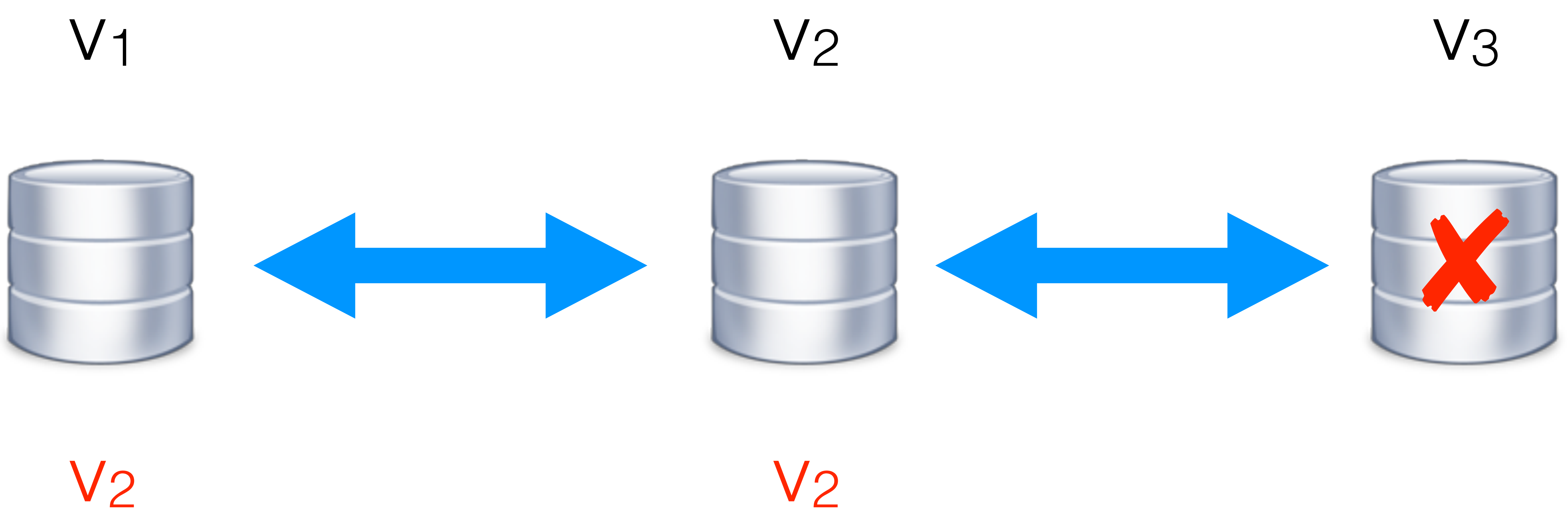
Linearizability Proofs for Distributed Consensus Protocols

Consensus

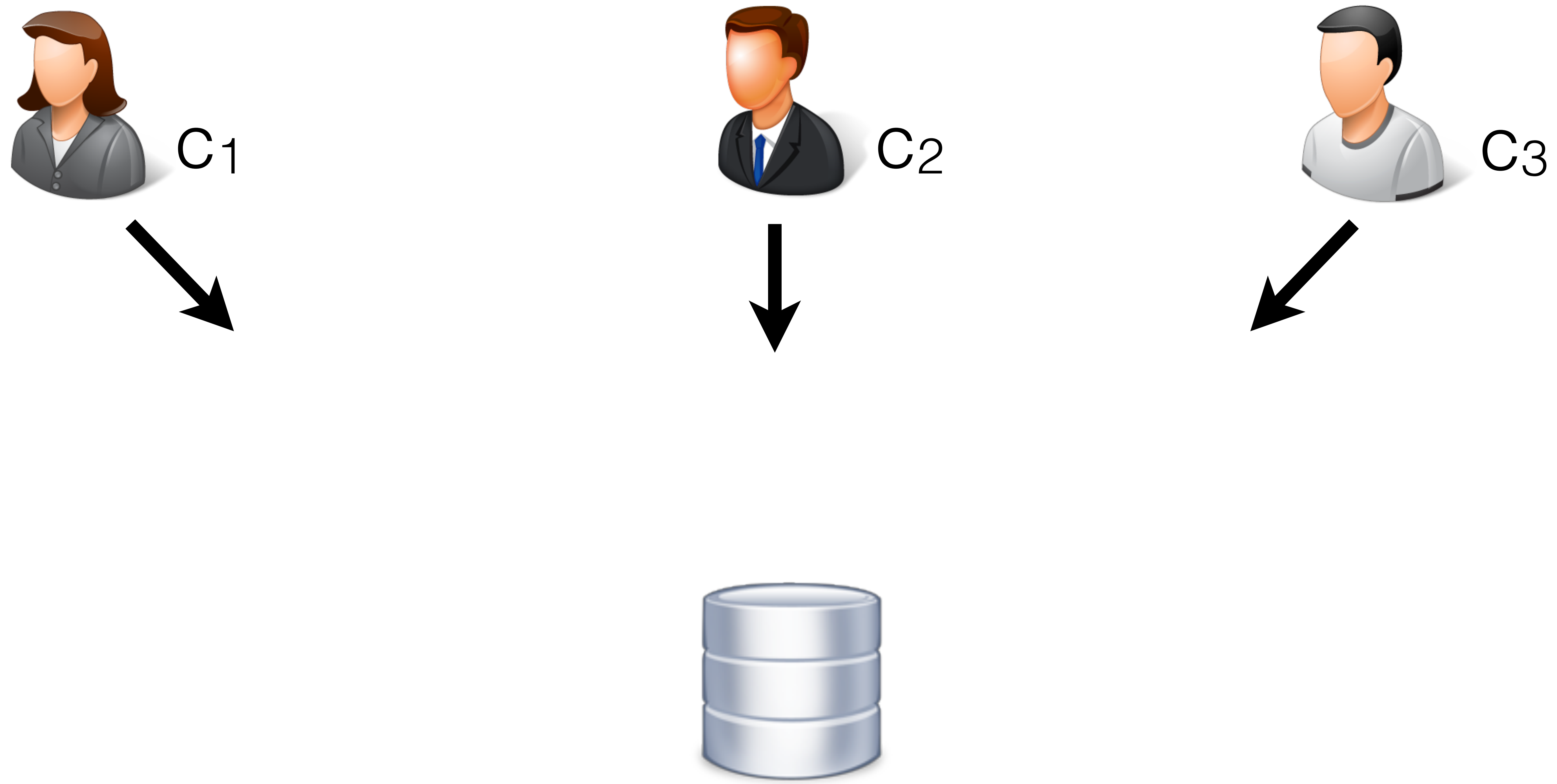
- Several nodes, which can crash
- Each proposes a value

Consensus



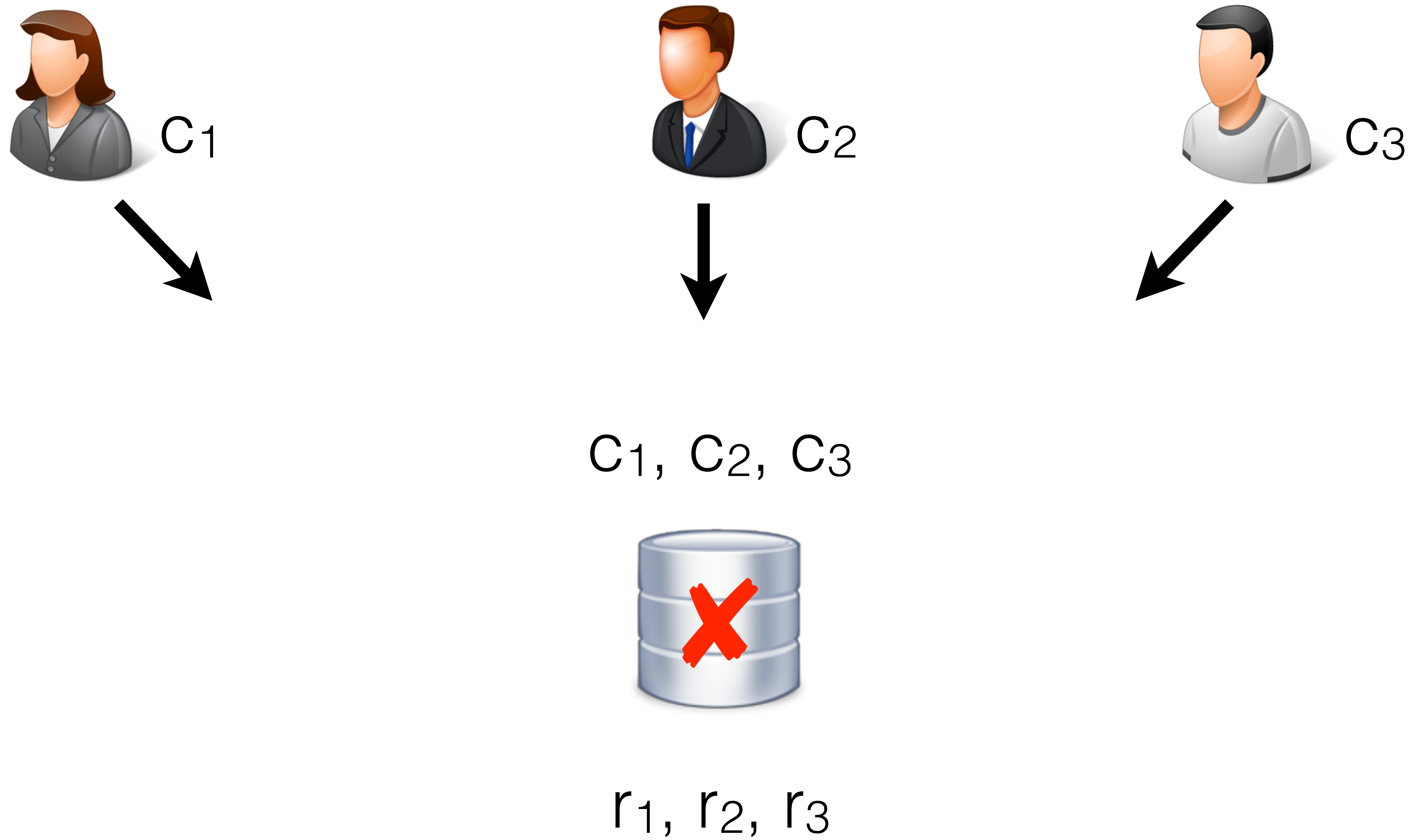
- Several nodes, which can crash
- Each proposes a value
- All non-crashed nodes agree on a single value

Deterministic state



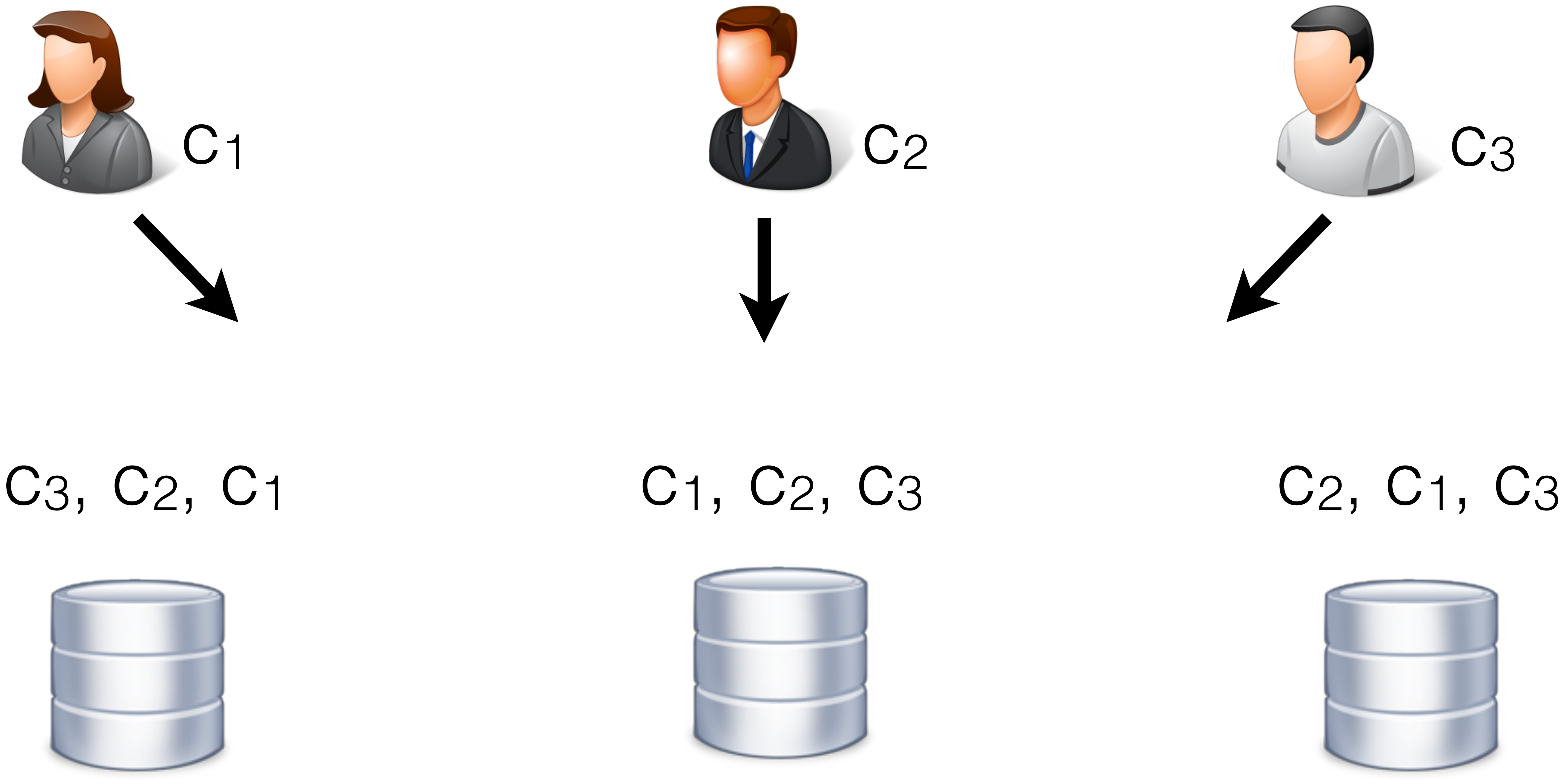
Clients submit commands

Deterministic state



Machine totally orders commands and computes the sequence of results

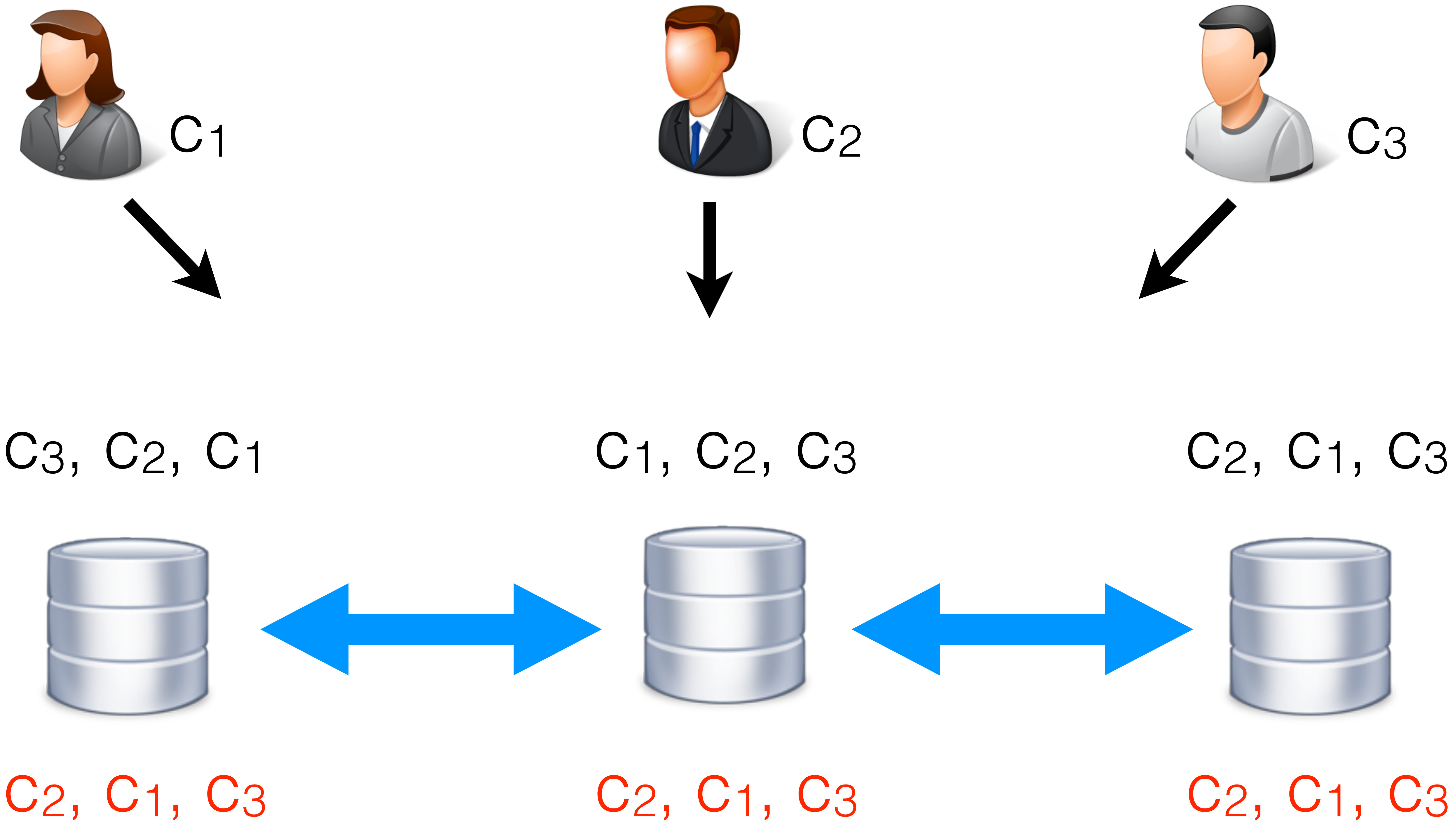
State machine



Clients send commands to all replicas

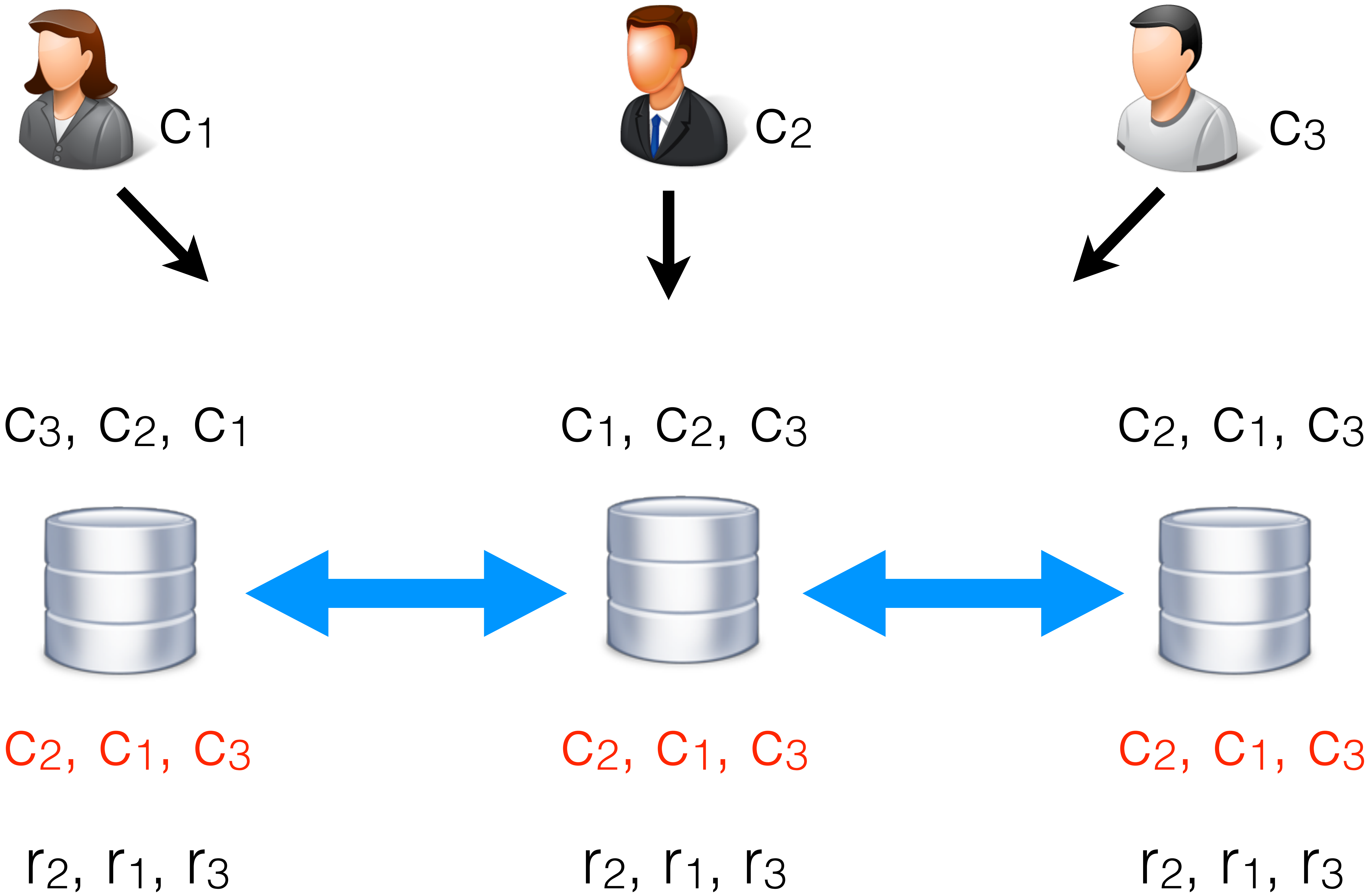
Replicas may receive commands in different orders

State machine



Order commands via a sequence of consensus instances

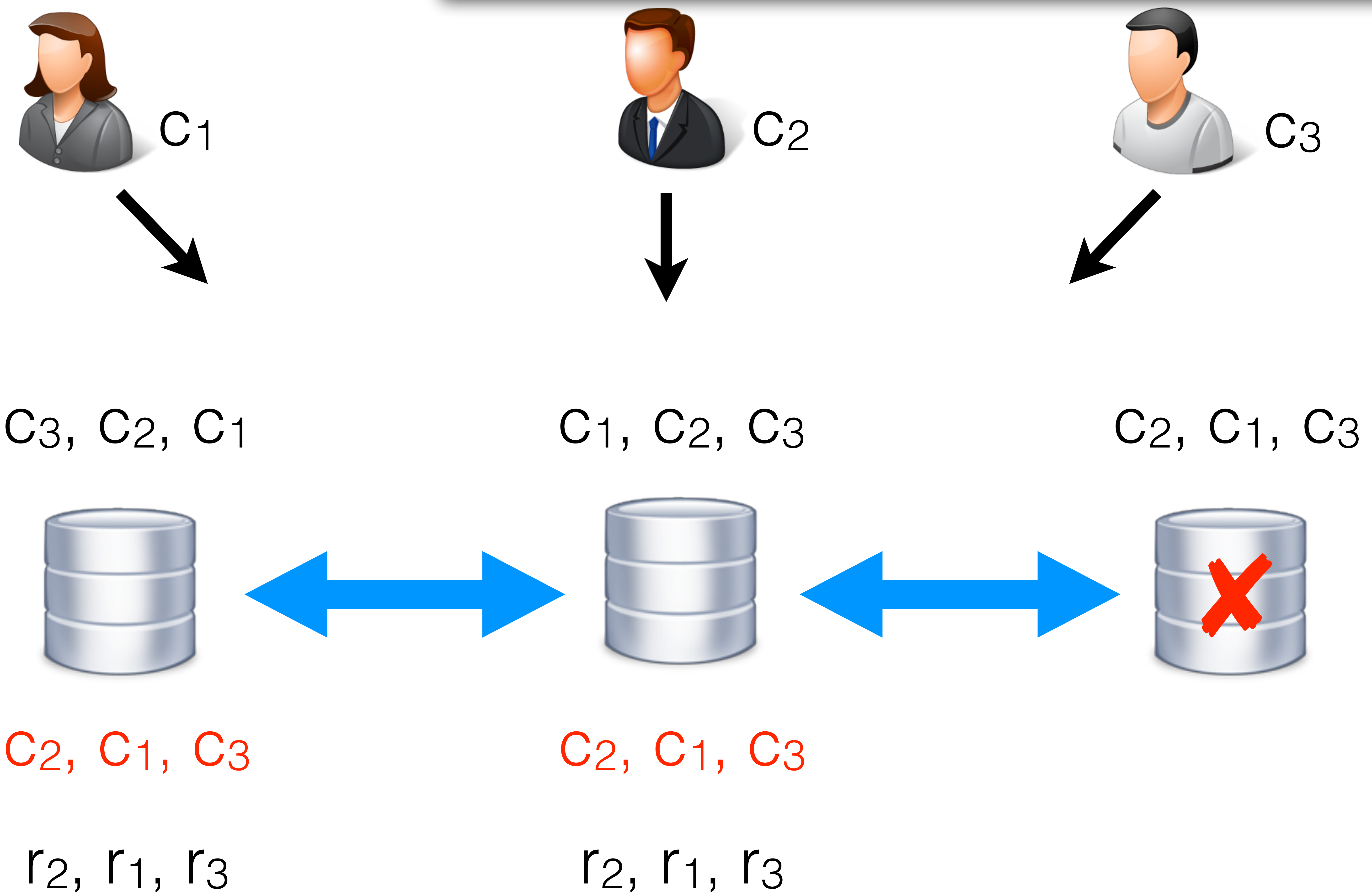
State machine



Replicas compute *the same* sequence of results

Stat

Correctness: *replicated* implementation is linearizable wrt. *single-server* one: replication transparent to clients



Replicas compute the same sequence of results

The zoo of cor

Complex protocols: constant fight for better performance

- Viewstamped replication (1988)
- Paxos (1998)
- Disk Paxos (2003)
- Cheap Paxos (2004)
- Generalized Paxos (2004)
- Paxos Commit (2004)
- Fast Paxos (2006)
- Stoppable Paxos (2008)
- Mencius (2008)
- Vertical Paxos (2009)
- ZAB (2009)
- Ring Paxos (2010)
- Egalitarian Paxos (2013)
- Raft (2014)
- M2Paxos (2016)
- Flexible Paxos (2016)
- Caesar (2017)

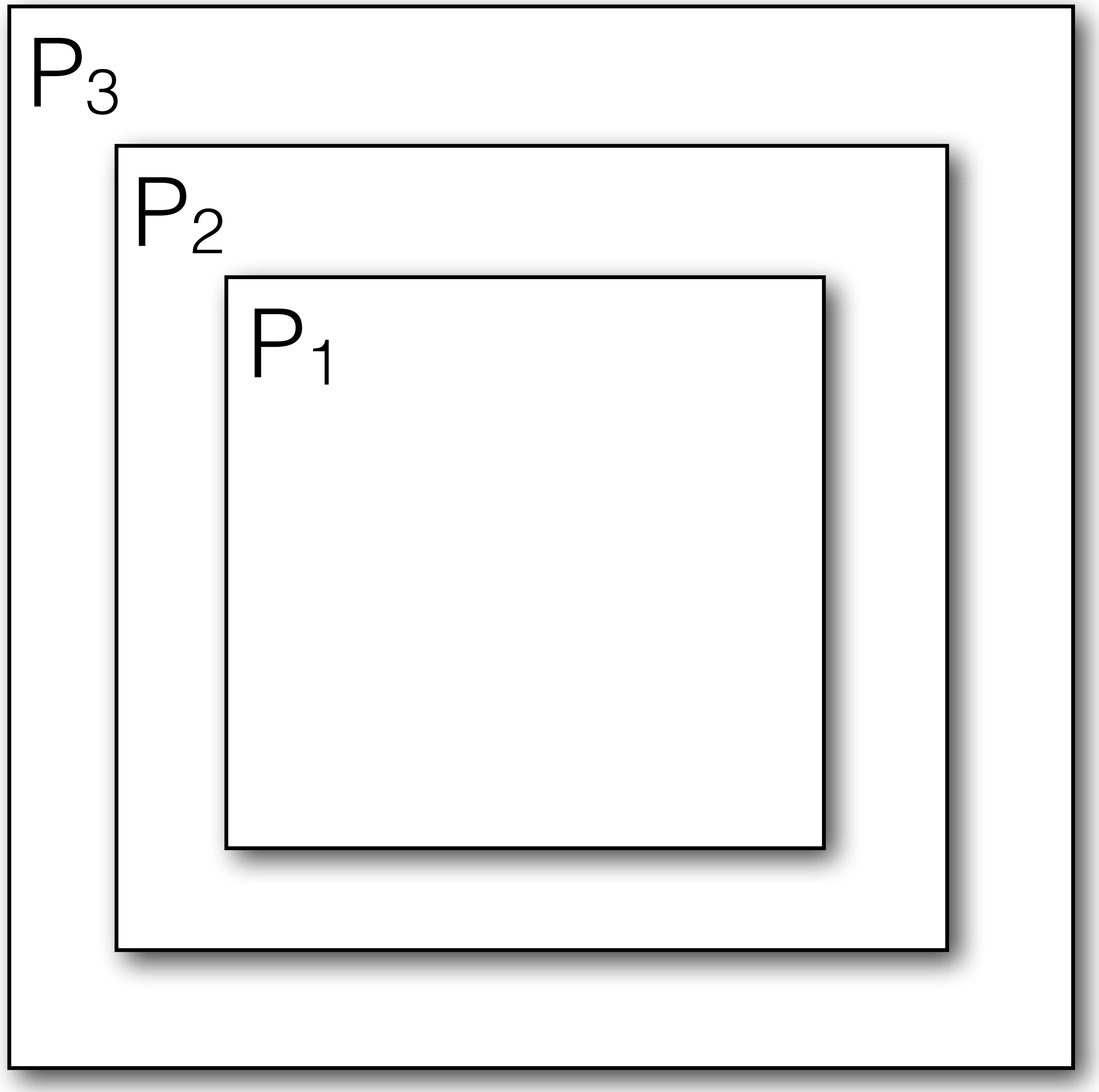
O

- Viewstamped replication (1988)
- Paxos (1998)
- Disk Paxos (2003)
- Cheap Paxos (2004)
- Generalized Paxos (2004)
- Paxos Commit (2004)
- Fast Paxos (2006)
- Stoppable Paxos (2008)
- Mencius (2008)
- Vertical Paxos (2009)
- ZAB (2009)
- Ring Paxos (2010)
- Egalitarian Paxos (2013)
- Raft (2014)
- M2Paxos (2016)
- Flexible Paxos (2016)
- Caesar (2017)

- Develop metho
  including realist
- Get insights into their structure;
- Design new and better protocols?

Approach

- Modular reasoning: verify parts of the protocol separately instead of the whole thing
- Linearizability implies refinement [Filipovic+ 2009]

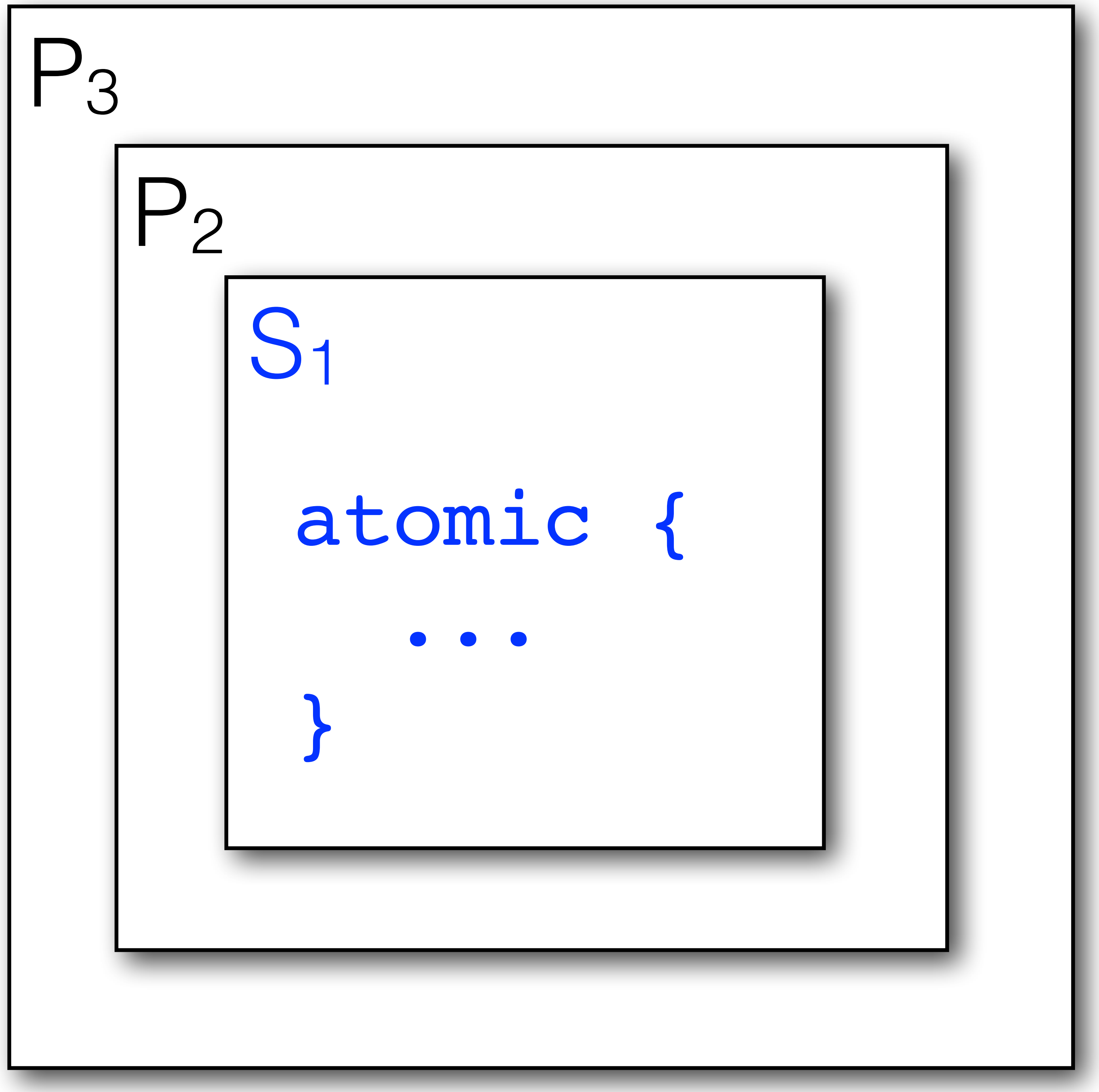


$P_1 \sqsubseteq S_1$

Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, Hongseok Yang: **Abstraction for Concurrent Objects.** ESOP 2009

Approach

- Modular reasoning: verify parts of the protocol separately instead of the whole thing
- Linearizability implies refinement [Filipovic+ 2009]

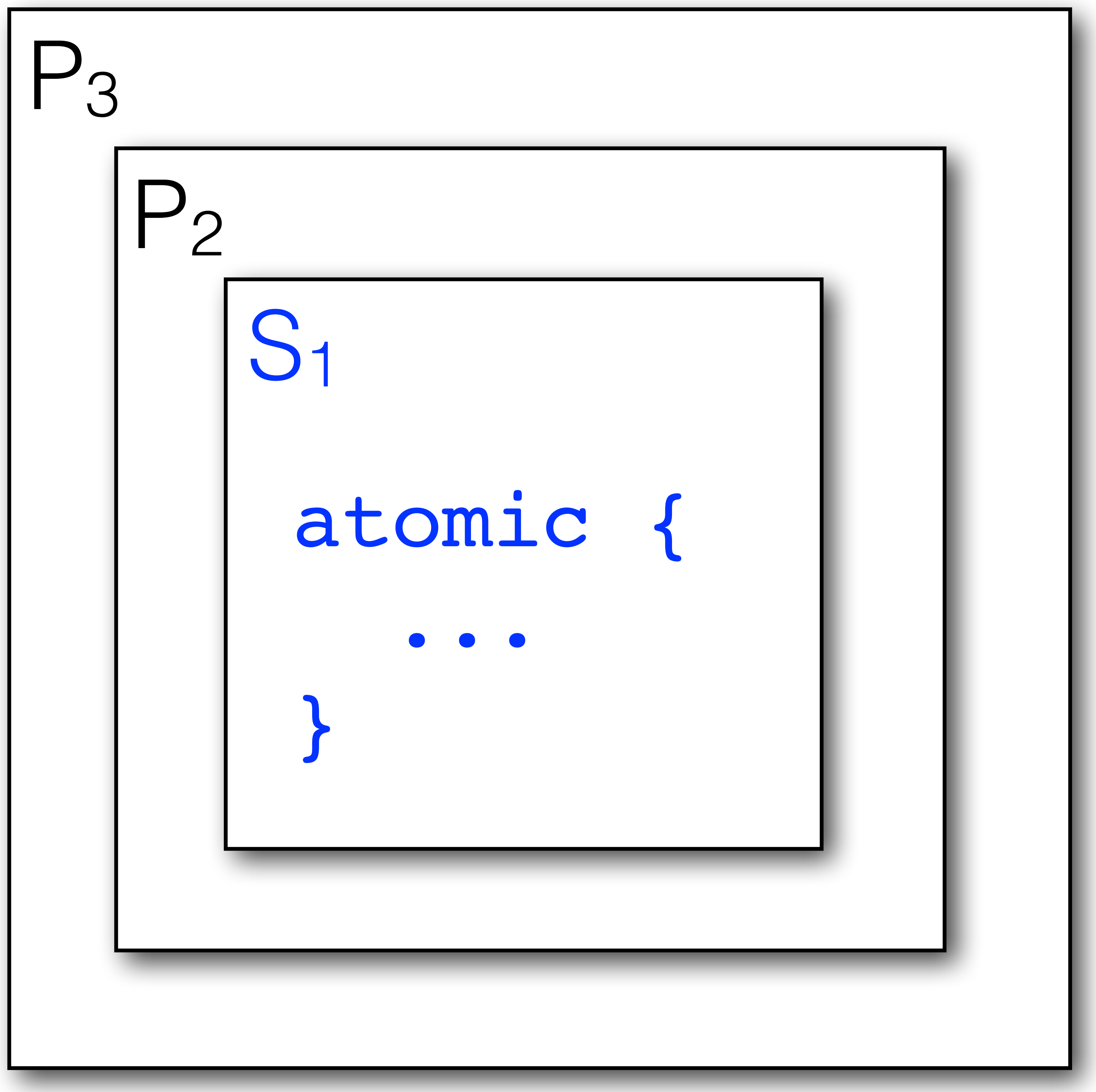


$P_1 \sqsubseteq S_1$

Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, Hongseok Yang: **Abstraction for Concurrent Objects.** ESOP 2009

Approach

- Modular reasoning: verify parts of the protocol separately instead of the whole thing
- Linearizability implies refinement [Filipovic+ 2009]



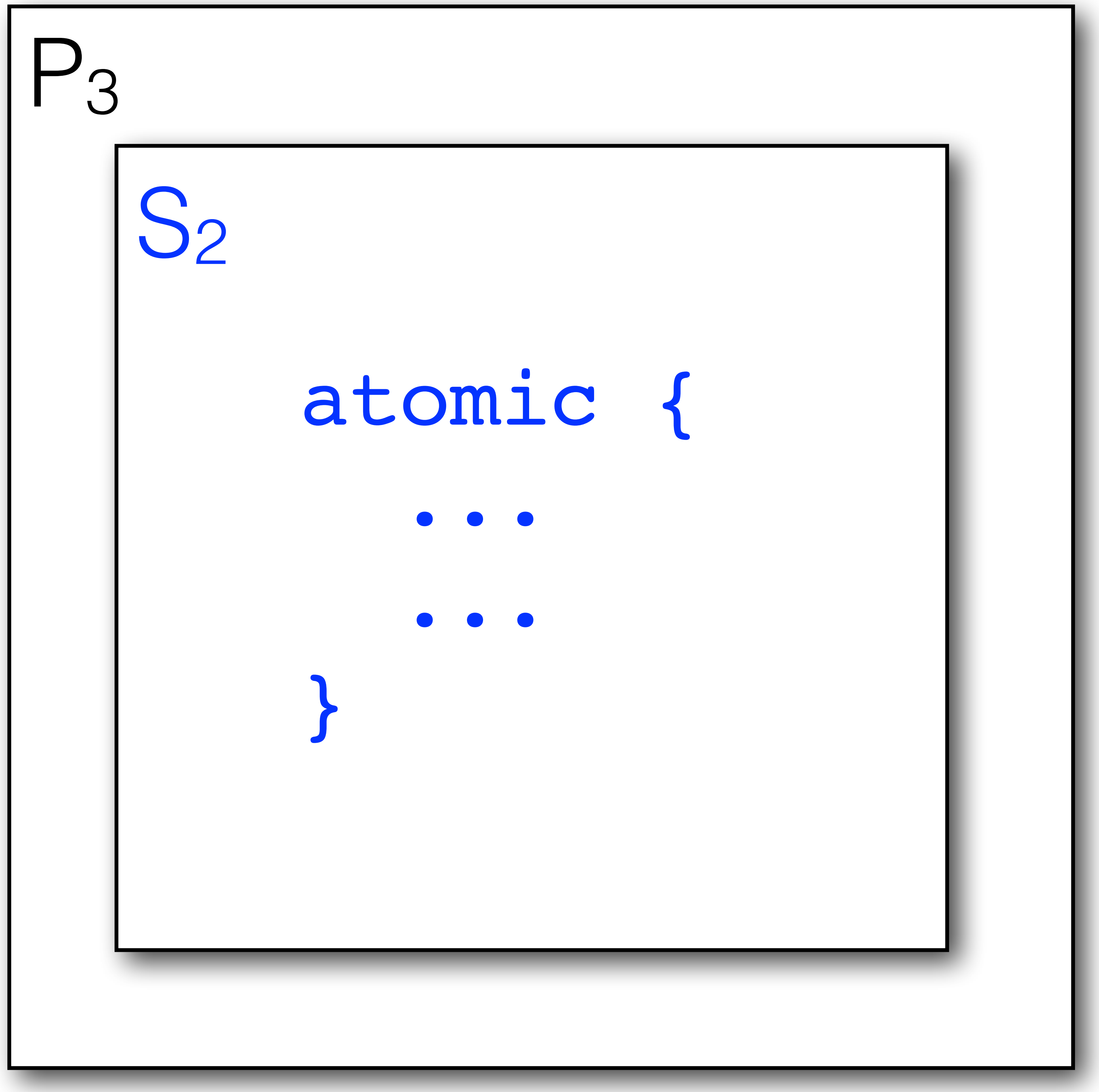
$P_1 \sqsubseteq S_1$

$P_2(S_1) \sqsubseteq S_2$

Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, Hongseok Yang: **Abstraction for Concurrent Objects.** ESOP 2009

Approach

- Modular reasoning: verify parts of the protocol separately instead of the whole thing
- Linearizability implies refinement [Filipovic+ 2009]



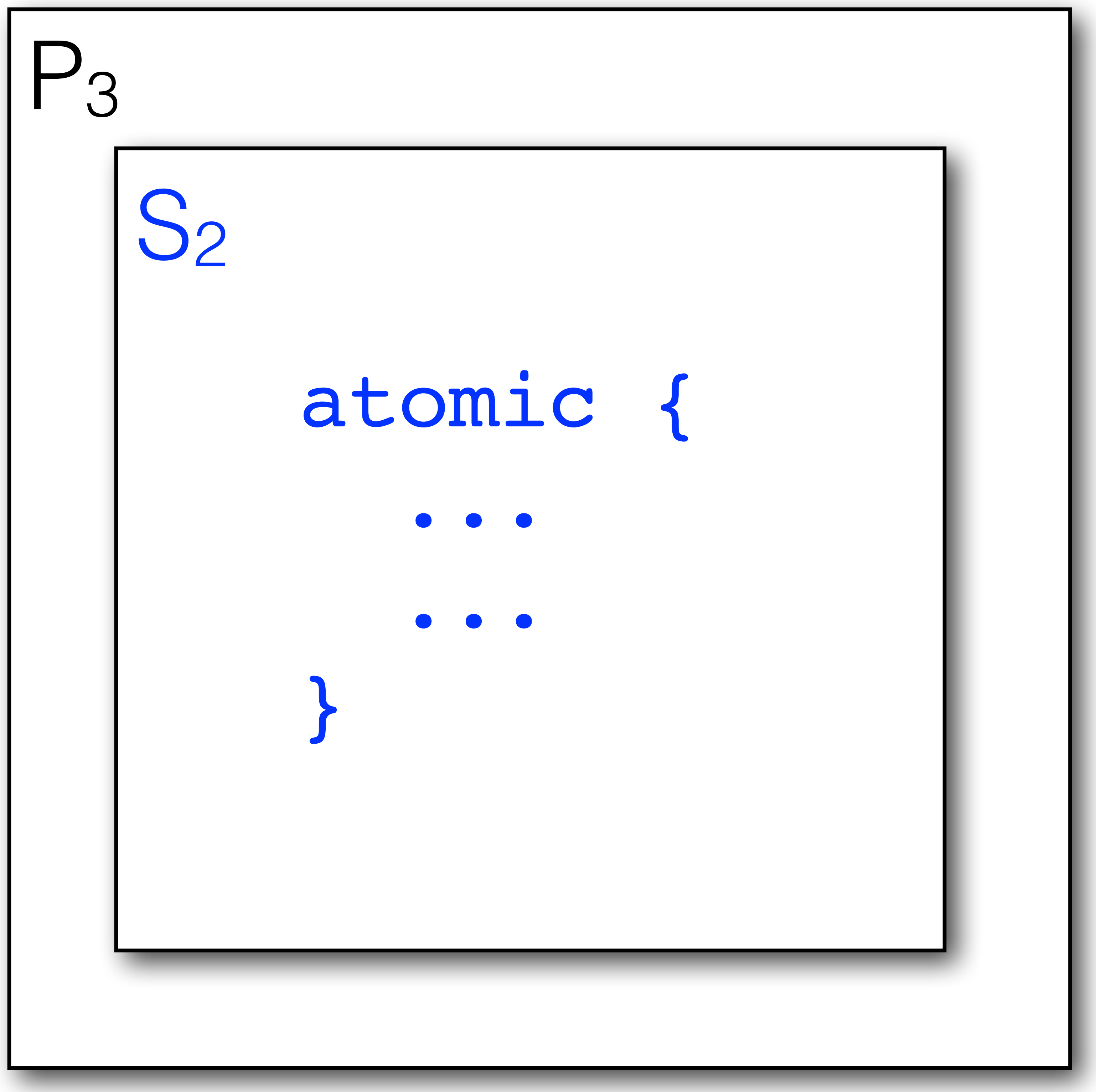
$P_1 \sqsubseteq S_1$

$P_2(S_1) \sqsubseteq S_2$

Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, Hongseok Yang: **Abstraction for Concurrent Objects.** ESOP 2009

Approach

- Modular reasoning: verify parts of the protocol separately instead of the whole thing
- Linearizability implies refinement [Filipovic+ 2009]



$P_1 \sqsubseteq S_1$

$P_2(S_1) \sqsubseteq S_2$

$P_3(S_2) \sqsubseteq S_3$

Approach

- Modular reasoning: verify parts of the protocol separately instead of the whole thing
- Linearizability implies refinement [Filipovic+ 2009]

S_3

```
atomic {
   ...
   ...
   ...
}
```

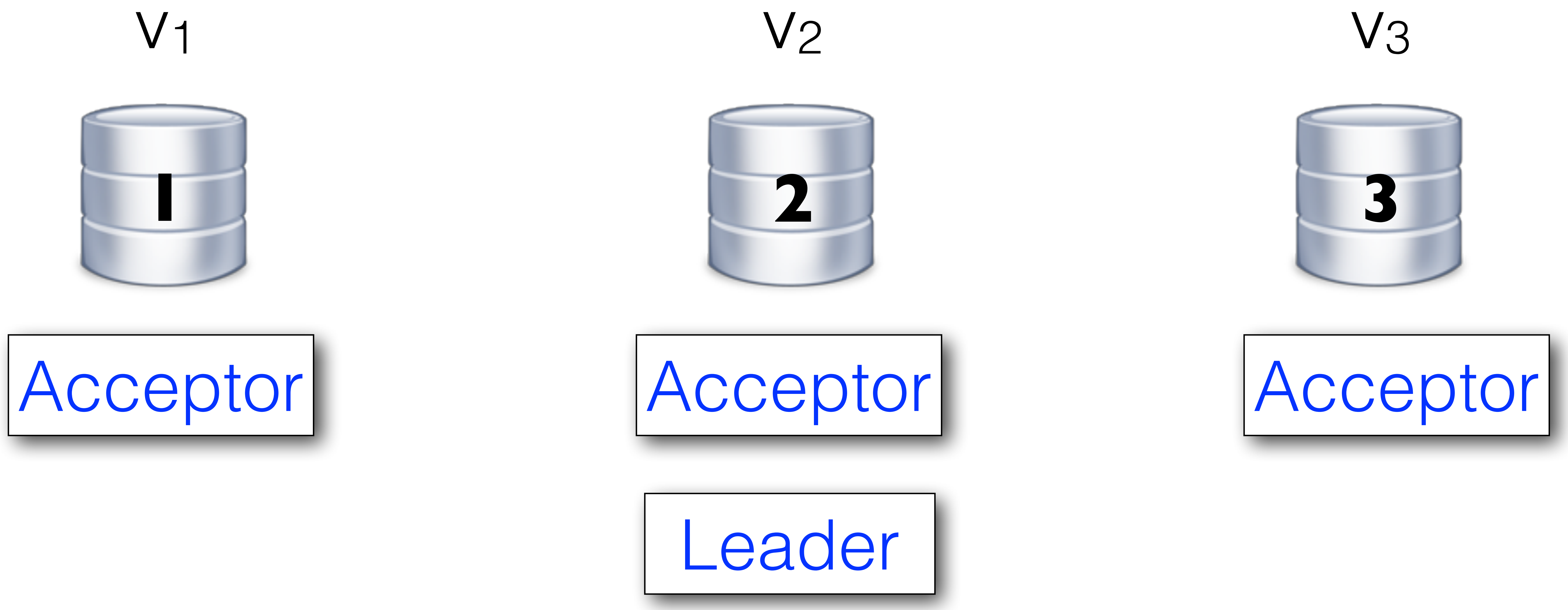
$P_1 \sqsubseteq S_1$

$P_2(S_1) \sqsubseteq S_2$

$P_3(S_2) \sqsubseteq S_3$

Layered structure in consensus

- Steal abstractions from an existing analysis of Paxos [Boichat+ 2003, Chockler+ 2002]
- Show their *linearizability* ⇒ modular proof of Paxos
- Generalise them to modularise proofs of other Paxos versions and consensus protocols (e.g., ZAB and Raft)



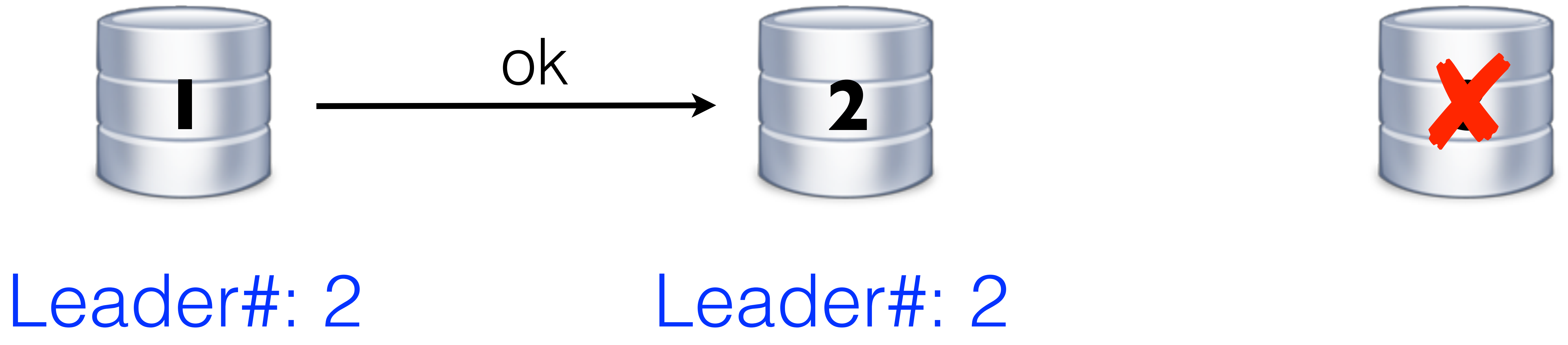
- Acceptors = members of parliament: can vote to accept a value, majority wins;
- Leader = parliament speaker: proposes its value to vote on
- Good for multi-consensus: can elect the leader once and get it to process multiple client requests

Leader **?**

- Phase 1: a prospective leader convinces a majority of acceptors to accept its authority

Leader#: 2

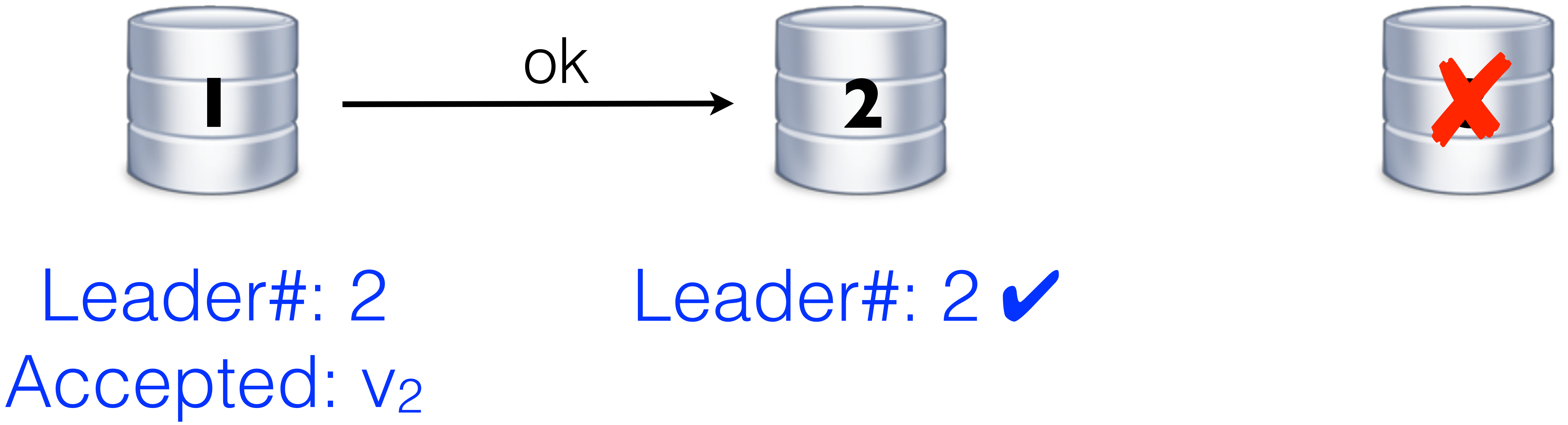
- Phase 1: a prospective leader convinces a majority of acceptors to accept its authority



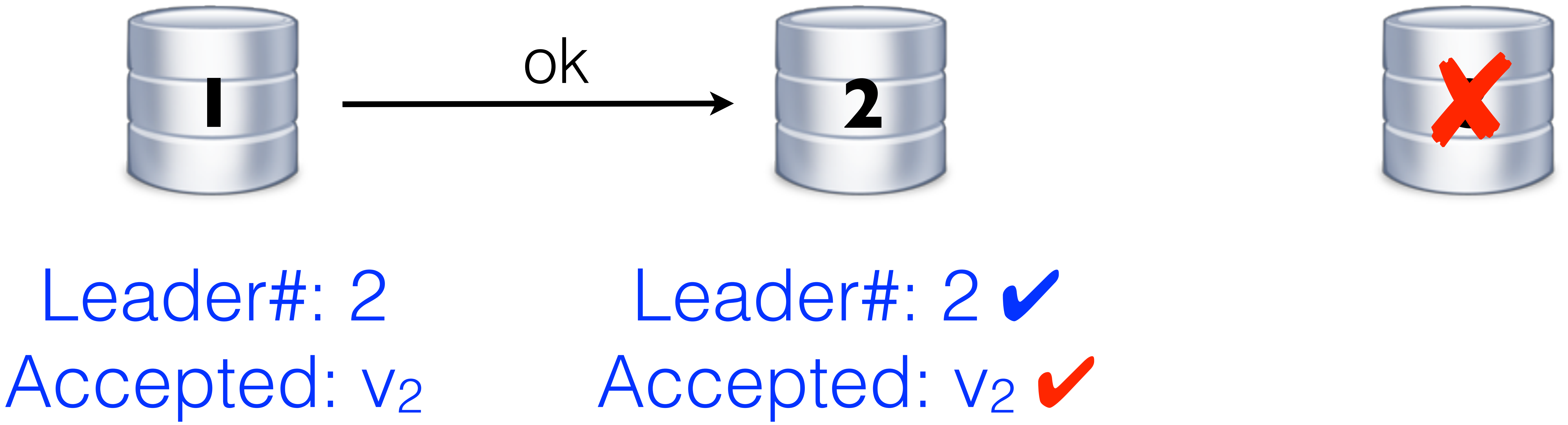
- Phase 1: a prospective leader convinces a majority of acceptors to accept its authority

- Phase 1: a prospective leader convinces a majority of acceptors to accept its authority

- Phase 1: a prospective leader convinces a majority of acceptors to accept its authority
- Phase 2: the leader gets a majority of acceptors to accept its value and replies to the client



- Phase 1: a prospective leader convinces a majority of acceptors to accept its authority
- Phase 2: the leader gets a majority of acceptors to accept its value and replies to the client



- Phase 1: a prospective leader convinces a majority of acceptors to accept its authority
- Phase 2: the leader gets a majority of acceptors to accept its value and replies to the client

Leader#: 2
Accepted: v_2

Leader#: 2 ✔
Accepted: v_2 ✔
Reply v_2 to client

- Phase 1: a prospective leader convinces a majority of acceptors to accept its authority
- Phase 2: the leader gets a majority of acceptors to accept its value and replies to the client

Leader#: 2
Accepted: v_2

Leader#: 2 ✔
Accepted: v_2 ✔
Reply v_2 to client

- Phase 1: a prospective leader convinces a majority of acceptors to accept its authority
- Phase 2: the leader gets a majority of acceptors to accept its value and replies to the client

Leader#: 3
Accepted: v_3

Leader#: 2 ✔
Accepted: v_2 ✔
Reply v_2 to client

Leader#: 3 ✔
Accepted: v_3 ✔
Reply v_3 to client

- Problem: node 3 may wake up,
  form *a majority of 1 and 3*, and accept value $v_{3;}$
- Need to ensure once a value is chosen by a majority, *it can't be changed*;
- Use *round numbers* to distinguish different votes.

Server 1	Server 2	Server 3
Leader#: ?	Leader#: ?	Leader#: ?
Round#: 0	Round#: 0	Round#: 0
Accepted: ?	Accepted: ?	Accepted: ?

- Phase 1: a prospective leader choses a unique round **r** and convinces a majority of acceptors to switch to **r**
- Acceptor switches only if it's current round *is less*

Leader#: ? | Leader#: 2 | Leader#: ?
Round#: 0 | Round#: r | Round#: 0
Accepted: ? | Accepted: ? | Accepted: ?

- Phase 1: a prospective leader choses a unique round **r** and convinces a majority of acceptors to switch to **r**
- Acceptor switches only if it’s current round *is less*

Server 1	Server 2	Server 3
Leader#: 2	Leader#: 2 ✔	Leader#: ?
Round#: r	Round#: r	Round#: 0
Accepted: ?	Accepted: ?	Accepted: ?

- Phase 1: a prospective leader choses a unique round **r** and convinces a majority of acceptors to switch to **r**
- Acceptor switches only if it's current round *is less*

Server 1	Server 2	Server 3
Leader#: 2	Leader#: 2 ✔	Leader#: ?
Round#: r	Round#: r	Round#: 0
Accepted: ?	Accepted: v_2	Accepted: ?

- Phase 2: the leader sends its value tagged with the round number;

- Acceptor only accepts a value *tagged* with the round it has agreed for before.

Leader#: 2
Round#: r
Accepted: v_2

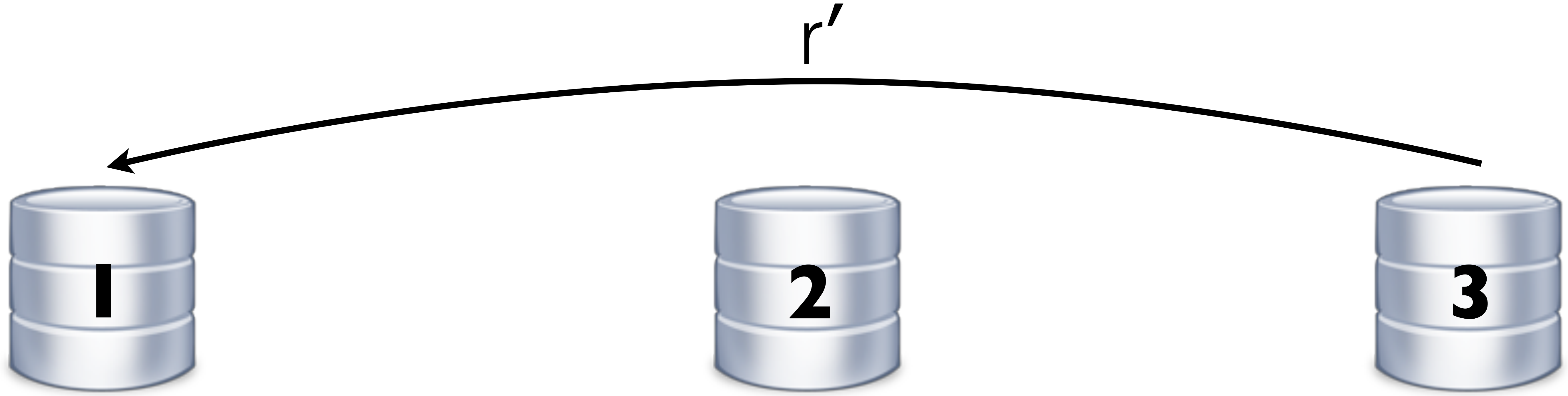
Leader#: 2 ✔
Round#: r
Accepted: v_2

Leader#: ?
Round#: 0
Accepted: ?

- Phase 2: the leader sends its value tagged with the round number;
- Acceptor only accepts a value *tagged* with the round it has agreed for before.

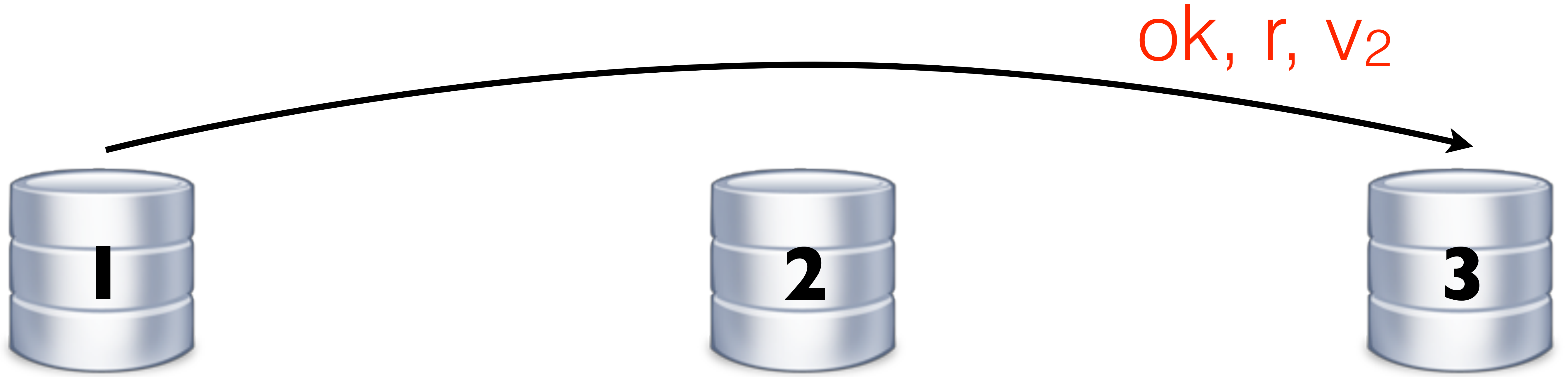
Leader#: 2	Leader#: 2 ✔	Leader#: ?
Round#: r	Round#: r	Round#: 0
Accepted: v_2	Accepted: v_2 ✔	Accepted: ?
	Reply v_2 to client	

- Phase 2: the leader sends its value tagged with the round number;
- Acceptor only accepts a value *tagged* with the round it has agreed for before.



Server 1	Server 2	Server 3
Leader#: 2	Leader#: 2 ✔	Leader#: 3
Round#: r	Round#: r	Round#: r′
Accepted: v_2	Accepted: v_2 ✔	Accepted: ?
	Reply v_2 to client	

- Phase 1: acceptor sends to the prospective leader its round number and value;

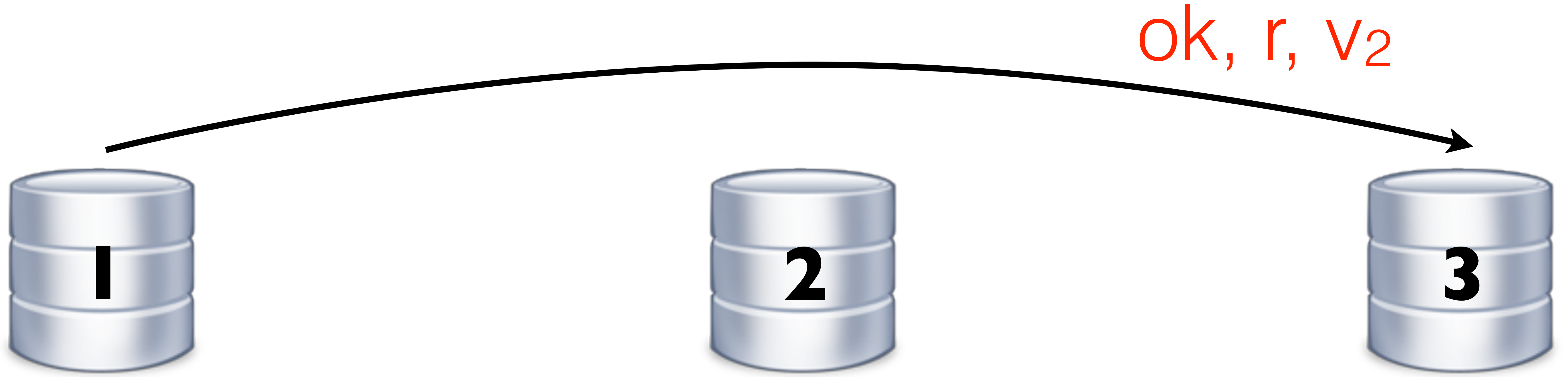


Leader#: 3
Round#: r′
Accepted: v_2

Leader#: 2 ✔
Round#: r
Accepted: v_2 ✔
Reply v_2 to client

Leader#: 3
Round#: r′
Accepted: ?

- Phase 1: acceptor sends to the prospective leader its round number and value;
- Acceptor sends to the prospective leader its round number and value

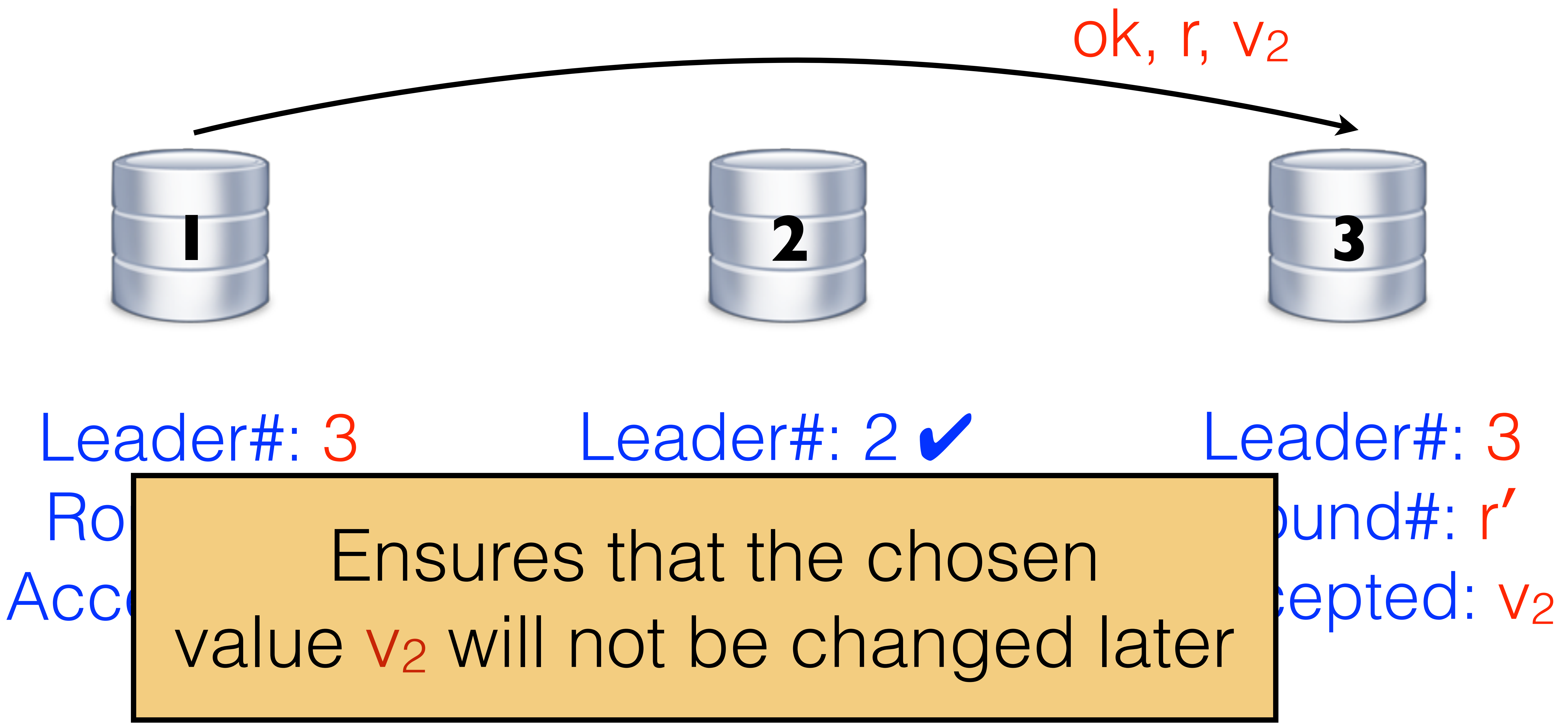


Leader#: 3
Round#: r′
Accepted: v_2

Leader#: 2 ✔
Round#: r
Accepted: v_2 ✔
Reply v_2 to client

Leader#: 3
Round#: r′
Accepted: v_2

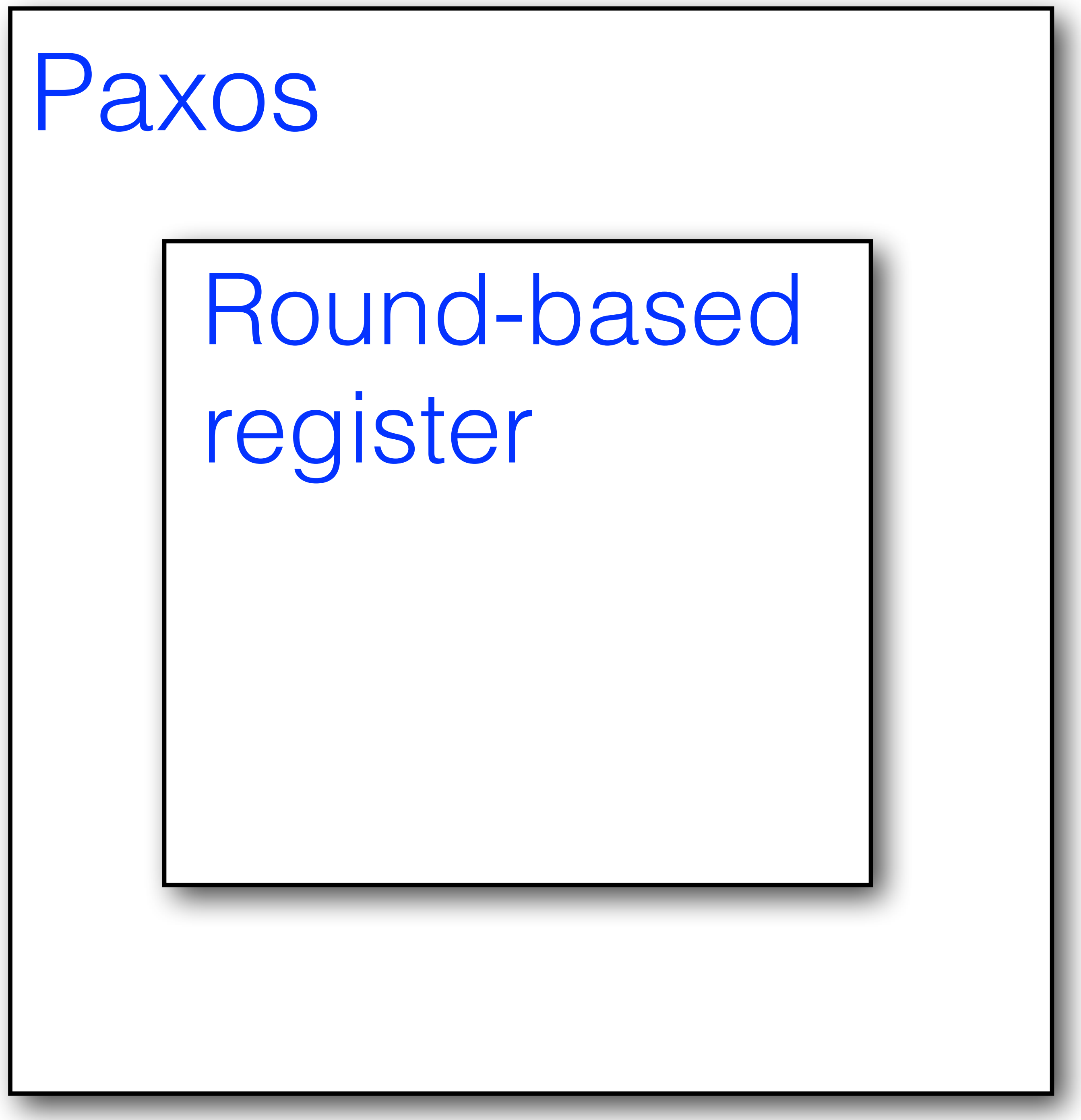
- Phase 1: acceptor sends to the prospective leader its round number and value;
- Acceptor sends to the prospective leader its round number and value;
- If some acceptor has accepted a value, the leader proposes the value with the highest round number.



- Phase 1: acceptor sends to the prospective leader its round number and value;
- Acceptor sends to the prospective leader its round number and value;
- If some acceptor has accepted a value, the leader proposes the value with the highest round number.

Round-based register

[Boichat+ 2003]



- Data type representing the "state" of acceptors as a *shared pointer*
- `read()` - Phase 1 of Paxos
- `write()` - Phase 2 of Paxos

Read - Paxos Phase 1

```
read(r) {
    if (a majority of acceptors has round < r) {
      switch them to round r
      if (no acceptor has a value accepted)
          return none
      else
        return the value at the acceptor
                  with the highest round
    } else
        return abort
}
```

Write - Paxos Phase 2

```
write(r, v) {
    if (a majority of acceptors has round r) {
        put v to all of them
        return commit
    } else {
        return abort
    }
}
```

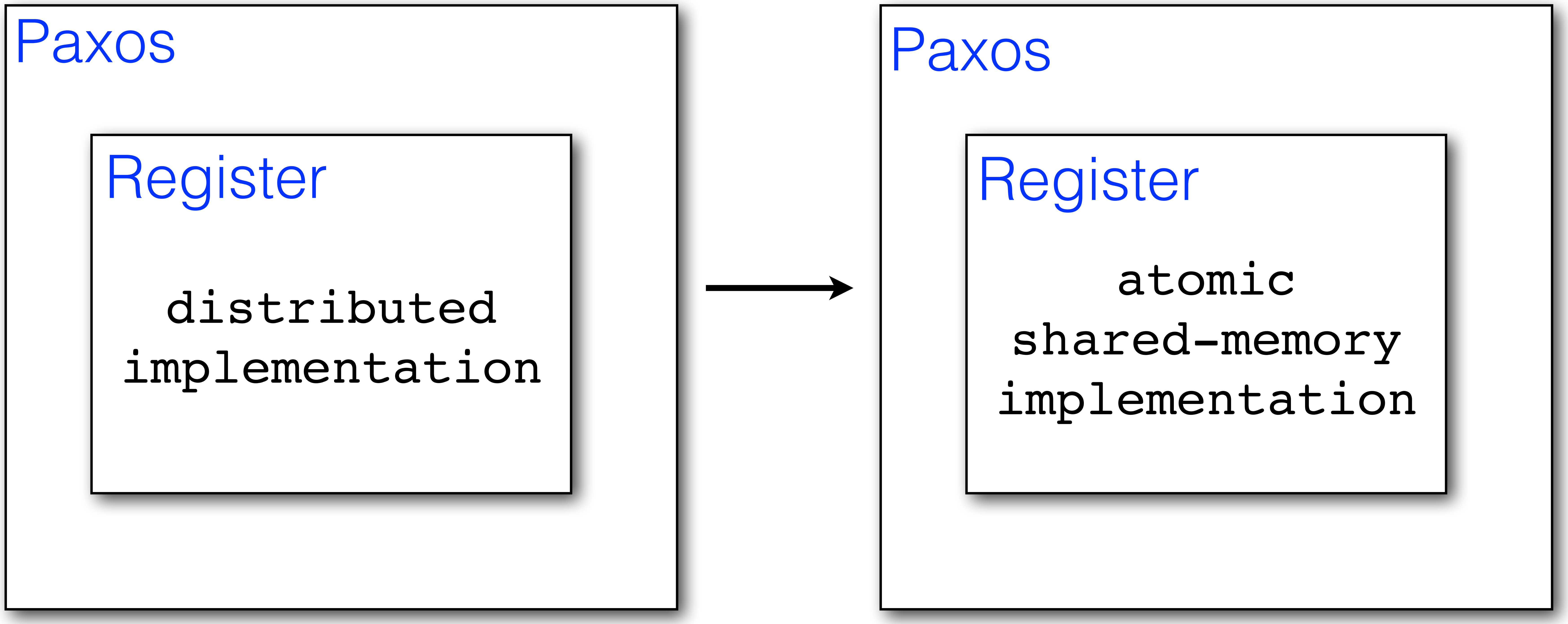
Consensus Using the Register

```
propose(v) {
    choose a round r
    v' = read(r)
    if (v' = abort)
        increase r and repeat
    if (v' = none) v' = v
    if (write(r, v') = commit)
        return v'
    else
        increase r and repeat
}
```

Conjecture

Round-based register is linearizable wrt an atomic specification strong enough to prove Paxos correct

** only safety, no liveness*



```
round = 0;
vals = {none};
```

```
atomic read(k) {
  if (round < k) {
   if (nondet()) {
    round = k;
    v = pickNondet(vals);
      return v;
    } else {
      return abort;
    }
  } else {
    return abort;
  }
}
```

```
atomic write(k, v) {
  if (round ≤ k) {
    if (nondet()) {
      vals = {v};
      round = k;
      return commit;
    } else {
      vals = vals U {v};
      return abort;
    }
  } else {
    return abort;
  }
}
```

```
round = 0;
vals = {none};
```

“Centralized state”

```
atomic read(k) {
 if (round < k) {
   if (nondet()) {
    round = k;
    v = pickNondet(vals);
    return v;
   } else {
    return abort;
   }
 } else {
   return abort;
 }
}
```

```
atomic write(k, v) {
  if (round ≤ k) {
    if (nondet()) {
      vals = {v};
      round = k;
      return commit;
    } else {
      vals = vals U {v};
      return abort;
    }
  } else {
    return abort;
  }
}
```

```
round = 0;
vals = {none};
```

Atomic methods

```
atomic read(k) {
 if (round < k) {
   if (nondet()) {
    round = k;
    v = pickNondet(vals);
    return v;
   } else {
    return abort;
   }
 } else {
   return abort;
 }
}
```

```
atomic write(k, v) {
  if (round ≤ k) {
    if (nondet()) {
      vals = {v};
      round = k;
      return commit;
    } else {
      vals = vals U {v};
      return abort;
    }
  } else {
    return abort;
  }
}
```

```
round = 0;
vals = {none};
```

Paxos becomes
a shared-memory algorithm

```
atomic read(k) {
 if (round < k) {
   if (nondet()
    round = k;
    v = pickNon
    return v;
   } else {
    return abort
   }
 } else {
   return abort
 }
}
```

```
atomic write(k, v) {
  if (round ≤ k) {
```

```
propose(v) {
     choose a round r
     v′ = read(r)
     if (v′ = abort)
          increase r and repeat
     if (v′ = none) v′ = v
     if (write(r, v′) = commit)
          return v′
     else
          increase r and repeat
}
```

```
round = 0;
vals = {none};
```

Single round number: the last round a majority of acceptors was switched to

Set of values stored at acceptors: singleton **{v}** if a quorum accepted **v**

- Tricky to simulate the implementation using a single round number;
- Different acceptors might have adopted different round numbers; the register *"acts"* differently depending on the underlying quorum;
- Solution: *highly non-deterministic* specification

```
round = 0;
vals = {none};
```

Methods can abort even if the parameter round is higher than the current one.

```
atomic read(k) {
  if (round < k) {
   if (nondet()) {
    round = k;
    v = pickNondet(vals);
      return v;
    } else {
      return abort;
    }
  } else {
    return abort;
  }
}
```

```
atomic write(k, v) {
  if (round ≤ k) {
    if (nondet()) {
      vals = {v};
      round = k;
      return commit;
    } else {
      vals = vals U {v};
      return abort;
    }
  } else {
    return abort;
  }
}
```

```
round = 0;
vals = {none};
```

Methods can abort even if the parameter round is higher than the current one.

OK for consensus safety - it just restarts.

```
atomic read(k) {
  if (round < k) {
   if (nondet())
    round = k;
    v = pickNonde
      return v;
    } else {
      return abor
    }
  } else {
    return abort;
  }
}
```

```
atomic write(k, v) {
  if (round ≤ k) {
```

```
propose(v) {
     choose a round r
     v′ = read(r)
     if (v′ = abort)
          increase r and repeat
     if (v′ = none) v′ = v
     if (write(r, v′) = commit)
          return v′
     else
          increase r and repeat
}
```

```
round = 0;
vals = {none};
```

Spec allows proving that a decision taken in consensus can't be changed

```
atomic read(k) {
  if (round < k) {
   if (nondet()) {
    round = k;
    v = pickNondet(vals);
      return v;
    } else {
      return abort;
    }
  } else {
    return abort;
  }
}
```

```
atomic write(k, v) {
  if (round ≤ k) {
    if (nondet()) {
      vals = {v};
      round = k;
      return commit;
    } else {
      vals = vals U {v};
      return abort;
    }
  } else {
    return abort;
  }
}
```

```
round = 0;
vals = {none};
```

Successful write of v sets vals to {v}

```
atomic read(k) {
  if (round < k) {
   if (nondet()) {
    round = k;
    v = pickNondet(vals);
      return v;
    } else {
      return abort;
    }
  } else {
    return abort;
  }
}
```

```
atomic write(k, v) {
  if (round ≤ k) {
    if (nondet()) {
      vals = {v};
      round = k;
      return commit;
    } else {
      vals = vals U {v};
      return abort;
    }
  } else {
    return abort;
  }
}
```

```
round = 0;
vals = {none};
```

Successful write of v sets vals to {v}

Following successful read will return v

```
atomic read(k) {
  if (round < k) {
   if (nondet()) {
    round = k;
    v = pickNondet(vals);
     return v;
    } else {
     return abort;
    }
  } else {
    return abort;
  }
}
```

```
atomic write(k, v) {
  if (round ≤ k) {
    if (nondet()) {
      vals = {v};
      round = k;
      return commit;
    } else {
      vals = vals ∪ {v};
      return abort;
    }
  } else {
    return abort;
  }
}
```

```
round = 0;
vals = {none};
```

Successful write of v sets vals to {v}.
Following successful read will return v.

propose() writes what it has read.

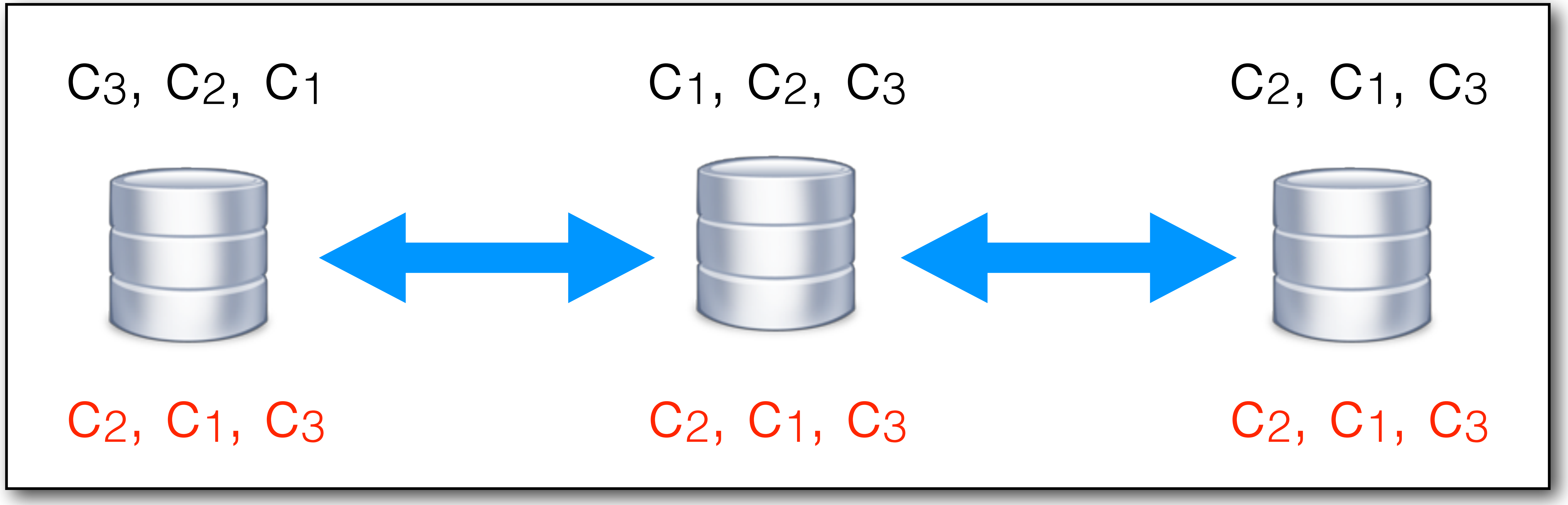
```
atomic read(k) {
  if (round < k) {
   if (nondet())
    round = k;
    v = pickNonde
      return v;
    } else {
      return abor
    }
  } else {
    return abort;
  }
}
```

```
atomic write(k, v) {
  if (round ≤ k) {
```

```
propose(v) {
    choose a round r
    v' = read(r)
    if (v' = abort)
        increase r and repeat
    if (v' = none) v' = v
    if (write(r, v') = commit)
        return v'
    else
        increase r and repeat
}
```

Multi-Paxos

State machine replication requires solving a sequence of consensus instances



- Naive solution: execute a separate Paxos instance for each sequence element
- Multi-Paxos: "Amortize" Phase 1 once for multiple sequence elements

Scaling to Multi-Paxos

Multi-Paxos refines the naive solution ➜
can be proven without unpacking the proof of Paxos

- Naive solution: execute a separate Paxos instance for each sequence element
- Multi-Paxos: “Amortize” Phase 1 once for multiple sequence elements
- See the ESOP’18 paper “Paxos Consensus, Deconstructed and Abstracted” for details.

To Take Away

- Viewstamped replication (1988)
- Paxos (1998)
- Disk Paxos (2003)
- Cheap Paxos (2004)
- Generalized Paxos (2004)
- Paxos Commit (2004)
- Fast Paxos (2006)
- Stoppable Paxos (2008)
- Mencius (2008)
- Vertical Paxos (2009)
- ZAB (2009)
- Ring Paxos (2010)
- Egalitarian Paxos (2013)
- Raft (2014)
- M2Paxos (2016)
- Flexible Paxos (2016)
- Caesar (2017)

- *Shared-memory* concurrency is simpler than synchronous *message-passing* concurrency;
- Linearizability is a good tool for *vertically structuring* protocols;
- *Non-determinism* is specs is your friend.

Thanks!