Northeastern University - Seattle

CS6650 Building Scalable Distributed Systems

Professor Ian Gorton

Building Scalable Distributed Systems

Week 5 – Strong Consistency and Distributed Databases

Learning Objectives

- Explain how 2 phase commit works and how it handles failures
- Describe the Raft consensus algorithm
- Explain the implications of the FLP result
- Explain causal consistency in Neo4j

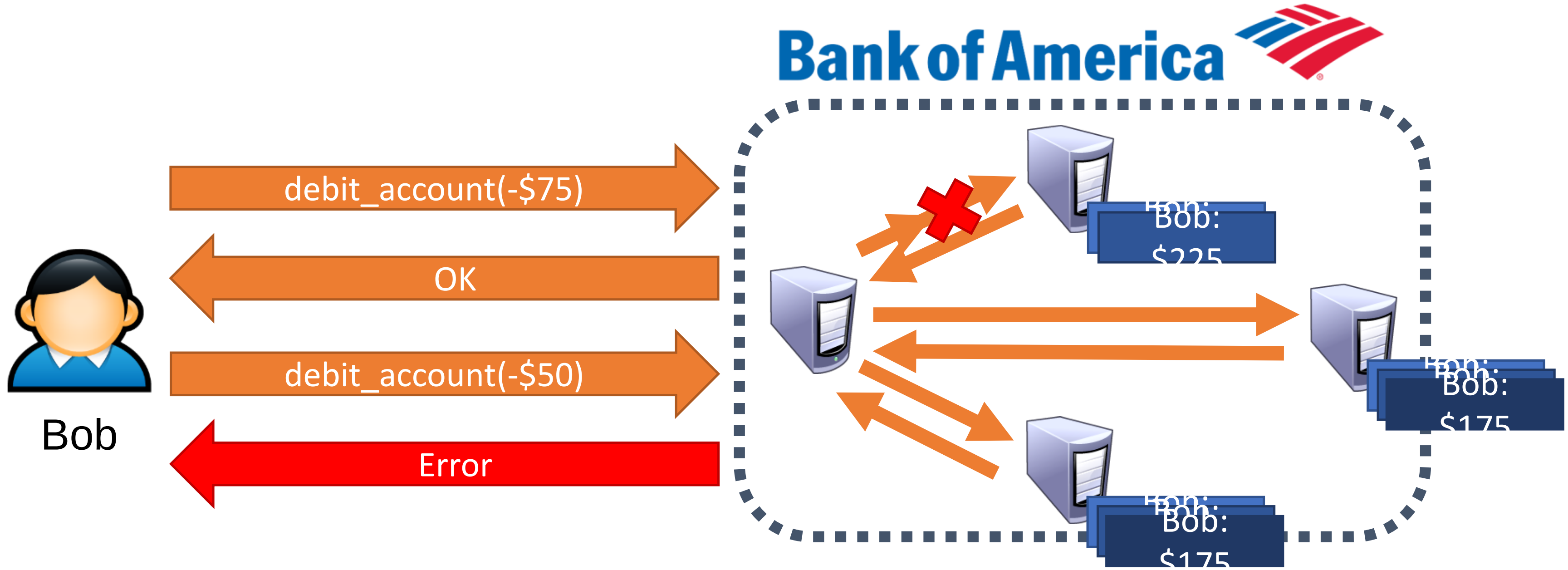
Remember Last Week?

- Leaderless data systems and replica consistency
- Mechanisms to handle conflicts
  - LWW – not great unless immutable objects used
  - Quorums
  - Conflict resolution with versions
    - Vector clocks/version vectors

Outline

- Strong consistency and transactions
- The Raft protocol
- Causal consistency in Neo4j

Strong Consistency*



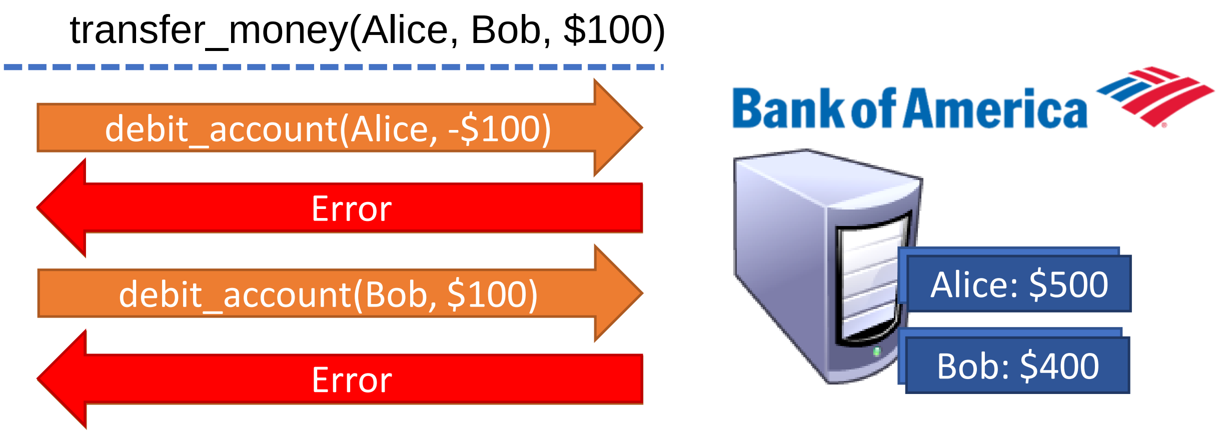
- One approach to building distributed systems is to force them to be consistent
  - Guarantee that all replicas receive an update…
  - …Or none of them do
- If consistency is guaranteed, then reaching consensus is trivial

**Strong consistency based on materials from Christo Wilson/David Choffnes at Northeastern*

Distributed Commit Problem

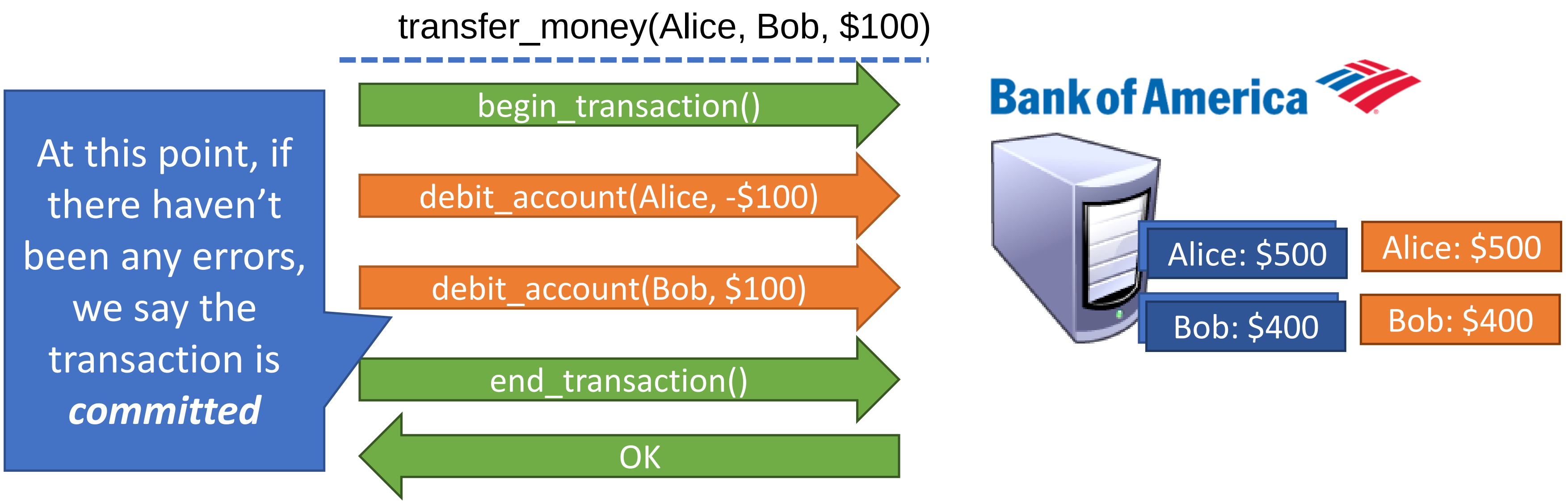
- Application that performs operations on multiple replicas or databases
  - We want to guarantee that all replicas get updated, or none do
- Distributed commit problem:
  - Operation is committed when all participants ***can*** perform the action
  - Once a commit decision is reached, all participants ***must*** perform the action
- Two steps gives rise to the Two Phase Commit protocol

Motivating Transactions



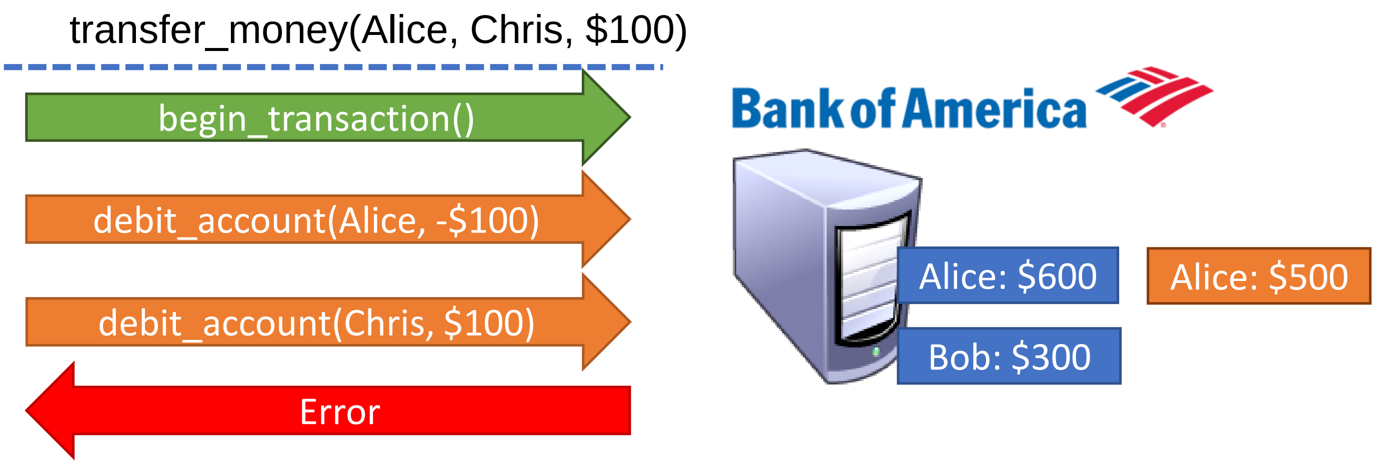
- System becomes inconsistent if any individual action fails

Simple Transactions



- Actions inside a transaction behave as a single action

Simple Transactions



- If any individual action fails, the whole transaction fails
  - Failed transactions have no side effects
- Incomplete results during transactions are hidden

ACID Properties

- Traditional transactional databases support the following:
  - Atomicity: all or none; if transaction fails then no changes are applied to the database
  - Consistency: there are no violations of database integrity
  - Isolation: partial results from incomplete transactions are hidden
  - Durability: the effects of committed transactions are permanent

Two Phase Commit (2PC)

- Well known techniques used to implement transactions in **centralized** databases
  - E.g. journaling (append-only logs)
  - Take a databases course ;)
- Two Phase Commit (2PC) is a protocol for implementing transactions in a **distributed** setting
  - Protocol operates in two rounds
  - Assume we have leader/coordinator that manages transactions
  - Each replica promises that it is ready to commit
  - Leader decides the outcome and instructs replicas to commit or abort

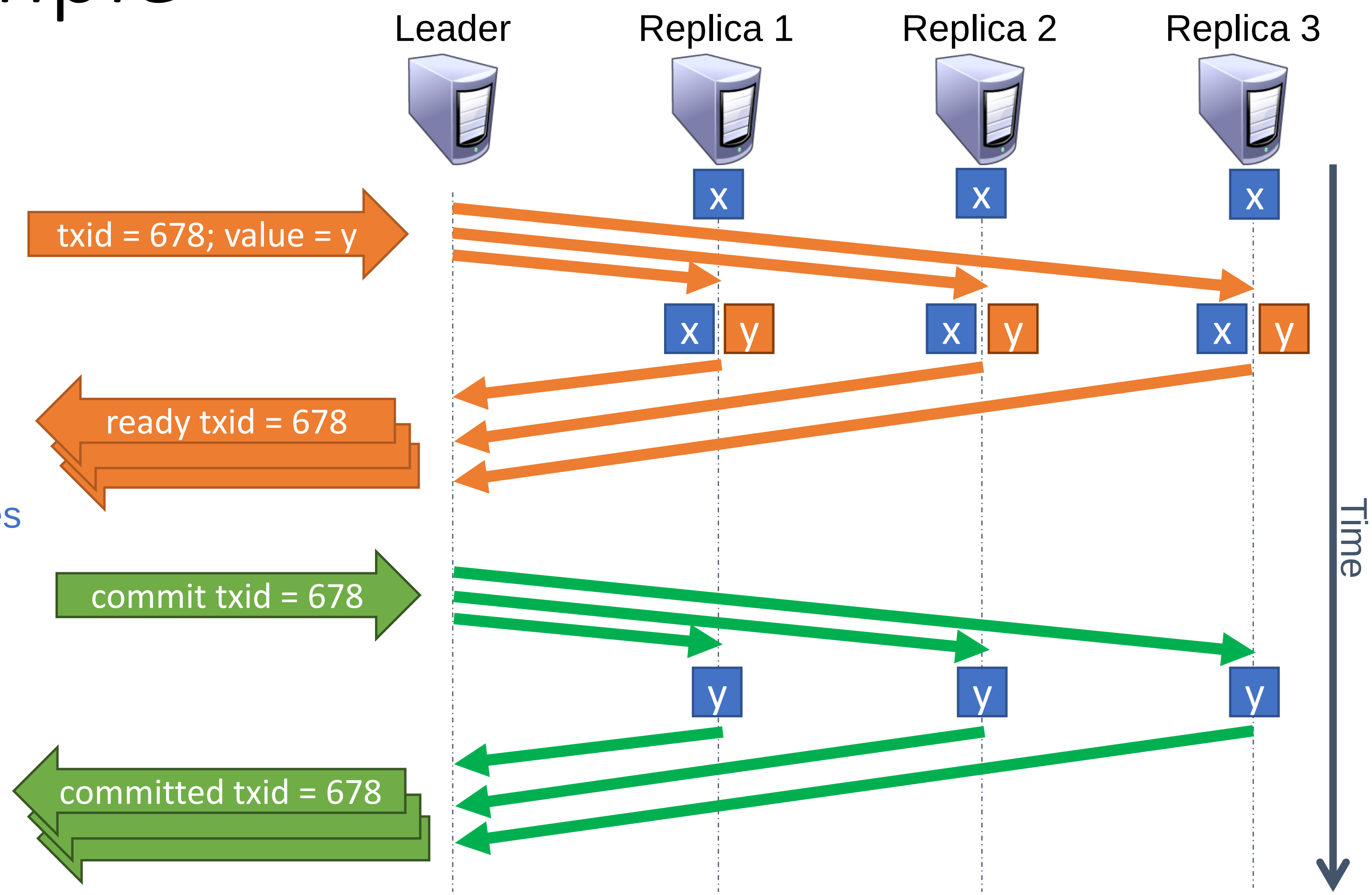
2PC Example

- Begin by distributing the update
- Txid is a logical clock

- Wait to receive "ready to commit" from all replicas
- Also called promises

- Tell replicas to commit

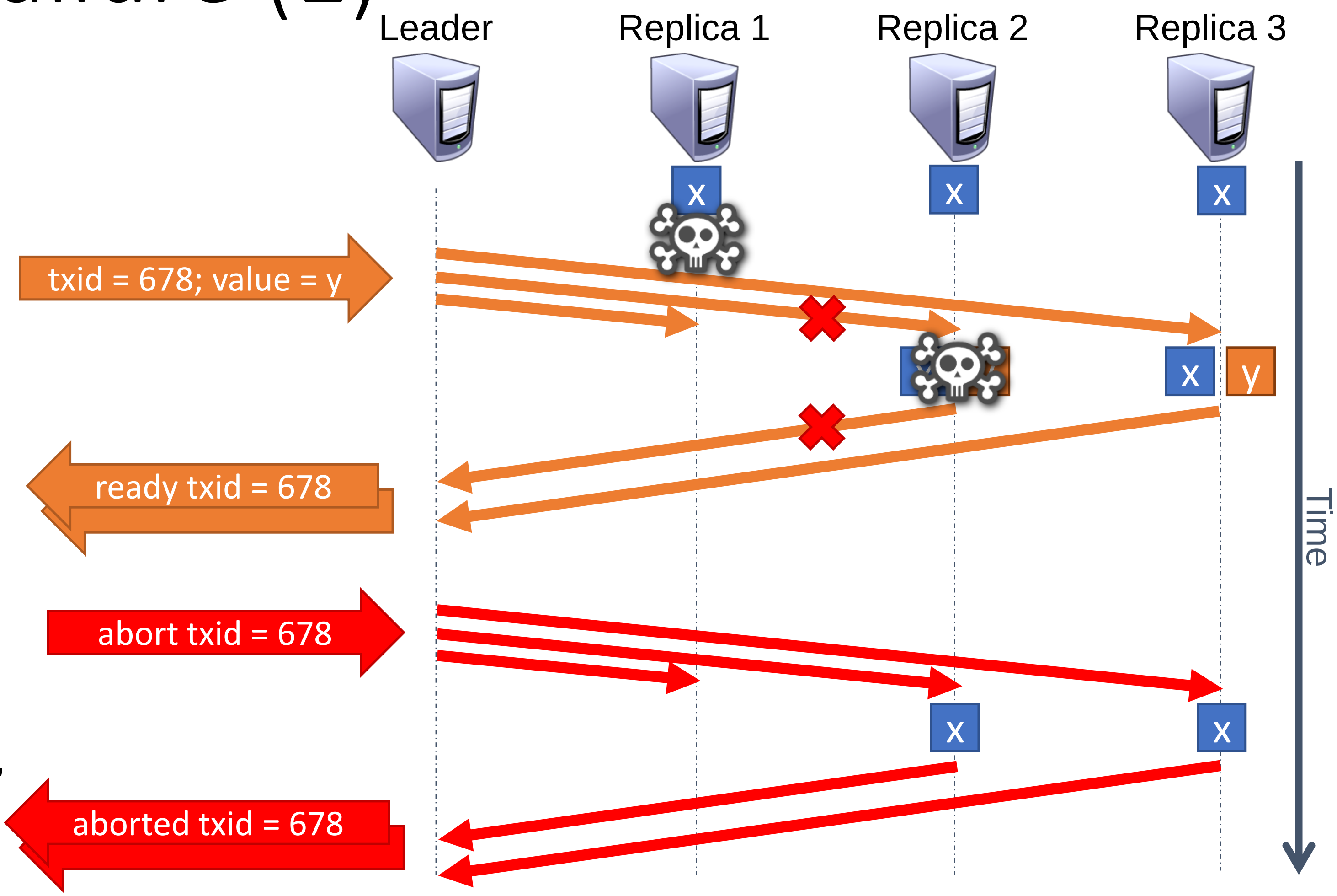
- At this point, all replicas are guaranteed to be up-to-date



Failure Modes

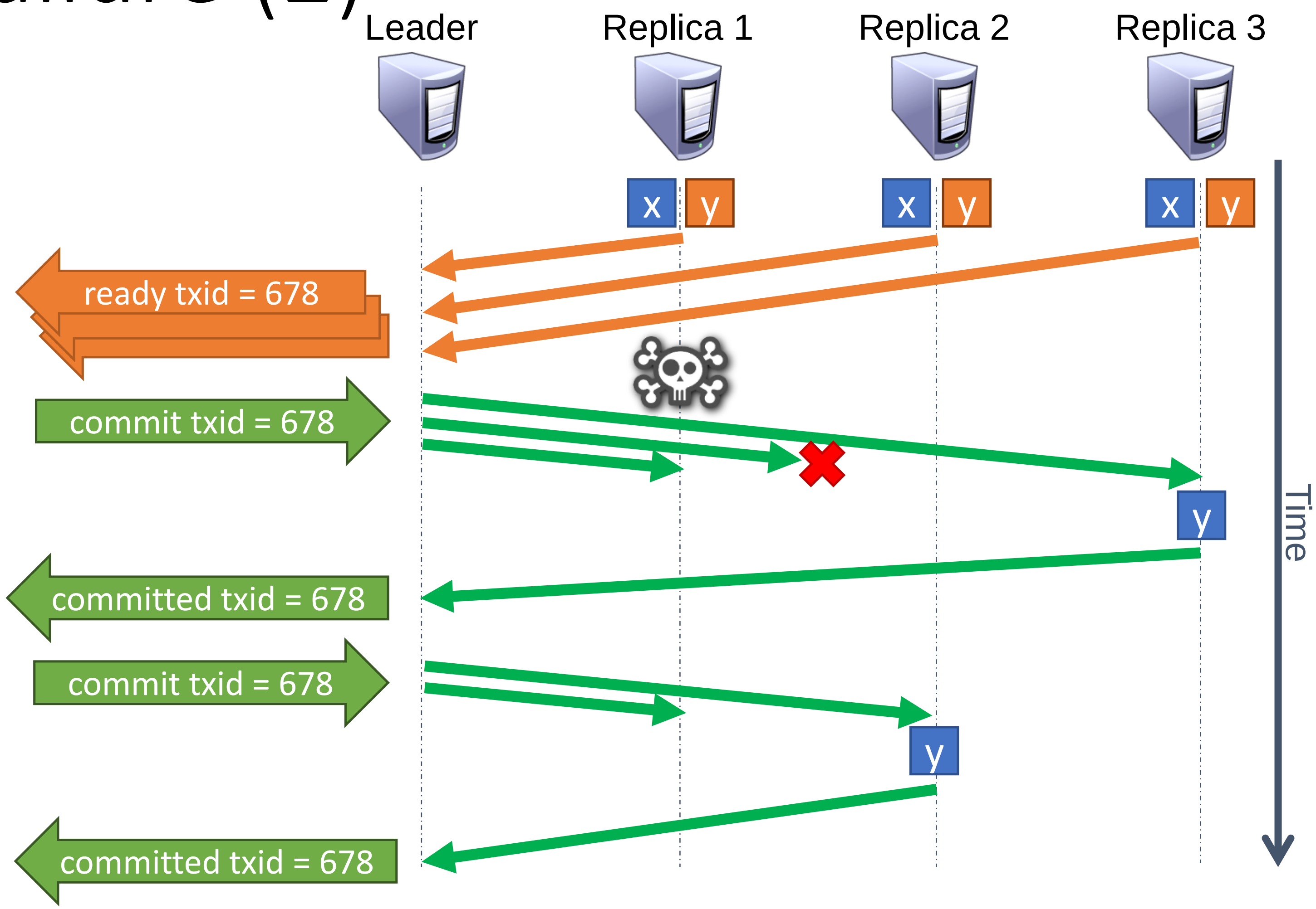
- Replica Failure
  - Before or during the initial promise phase
  - Before or during the commit
- Leader Failure
  - Before receiving all promises
  - Before or during sending commits
  - Before receiving all committed messages

Replica Failure (1)



- Error: not all replicas are "ready"
- The same thing happens if a write or a "ready" is dropped, a replica times out, or a replica returns an error

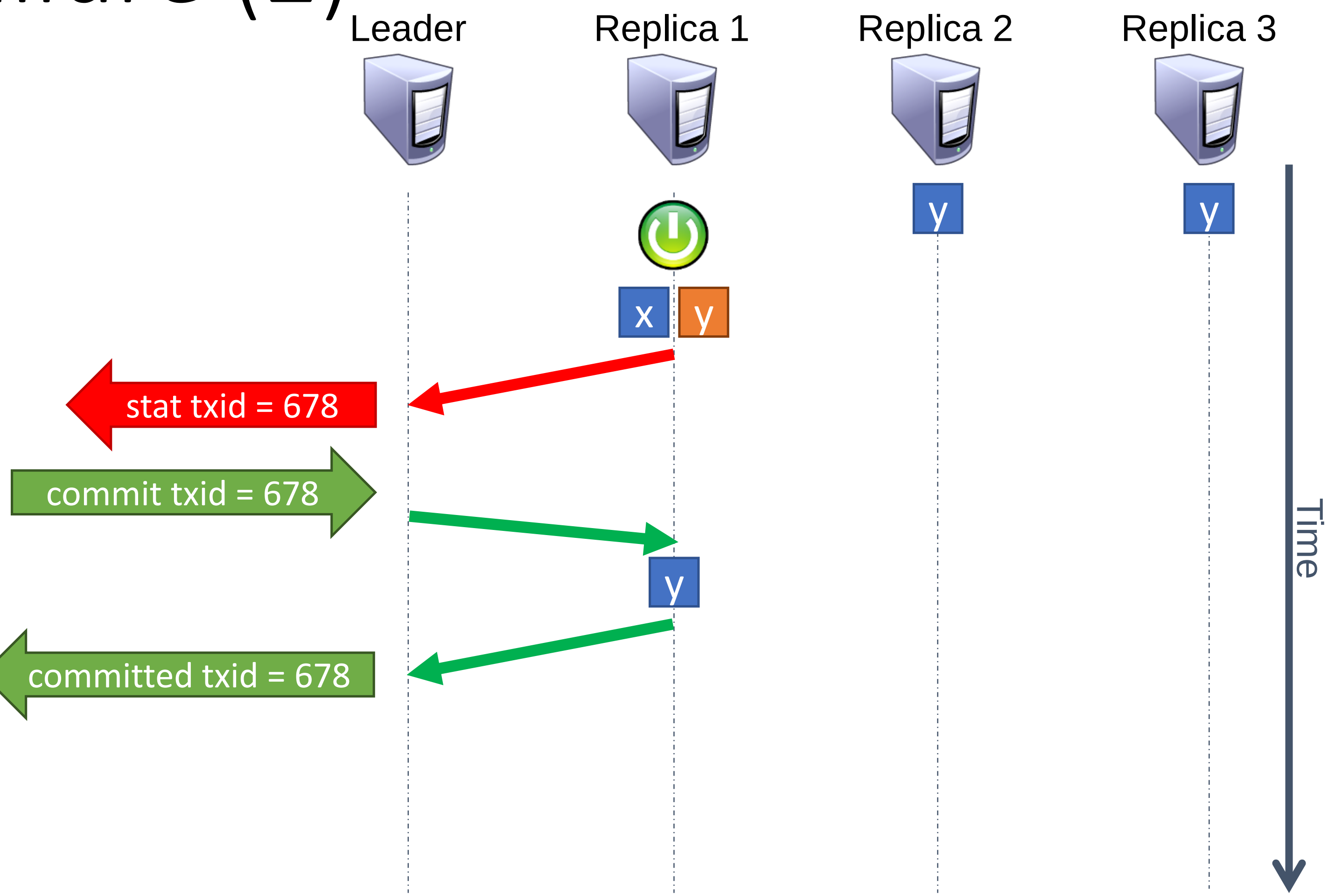
Replica Failure (2)



- Known inconsistent state
- Leader must keep retrying until all commits succeed

Replica Failure (2)

- Replicas attempt to resume unfinished transactions when they reboot
- Finally, the system is consistent and may proceed



Leader Failure

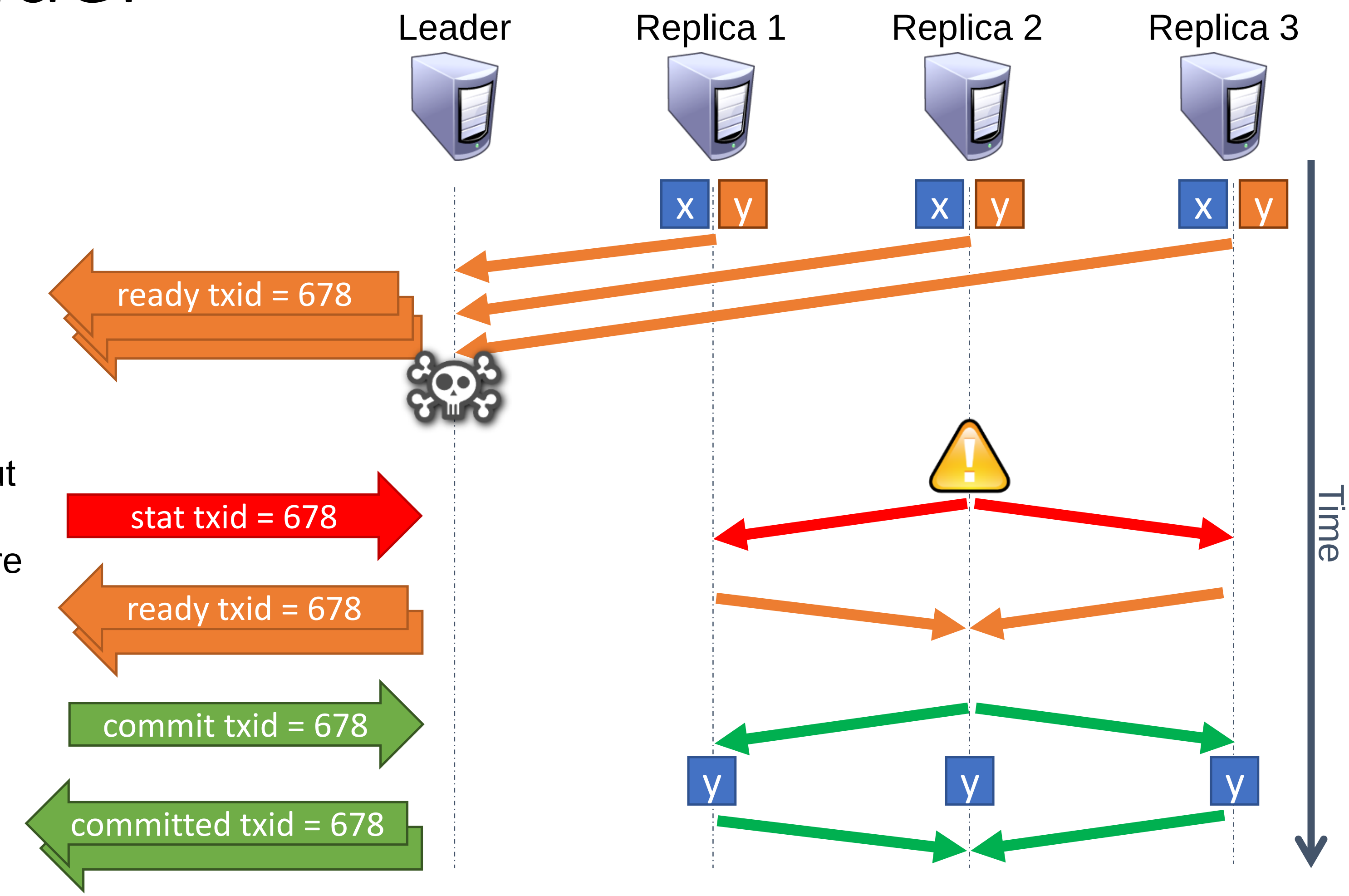
- What happens if the leader crashes?
  - Leader must constantly write its state to permanent storage
  - It must pick up where it left off once it reboots
- If there are unconfirmed transactions
  - Send new write messages, wait for “ready to commit” replies
- If there are uncommitted transactions
  - Send new commit messages, wait for “committed” replies
- Replicas may see duplicate messages during this process
  - Thus, it’s important that every transaction have a unique txid

Leader Failure

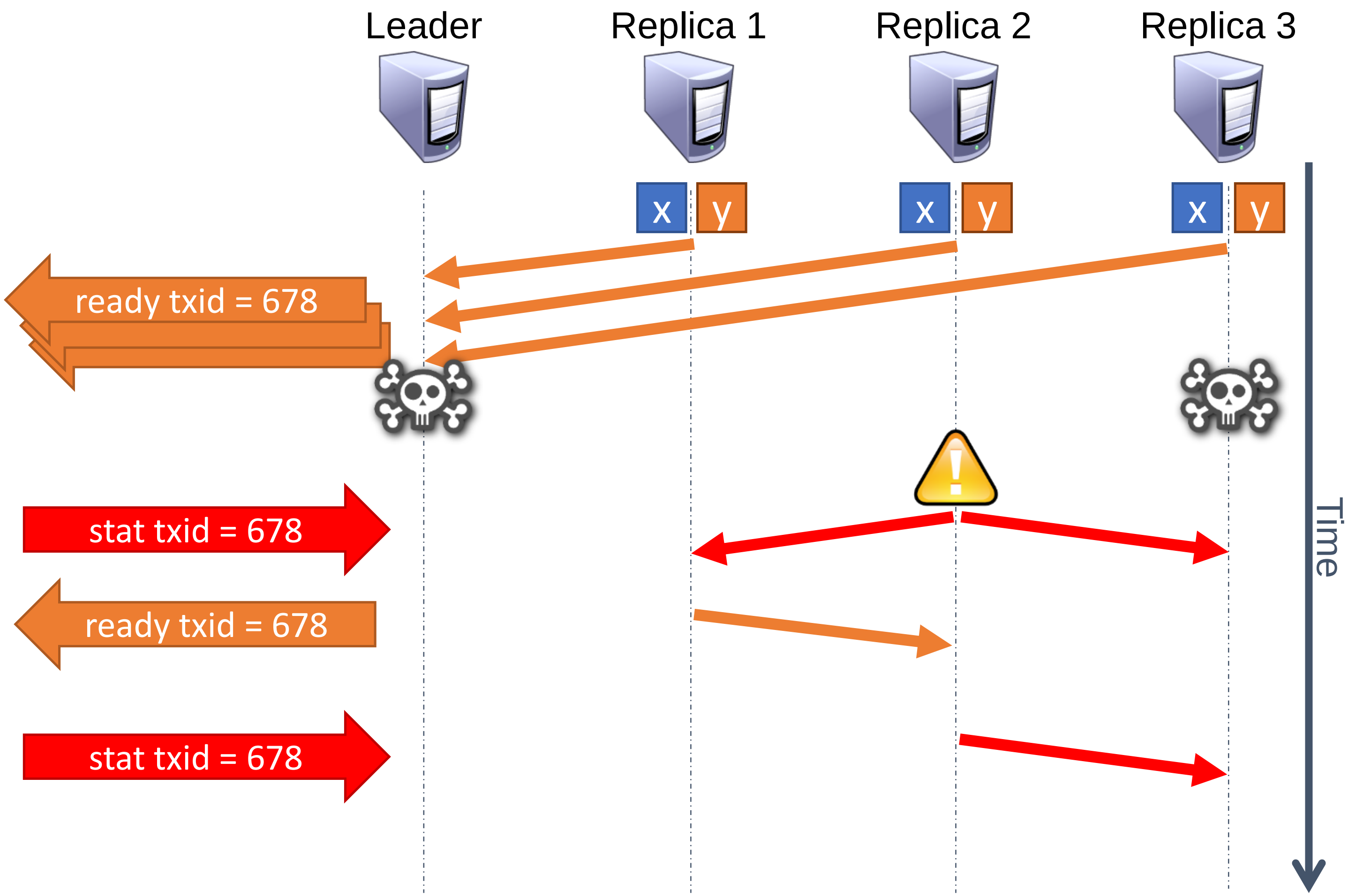
- Key problem: what if the leader crashes and never recovers?
  - By default, replicas block until contacted by the leader
  - Can the system make progress?
- Yes, under limited circumstances
  - After sending a “ready to commit” message, each replica starts a timer
  - The first replica whose timer expires elects itself as the new leader
  - Query the other replicas for their status
  - Send “commits” to all replicas if they are all “ready”
- However, **this only works if all the replicas are alive and reachable**
  - If a replica crashes or is unreachable, deadlock is unavoidable

New Leader

- Replica 2's timeout expires, begins recovery procedure
- System is consistent again



Deadlock



- Replica 2's timeout expires, begins recovery procedure
- Cannot proceed, but cannot abort

Garbage Collection

- Replicas must retain records of past transactions, in case the leader fails
  - Example, suppose the leader crashes, reboots, and attempts to commit a transaction that has already been committed
  - Replicas must remember that this past transaction was already committed, since committing a second time may lead to inconsistencies
- In practice, leader periodically tells replicas to garbage collect
  - All transactions <= some txid may be deleted

2PC Summary

- Message complexity: O(2n)
- The good: guarantees consistency
- The bad:
  - Write performance suffers if there are failures during the commit phase
  - Does not scale easily
  - A pure 2PC system blocks all writes if the leader fails
  - Smarter 2PC systems still blocks all writes if the leader + 1 replica fail
- 2PC sacrifices **availability** in favor of **consistency**

Can 2PC be Fixed?

- Problem with 2PC is need for centralized leader
  - Only leader knows if a transaction is 100% ready to commit or not
  - Thus, if the leader + 1 replica fail, recovery is impossible
- Potential solution: Three Phase Commit
  - Add an additional round of communication informing all replicas to prepare to commit, before committing
- 3PC is not robust against network partitions
  - i.e. not all servers can contact each other

3PC

- In practice, nobody uses 3PC
  - Additional complexity and performance penalty is too high
  - Loss of consistency during partitions is a deal breaker

Class Exercise

- Let’s implement Lamport’s Clocks from last week’s lecture
- Start with the code here:
  - https://github.com/gortonator/logicalclock
- Understand how it works and clean up outputs so you can see the effect of the clocks

Consensus with Raft

Consensus

- Build a distributed system that meets the following goals:
  - The system should be able to reach consensus
    - *Consensus [n]: general agreement*
  - The system should be consistent
    - Data should be correct; no integrity violations
  - The system should be highly available
    - Data should be accessible even in the face of arbitrary failures
- Challenges:
  - Huge number of failure modes
  - Network partitions are difficult to cope with
  - We haven’t even considered byzantine failures

First: Some Basic Theory

- Assume asynchronous, reliable network
  - Replicas may take an arbitrarily long time to respond to messages
- Assume all faults are *crash faults*
  - *i.e.* if a replica has a problem it must have crashed and never wakes up
  - No byzantine faults

The FLP Result

There is no asynchronous algorithm that achieves consensus on a 1-bit value in the presence of crash faults. The result is true even if no crash actually occurs!

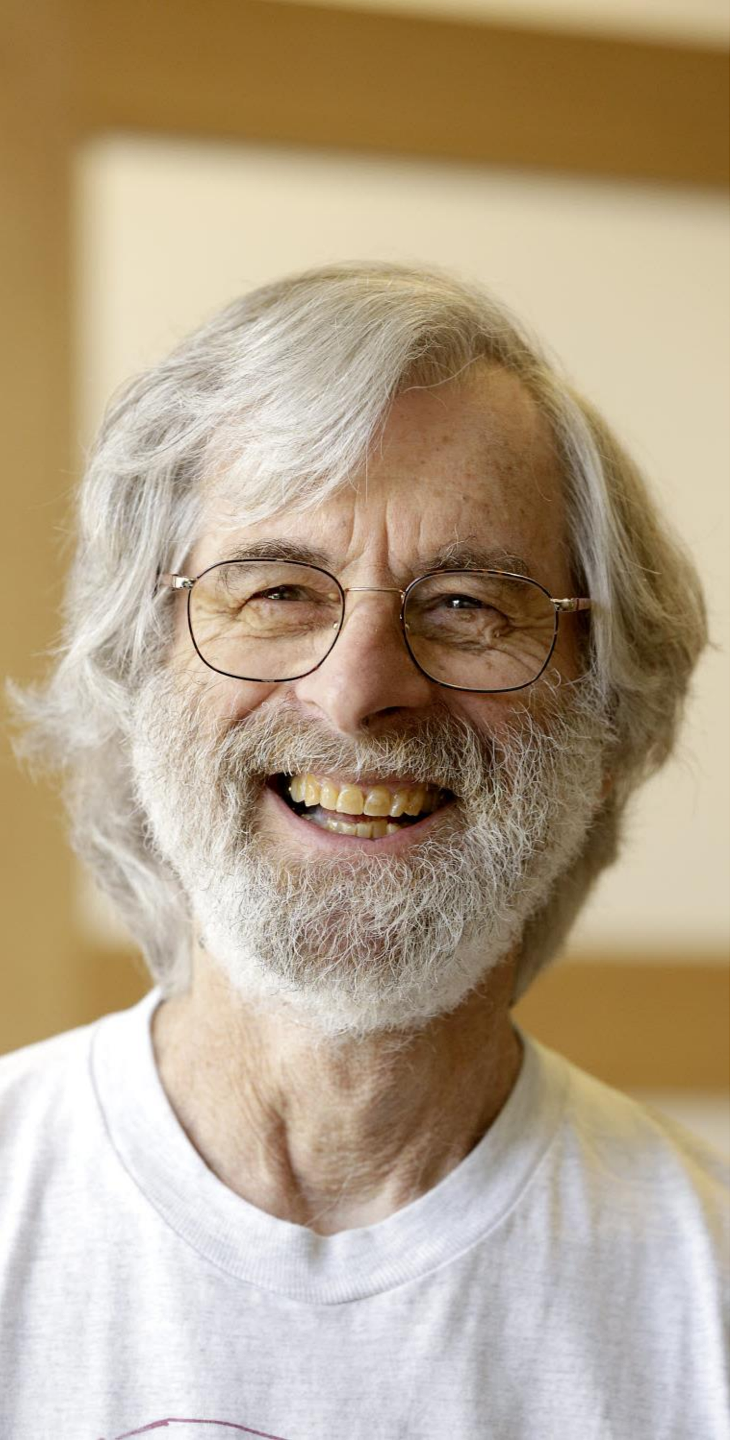
- This is known as the FLP result
  - Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson, 1985
- Extremely powerful theoretical result because:
  - If you can’t agree on 1-bit, generalizing to larger values isn’t going to help you
  - If you can’t guarantee convergence with crash faults, no way you can guarantee convergence with byzantine faults
  - If you can’t guarantee convergence on a reliable network, no way you can on an unreliable network

FLP Proof Sketch

- In an asynchronous system, a replica *x* cannot tell whether a non-responsive replica *y* has crashed or is just slow
- What can *x* do?
  - If *x* waits, it will block indefinitely since it might never receive the message from *y*
  - If *x* decides, it may find out later that *y* made a different decision
- Proof constructs a scenario where each attempt to decide is overruled by a delayed, asynchronous message
  - Thus, the system oscillates between 0 and 1 never converges

Impact of FLP

- FLP proves that any distributed algorithm attempting to reach consensus has runs that never terminate
  - Hence always achieving consensus is impossible
  - Unrealistic model however
- Distributed systems in practice achieve consensus all the time! How?
  - Use randomization, probabilistic guarantees (gossip protocols)
  - Use quorum systems (e.g. Paxos or Raft)
- Essentially trade-off consistency in favor of availability



Paxos

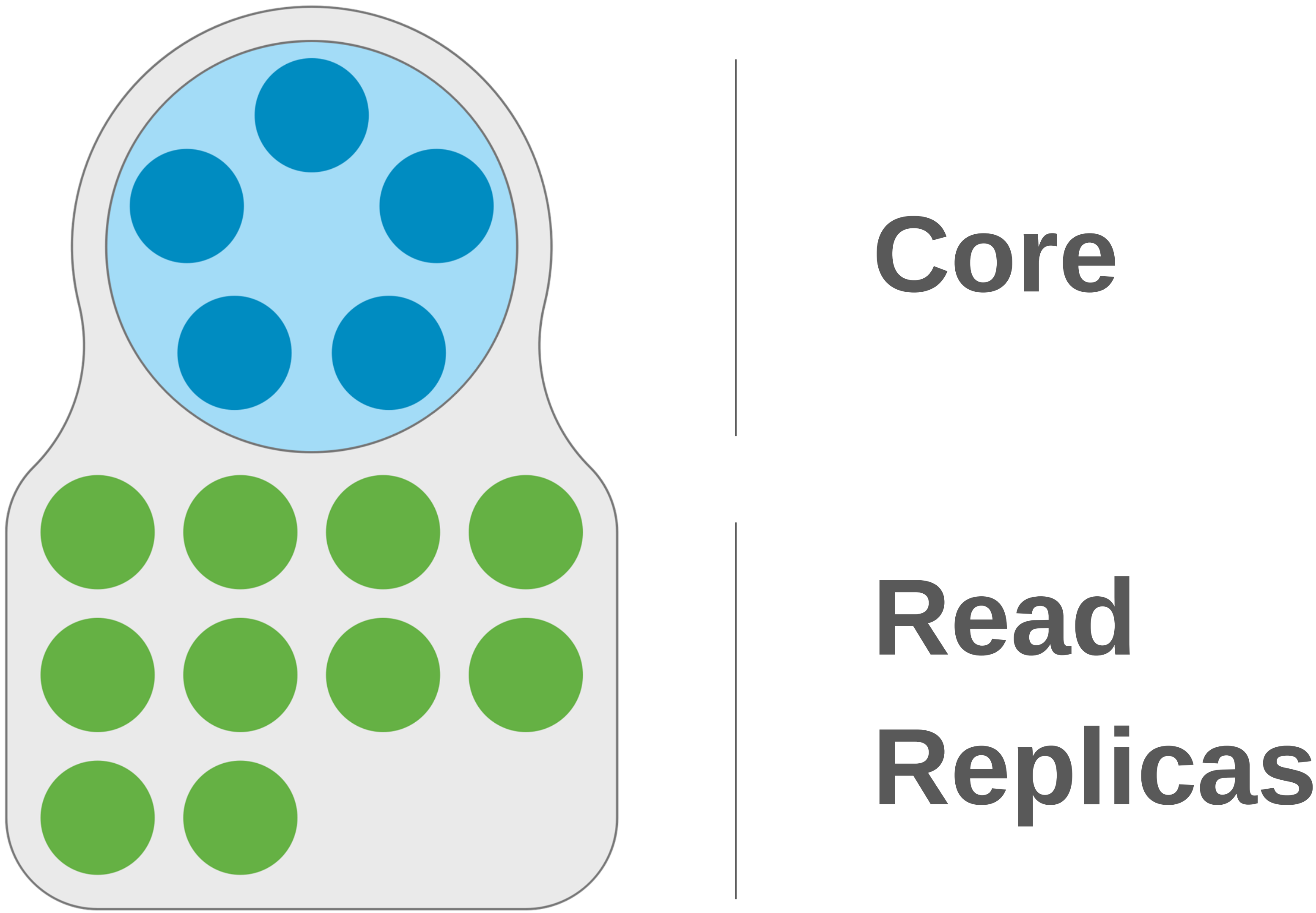
- Developed by Turing award winner Leslie Lamport
  - First published as a tech report in 1989
  - Journal refused to publish it, nobody understood the protocol
- Formally published in 1998
  - Again, nobody understands it
- Leslie Lamport publishes "Paxos Made Simple" in 2001
  - People start to get the protocol
- Reaches widespread fame in 2006-2007
  - Used by Google in their Chubby distributed mutex system - https://static.googleusercontent.com/media/research.google.com/en//archive/paxos_made_live.pdf
  - Implementation deviates from published algorithm to make it work at scale
  - Zookeeper is the open-source version of Chubby

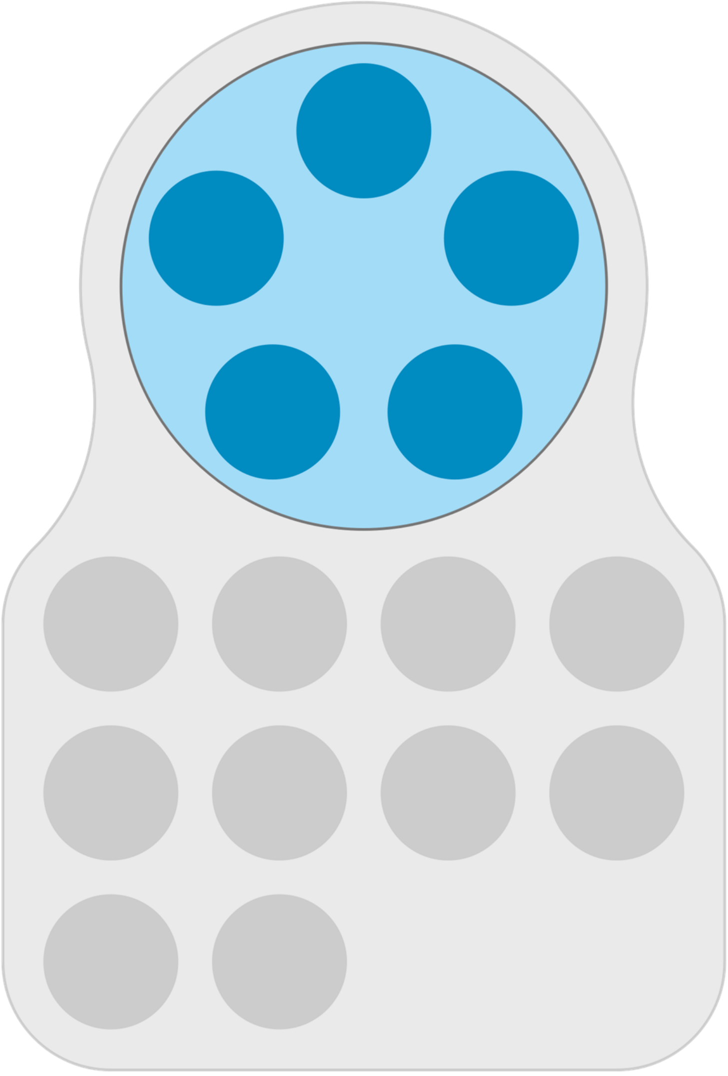
Paxos

- Replication protocol that ensures a global ordering of updates
  - Kind of like a **shared log**
  - All writes into the system are ordered in logical time
  - Replicas agree on the order of committed writes
- Uses a quorum approach to consensus
  - Typical implementations choose one replica as the leader
  - The protocol moves forward as long as $\lfloor N/2 \rfloor + 1$ replicas agree
- The “Paxos protocol” is actually a theoretical proof
  - Concrete implementation of the protocol described in *Paxos for System Builders*, Jonathan Kirsch and Yair Amir. http://www.cs.jhu.edu/~jak/docs/paxos_for_system_builders.pdf

Paxos and Raft

- Neo technologies implemented Paxos in the Neo4j database
- Found it error prone costly to maintain
- Replaced Paxos with Raft
- Proven to be more reliable and less costly to maintain
- So let’s look at Neo4j and Raft ….
    - Slides thanks to Jim Webber, Neo Technologies

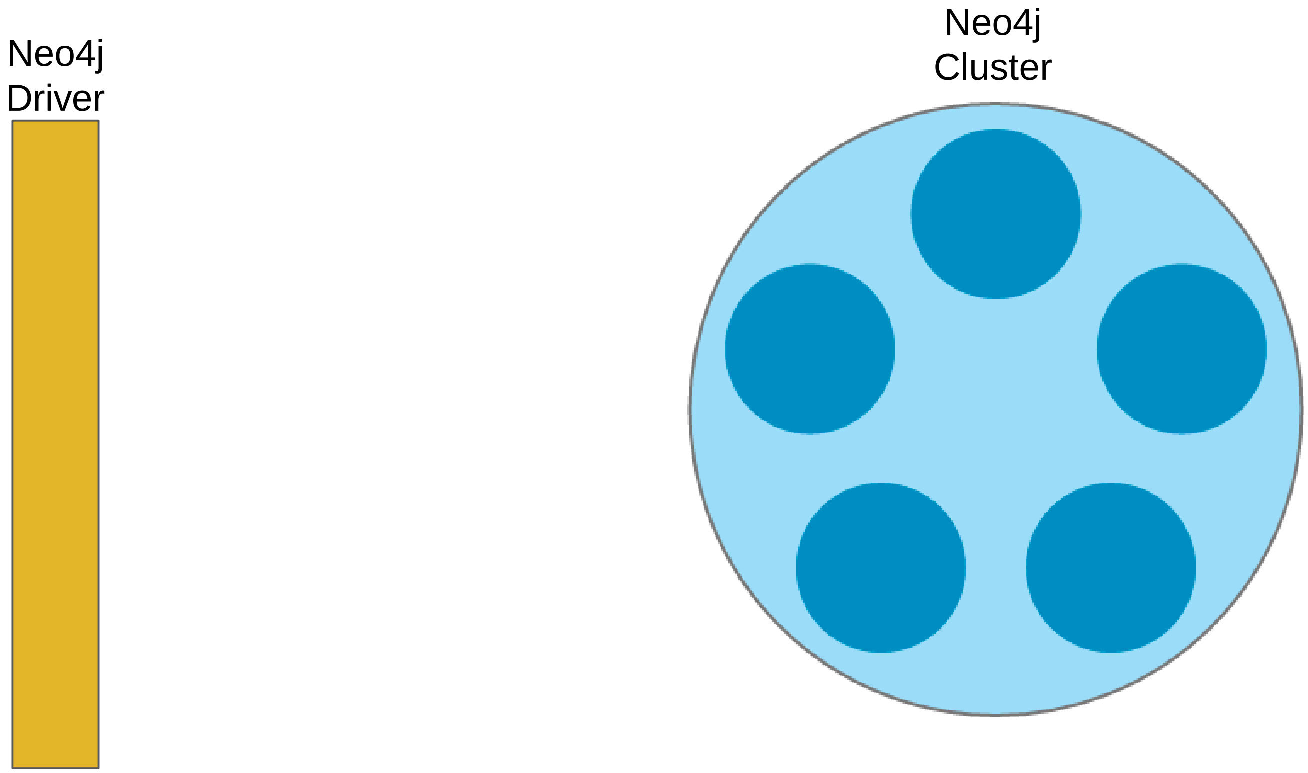

Core
Read
Replicas



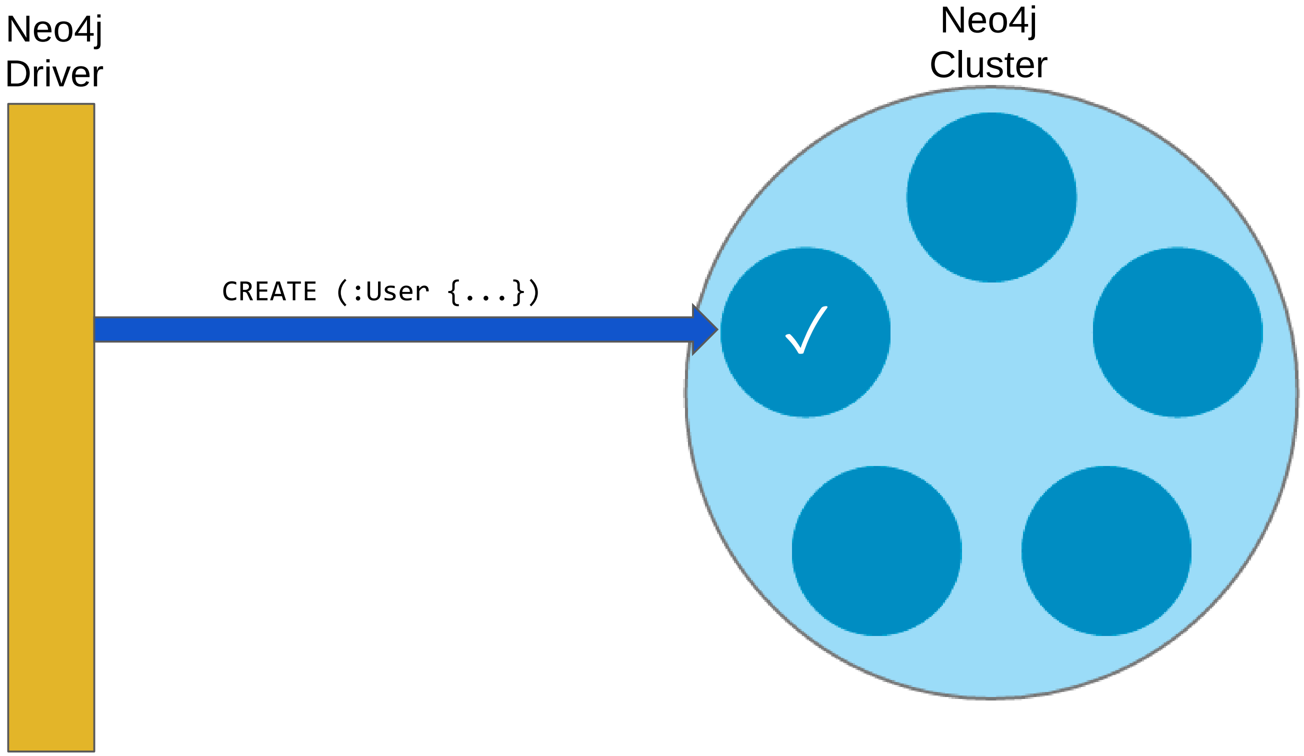
Core

- Small group of Neo4j databases maintain same copy of data
- Fault-tolerant Consensus Commit
- Responsible for data safety

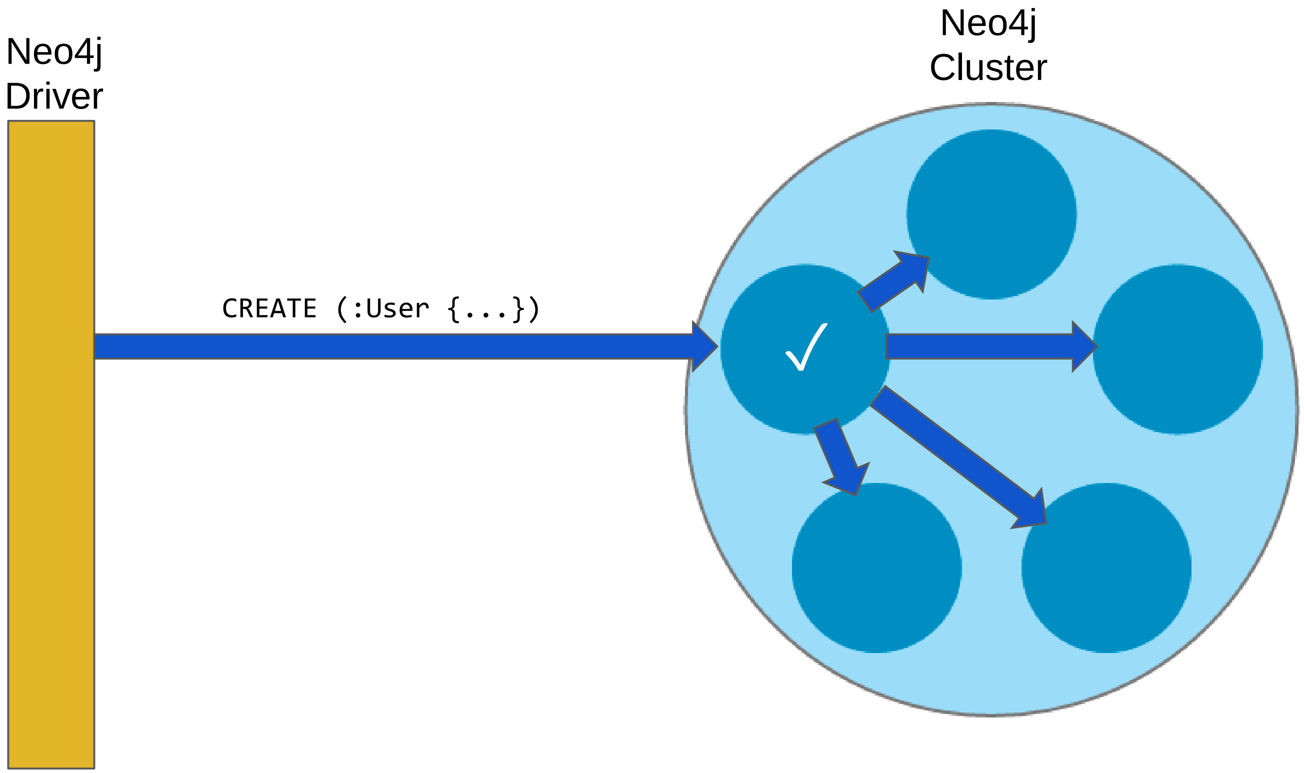
Writing to the Core Cluster



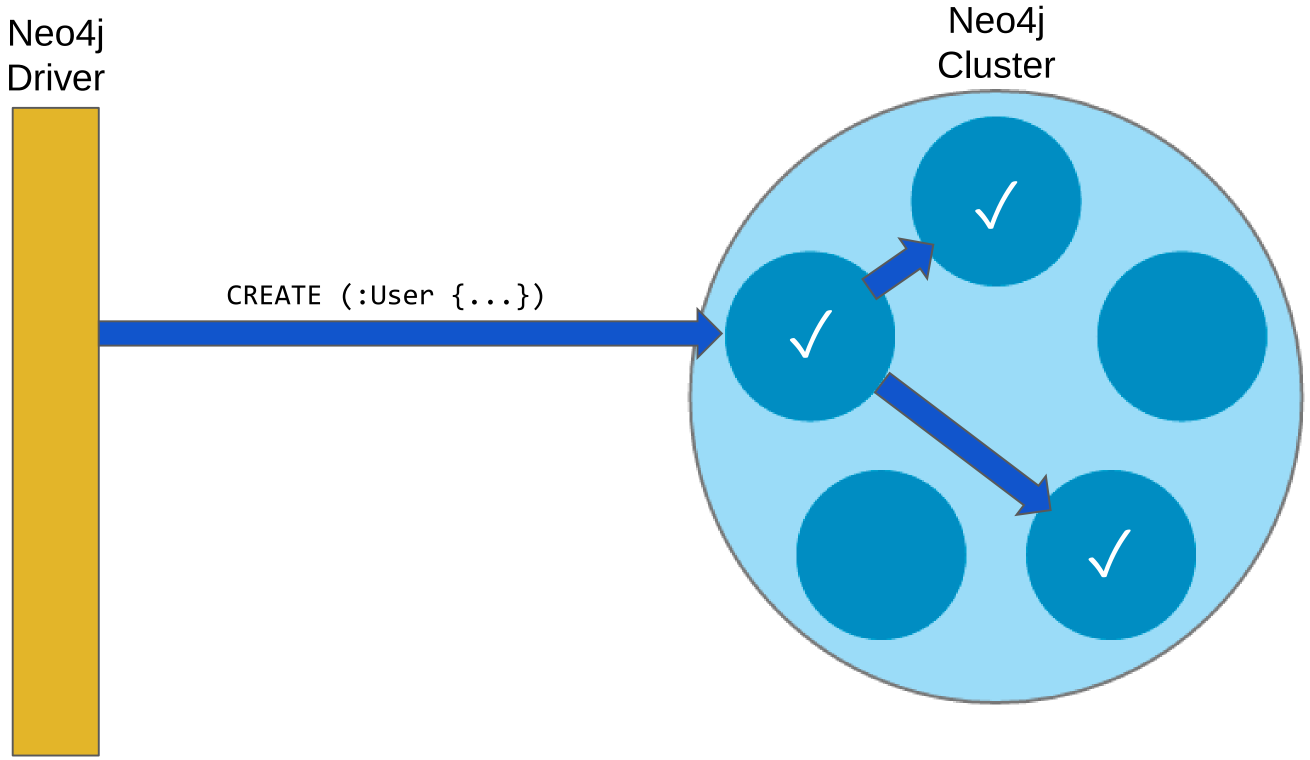
Writing to the Core Cluster



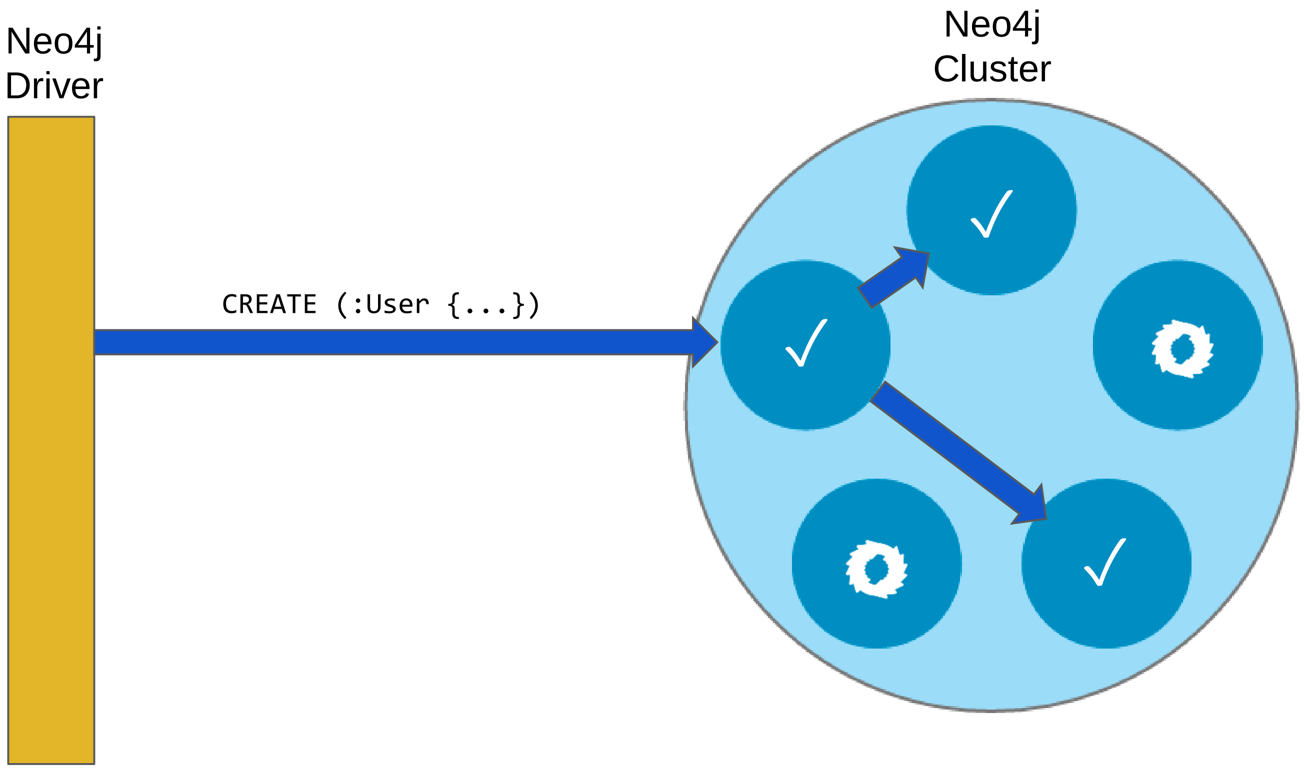
Writing to the Core Cluster



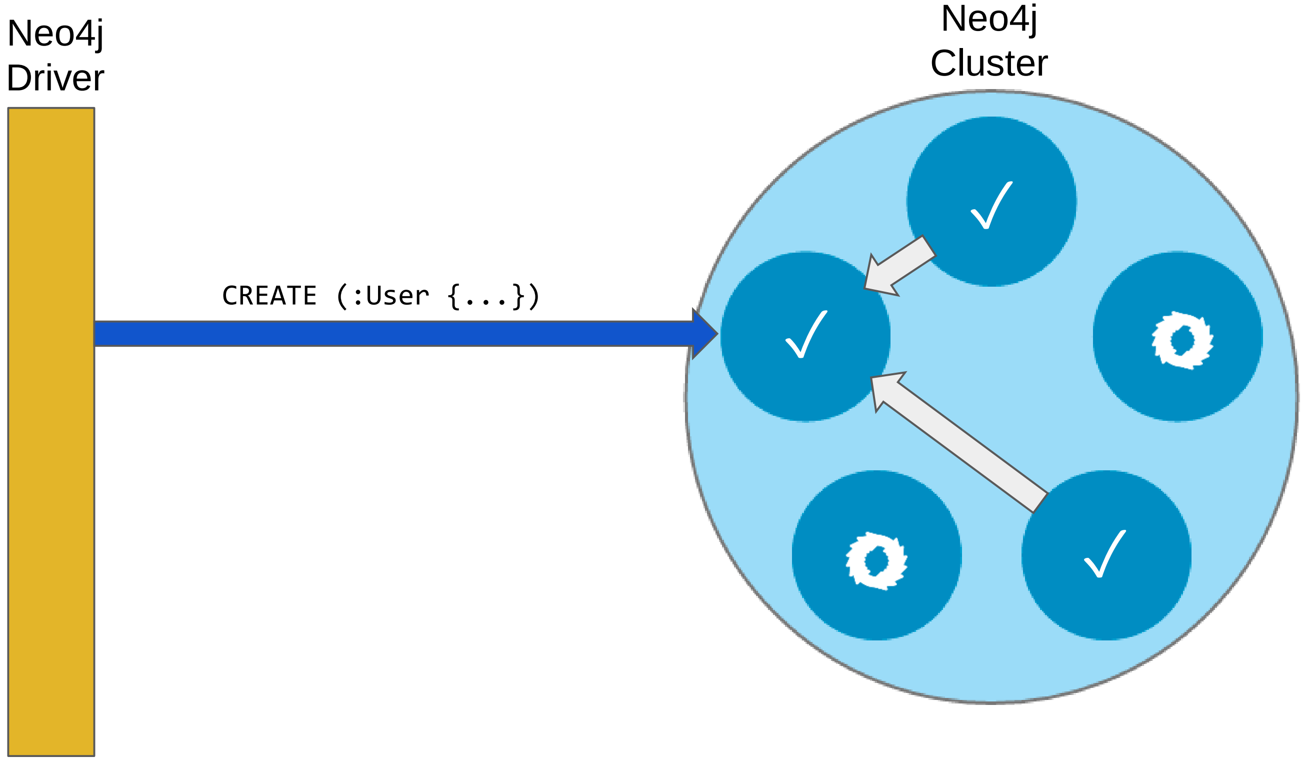
Writing to the Core Cluster



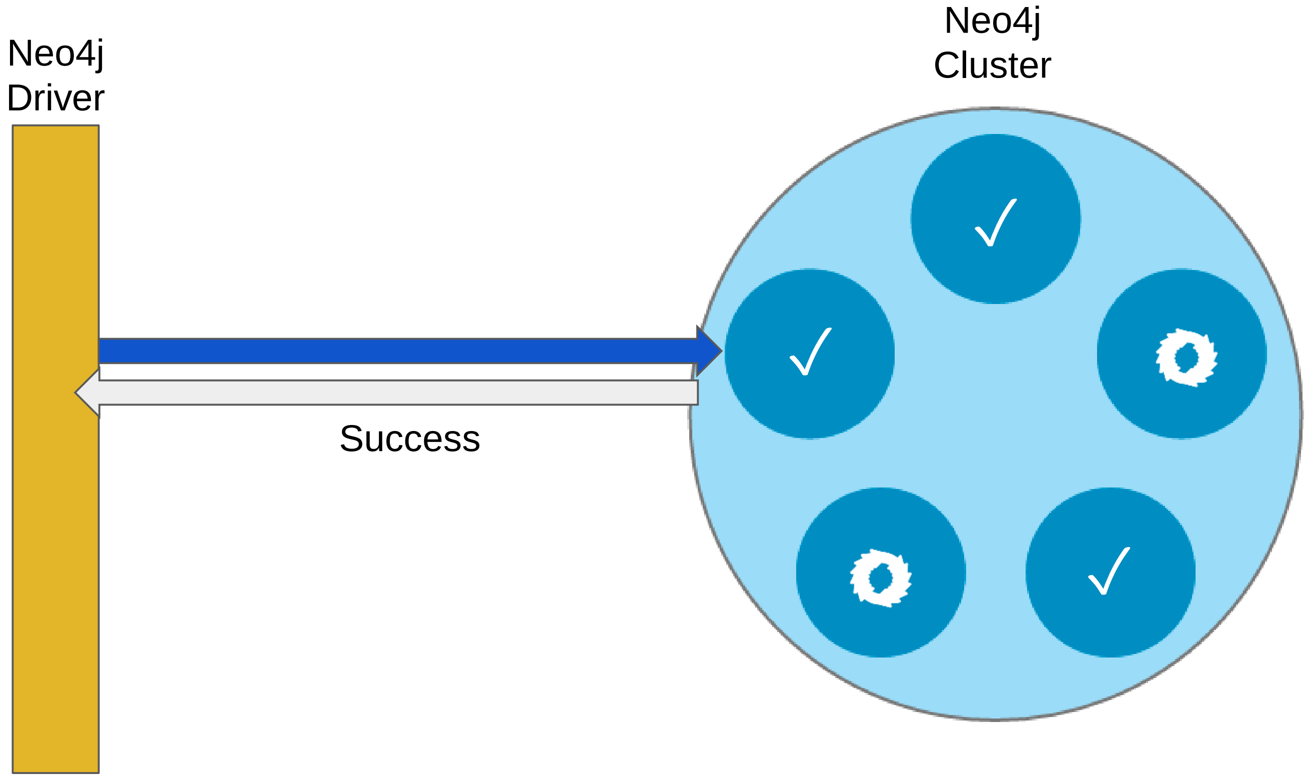
Writing to the Core Cluster



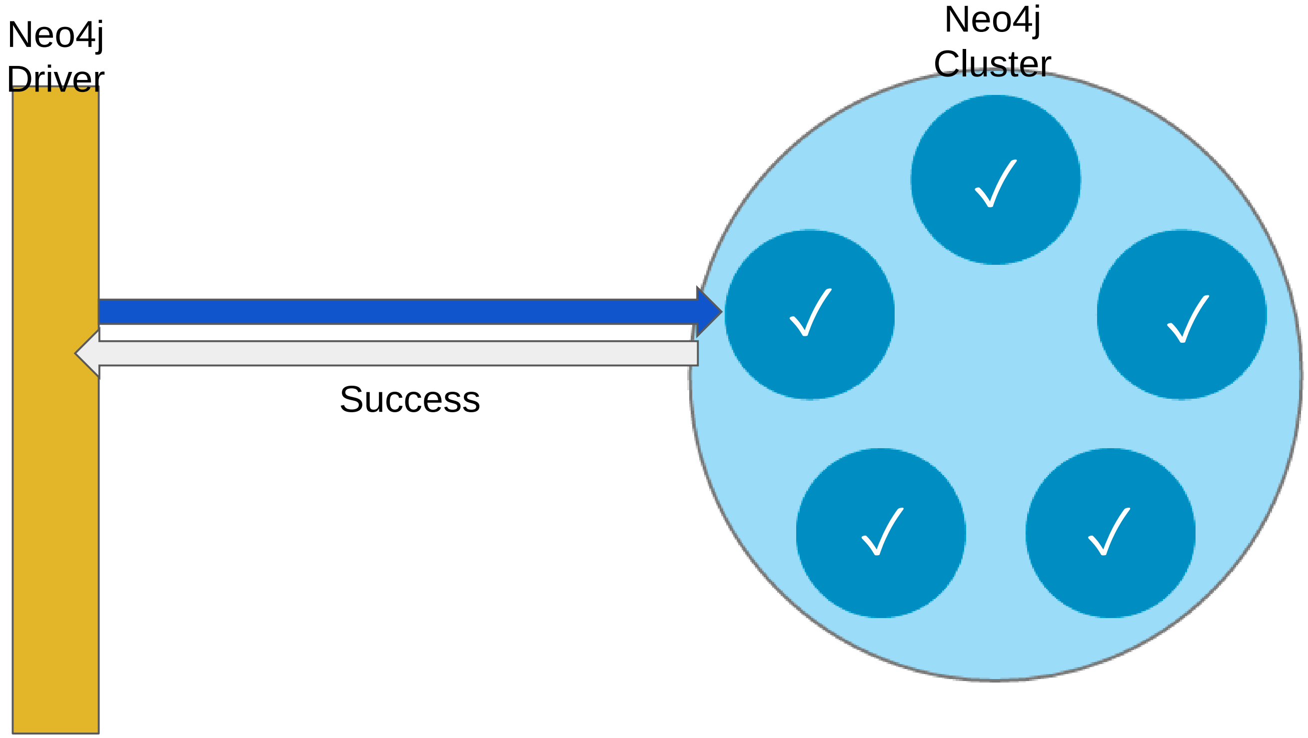
Writing to the Core Cluster



Writing to the Core Cluster



Writing to the Core Cluster



EASY TO UNDERSTAND

In Search of an Understandable Consensus Algorithm (Extended Version)

Diego Ongaro and John Ousterhout
Stanford University

Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety.

1 Introduction

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members. Because of this, they play a key role in building reliable large-scale software systems. Paxos [15, 16] has dominated the discussion of consen-

state space reduction (relative to Paxos, Raft reduces the degree of nondeterminism and the ways servers can be inconsistent with each other). A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [29, 22]), but it has several novel features:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.
- **Leader election:** Raft uses randomized timers to elect leaders. This adds only a small amount of mechanism to the heartbeats already required for any consensus algorithm, while resolving conflicts simply and rapidly.
- **Membership changes:** Raft's mechanism for

Class Exercise: Raft explained

http://thesecretlivesofdata.com/raft/

Raft in Neo4j

For safety, there is only one Leader able to make forward progress in any given term.

The Leader imposes order on Raft log entries and driving the log forward with respect to *Followers*.

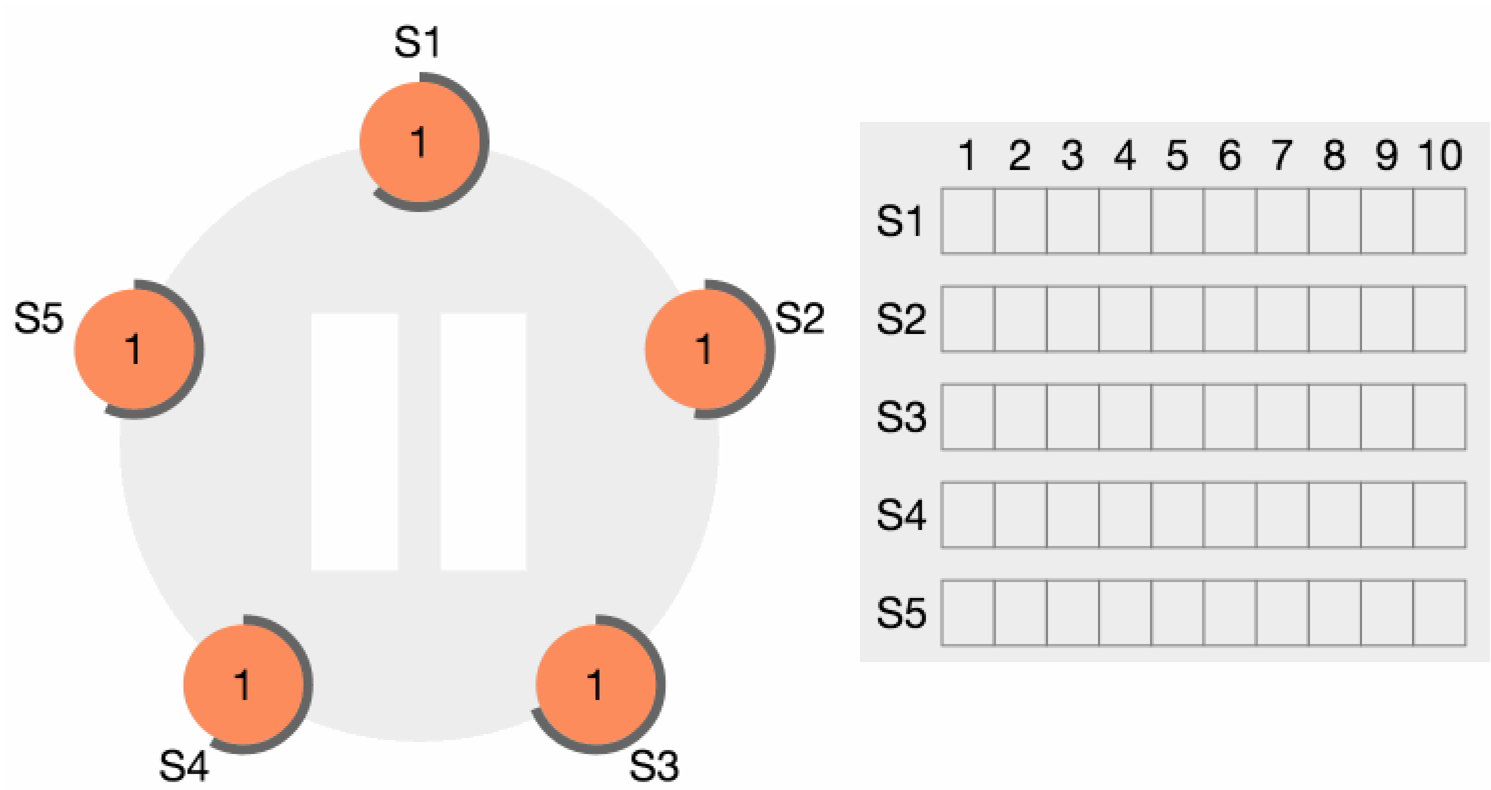
Followers maintain their logs with respect to the current Leader's log.

Any follower suspects that the Leader has failed, they instigate a leadership election by entering the *Candidate* state. (timeout is 500ms default.)

Instance is in the best state (including the existing Leader, if it remains available) can emerge from the election as Leader.

The "best state" for a Leader is decided by highest term, then by longest log, then by highest committed entry.

Raft Protocol



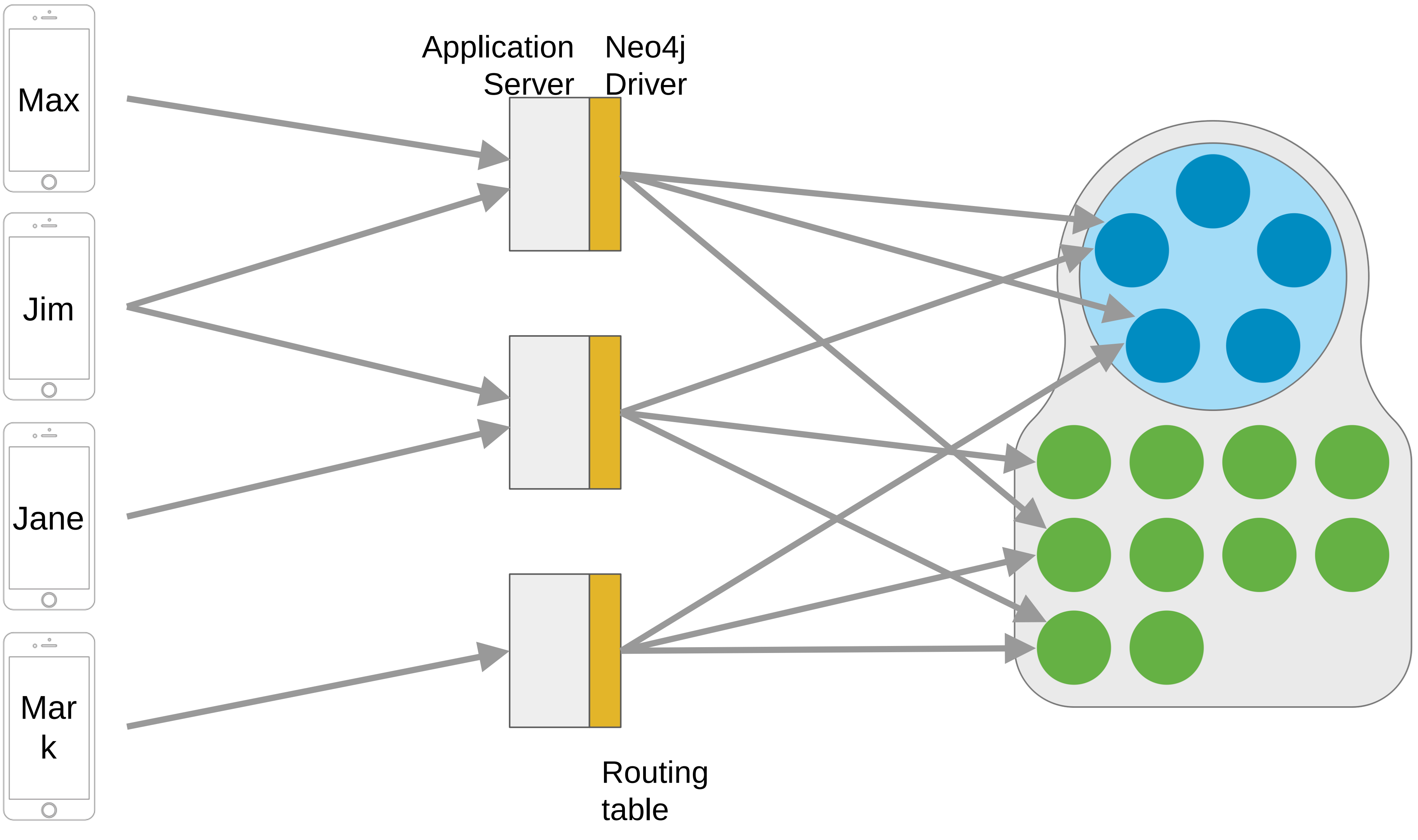
https://github.com/ongardie/raftscope

Load balancing requests

bolt+routing://

```
GraphDatabase.driver( "bolt+routing://aCoreServer" )
```

Bootstrap: specify any core server to route load across the whole cluster


Max
Jim
Jane
Mar
k
Application
Server
Neo4j
Driver
Routing
table

Routing Table

Address	Role
192.168.1.101:7687	ROUTE
192.168.1.101:7687	WRITE
192.168.1.102:7687	ROUTE
192.168.1.102:7687	READ
:	:
10.0.15.202:7687	READ
10.0.15.203:7687	READ
	TTL = 300s

The leader will typically present itself as the only **WRITE** server

All core machines can **ROUTE**, i.e. provide routing table information

Read replicas fulfil the **READ** role

The TTL is generated by the cluster based on configuration

Routed write statements

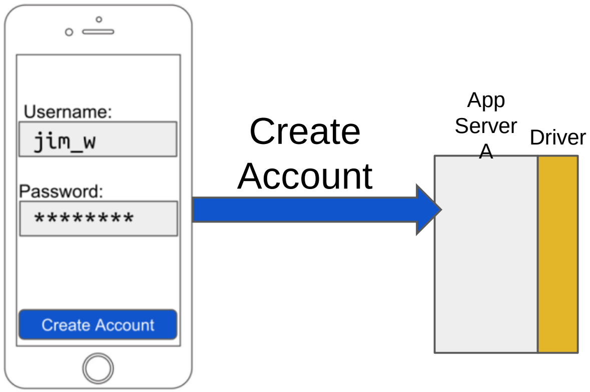
```
driver = GraphDatabase.driver( "bolt+routing://aCoreServer" );

try ( Session session = driver.session( AccessMode.WRITE ) )
{
   try ( Transaction tx = session.beginTransaction() )
   {
       tx.run( "MERGE (user:User {userId: {userId}})",
               parameters( "userId", userId ) );

       tx.success();
   }
}
```

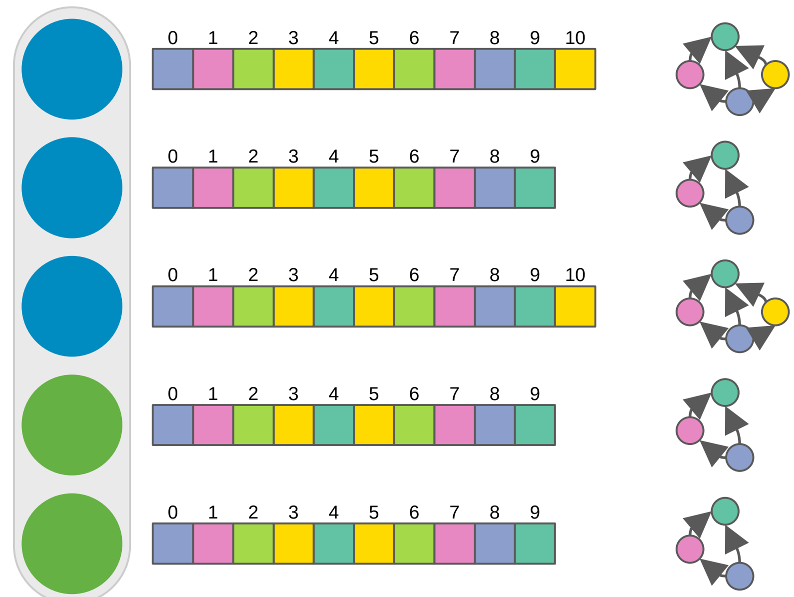
Routed read queries

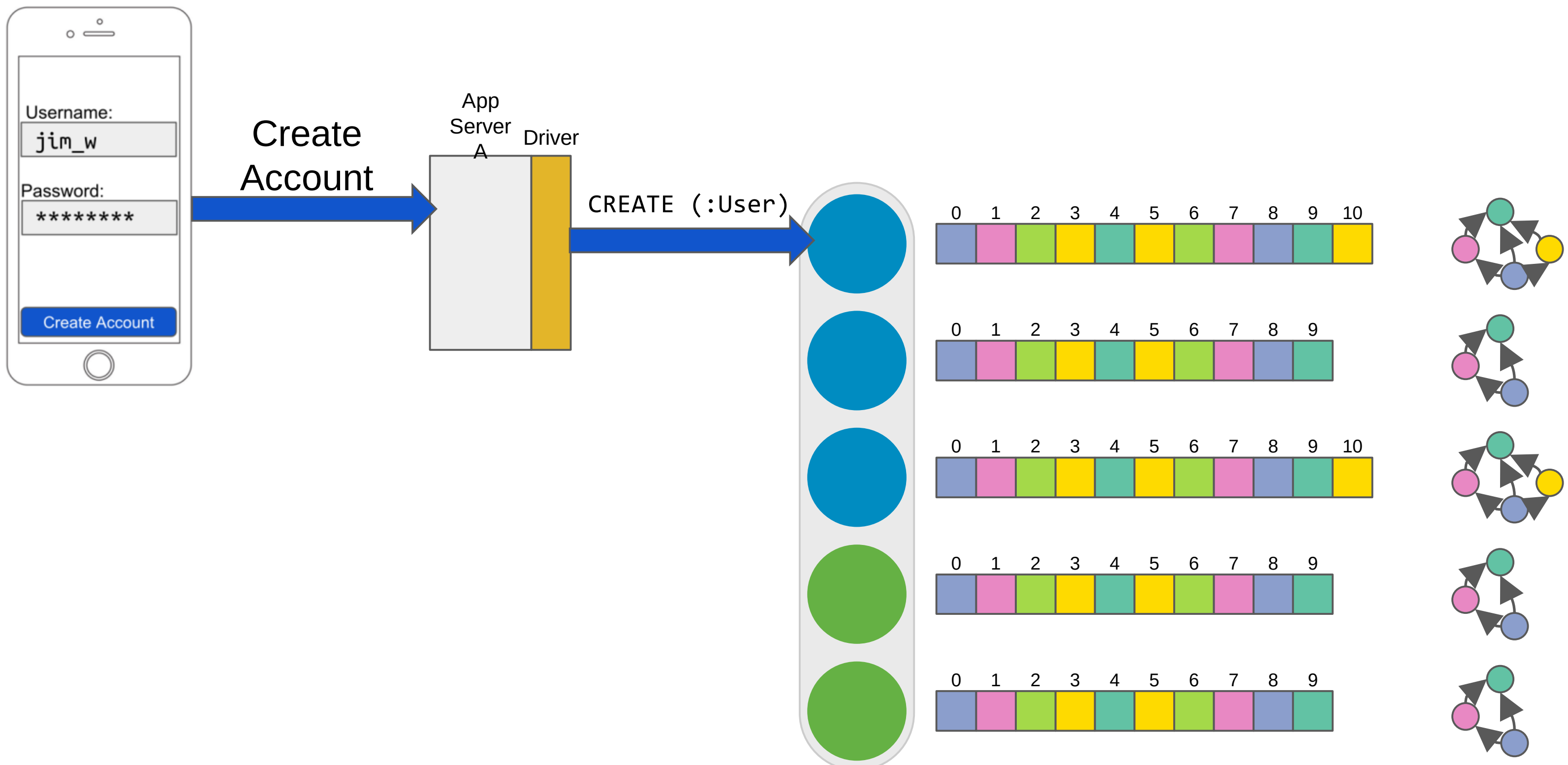
```
driver = GraphDatabase.driver( "bolt+routing://aCoreServer" );

try ( Session session = driver.session( AccessMode.READ ) )
{
   try ( Transaction tx = session.beginTransaction() )
   {
       tx.run( "MATCH (user:User {userId: {userId}})-[*]-(:Product) RETURN *",
               parameters( "userId", userId ) );

       tx.success();
   }
}
```

Routing Table

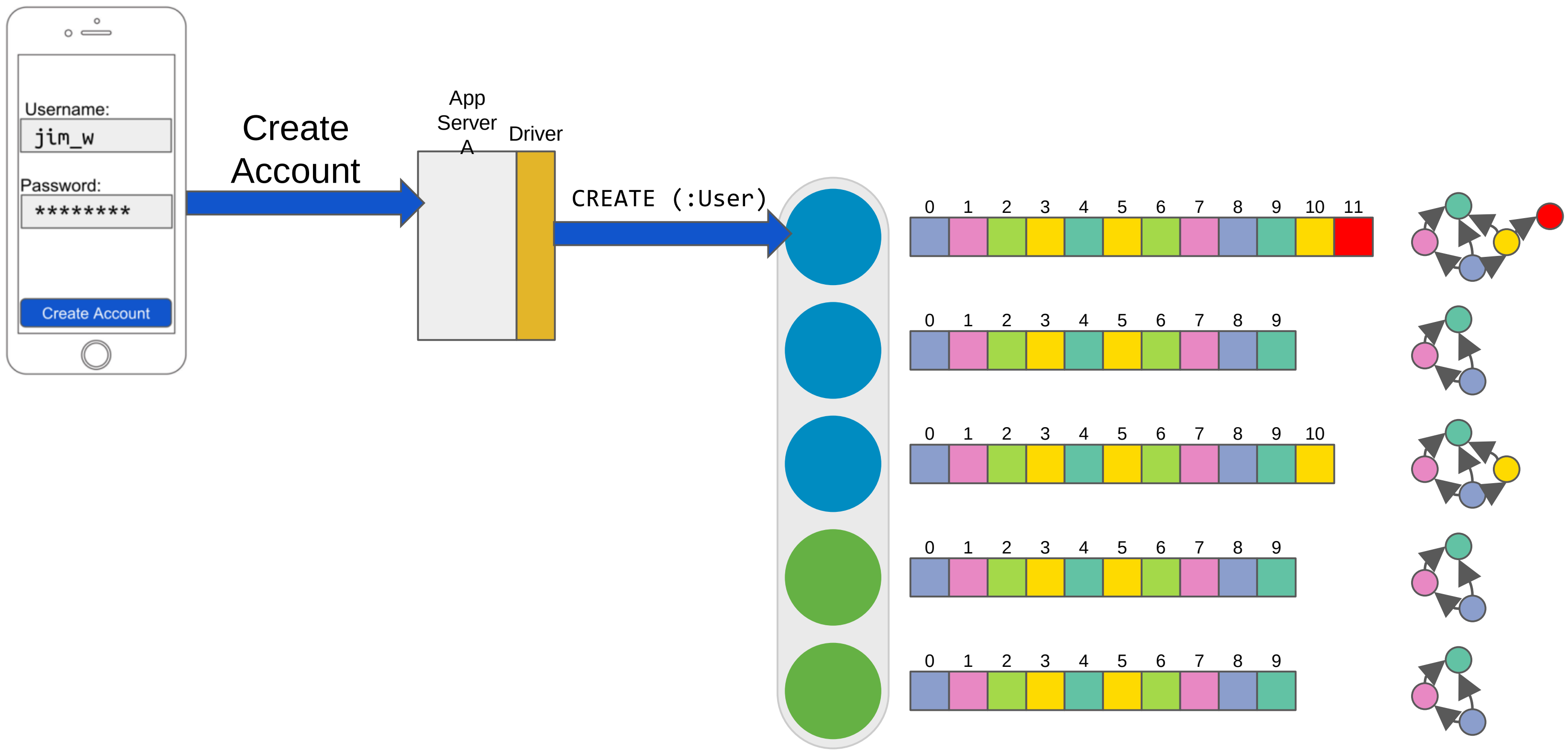
- Refreshed by default at driver every 300 secs
- What if new leader elected?
- What about casual consistency? Read your own writes ….

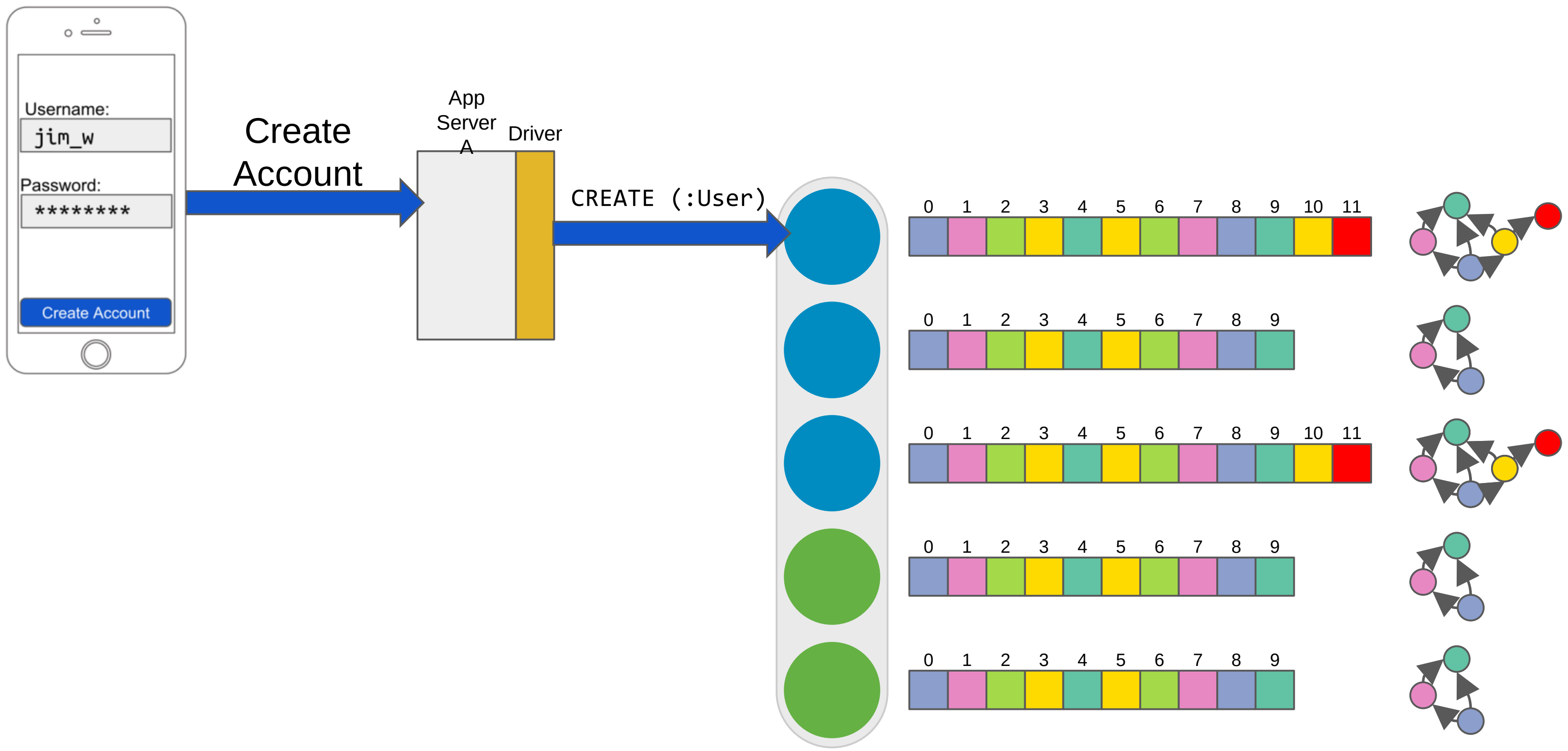

Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver


0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9

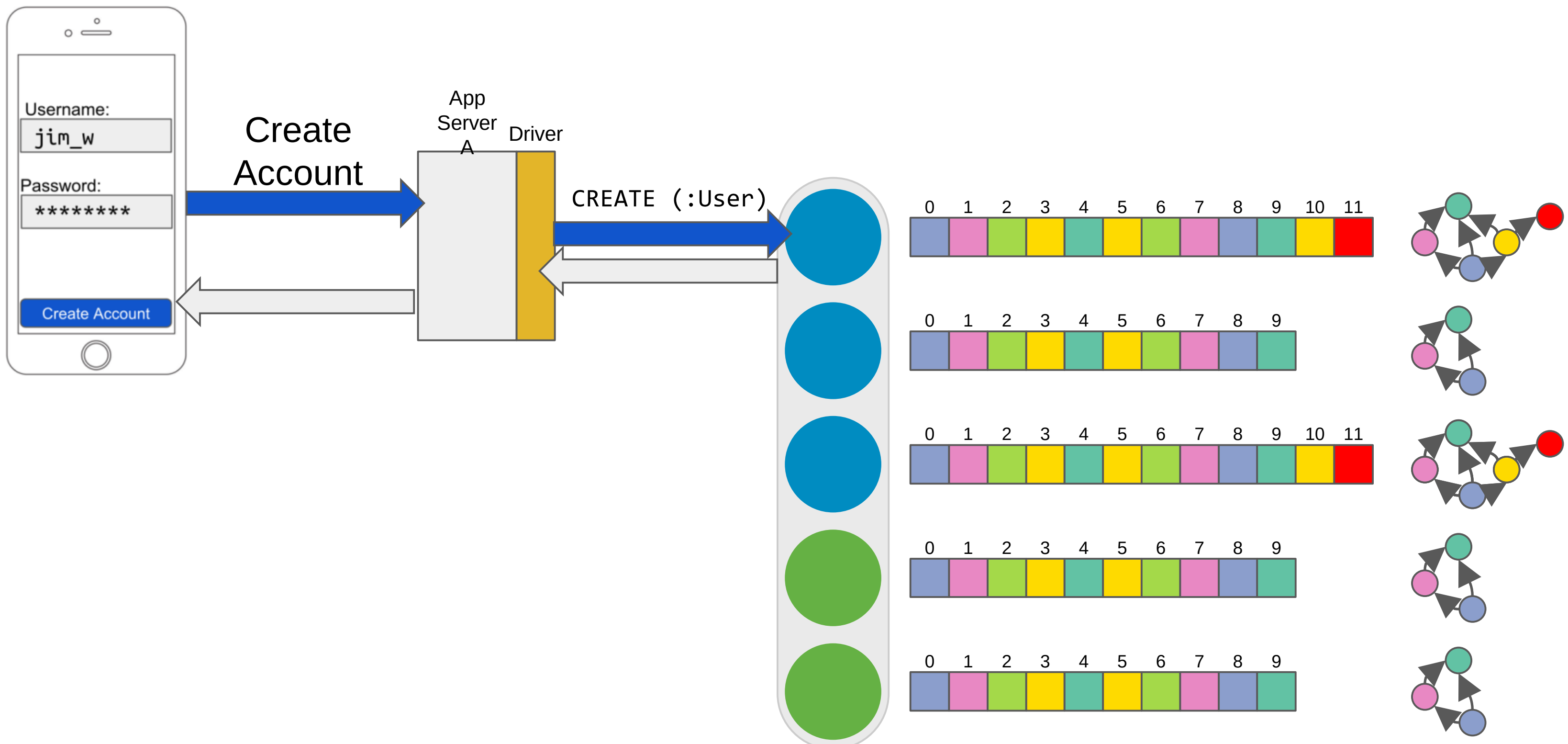


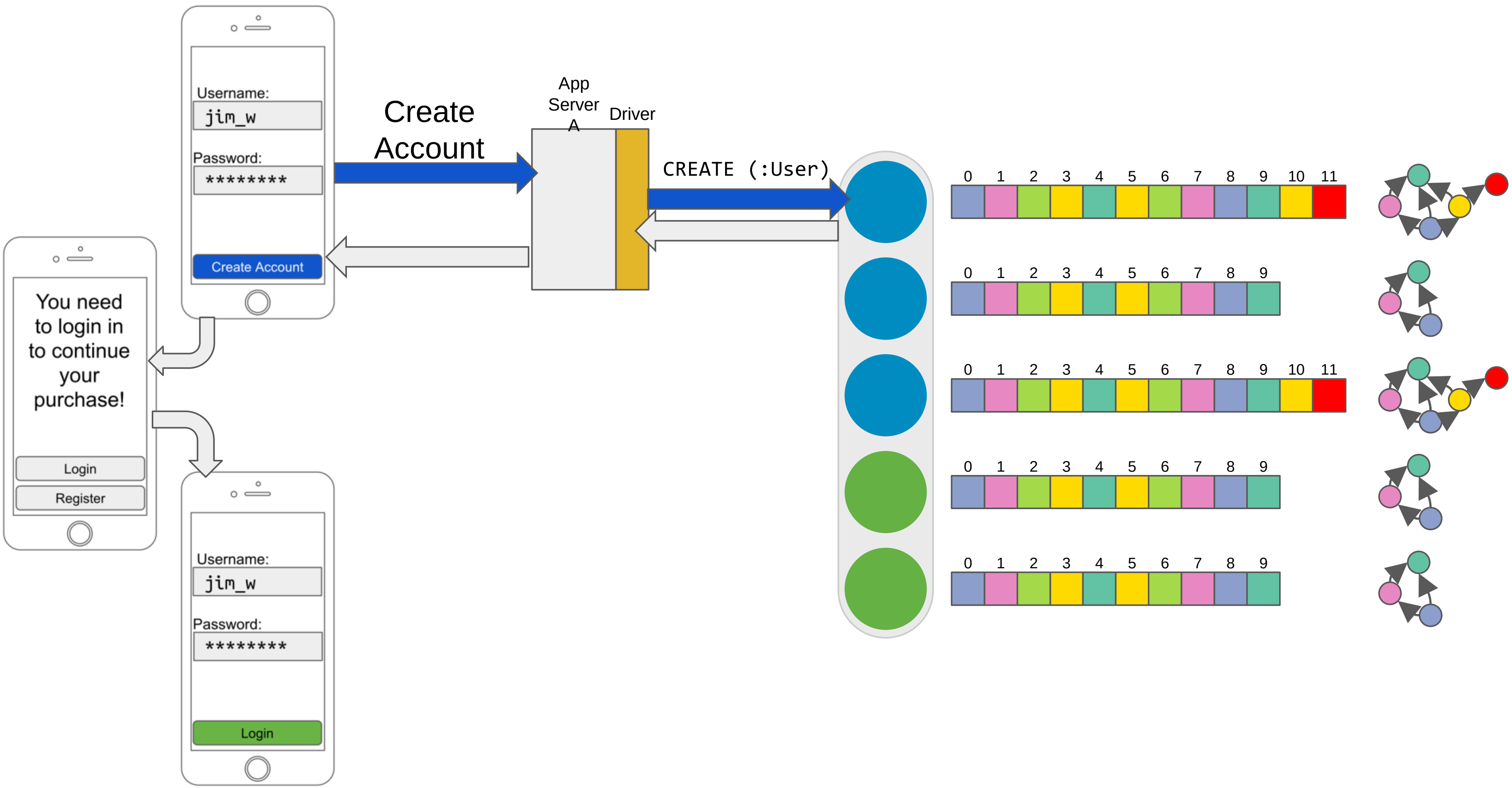
Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver
CREATE (:User)
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9

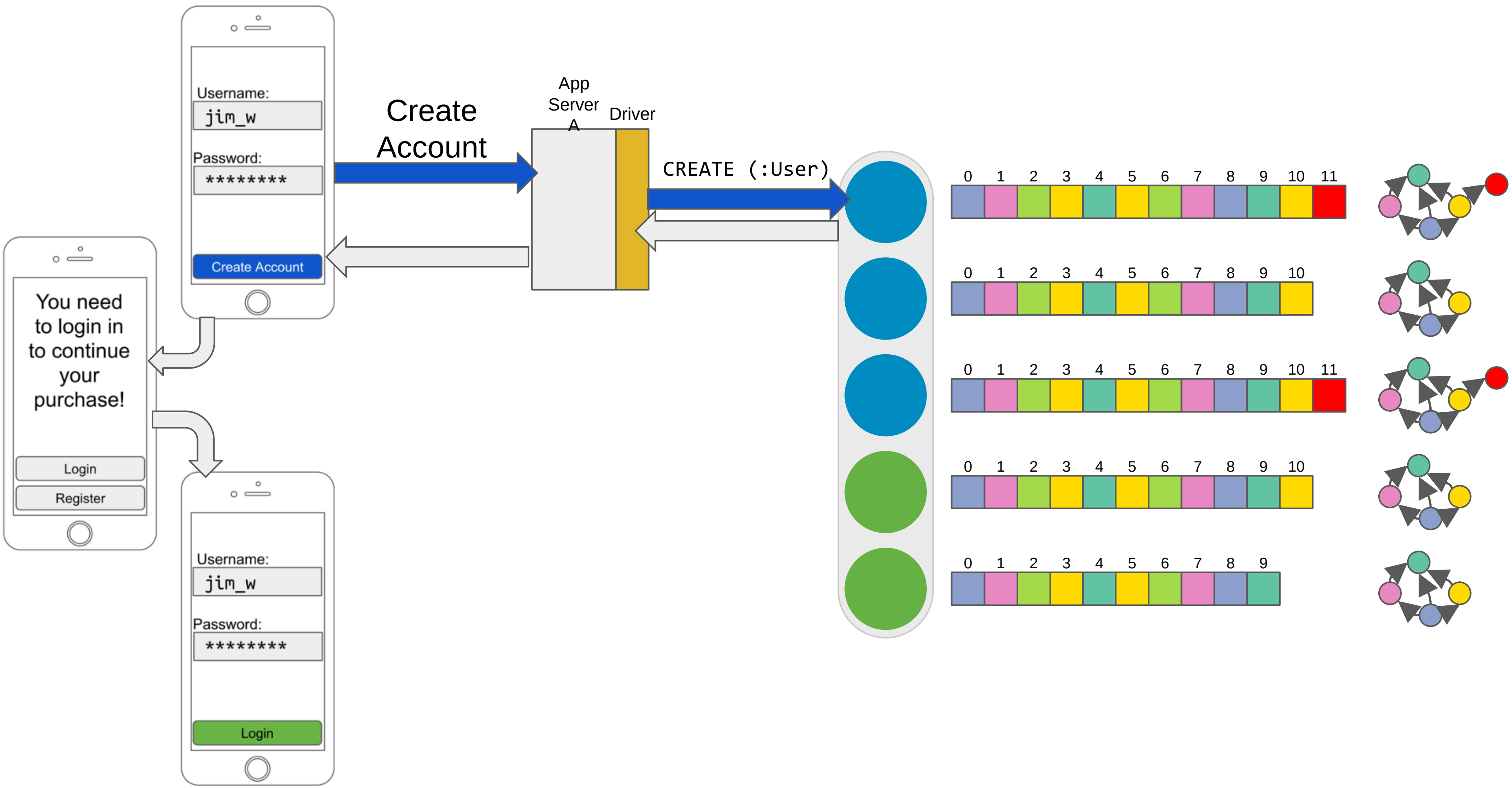

Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver
CREATE (:User)
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9



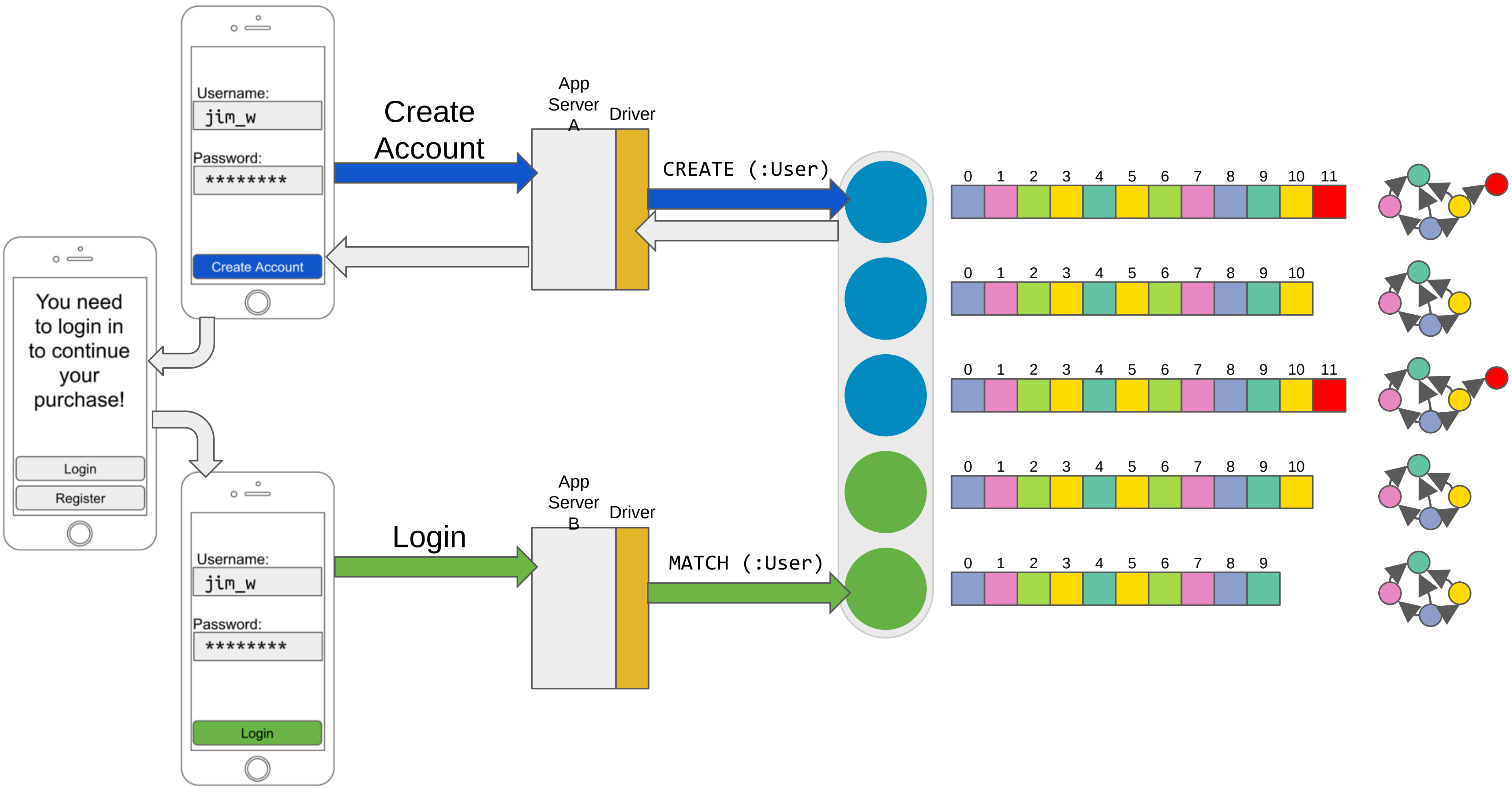
Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver
CREATE (:User)
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9


Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver
CREATE (:User)
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9

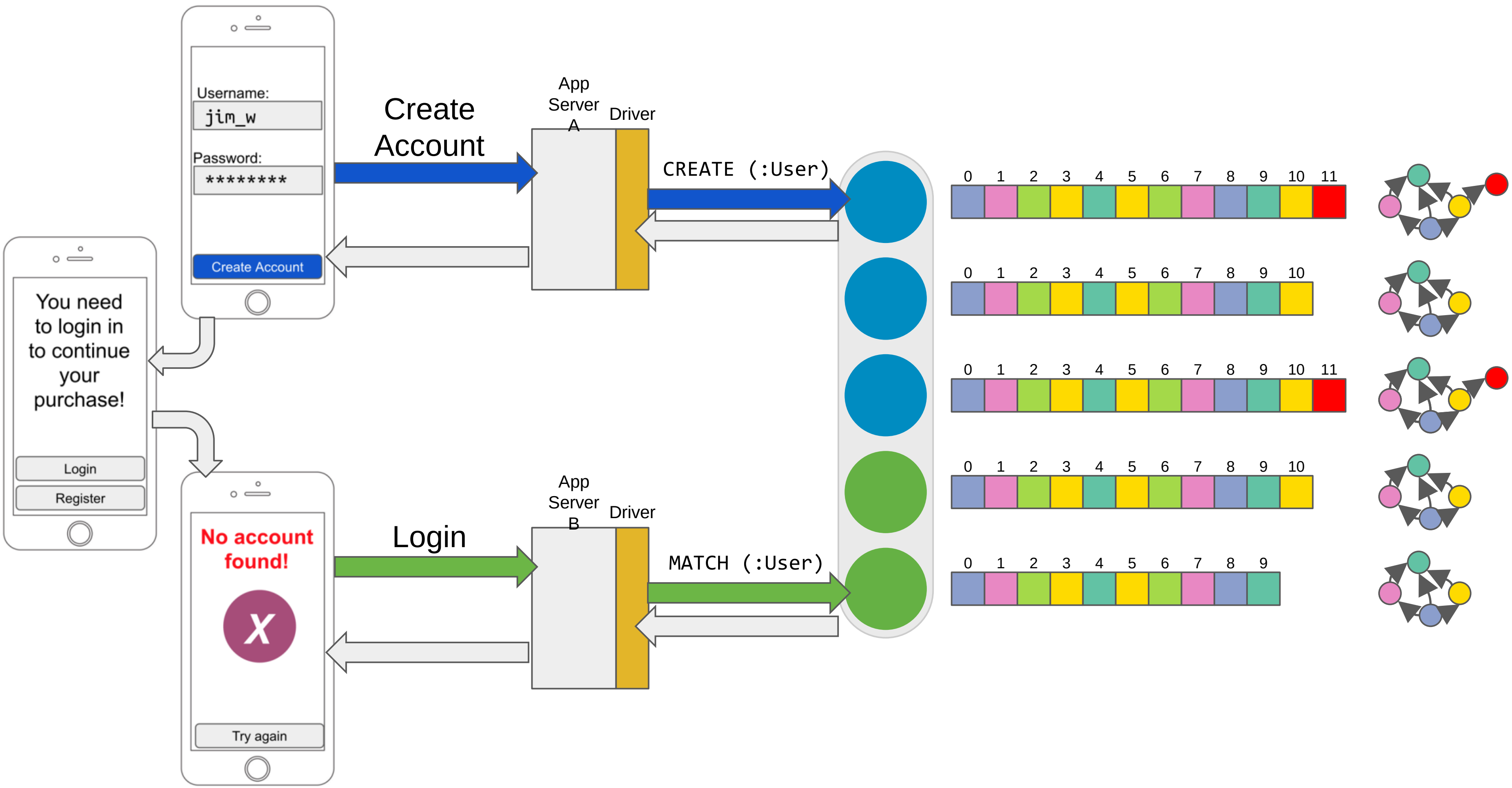

Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver
CREATE (:User)
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
You need to login in to continue your purchase!
Login
Register
Username:
jim_w
Password:
****
Login



Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver
CREATE (:User)
You need to login in to continue your purchase!
Login
Register
Username:
jim_w
Password:
****
Login
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9

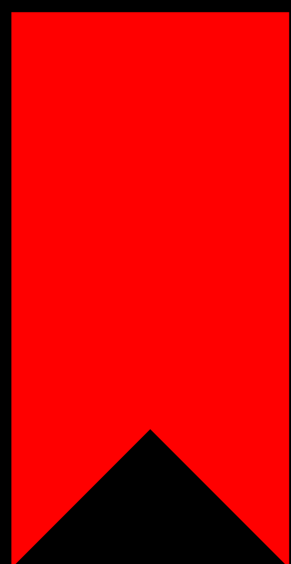


Username:
jim_w
Password:
****
Create Account
Create
Account
App
Server
A
Driver
CREATE (:User)
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9
You need to login in to continue your purchase!
Login
Register
Username:
jim_w
Password:
****
Login
Login
App
Server
B
Driver
MATCH (:User)



Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver
CREATE (:User)
You need to login in to continue your purchase!
Login
Register
No account found!
X
Try again
Login
App Server B
Driver
MATCH (:User)
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9

Bookmark



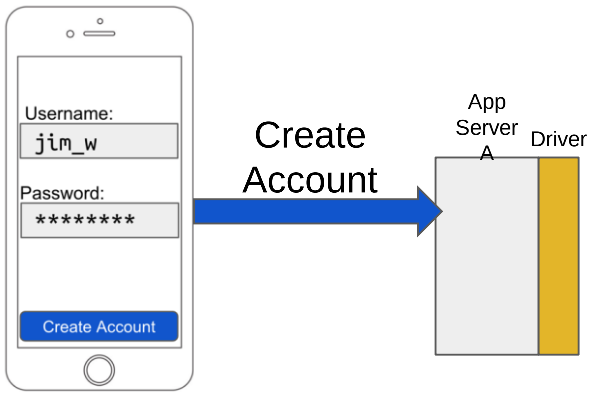
Session token

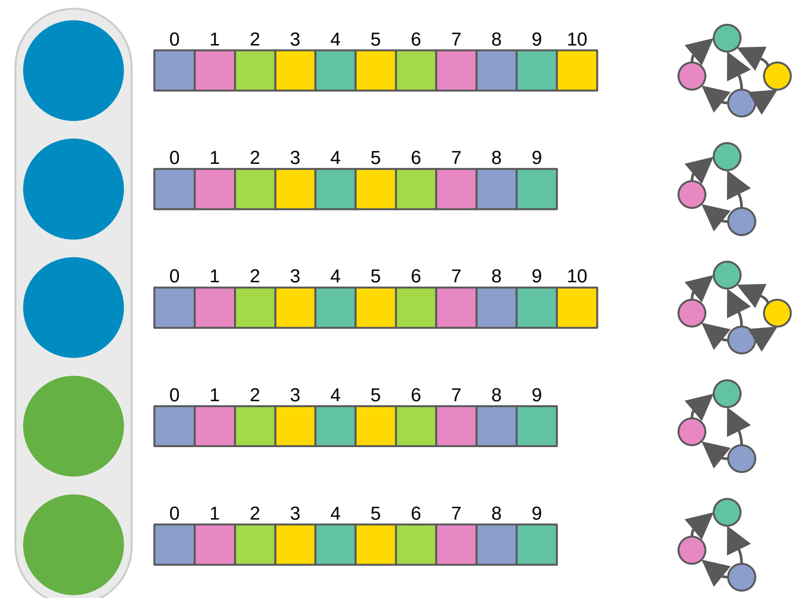
String (for portability)

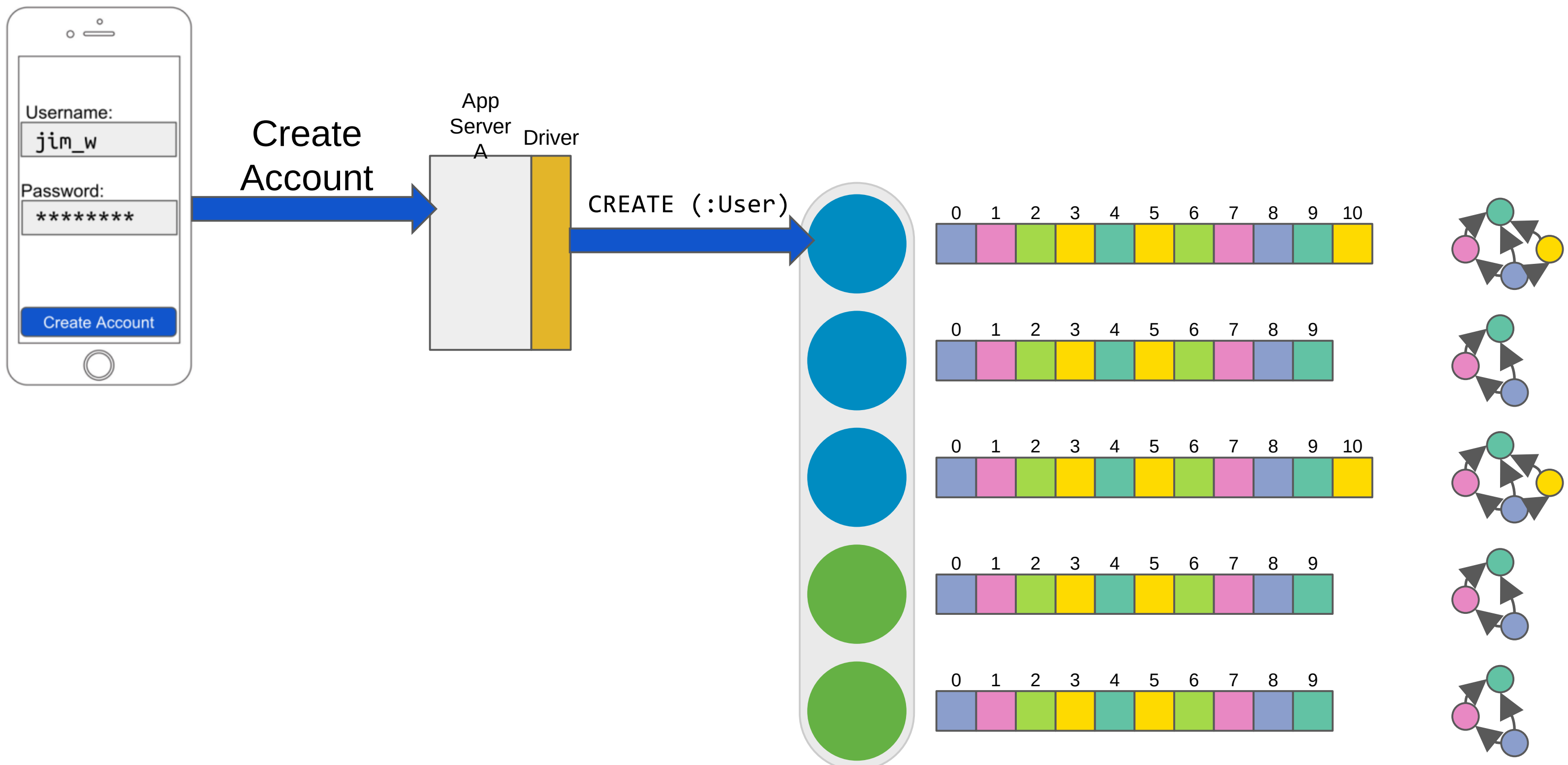
Opaque to application

Represents user’s most recent view of the graph

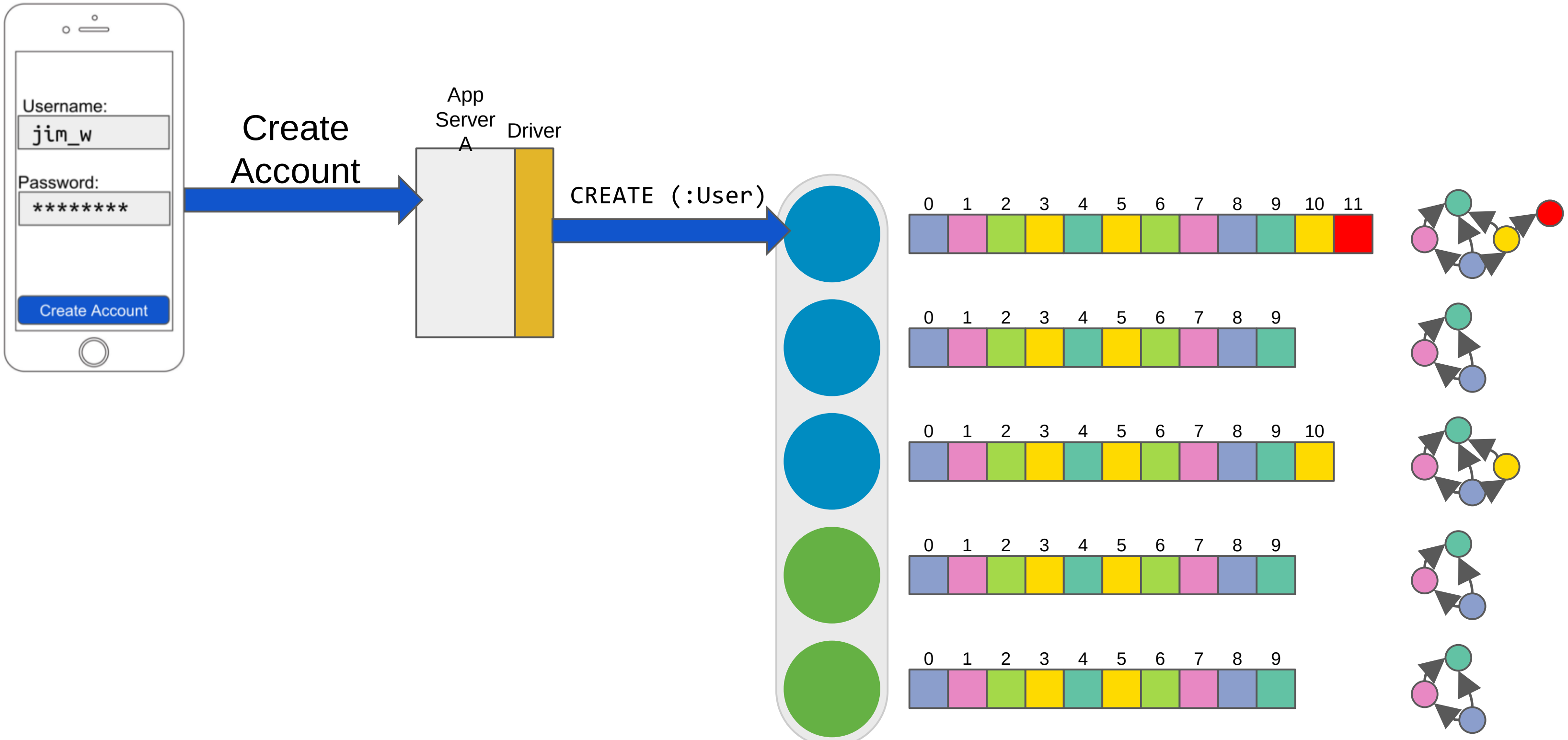
Let’s try again, with Causal Consistency


Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver

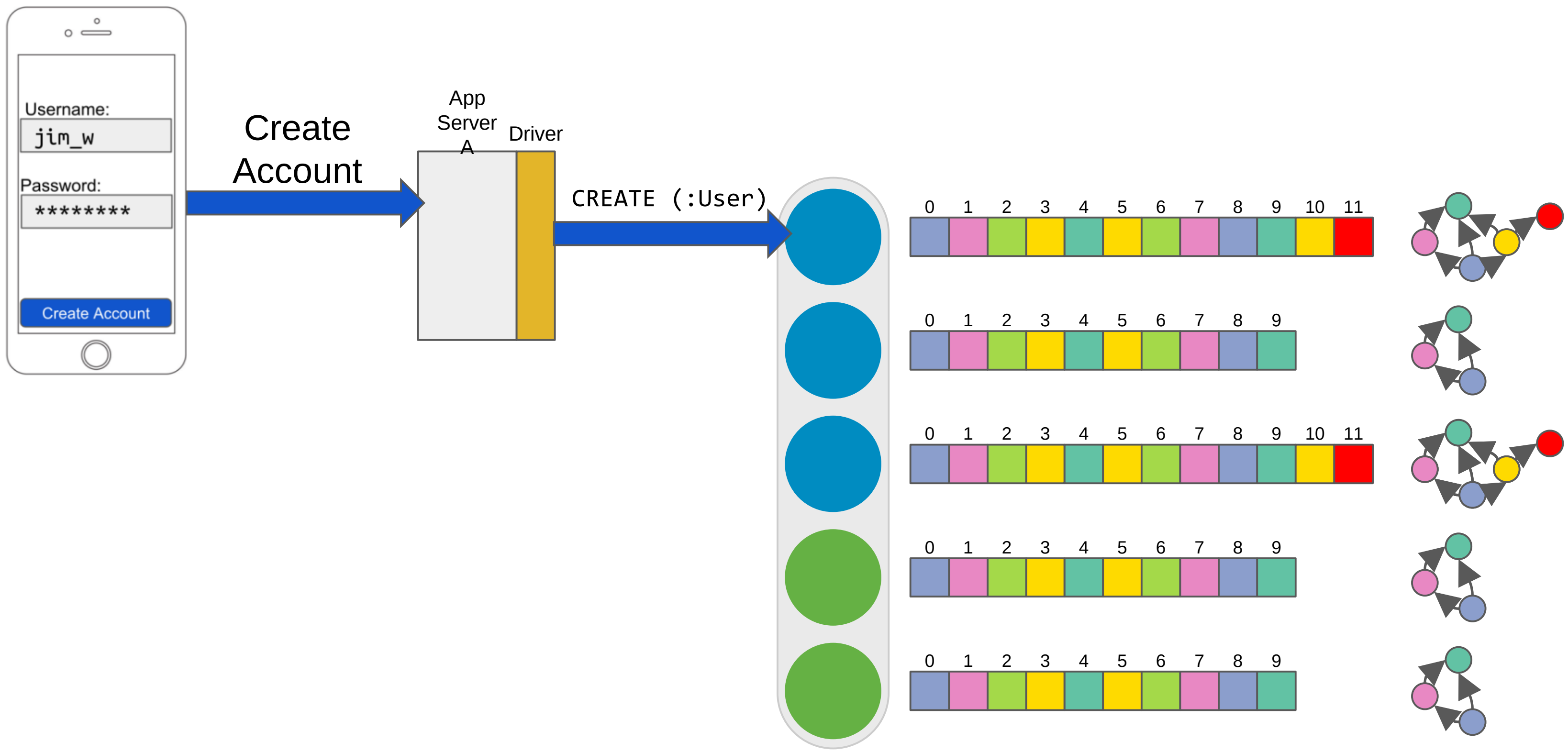

0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9



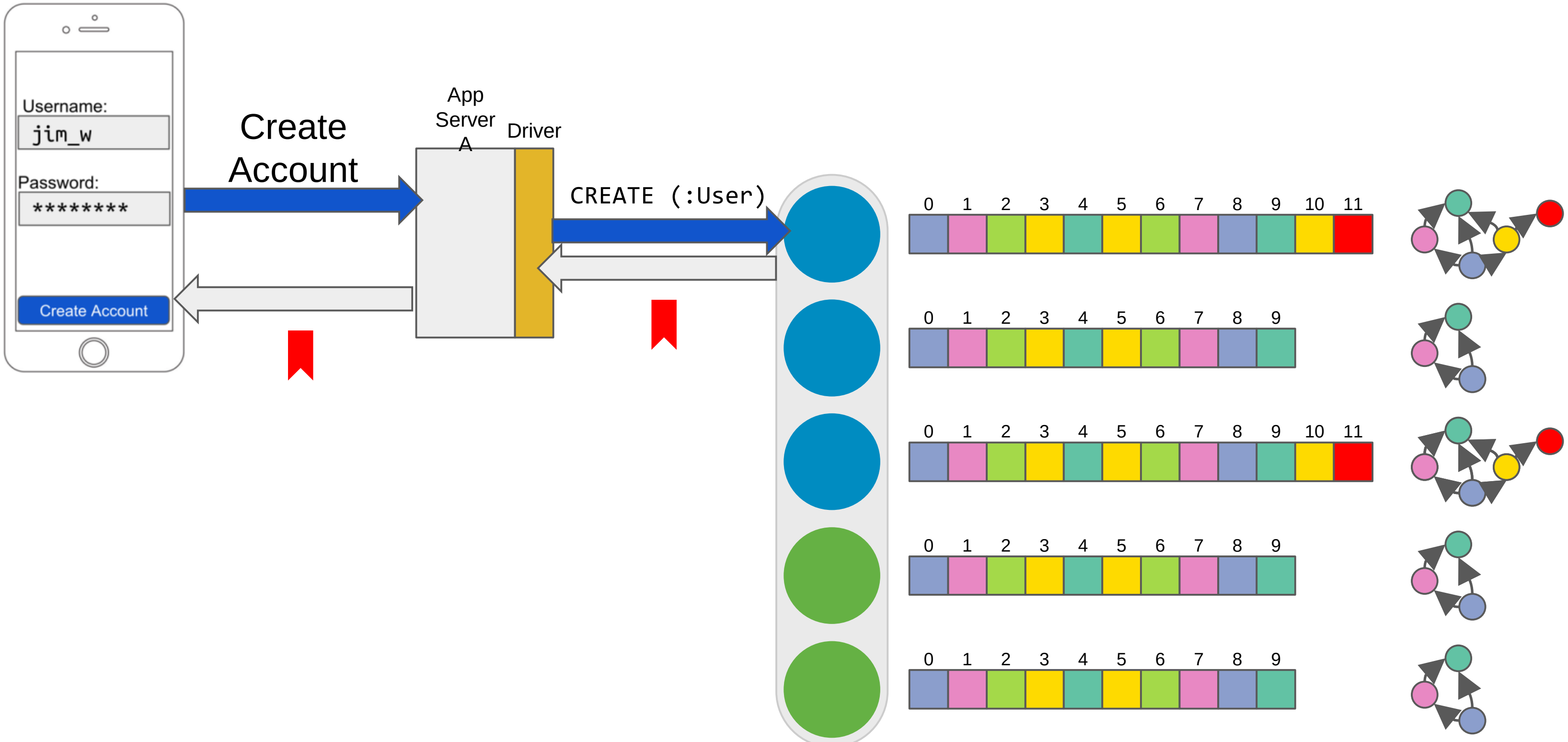
Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver
CREATE (:User)
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9



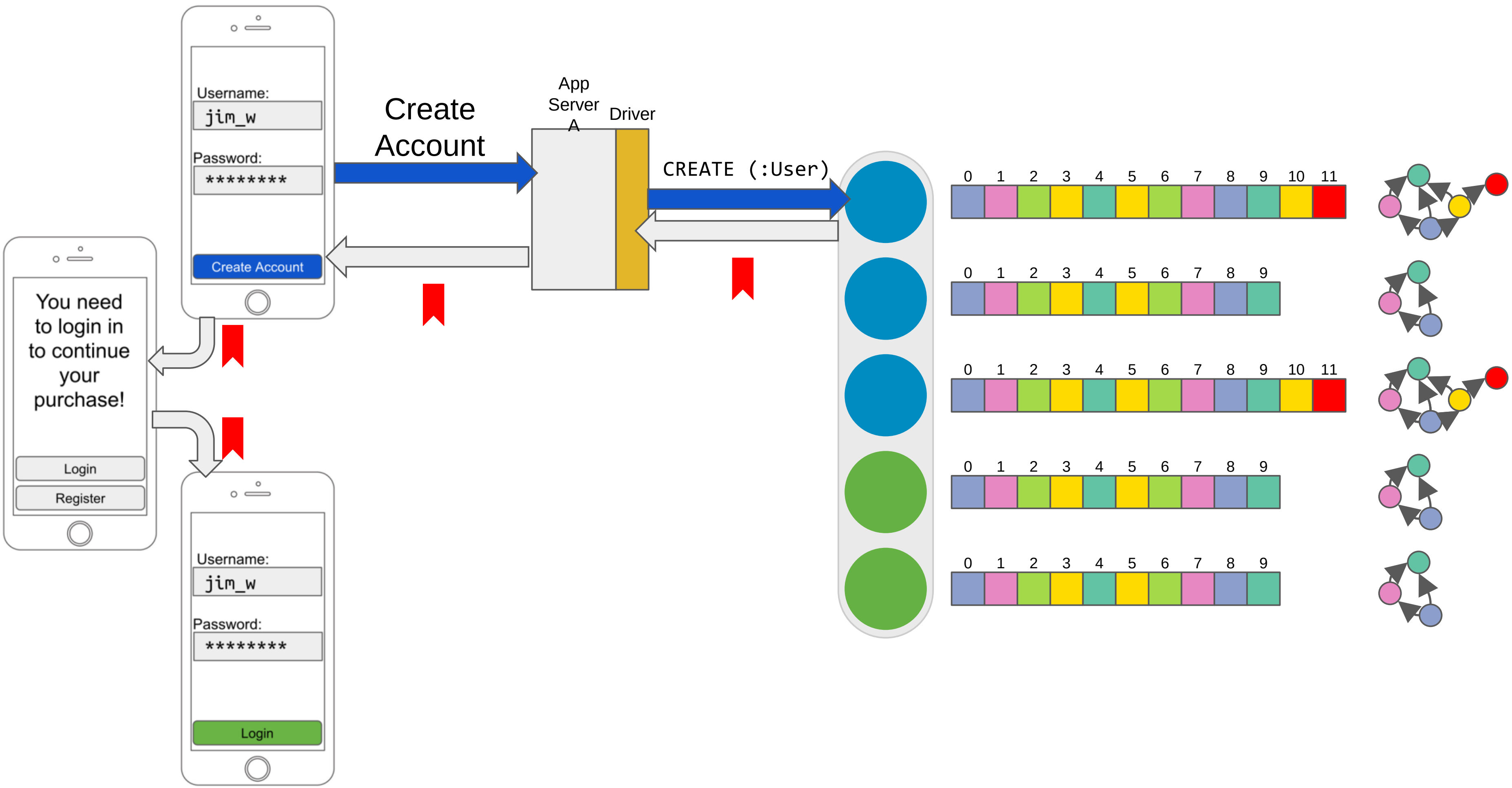
Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver
CREATE (:User)
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9



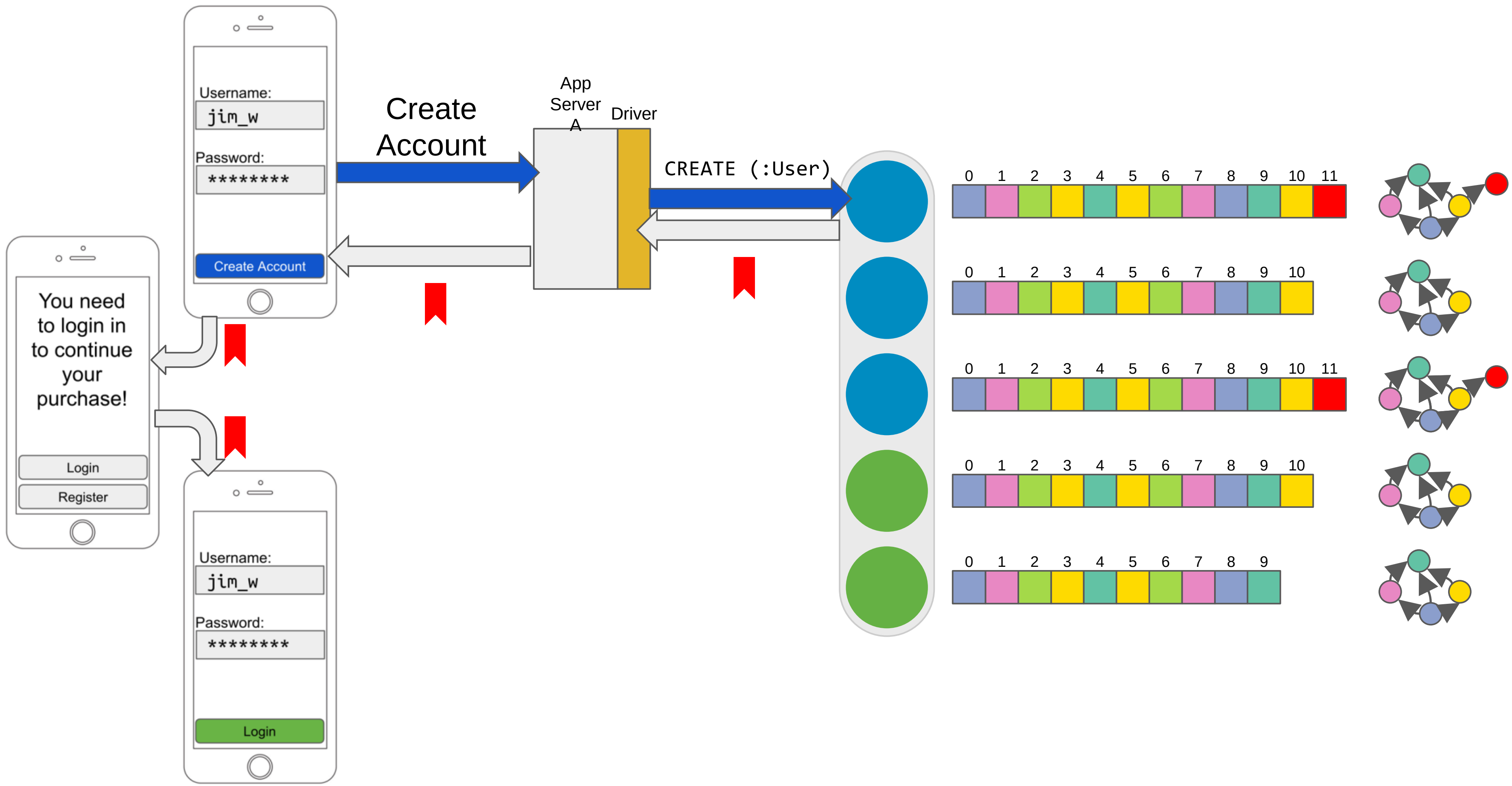
Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver
CREATE (:User)
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9



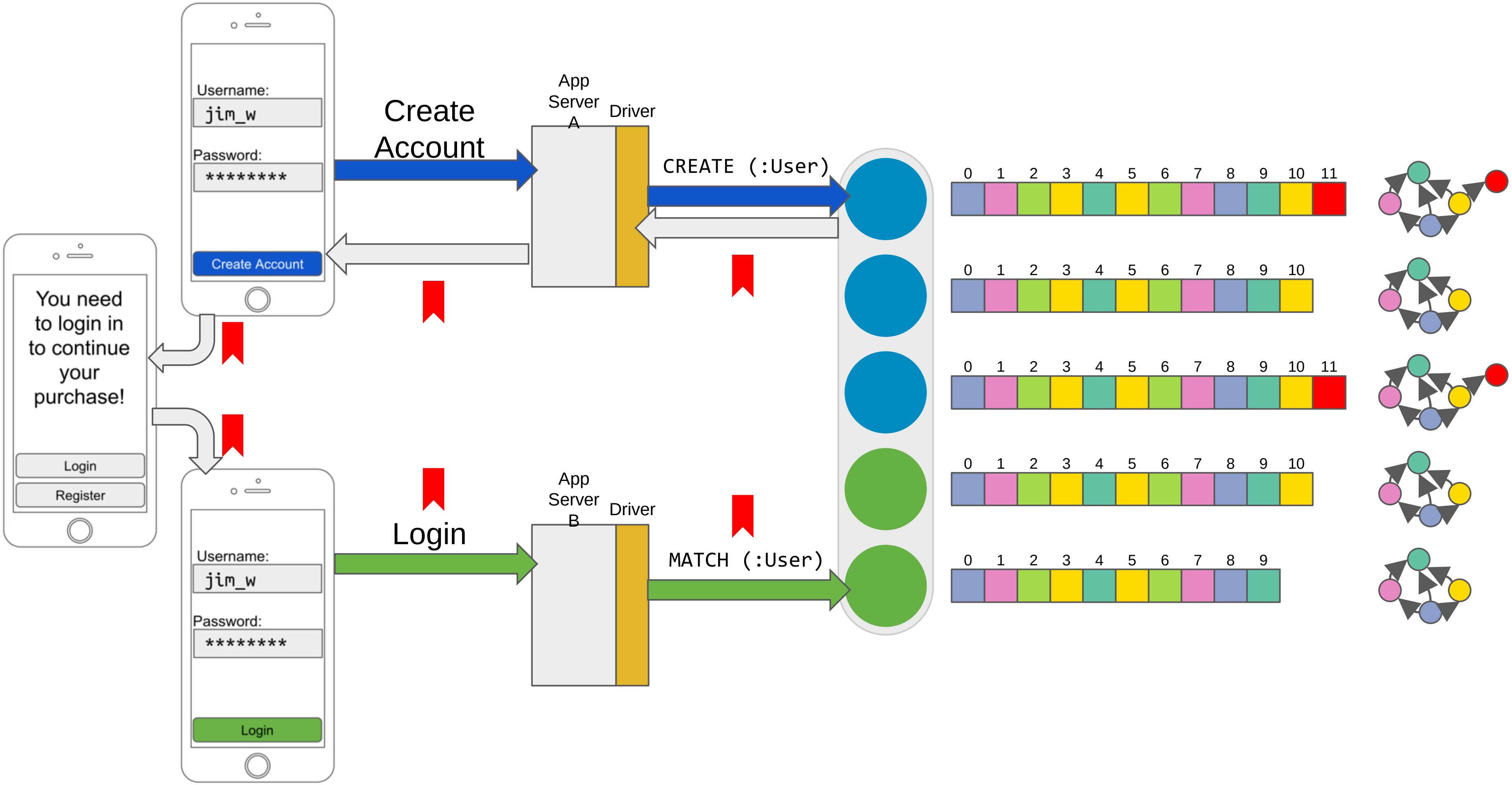
Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver
CREATE (:User)
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9



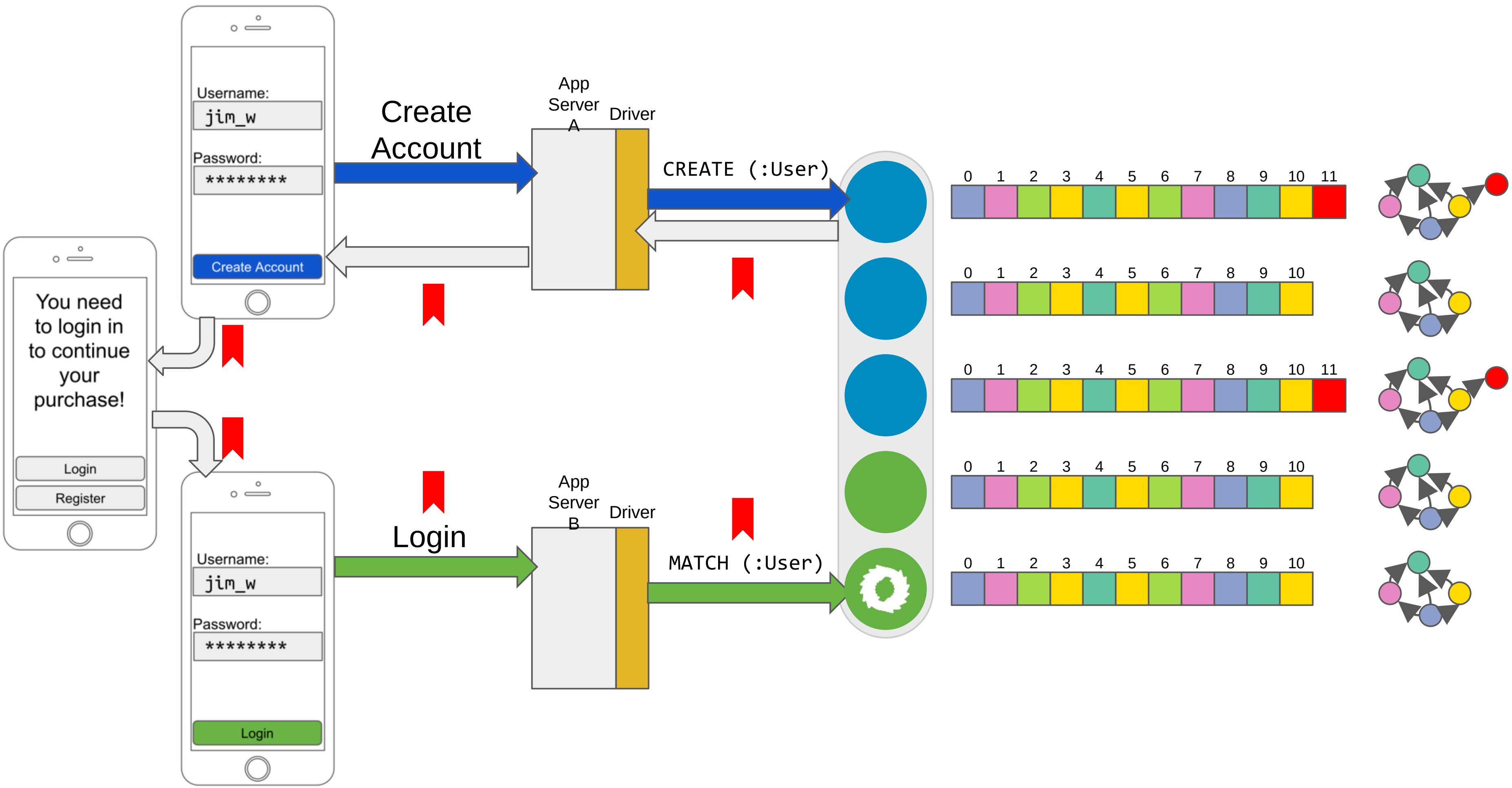
Username:
jim_w
Password:
****
Create Account
Create
Account
App
Server
A
Driver
CREATE (:User)
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
You need to login in to continue your purchase!
Login
Register
Username:
jim_w
Password:
****
Login



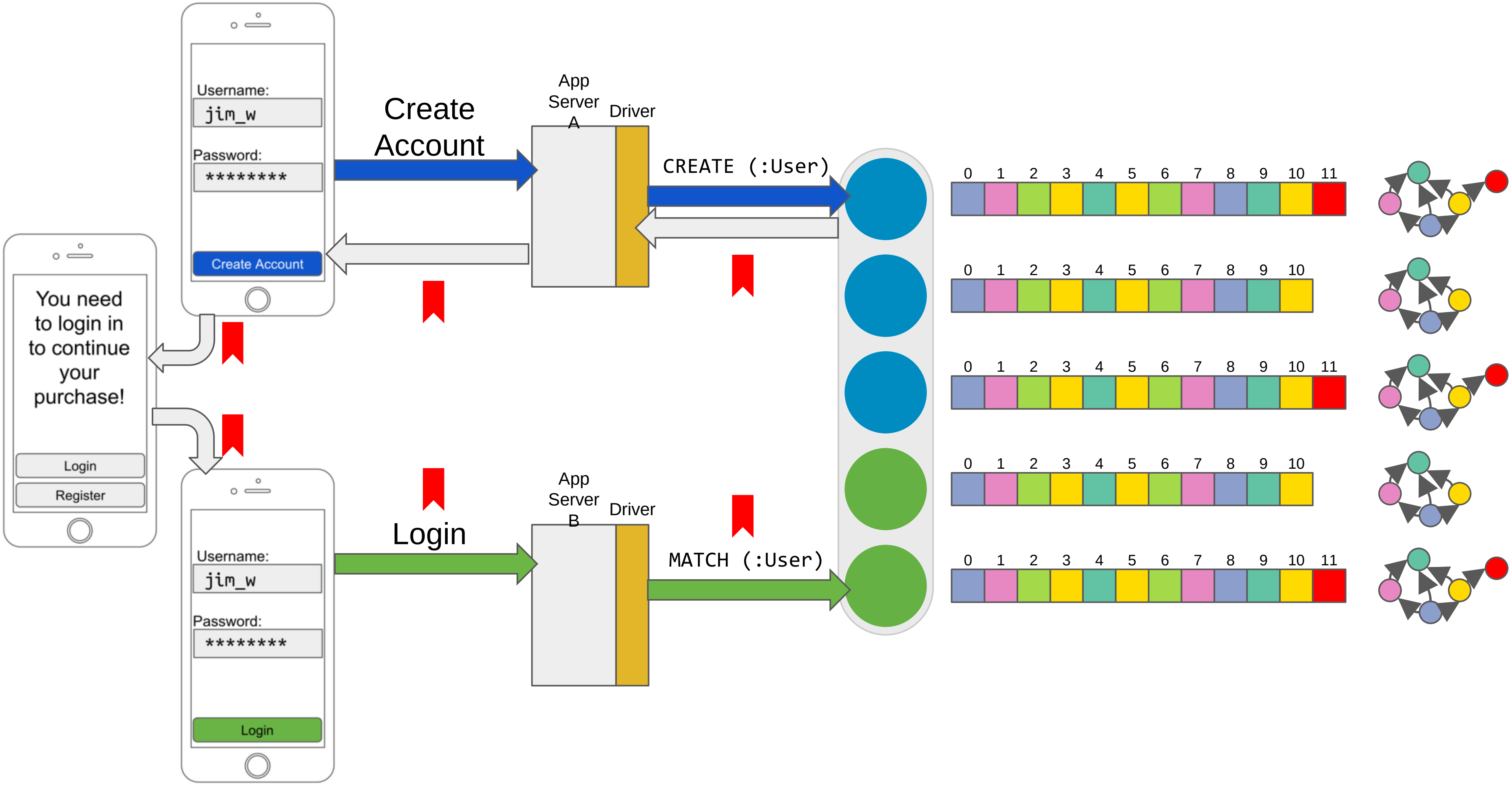
Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver
CREATE (:User)
You need to login in to continue your purchase!
Login
Register
Username:
jim_w
Password:
****
Login
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9



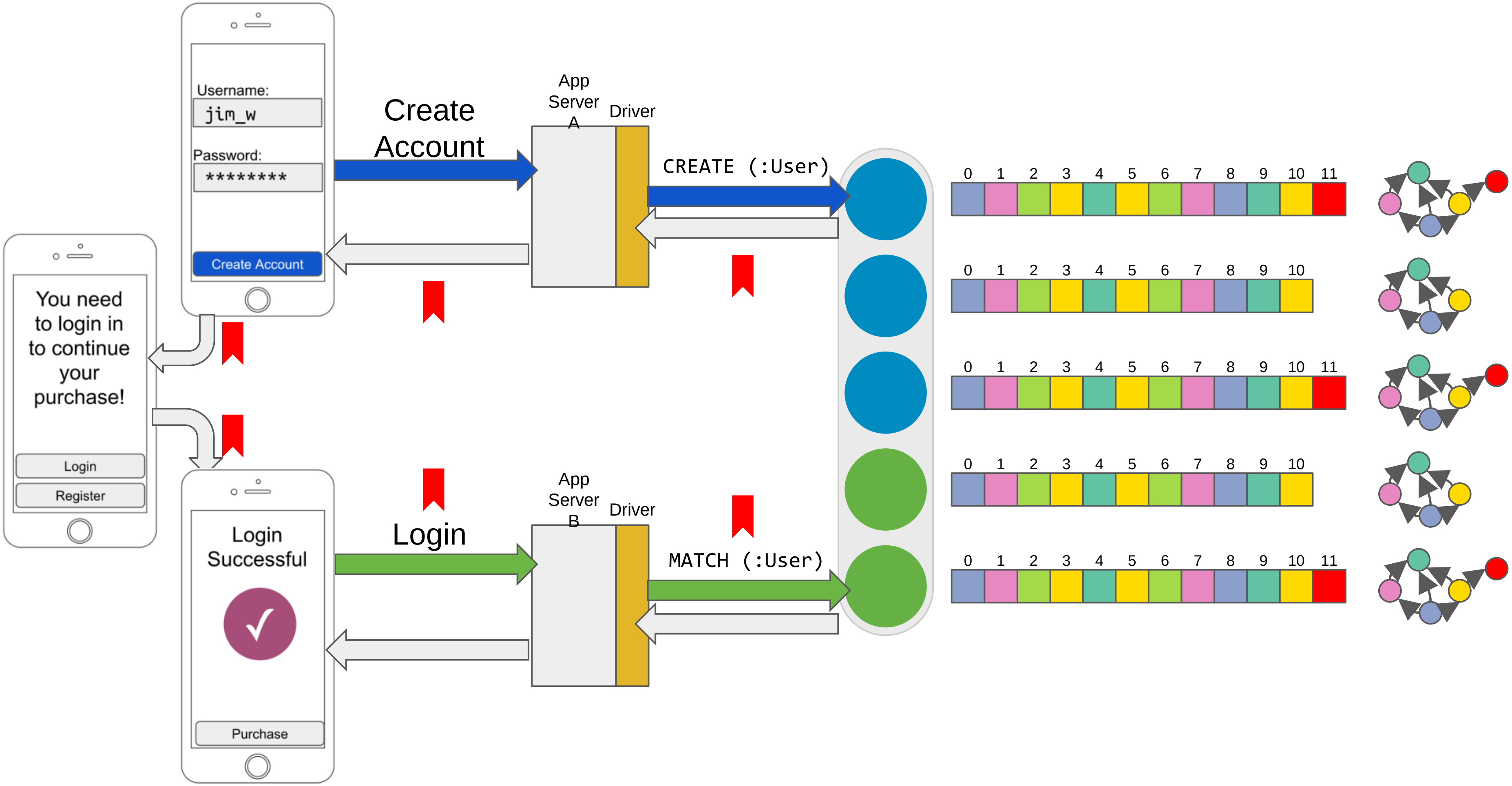
Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver
CREATE (:User)
You need to login in to continue your purchase!
Login
Register
Login
App Server B
Driver
MATCH (:User)
Username:
jim_w
Password:
****
Login
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9



Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver
CREATE (:User)
You need to login in to continue your purchase!
Login
Register
Login
App Server B
Driver
MATCH (:User)
Username:
jim_w
Password:
****
Login
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10



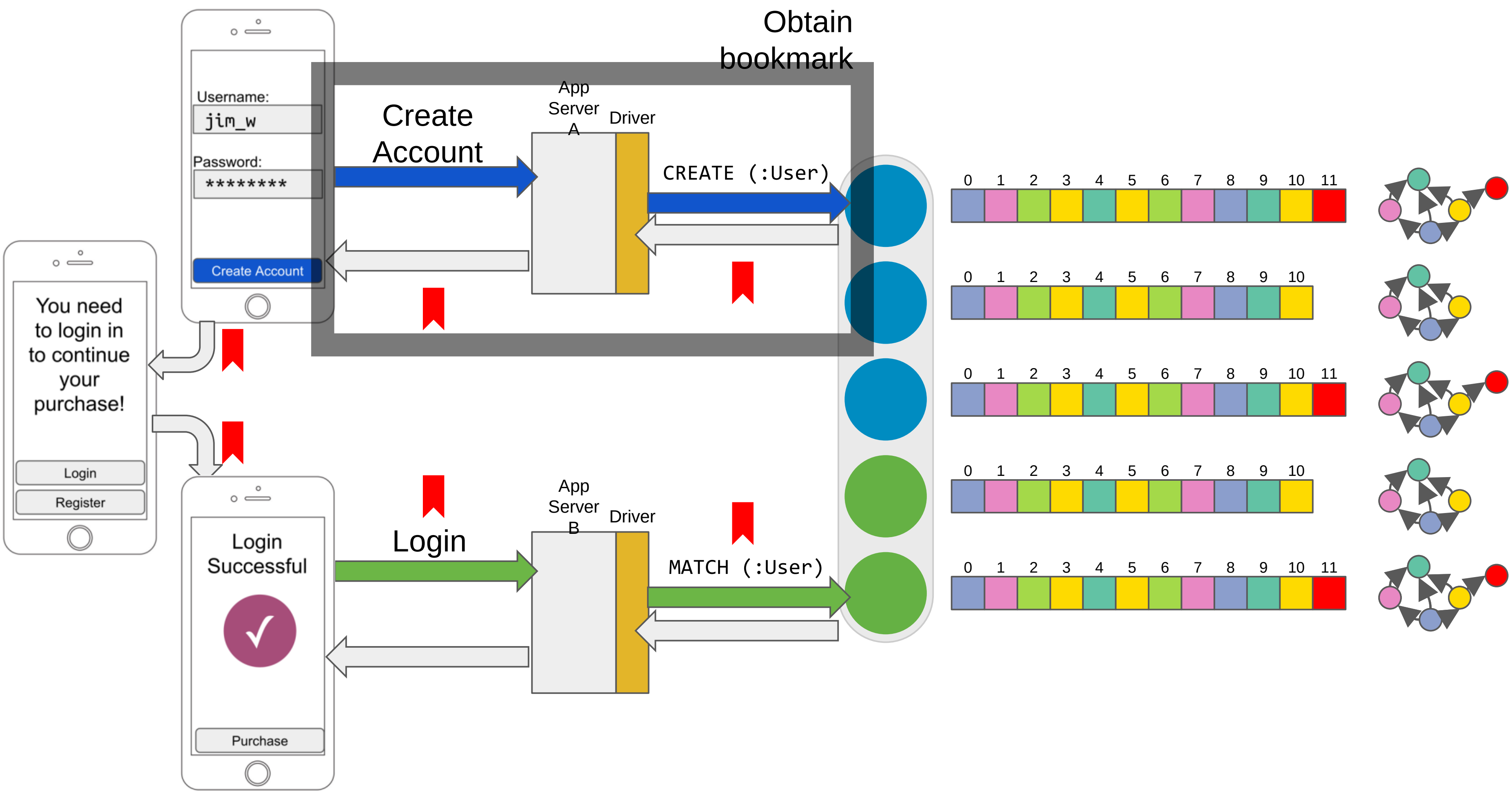
Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver
CREATE (:User)
You need to login in to continue your purchase!
Login
Register
Username:
jim_w
Password:
****
Login
Login
App Server B
Driver
MATCH (:User)
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10 11



Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver
CREATE (:User)
You need to login in to continue your purchase!
Login
Register
Login Successful
Purchase
Login
App Server B
Driver
MATCH (:User)
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10 11

Obtain bookmark

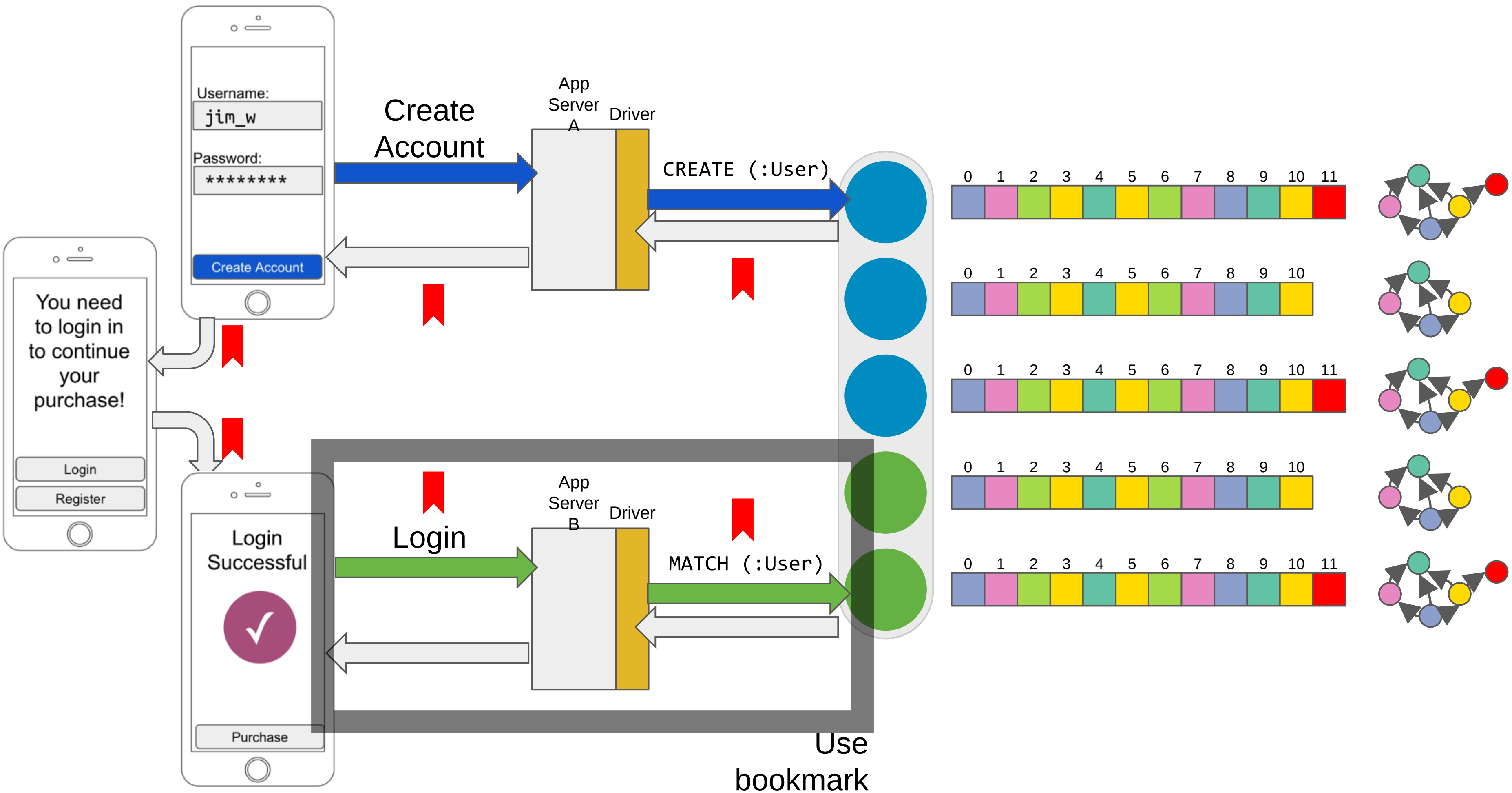
```
try ( Session session = driver.session( AccessMode.WRITE ) )
{
    try ( Transaction tx = session.beginTransaction() )
    {
        tx.run( "CREATE (user:User {userId: {userId}, passwordHash:
{passwordHash})",
                parameters( "userId", userId, "passwordHash", passwordHash ) );

        tx.success();
    }

    String bookmark = session.lastBookmark();
}
```



Obtain bookmark
Username:
jim_w
Password:
****
Create Account
Create Account
App Server A
Driver
CREATE (:User)
You need to login in to continue your purchase!
Login
Register
Login Successful
Purchase
Login
App Server B
Driver
MATCH (:User)
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10 11

Use a bookmark

```
try ( Session session = driver.session( AccessMode.READ ) )
{
    try ( Transaction tx = session.beginTransaction( bookmark ) )
    {
        tx.run( "MATCH (user:User {userId: {userId}}) RETURN *",
                parameters( "userId", userId ) );

        tx.success();
    }
}
```


Username:
jim_w
Password:
****
Create Account
You need to login in to continue your purchase!
Login
Register
Login Successful
Purchase
Create Account
Login
App Server A
Driver
App Server B
Driver
CREATE (:User)
MATCH (:User)
Use bookmark
0 1 2 3 4 5 6 7 8 9 10 11

Summary

- Strong consistency commit protocols
  - 2PC/3PC
- Consensus algorithms
  - Raft/Paxos