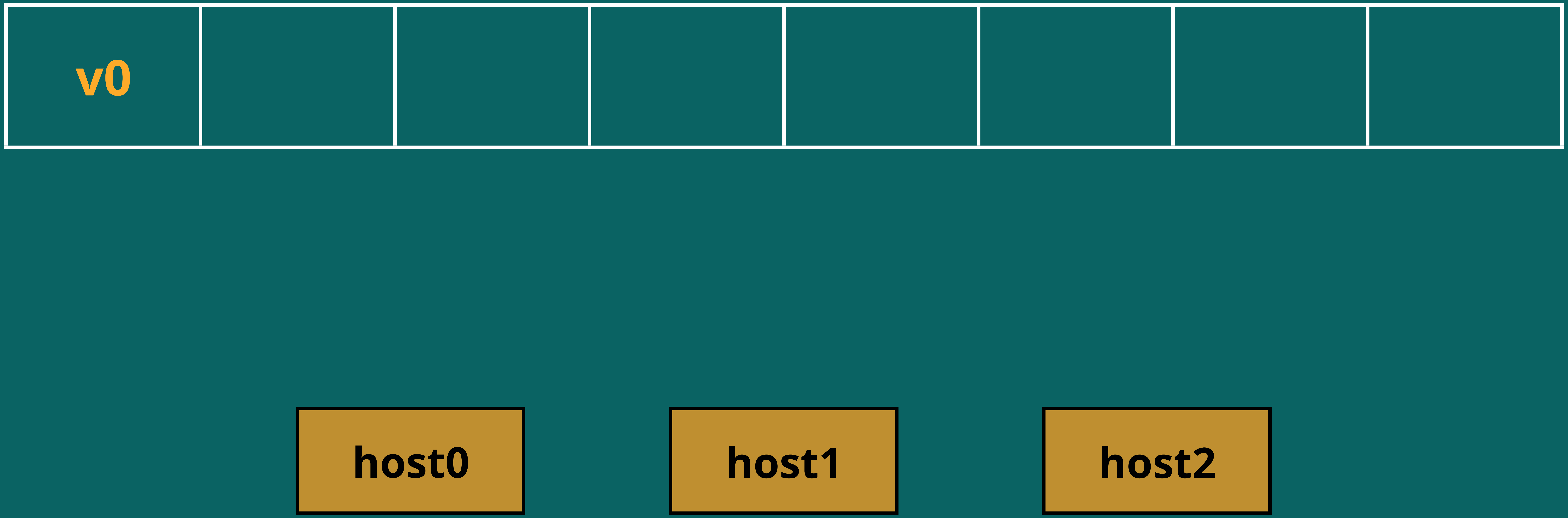
multiple processes

agreeing *on* a value

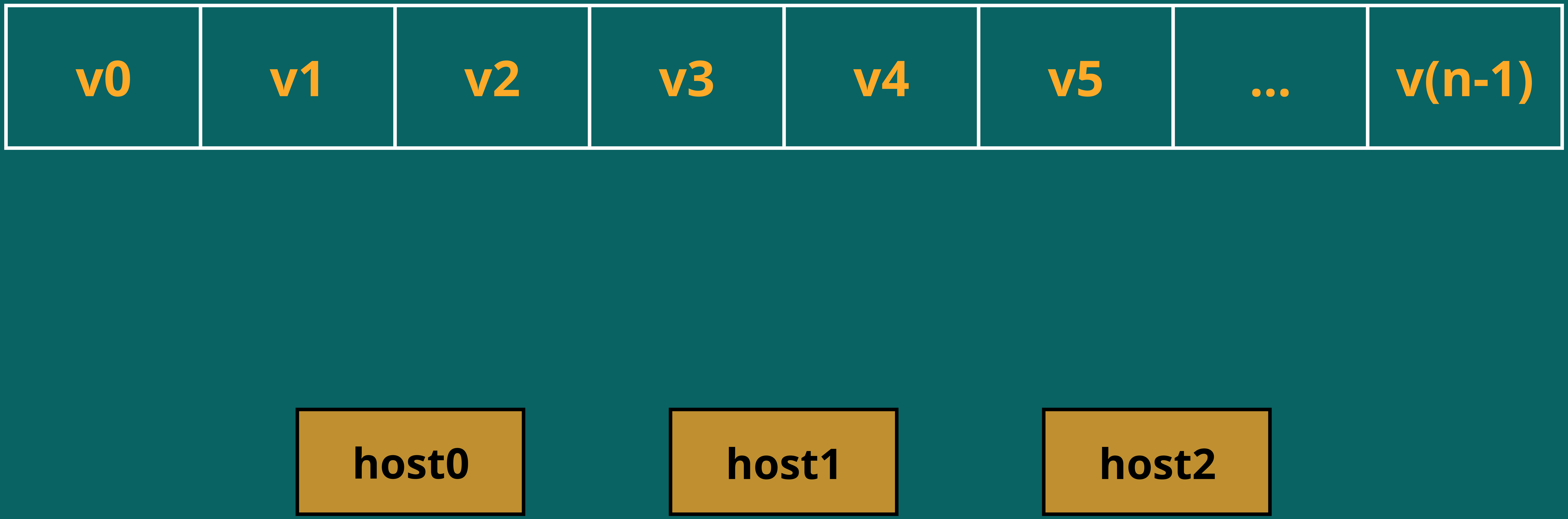
despite failures.


Replicated Log
host0
host1
host2

Replicated Log



Replicated Log



Consensus

- termination
- agreement
- validity

Consensus

termination

non faulty processes eventually decide on a value

agreement

validity

Consensus

termination

non faulty processes eventually decide on a value

agreement

processes that decide do so on the same value

validity

Consensus

termination

non faulty processes eventually decide on a value

agreement

processes that decide do so on the same value

validity

value must have been proposed


Theoretical
Real World

Back to 1985...


Back to 1985...
The
FLP
Result

Safety & Liveness

bad things can’t happen

good things
<u>eventually</u> happen

Consensus

termination

non faulty processes eventually decide on a value

agreement

processes that decide do so on the same value

validity

value must have been proposed

Safety

Liveness

termination

non faulty processes eventually decide on a value

agreement

processes that decide do so on the same value

validity

value must have been proposed

Safety

Liveness

termination

non faulty processes eventually decide on a value

agreement

processes that decide do so on the same value

validity

value must have been proposed

Safety

Liveness

termination

non faulty processes eventually decide on a value

agreement

processes that decide do so on the same value

validity

value must have been proposed

Safety

non-triviality

Liveness

termination

non faulty processes eventually decide on a value

agreement

processes that decide do so on the same value

validity

value must have been proposed

The FLP Result:

perfect Safety and Liveness in async consensus is impossible

Symmetric

vs

Asymmetric

Raft

Motivation: RAMCloud

large scale, general purpose, distributed storage

all data lives in DRAM

strong consistency model

https://ramcloud.stanford.edu/

Motivation: RAMCloud

large scale, general purpose, distributed storage

all data lives in DRAM

strong consistency model

100 byte object reads in 5μs

https://ramcloud.stanford.edu/

In Search of an Understandable Consensus Algorithm

Diego Ongaro John Ousterhout

https://ramcloud.stanford.edu/raft.pdf

“Unfortunately, Paxos is quite difficult to understand, in spite of numerous attempts to make it more approachable. Furthermore, its architecture is unsuitable for building practical systems, requiring complex changes to create an efficient and complete solution. As a result, both system builders and students struggle with Paxos.”

mrb
@mrb_bk

Following

@argvo I actually explained Raft (properly, I believe) to someone after like 3 drinks last night so that's why I think it's here to stay

Reply Retweet Favorite More

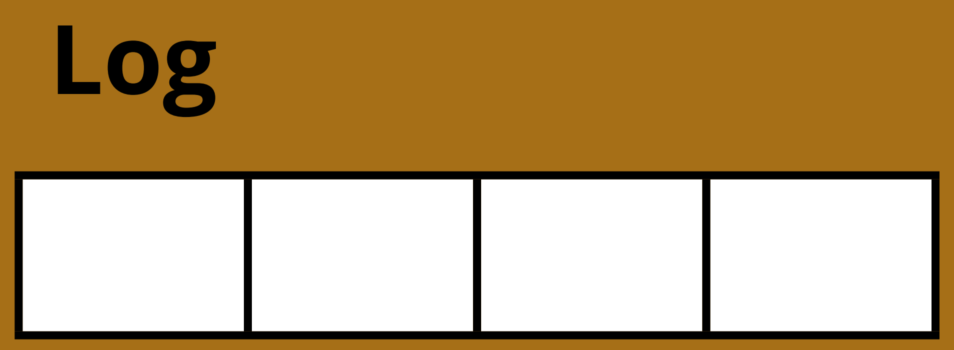
Design Goals:

Understandability & Decomposition

Strong Leadership Model

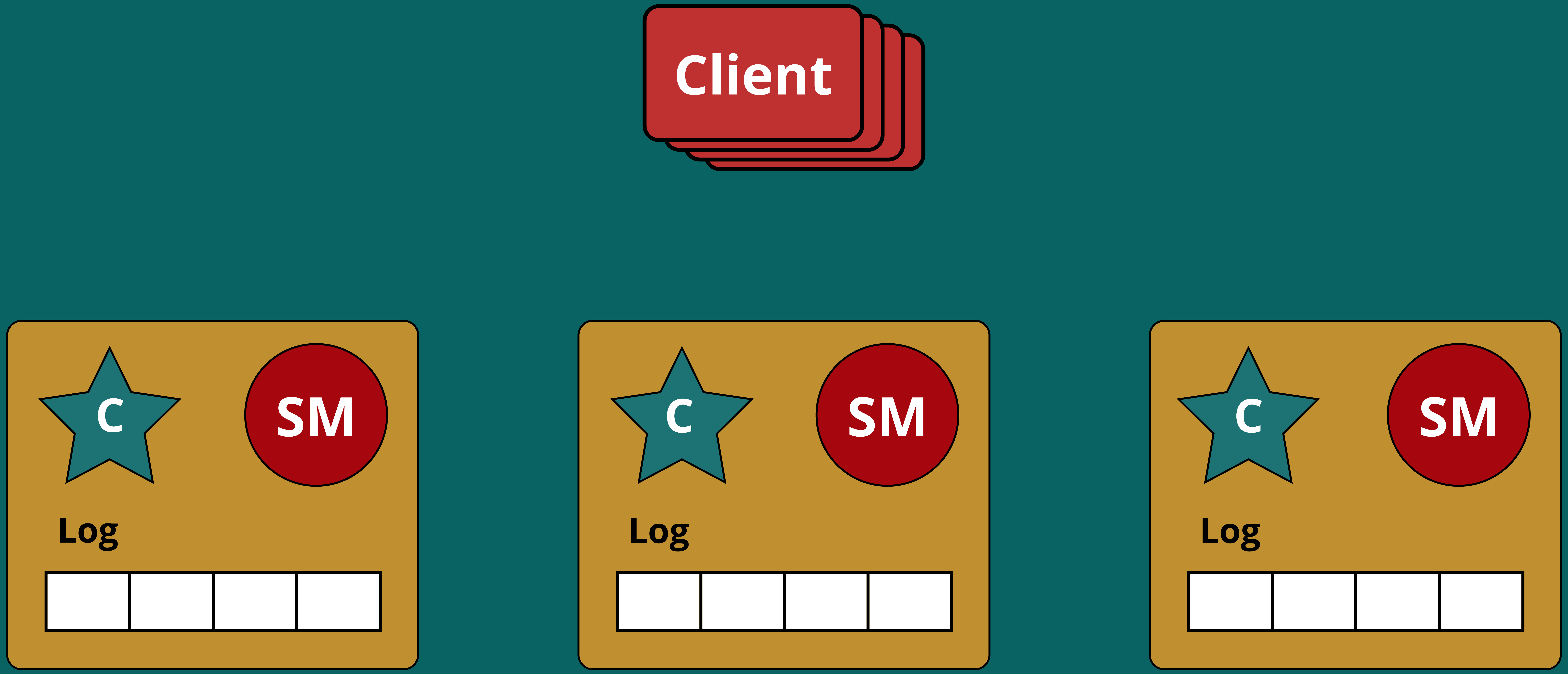
Joint Consensus for Membership Changes

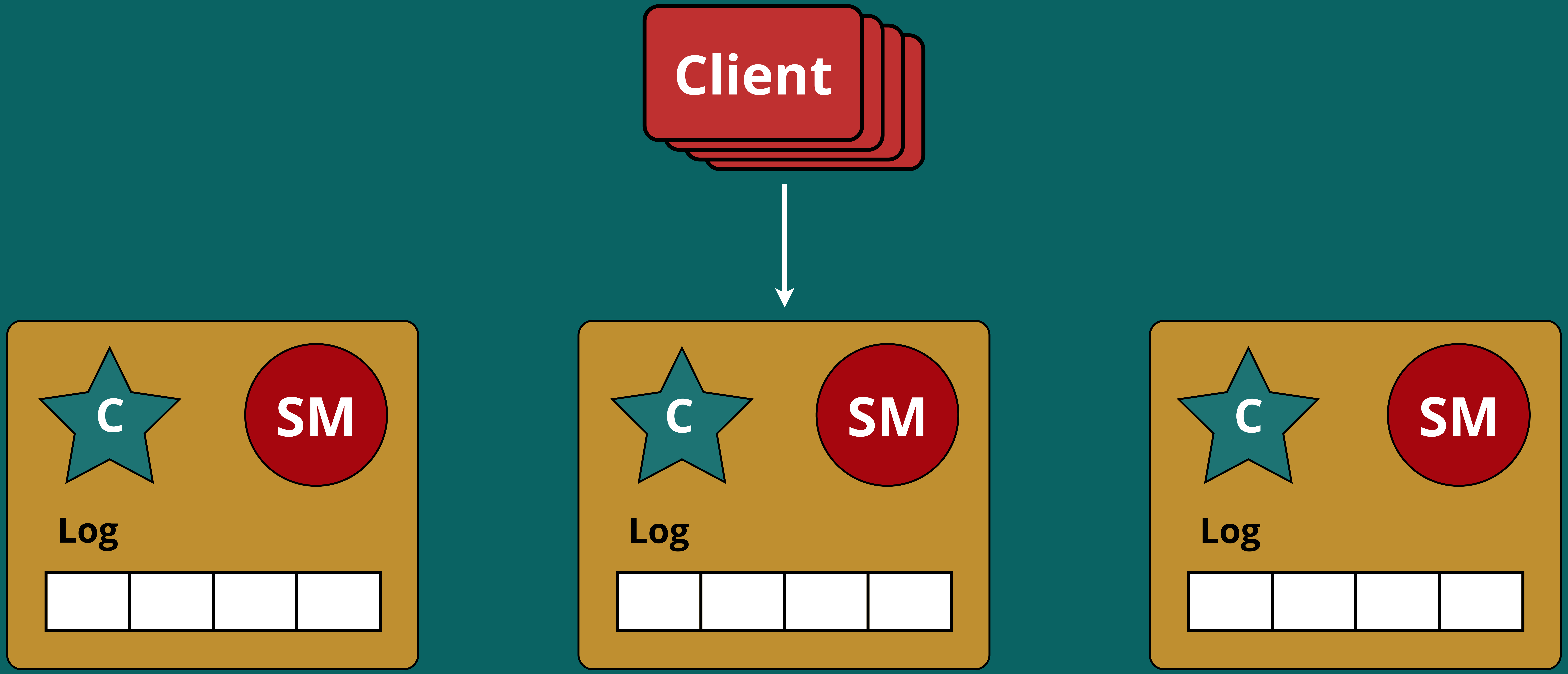
Consensus Module



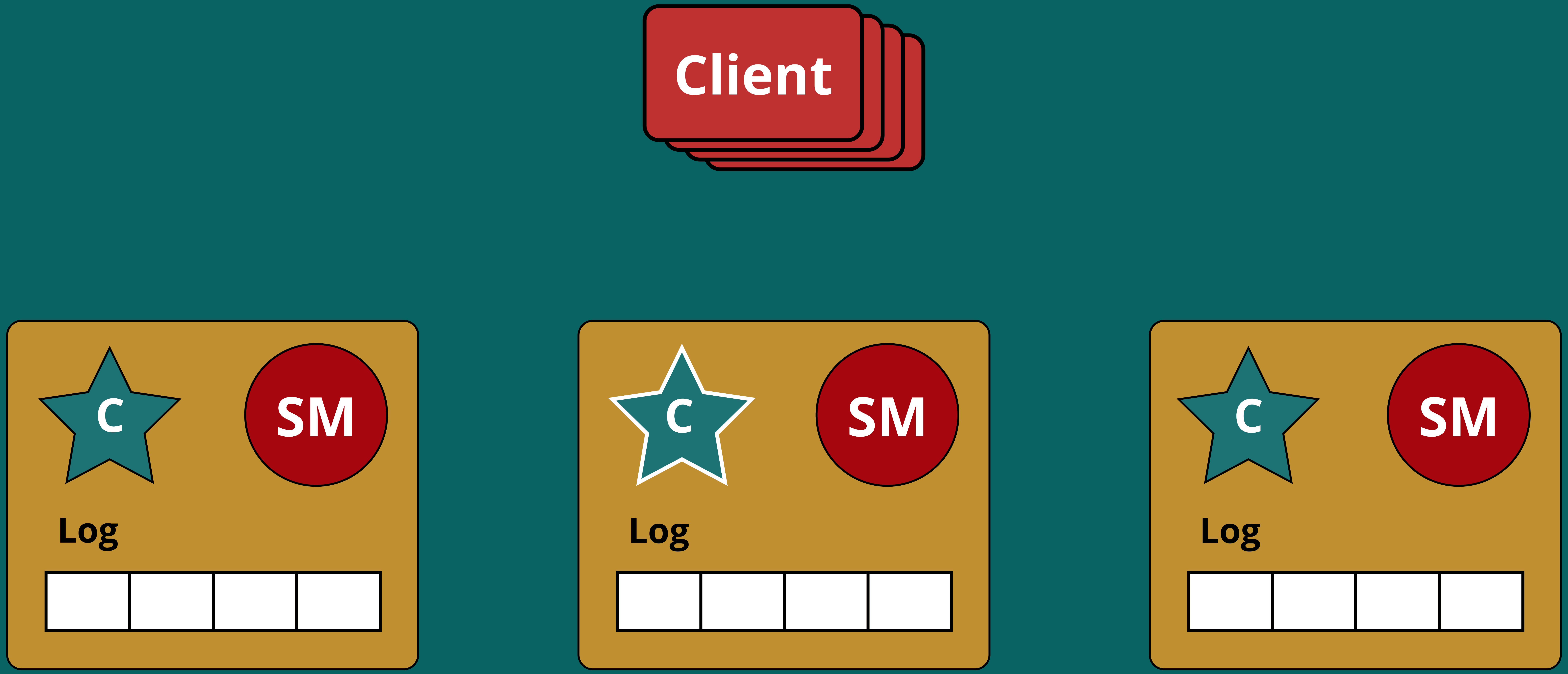
Replicated Log

State Machine

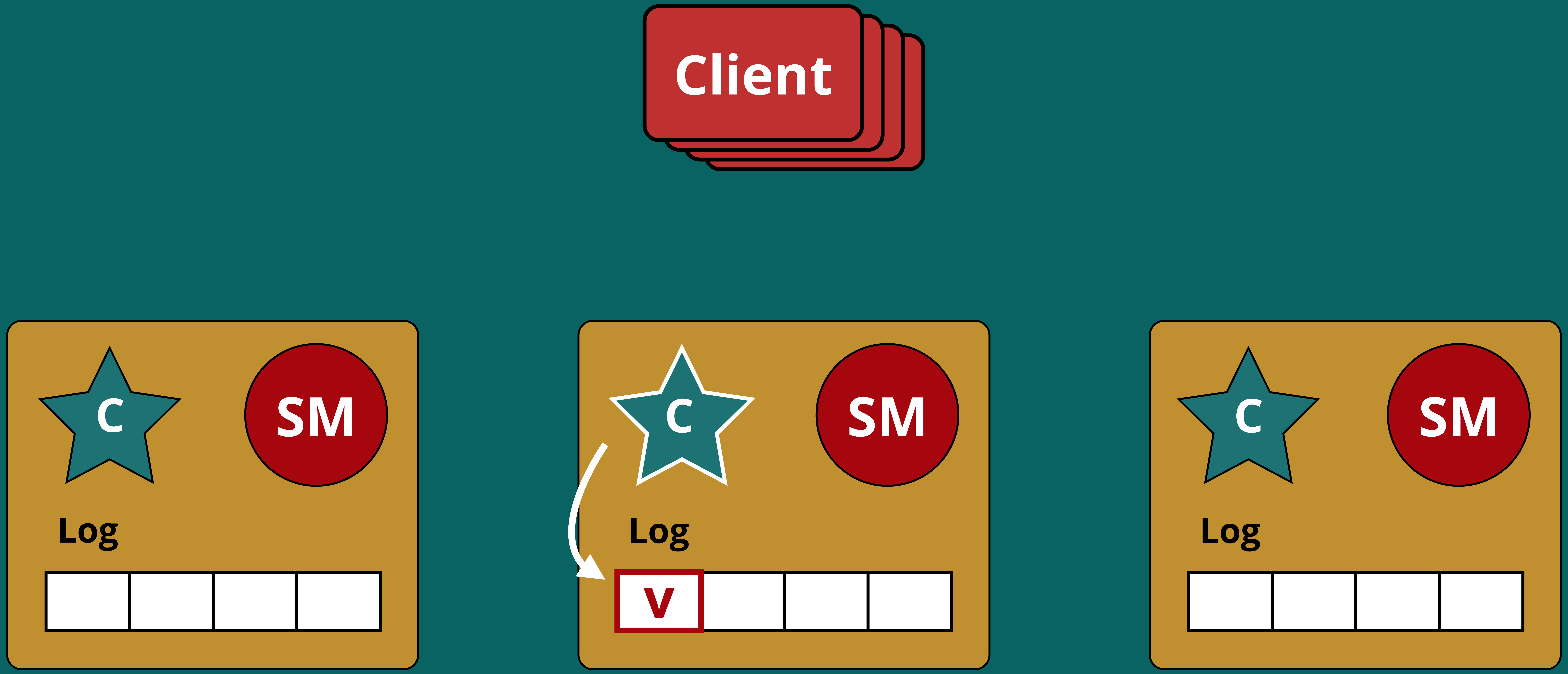

Client
C
SM
Log
C
SM
Log
C
SM
Log



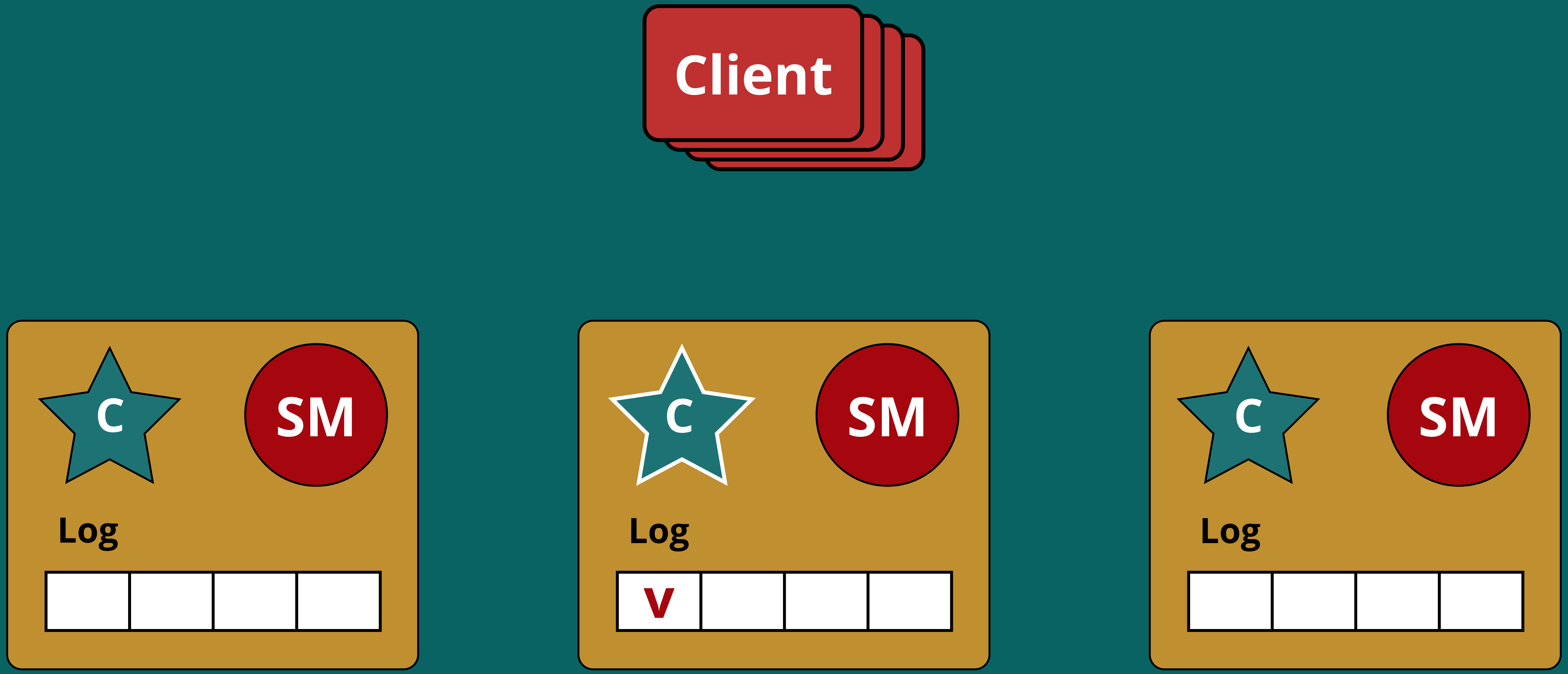
1. client makes request to Leader

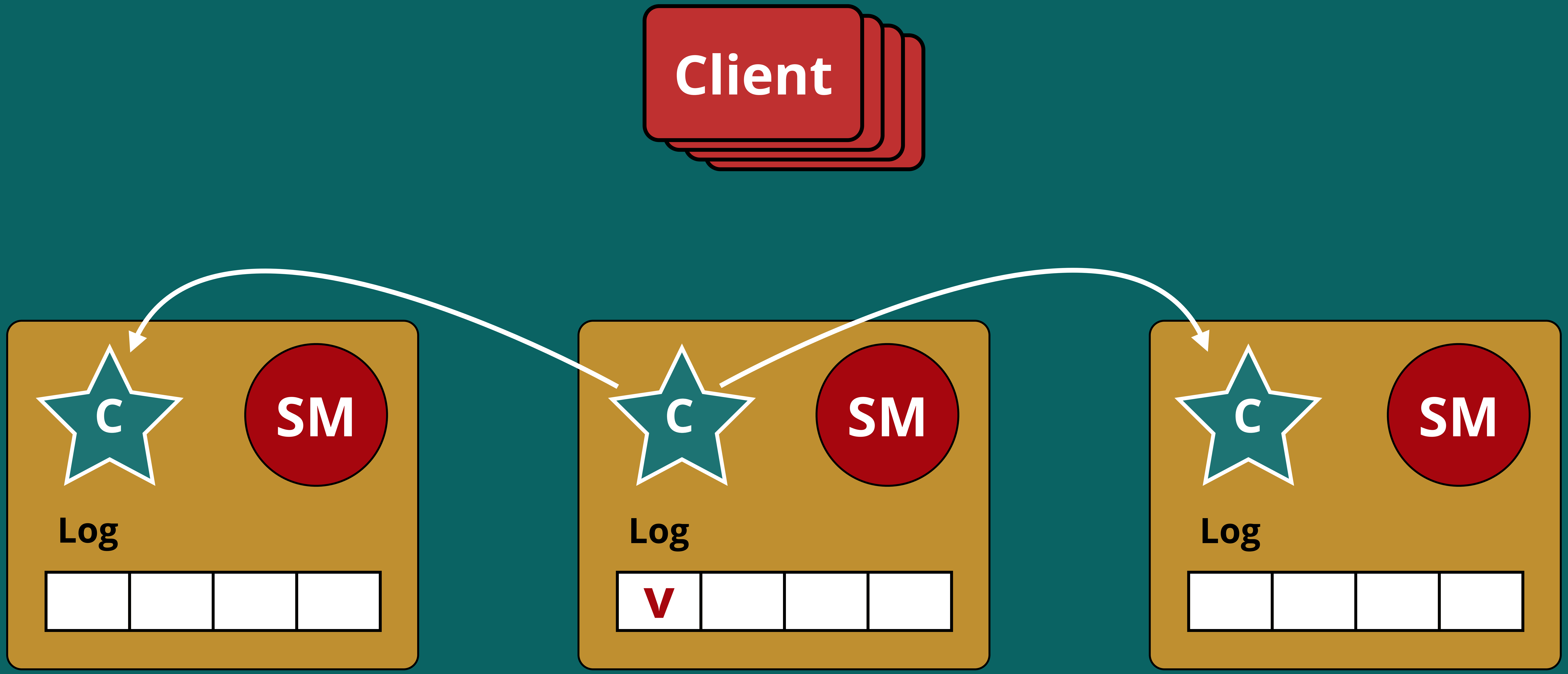


2. consensus module manages request

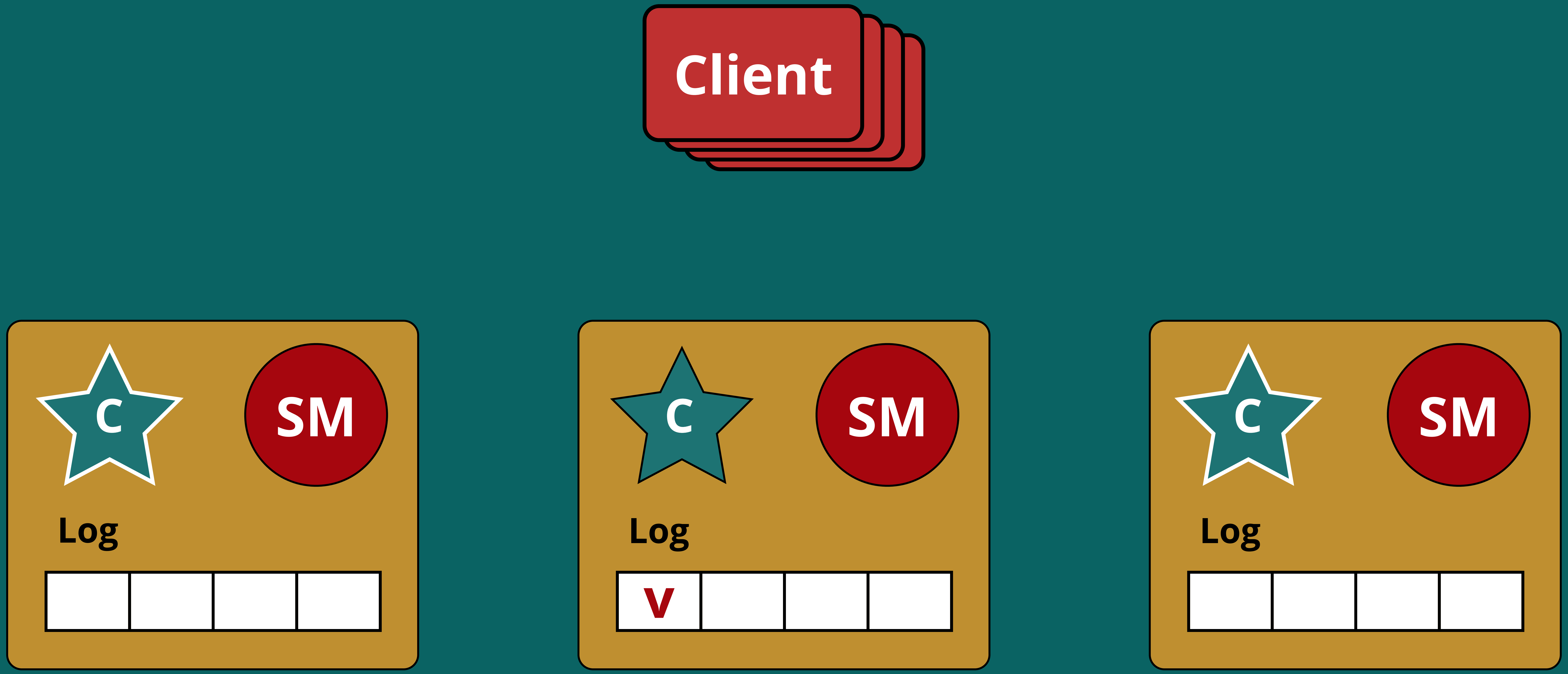


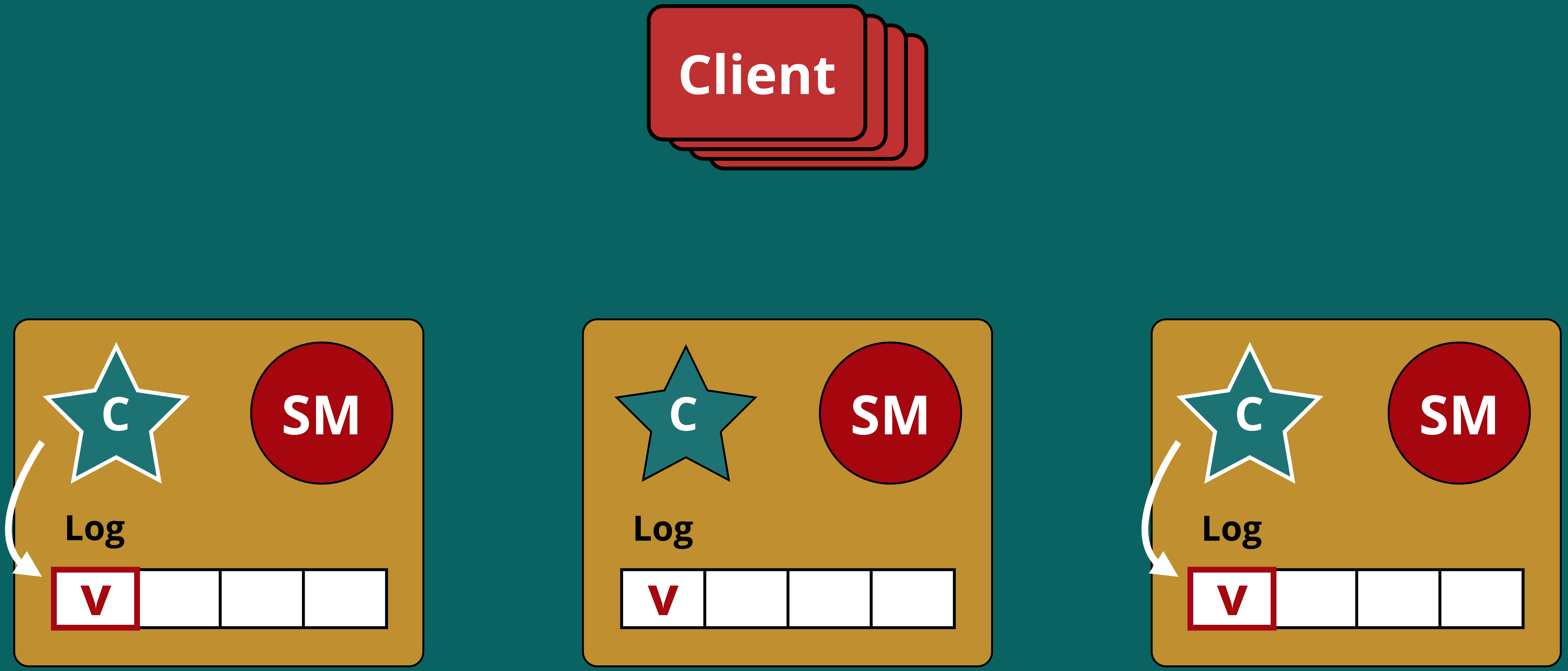
3. persist instruction to local log


Client
C
SM
Log
C
SM
Log
V
C
SM
Log

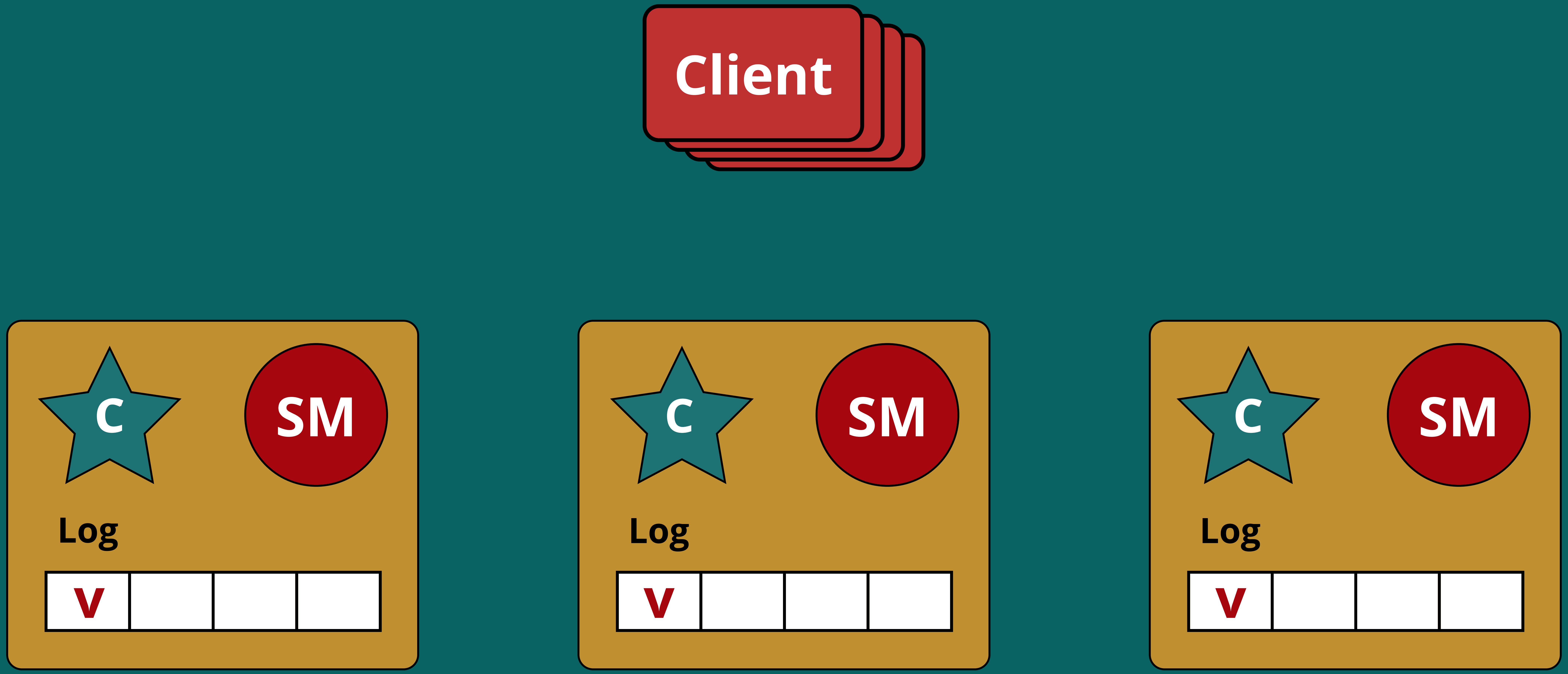


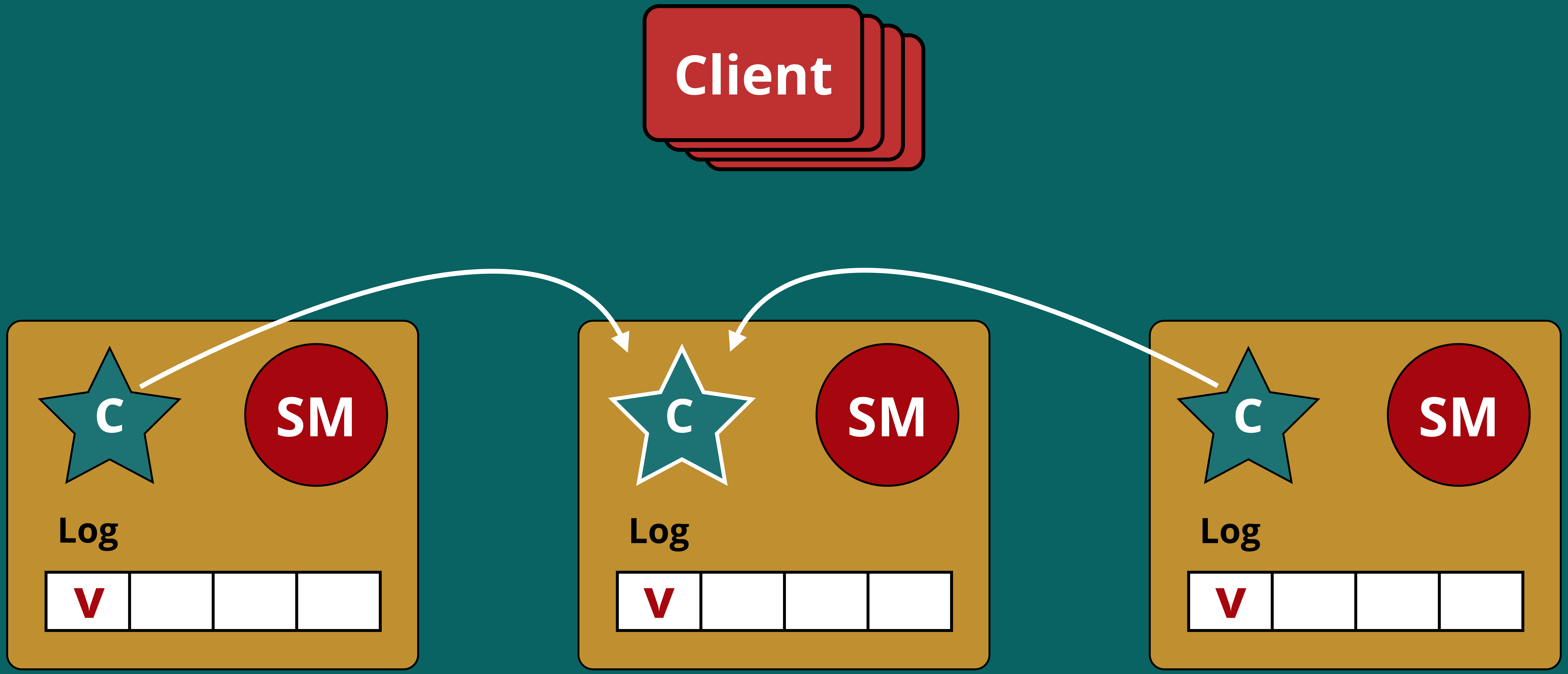
4. leader replicates command to other machines

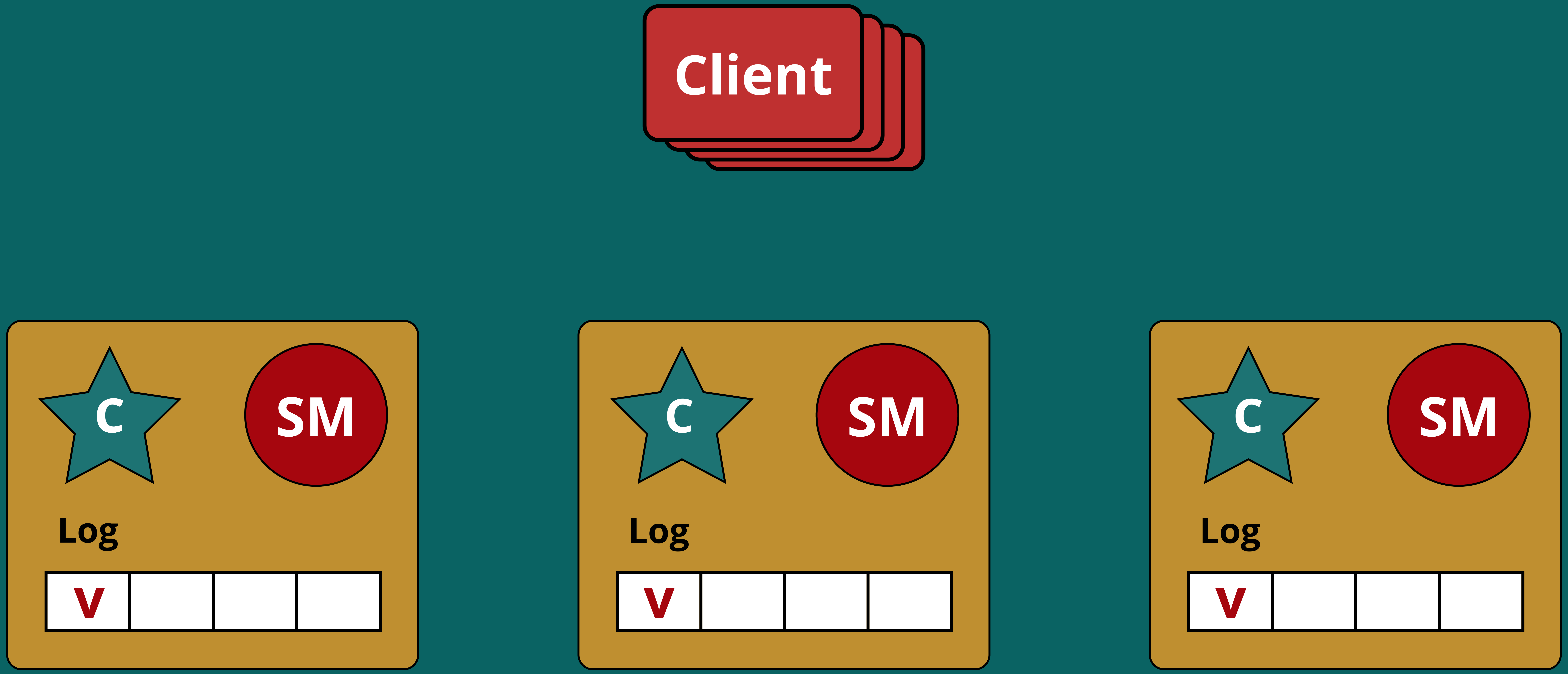

Client
C
SM
Log
C
SM
Log
V
C
SM
Log

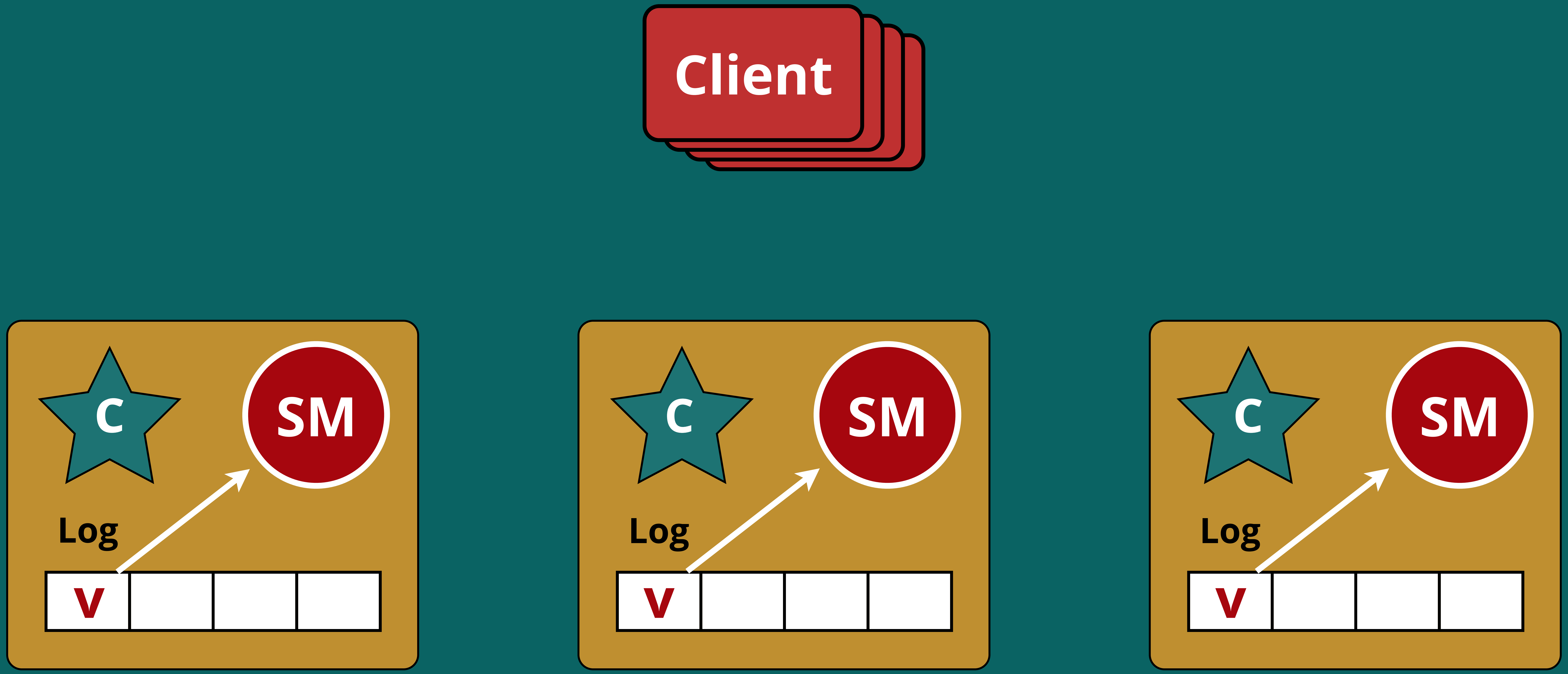


5. command recorded to local machines' log

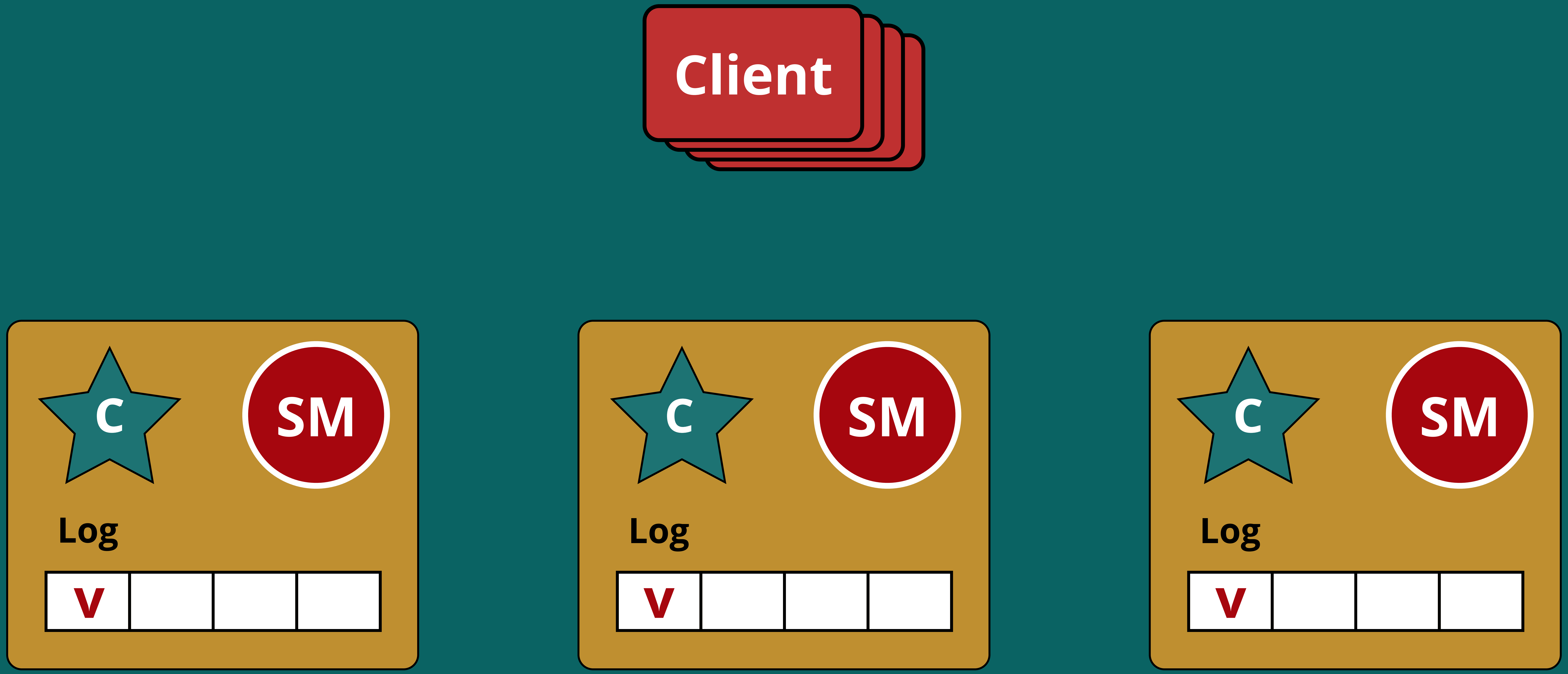

Client
C
SM
Log
V
C
SM
Log
V
C
SM
Log
V


Client
C
SM
Log
V
C
SM
Log
V
C
SM
Log
V

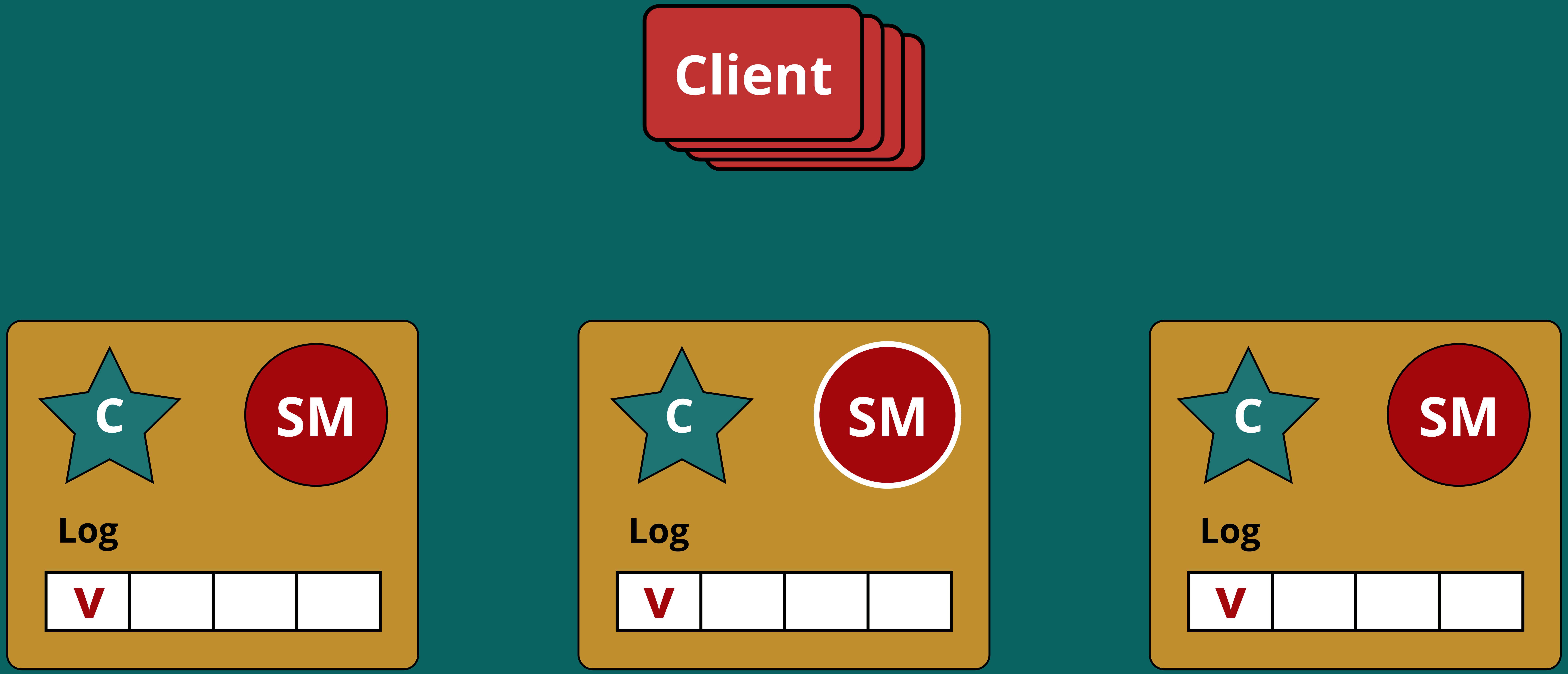

Client
C
SM
Log
V
C
SM
Log
V
C
SM
Log
V

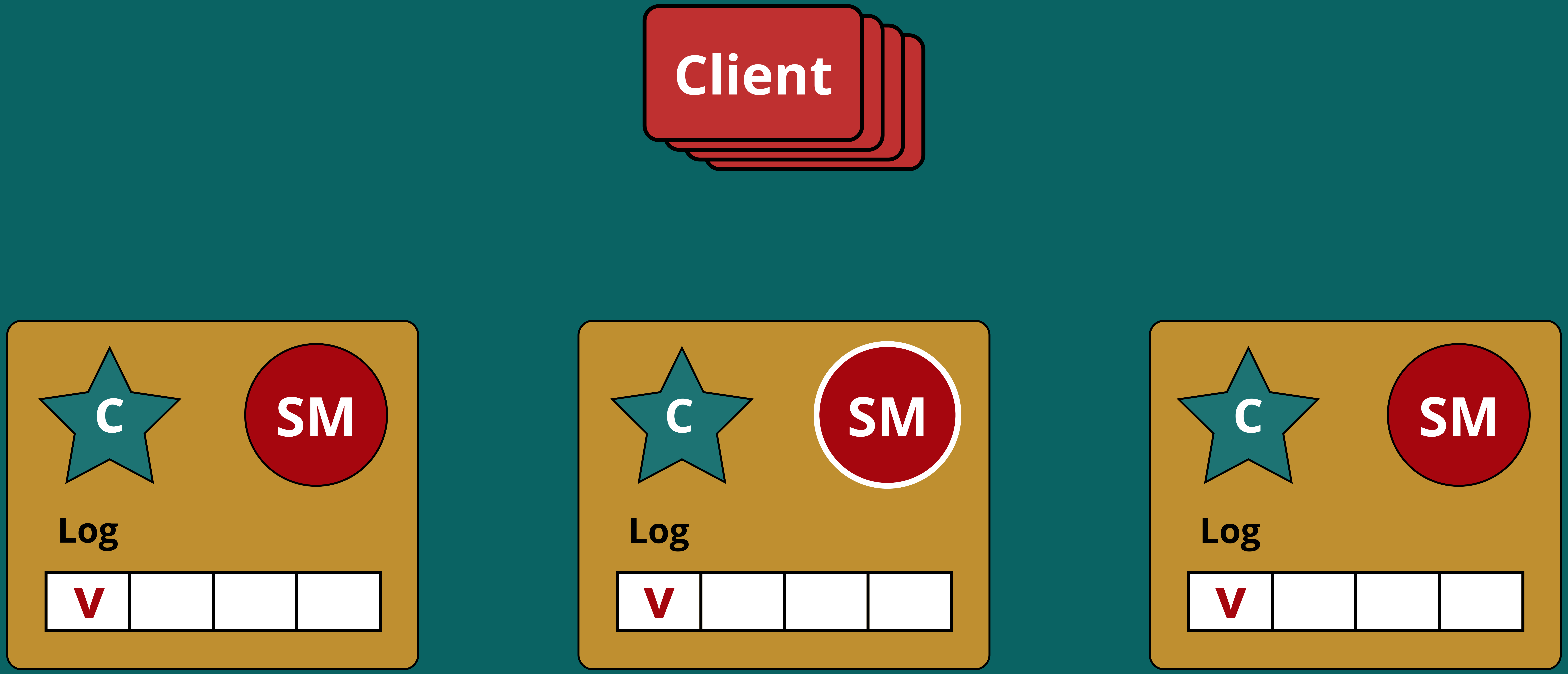


7. command forwarded to state machines for processing

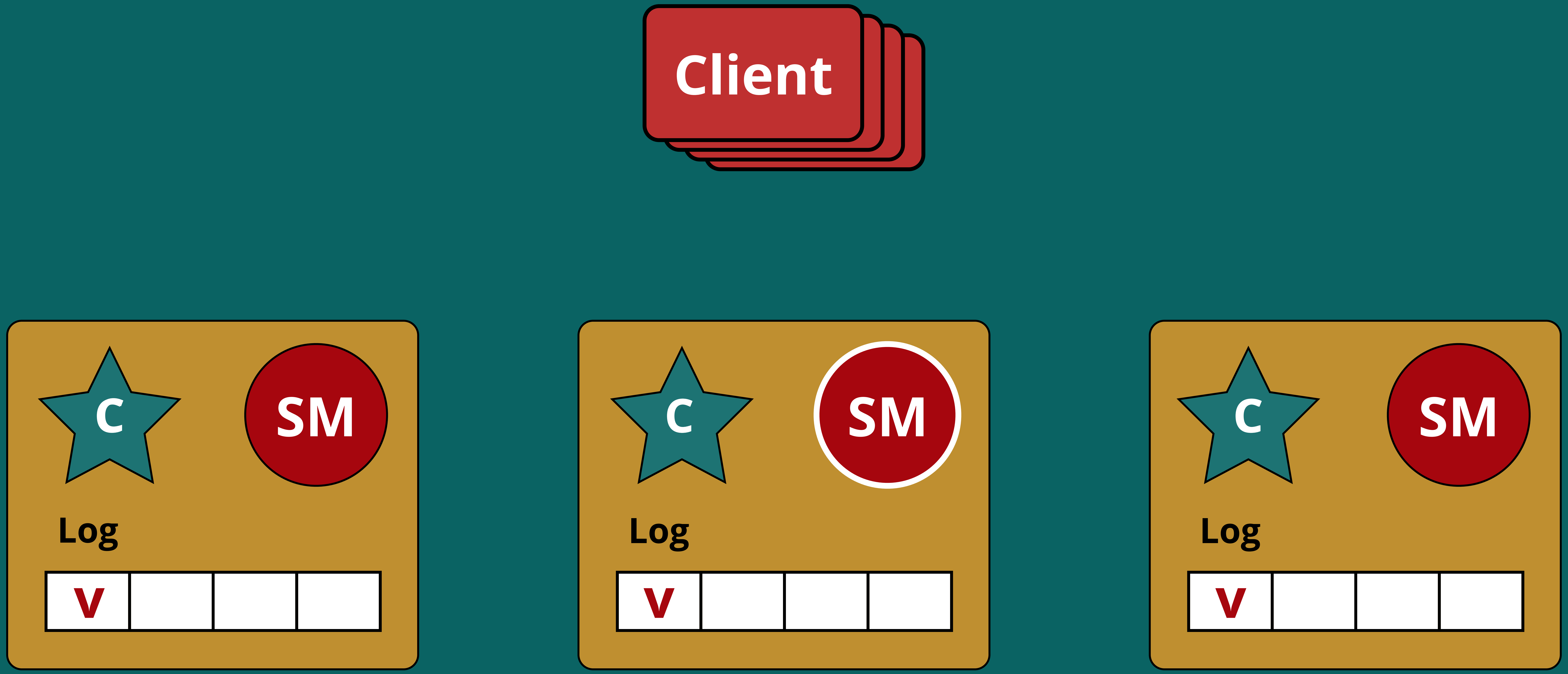


7. command forwarded to state machines for processing


Client
C
SM
Log
V
C
SM
Log
V
C
SM
Log
V



8. SM processes
command, ACKs to client



Client
C
SM
Log
V
C
SM
Log
V
C
SM
Log
V

Why does that work?

job of the consensus module to:

manage replicated logs

determine when it’s safe to pass
to state machine for execution

only requires majority participation

Why does that work?

C job of the consensus module to:

Safety {
- manage replicated logs
- determine when it's safe to pass to state machine for execution

Liveness { only requires majority participation

$2F + 1$

solve for F

$$2F + 1$$

service

F + 1

unavailable

Fail-Stop
Behavior

What If The
Leader
DIES?

Leader Election!

1. Select 1/N servers to act as Leader
2. Leader ensures Safety and Linearizability
3. Detect crashes + Elect new Leader
4. Maintain consistency after Leadership “coups”
5. Depose old Leaders if they return
6. Manage cluster topology

Possible Server Roles:

Leader

Follower

Candidate

Possible Server Roles:

Leader

Follower

Candidate

At most only 1 valid Leader at a time

Receives commands from clients

Commits entries

Sends heartbeats

Possible Server Roles:

Replicate state changes

Passive member of cluster during normal operation

Vote for Candidates

Leader

Follower

Candidate

Possible Server Roles:

Initiate and coordinate Leader Election

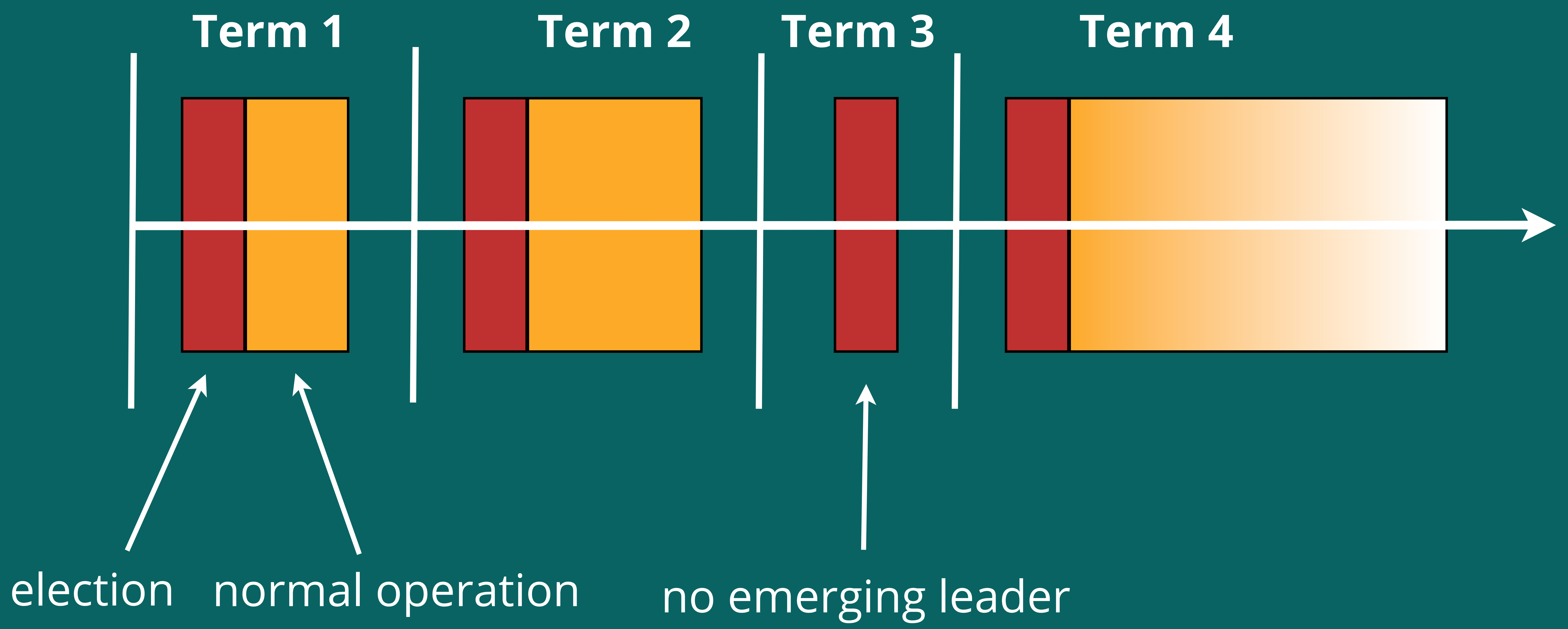
Was previously a Follower

Leader

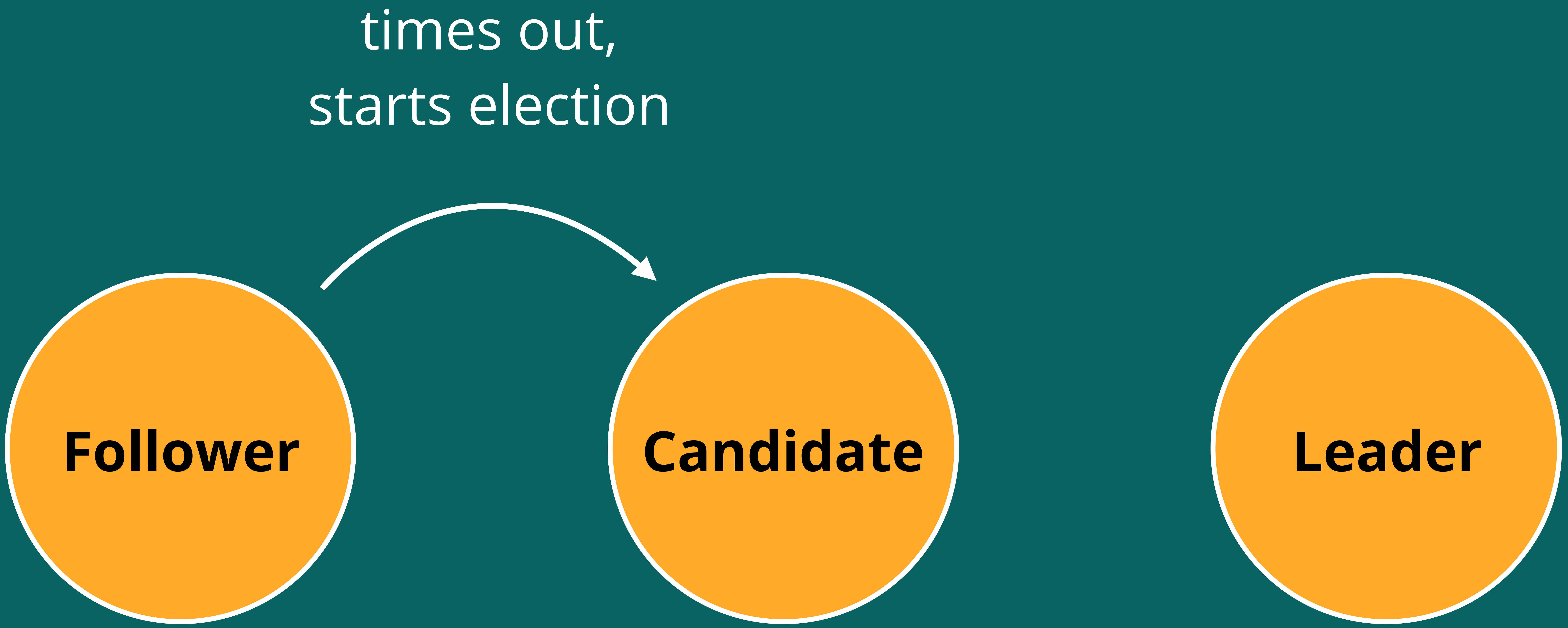
Follower

Candidate

Terms:


Term 1
Term 2
Term 3
Term 4
election
normal operation
no emerging leader

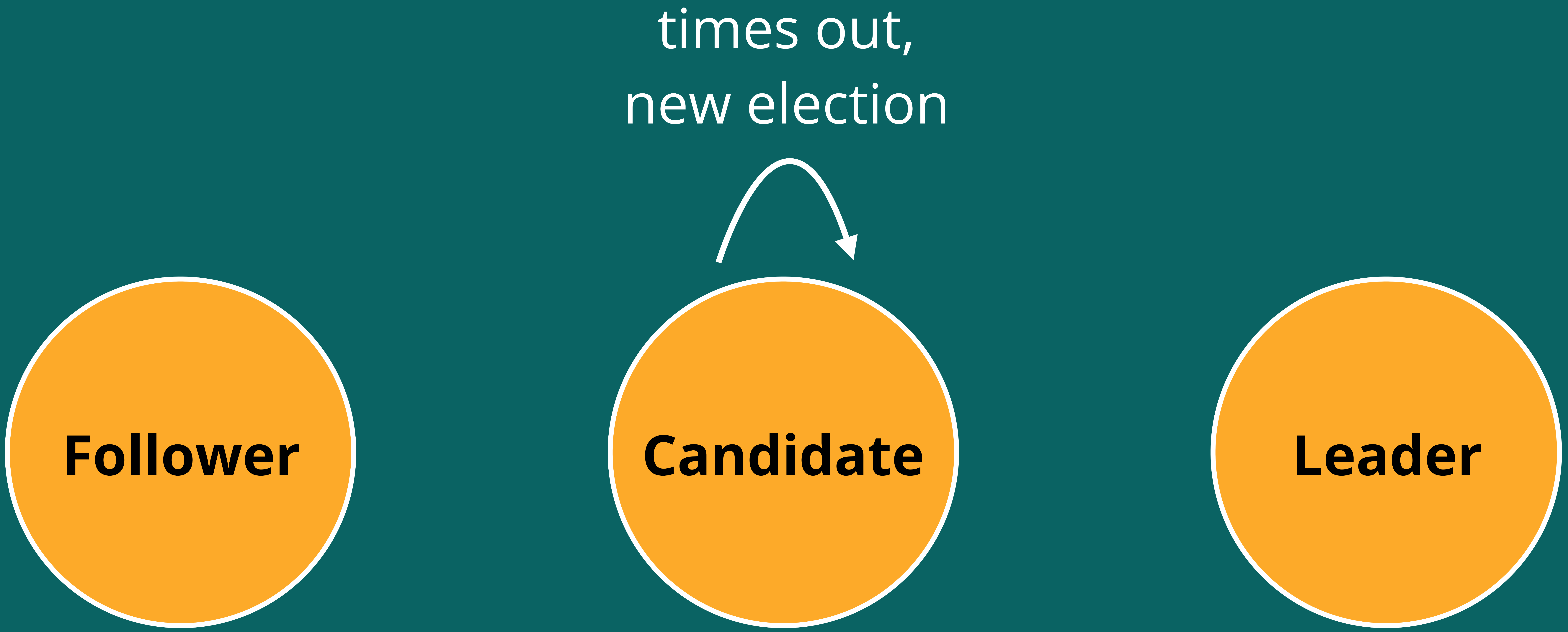

Follower
Candidate
Leader


times out,
starts election
Follower
Candidate
Leader

Follower

Candidate

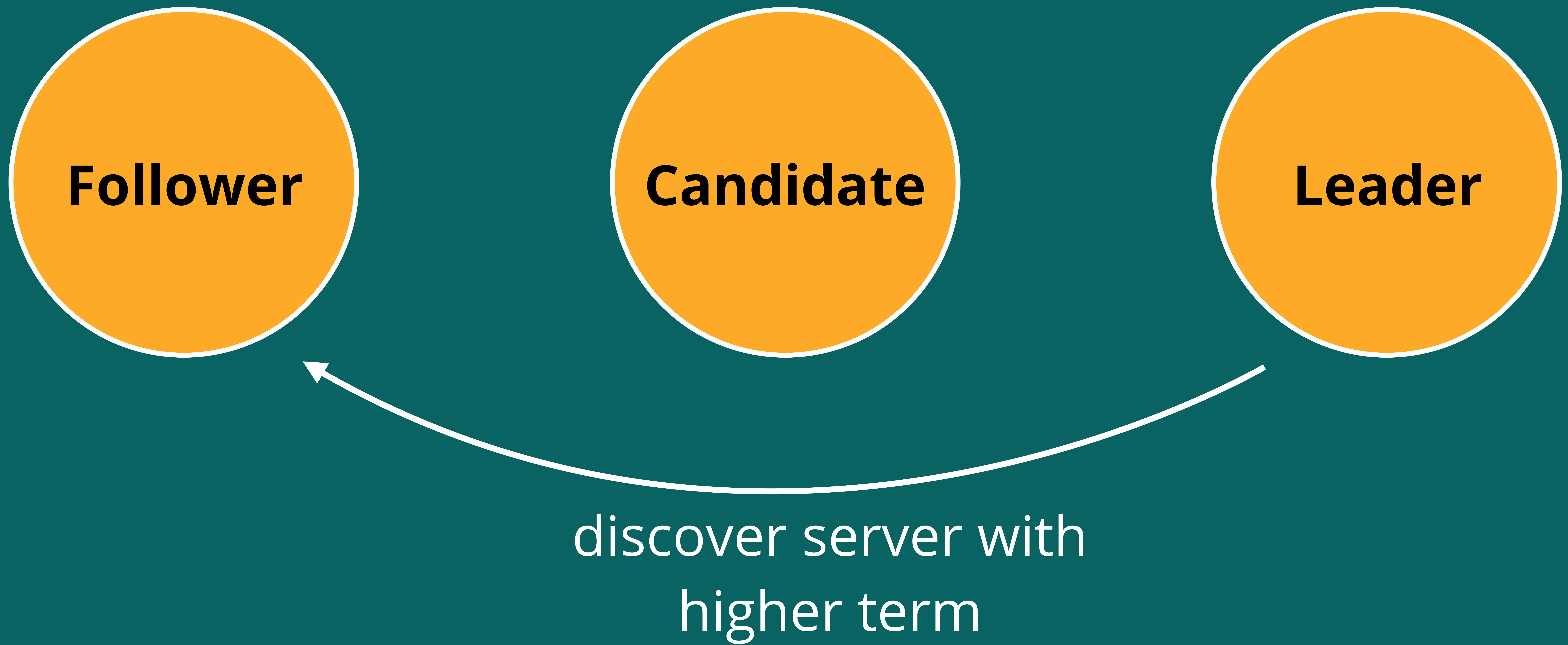
Leader


times out,
new election
Follower
Candidate
Leader

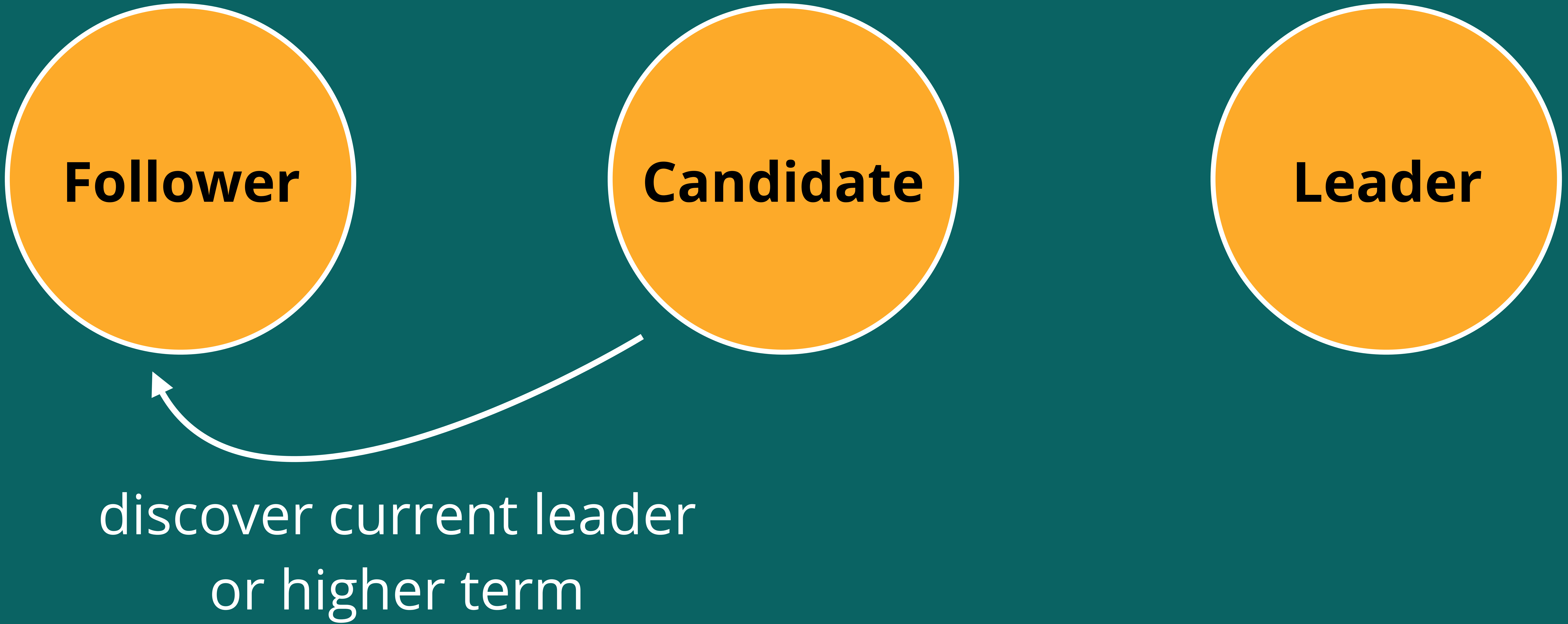

Follower
Candidate
Leader


receives votes from majority of servers
Follower
Candidate
Leader


Follower
Candidate
Leader


Follower
Candidate
Leader
discover server with
higher term


Follower
Candidate
Leader


Follower
Candidate
Leader
discover current leader
or higher term


Follower
Candidate
Leader

Potential Use Cases:

- Distributed Lock Manager
- Configuration Management
- Database Transactions
- Automated Failover
- Service Discovery
- etc...

http://coreos.com/blog/distributed-configuration-with-etcd/index.html

Rafter

github.com/andrewjstone/rafter

What:

- A labor of love, a work in progress
- A library for building strongly consistent distributed systems in Erlang
- Implements the raft consensus protocol in Erlang
- Fundamental abstraction is the replicated log

Replicated Log

- API operates on log **entries**
- Log entries contain **commands**
- Commands are transparent to Rafter
- Systems build on top of rafter with pluggable state machines that process commands upon log entry commit.

Erlang

Erlang: A Concurrent Language

- Processes are the fundamental abstraction
- Processes can only communicate by sending each other messages
- Processes do not share state
- Processes are managed by supervisor processes in a hierarchy

Erlang: A Concurrent Language

```
 loop() ->
      receive
          {From, Msg} ->
               From ! Msg,
               loop()
      end.

%% Spawn 100,000 echo servers
Pids = [spawn(fun loop/0) || _ <-
lists:seq(1,100000)]

%% Send a message to the first process
lists:nth(0, Pids) ! {self(), ayo}.
```

Erlang: A Functional Language

- Single Assignment Variables
- Tail-Recursion
- Pattern Matching

```
{op, {set, Key,  Val}} = {op, {set, <<“job”>>, <<“developer”>>}}
```

- Bit Syntax

```
Header = <<Sha1:20/binary, Type:8, Term:64, Index:64, DataSize:32>>
```

Erlang: A Distributed Language

Location Transparency: Processes can send messages to other processes without having to know if the other process is local.

```
%% Send to a local gen_server process
gen_server:cast(peer1, do_something)

%% Send to a gen_server on another machine
gen_server:cast({‘peer1@rafter1.basho.com’}, do_something)

%% wrapped in a function with a variable name for a clean client API
do_something(Name) -> gen_server:cast(Name, do_something).

%% Using the API
Result = do_something(peer1).
```

Erlang: A Reliable Language

- Erlang embraces “Fail-Fast”
  - Code for the good case. Fail otherwise.
  - Supervisors relaunch failed processes
  - Links and Monitors alert other processes of failure
  - Avoids coding most error paths and helps prevent logic errors from propagating

OTP

- OTP is a set of modules and standards that simplifies the building of reliable, well engineered erlang applications.
- The **gen_server**, **gen_fsm** and **gen_event** modules are the most important parts of OTP
  - They wrap processes as server “behaviors” in order to facilitate building common, standardized distributed applications that integrate well with the Erlang Runtime

Implementation

github.com/andrewjstone/rafter

Peers

- Each peer is made up of two supervised processes
  - A **gen_fsm** that implements the raft consensus fsm
  - A **gen_server** that wraps the persistent log
- An API module hides the implementation

Rafter API

- The entire user api lives in rafter.erl
- rafter:start_node(peer1, kv_sm).
- rafter:set_config(peer1, [peer1, peer2, peer3, peer4, peer5]).
- rafter:op(peer1, {set, <<“Omar”>>, <<“gonna get got”>>}).
- rafter:op(peer1, {get, <<“Omar”>>}).

Output State Machines

- Commands are applied in order to each peer's state machine as their entries are committed
- All peers in a consensus group can only run one type of state machine passed in during **start_node/2**
- Each State machine must export **apply/1**

Hypothetical KV store

```
%% API
kv_sm:set(Key, Val) ->
    Peer = get_local_peer(),
    rafter:op(Peer, {set, Key, Value}).

%% State Machine callback
kv_sm:apply({set, Key, Value}) -> ets:insert({kv_sm_store,
{Key, Value});

kv_sm:apply({get, Key}) -> ets:lookup(kv_sm_store, Key).
```

rafter_consensus_fsm

- **gen_fsm** that implements Raft
- 3 states - follower, candidate, leader
- Messages sent and received between fsm's according to raft protocol
- State handling functions pattern match on messages to simplify and shorten handler clauses.

rafter_log.erl

- Log API used by **rafter_consensus_fsm** and **rafter_config**
- Utilizes Binary pattern matching for reading logs
- Writes out entries to append only log.
- State machine commands encoded with **term_to_binary/1**

rafter_config.erl

- Rafter handles dynamic reconfiguration of it’s clusters at runtime
- Depending upon the configuration of the cluster, different code paths need navigating, such as whether a majority of votes has been received.
- Instead of embedding this logic in the consensus fsm, it was abstracted out into a module of pure functions

rafter_config.erl API

```
-spec quorum_min(peer(),  #config{}, dict()) -> non_neg_integer().

-spec has_vote(peer(), #config{}) -> boolean().

-spec allow_config(#config{}, list(peer())) -> boolean().

-spec voters(#config{}) -> list(peer()).
```

Testing

Property Based Testing

- Use Erlang QuickCheck
- Too complex to get into now
- Come hear me talk about it at Erlang Factory Lite in Berlin!

shameless plug

Other Raft Implementations

https://ramcloud.stanford.edu/wiki/display/logcabin/LogCabin

https://github.com/coreos/etcd

https://github.com/benbjohnson/go-raft

http://coreos.com/blog/distributed-configuration-with-etcd/index.html

github.com/andrewjstone/rafter

(a few more)

Shameless Plugs

RICON West

http://ricon.io/west.html

Études for Erlang

http://meetup.com/Erlang-NYC

Thanks File

Andy Gross - Introducing us to Raft

Diego Ongaro - writing Raft, clarifying Tom's understanding, reviewing slides

Chris Meiklejohn - http://thinkdistributed.io - being an inspiration

Justin Sheehy - reviewing slides, correcting poor assumptions

Reid Draper - helping rubber duck solutions

Kelly McLaughlin - helping rubber duck solutions

John Daily - for his consistent pedantry concerning Tom's abuse of English

Basho - letting us indulge our intellect on the company's dime (we're hiring)

Any and all questions can be sent to /dev/null

@tsantero @andrew_j_stone