Paxos Consensus, Abstracted and Deconstructed

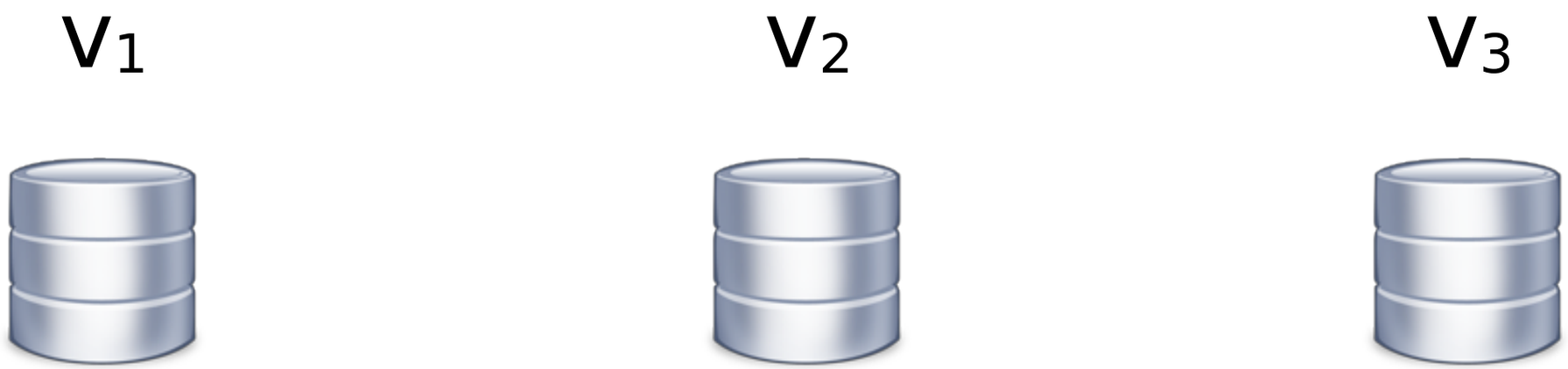
Álvaro García Pérez, Alexey Gotsman,
Yuri Meshamn, and Ilya Sergey

April 19th 2008

Consensus

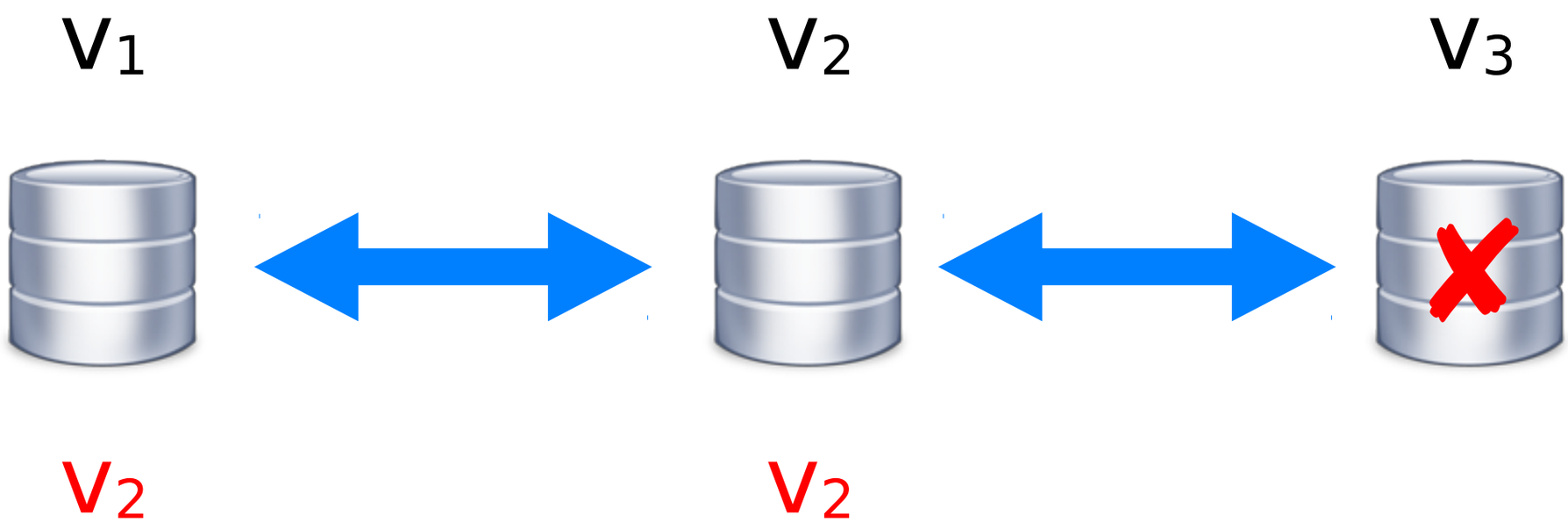
- Several nodes, which can crash

Consensus



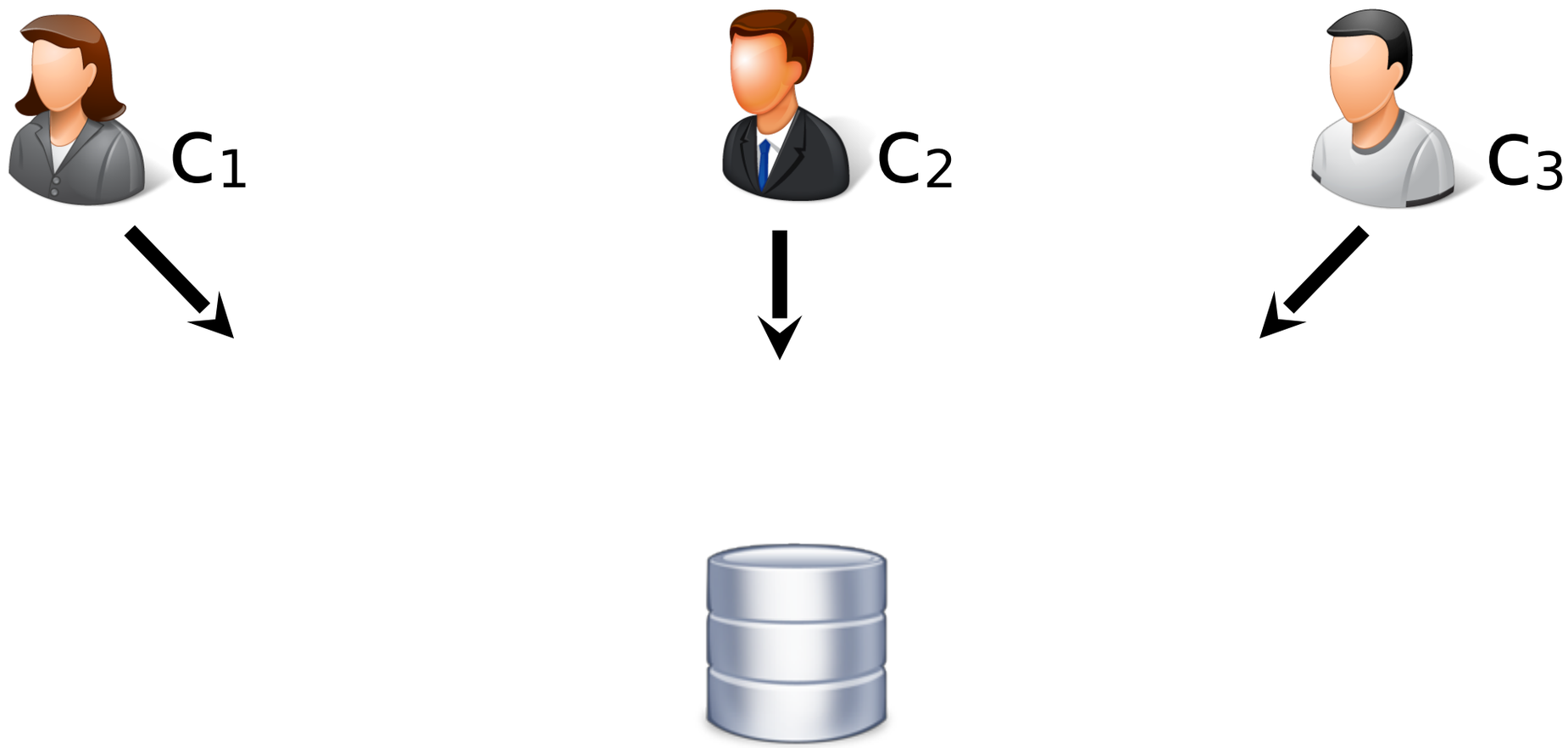
- Several nodes, which can crash
- Each node proposes a value

Consensus



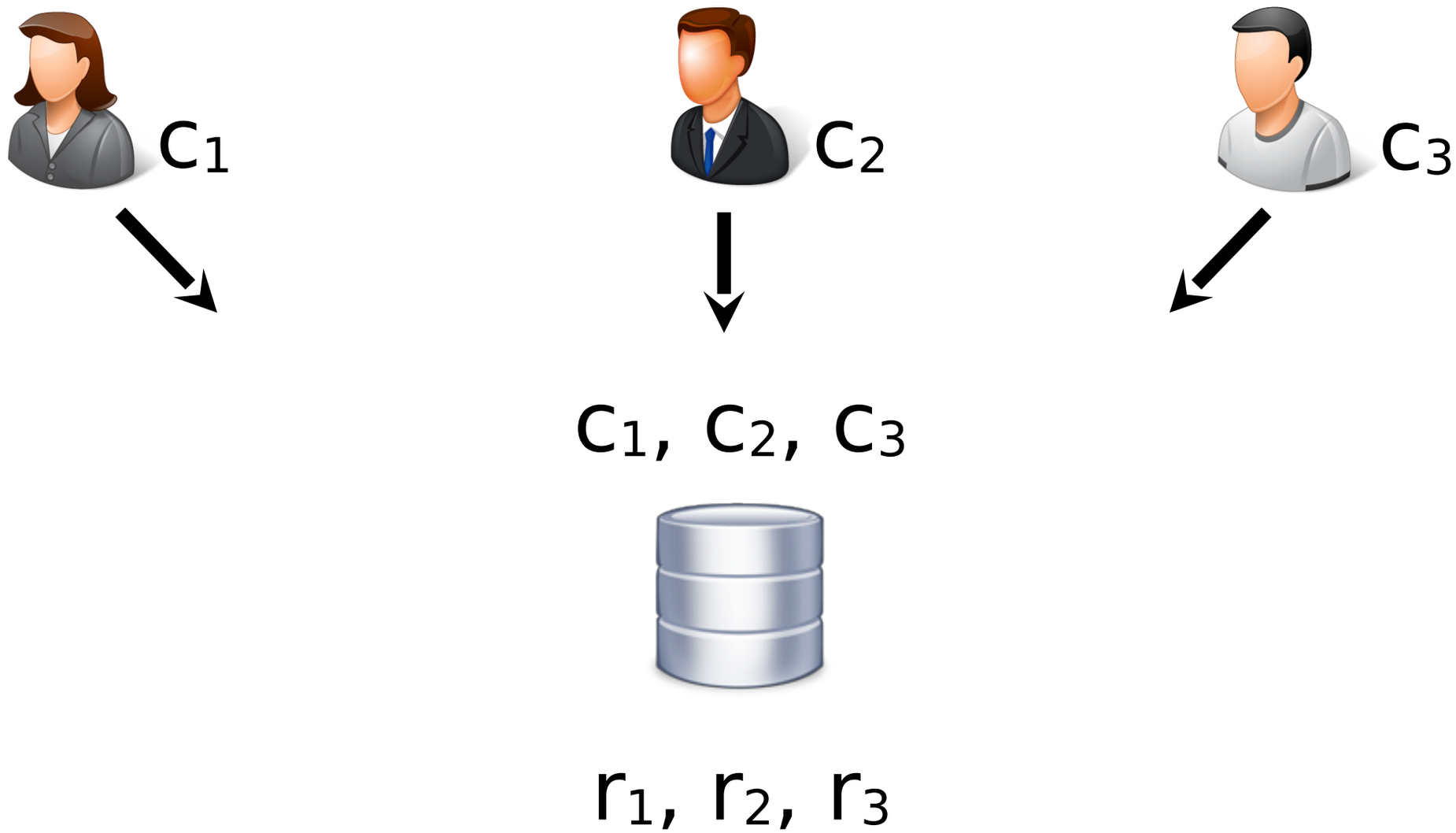
- Several nodes, which can crash
- Each node proposes a value
- All non-crashed nodes agree on a single value

Deterministic state machine



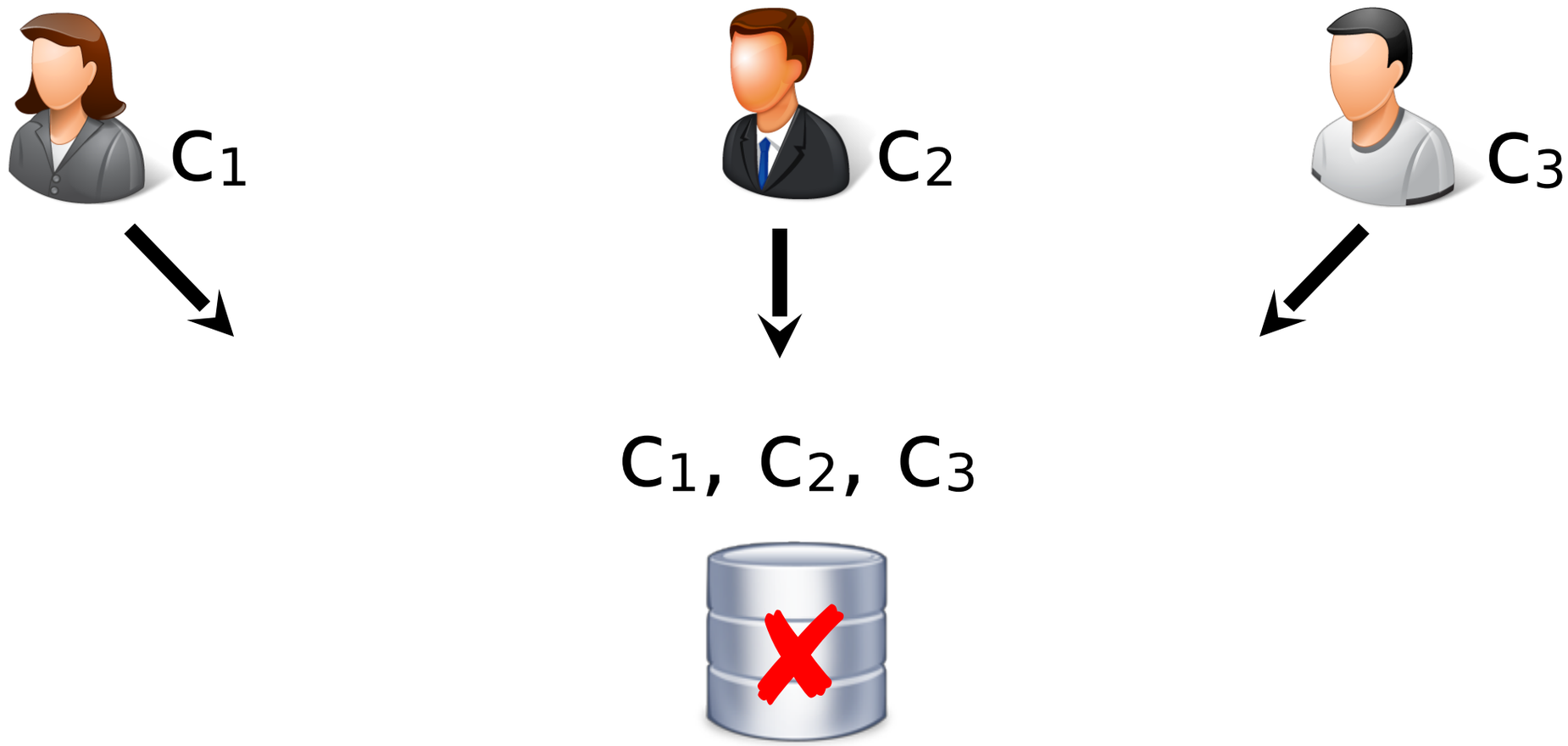
Clients submit commands

Deterministic state machine



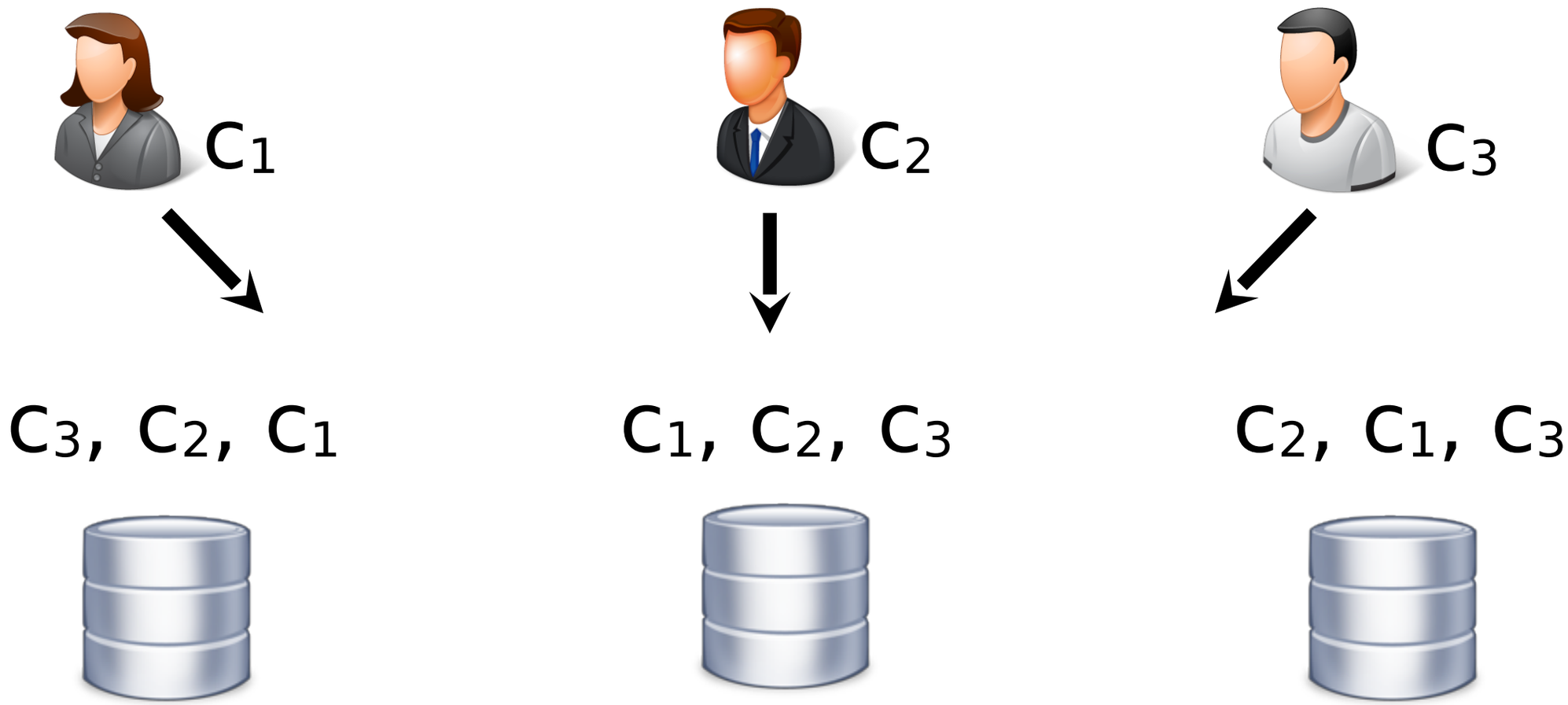
Machine totally orders commands and computes the sequence of results

Deterministic state machine



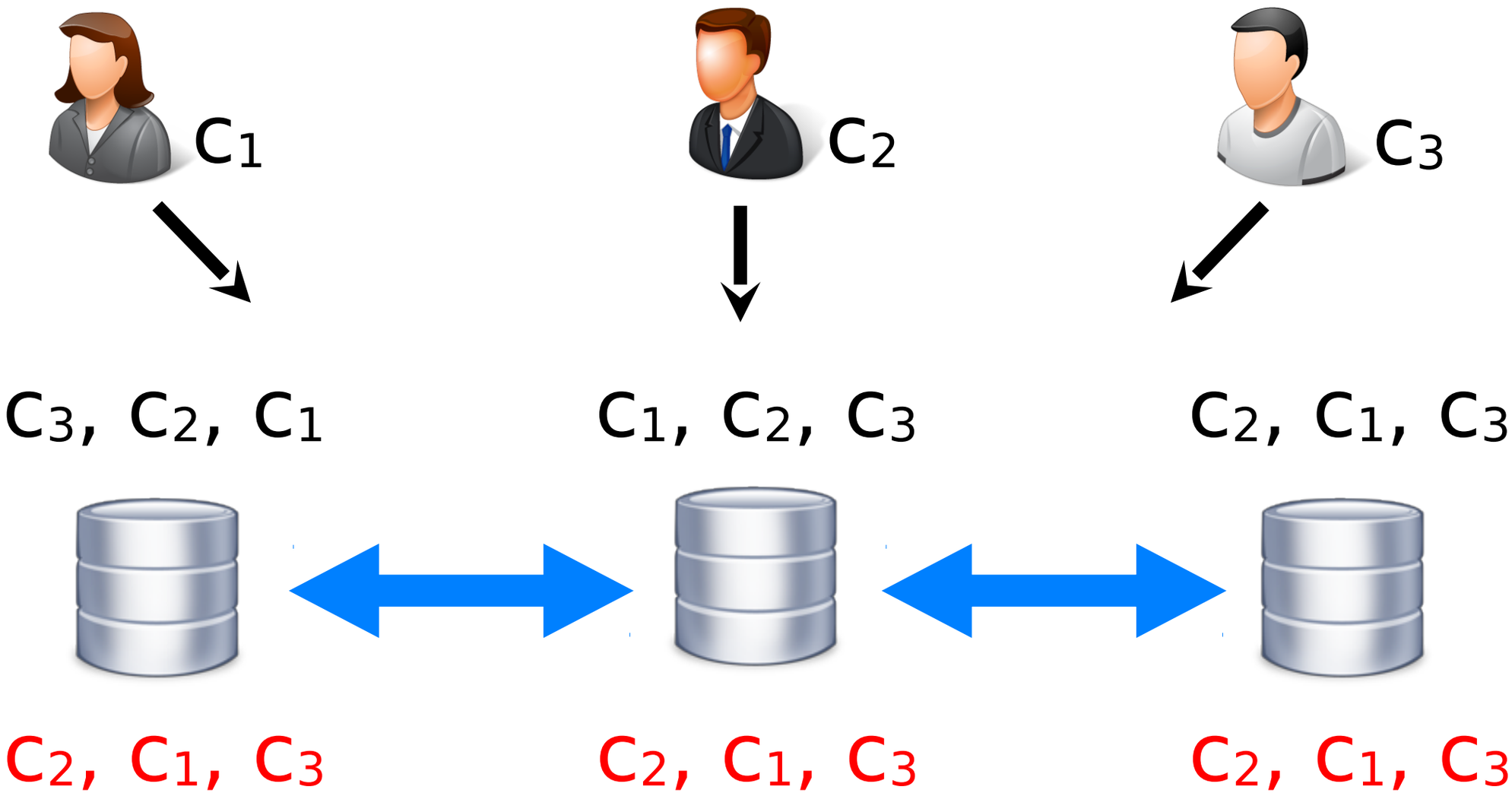
Machine totally orders commands and computes the sequence of results

State machine replication



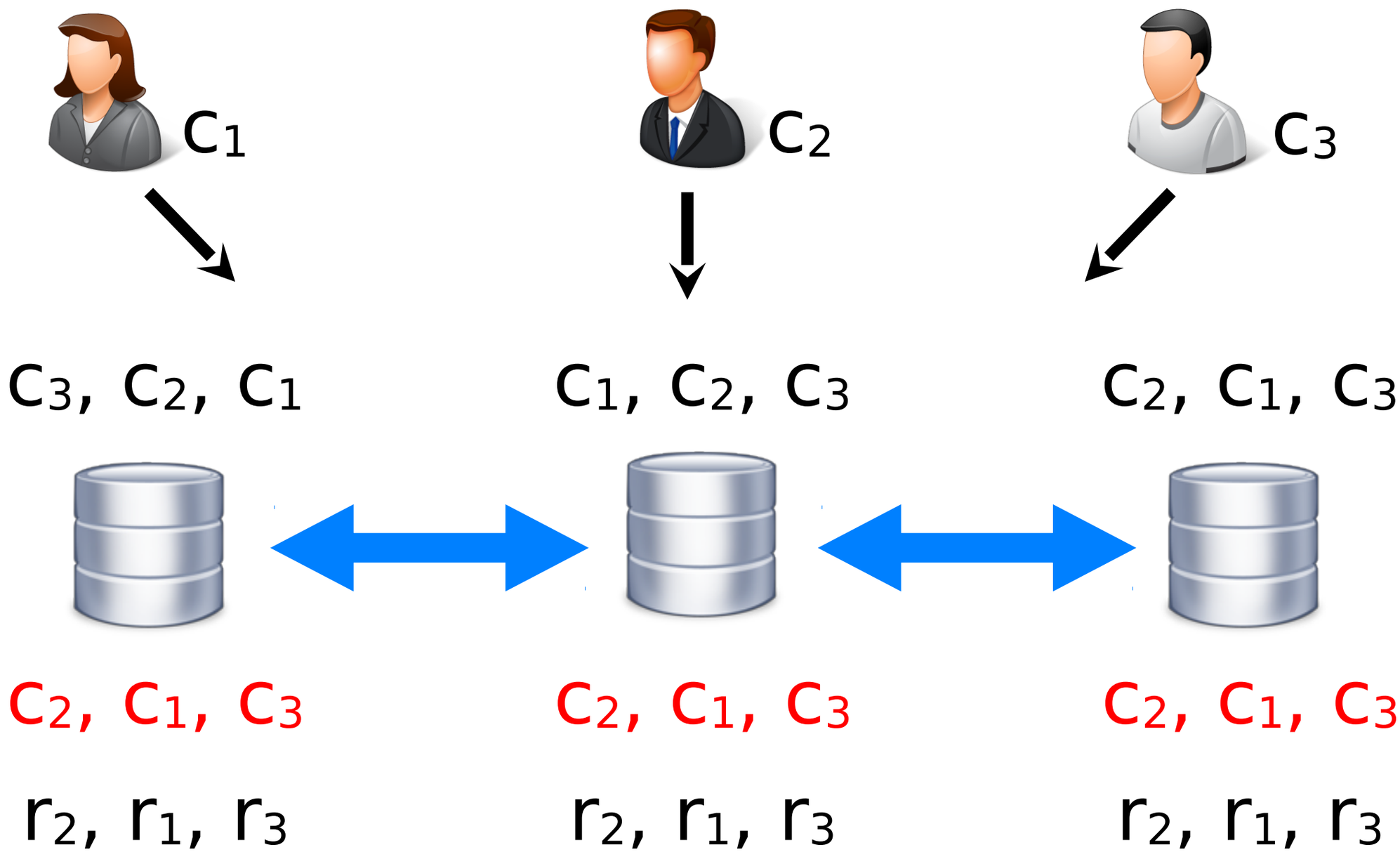
Clients send commands to all replicas
Replicas may receive commands in different orders

State machine replication



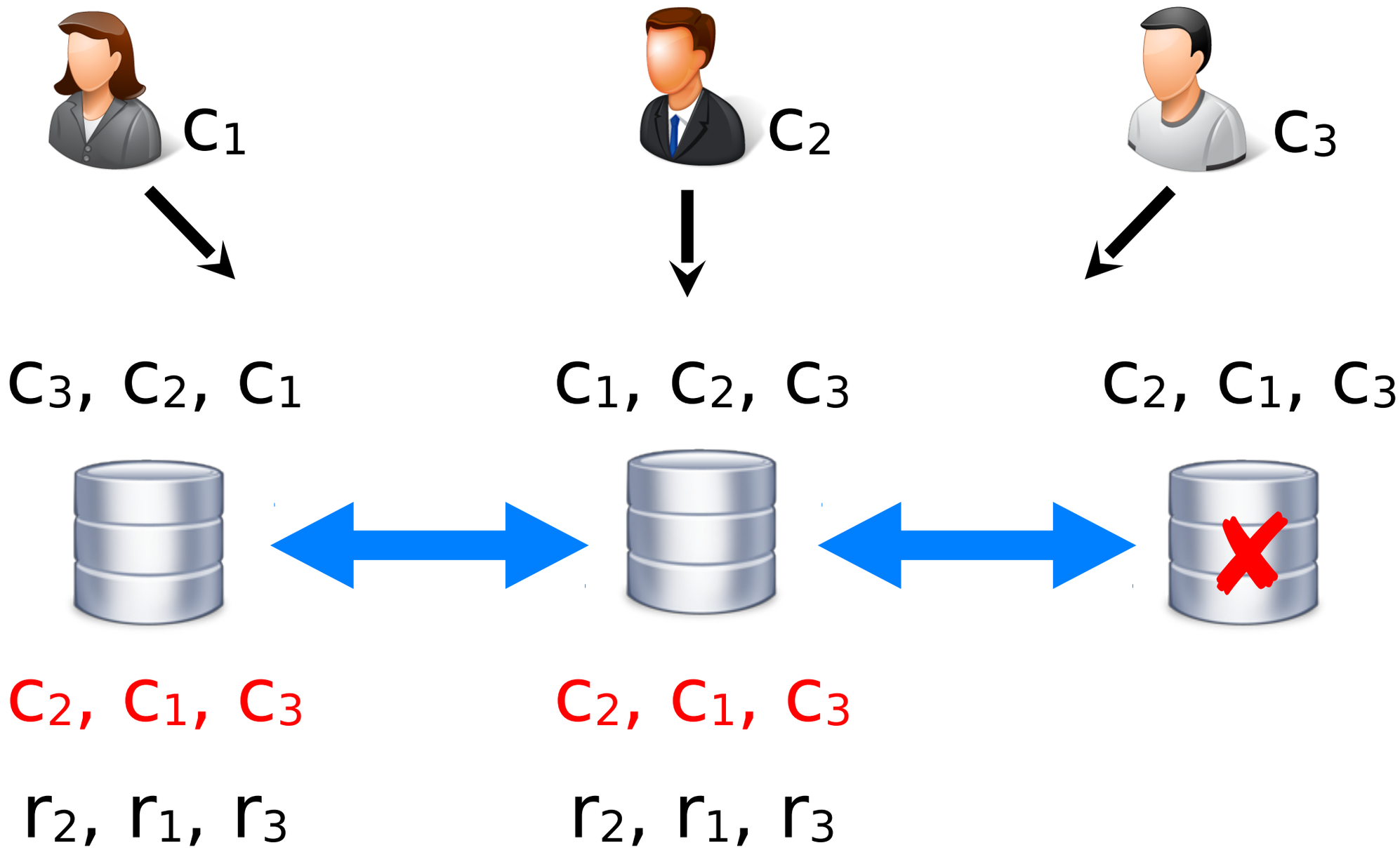
Totally order commands via a sequence of consensus instances

State machine replication



Replicas compute the same sequence of results

State machine replication



Replicas compute the same sequence of results

State ma

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

c_1

c_2

c_3

c_3, c_2, c_1

c_1, c_2, c_3

c_2, c_1, c_3

c_2, c_1, c_3

c_2, c_1, c_3

r_2, r_1, r_3

r_2, r_1, r_3

Replicas compute the same sequence of results

The zoo of consensus protocols

- 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)

The zoo of consensus protocols

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)

The zoo of consensus protocols

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)

The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the state-machine approach to the design of distributed systems.

Categories and Subject Descriptors: C2.4 [**Computer-Communications Networks**]: Distributed Systems—*Network operating systems*; D4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; J.1 [**Administrative Data Processing**]: Government

General Terms: Design, Reliability

Additional Key Words and Phrases: State machines, three-phase commit, voting

This submission was recently discovered behind a filing cabinet in the *TOCS* editorial office. Despite its age, the editor-in-chief felt that it was worth publishing. Because the author is currently doing field work in the Greek isles and cannot be reached, I was asked to prepare it for publication.

The author appears to be an archeologist with only a passing interest in computer science. This is unfortunate; even though the obscure ancient Paxon civilization he describes is of little interest to most computer scientists, its legislative system is an excellent model for how to implement a distributed computer system in an asynchronous environment. Indeed, some of the refinements the Paxons made to their protocol appear to be unknown in the systems literature.

The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Paxos Made Simple

Leslie Lamport

Abstract

The Paxos algorithm, when presented in plain English, is very simple.

The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Paxos Made Simple

Paxos Made Moderately Complex

ROBBERT VAN RENESSE and DENIZ ALTINBUKEN, Cornell University

This article explains the full reconfigurable multidecree Paxos (or multi-Paxos) protocol. Paxos is by no means a simple protocol, even though it is based on relatively simple invariants. We provide pseudocode and explain it guided by invariants. We initially avoid optimizations that complicate comprehension. Next we discuss liveness, list various optimizations that make the protocol practical, and present variants of the protocol.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Network operating systems*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*

General Terms: Design, Reliability

Additional Key Words and Phrases: Replicated state machines, consensus, voting

ACM Reference Format:
Robbert van Renesse and Deniz Altinbuken. 2015. Paxos made moderately complex. ACM Comput. Surv. 47, 3, Article 42 (February 2015), 36 pages.
DOI: http://dx.doi.org/10.1145/2673577

The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Paxos Made Simple

Paxos Made Moderately Complex

In Search of an Understandable Consensus Algorithm

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 majori-

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 [27, 20]), 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 al-

The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Paxos Made Simple

Paxos Made Moderately Complex

In Search of an Understandable Consensus Algorithm

Abstract

Raft is a consensus algorithm for m
log. It produces a result equivalent tc
it is as efficient as Paxos, but its s
from Paxos; this makes Raft more u
Paxos and also provides a better fc
ing practical systems. In order to enha
ity, Raft separates the key elements of
leader election, log replication, and sa
a stronger degree of coherency to re
states that must be considered. Resul
demonstrate that Raft is easier for st
Paxos. Raft also includes a new mecl
the cluster membership, which uses

Unfortunately, Paxos has two significant drawbacks. The first drawback is that Paxos is exceptionally difficult to understand. The full explanation [15] is notoriously opaque; few people succeed in understanding it, and only with great effort. As a result, there have been several attempts to explain Paxos in simpler terms [16, 20, 21]. These explanations focus on the single-decree subset, yet they are still challenging. In an informal survey of attendees at NSDI 2012, we found few people who were comfortable with Paxos, even among seasoned researchers. We struggled with Paxos ourselves; we were not able to understand the complete protocol until after reading several simplified explanations and designing our own alter-

Paxos Made Live - An Engineering Perspective (2006 Invited Talk)

Tushar Chandra, Robert Griesemer, and Joshua Redstone

Google Inc.

ABSTRACT

We describe our experience in building a fault-tolerant database using the Paxos consensus algorithm. Despite the existing literature in the field, building such a database proved to be non-trivial. We describe selected algorithmic and engineering problems encountered, and the solutions we found for them. Our measurements indicate that we have built a competitive system.

Categories and Subject Descriptors

D.4.5 [**Operating systems**]: Reliability—*Fault-tolerance*; B.4.5 [**Input/output and data communications**]: Reliability, Testing, and Fault-Tolerance—*Redundant design*

General Terms

Experimentation, Performance, Reliability

Keywords

Experiences, Fault-tolerance, Implementation, Paxos

database is just an example. As a result, the consensus problem has been studied extensively over the past two decades. There are several well-known consensus algorithms that operate within a multitude of settings and which tolerate a variety of failures. The Paxos consensus algorithm [8] has been discussed in the theoretical [16] and applied community [10, 11, 12] for over a decade.

We used the Paxos algorithm ("Paxos") as the base for a framework that implements a fault-tolerant log. We then relied on that framework to build a fault-tolerant database. Despite the existing literature on the subject, building a production system turned out to be a non-trivial task for a variety of reasons:

- While Paxos can be described with a page of pseudocode, our complete implementation contains several thousand lines of C++ code. The blow-up is not due simply to the fact that we used C++ instead of pseudo notation, nor because our code style may have been verbose. Converting the algorithm into a practical, production-ready system involved implementing many features and optimizations – some published in the lit-

Paxos Made Live - An Engineering Perspective (2006 Invited Talk)

Tushar Chandra, Robert Griesemer, and Joshua Redstone

Google Inc.

ABSTRACT

We describe our experience in building a fault-t
base using the Paxos consensus algorithm. De
isting literature in the field, building such a dat
to be non-trivial. We describe selected algorith
gineering problems encountered, and the soluti
for them. Our measurements indicate that we
competitive system.

- There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. In order to build a real-world system, an expert needs to use numerous ideas scattered in the literature and make several relatively small protocol extensions. The cumulative effort will be substantial and the final system will be based on an unproven protocol.

a framework that implements a fault-tolerant log. We then relied on that framework to build a fault-tolerant database. Despite the existing literature on the subject, building a production system turned out to be a non-trivial task for a variety of reasons:

- While Paxos can be described with a page of pseudo-code, our complete implementation contains several thousand lines of C++ code. The blow-up is not due simply to the fact that we used C++ instead of pseudo notation, nor because our code style may have been verbose. Converting the algorithm into a practical, production-ready system involved implementing many features and optimizations – some published in the lit-

Categories and Subject Descriptors

D.4.5 [**Operating systems**]: Reliability—*Fault-tolerance*; B.4.5 [**Input/output and data communications**]: Reliability, Testing, and Fault-Tolerance—*Redundant design*

General Terms

Experimentation, Performance, Reliability

Keywords

Experiences, Fault-tolerance, Implementation, Paxos

Paxos Made Live - (200

Tushar Chandra, Rob

ABSTRACT

We describe our experience in building a fault-tolera
base using the Paxos consensus algorithm. Despite
isting literature in the field, building such a database
to be non-trivial. We describe selected algorithmic
gineering problems encountered, and the solutions w
for them. Our measurements indicate that we have
competitive system.

Categories and Subject Descriptors

D.4.5 [**Operating systems**]: Reliability—*Fault-to*
B.4.5 [**Input/output and data communication**
ability, Testing, and Fault-Tolerance—*Redundant d*

General Terms

Experimentation, Performance, Reliability

Keywords

Experiences, Fault-tolerance, Implementation, Paxos

5.1 Handling disk corruption

Replicas witness disk corruption from time to time. A disk may be corrupted due to a media failure or due to an operator error (an operator may accidentally erase critical data). When a replica's disk is corrupted and it loses its persistent state, it may renege on promises it has made to other replicas in the past. This violates a key assumption in the Paxos algorithm. We use the following mechanism to address this problem [14].

Disk corruptions manifest themselves in two ways. Either file(s) contents may change or file(s) may become inaccessible. To detect the former, we store the checksum of the contents of each file in the file[2]. The latter may be indistinguishable from a new replica with an empty disk – we detect this case by having a new replica leave a marker in GFS after start-up. If this replica ever starts again with an empty disk, it will discover the GFS marker and indicate that it has a corrupted disk.

A replica with a corrupted disk rebuilds its state as follows. It participates in Paxos as a non-voting member; meaning that it uses the catch-up mechanism to catch up but does not respond with promise or acknowledgment messages. It remains in this state until it observes one complete instance of Paxos that was started after the replica started rebuilding its state. By waiting for the extra instance of Paxos, we ensure that this replica could not have reneged on an earlier promise.

production-ready system involved implementing many features and optimizations – some published in the lit-

Paxos Made Live - (200

Tushar Chandra, Rob

ABSTRACT

We describe our experience in building a fault-tolera
base using the Paxos consensus algorithm. Despite
isting literature in the field, building such a database
to be non-trivial. We describe selected algorithmic
gineering problems encountered, and the solutions w
for them. Our measurements indicate that we have
competitive system.

Categories and Subject Descriptors

D.4.5 [**Operating systems**]: Reliability—*Fault-to*
B.4.5 [**Input/output and data communication**
ability, Testing, and Fault-Tolerance—*Redundant d*

General Terms

Experimentation, Performance, Reliability

Keywords

Experiences, Fault-tolerance, Implementation, Paxos

5.1 Handling disk corruption

Replicas witness disk corruption from time to time. A disk may be corrupted due to a media failure or due to an operator error (an operator may accidentally erase critical data). When a replica's disk is corrupted and it loses its persistent state, it may renege on promises it has made to other replicas in the past. This violates a key assumption in the Paxos algorithm. We use the following mechanism to address this problem [14].

Disk corruptions manifest themselves in two ways. Either file(s) contents may change or file(s) may become inaccessible. To detect the former, we store the checksum of the contents of each file in the file[2]. The latter may be indistinguishable from a new replica with an empty disk – we detect this case by having a new replica leave a marker in GFS aft
empty d
that it has a corrupted disk.

Broken [Michael+ 2016]

A replica with a corrupted disk rebuilds its state as follows. It participates in Paxos as a non-voting member; meaning that it uses the catch-up mechanism to catch up but does not respond with promise or acknowledgment messages. It remains in this state until it observes one complete instance of Paxos that was started after the replica started rebuilding its state. By waiting for the extra instance of Paxos, we ensure that this replica could not have reneged on an earlier promise.

production-ready system involved implementing many features and optimizations – some published in the lit-

Goals

- Develop methods for proving protocols correct, including realistic deployments
- Get insights into their structure

Goals

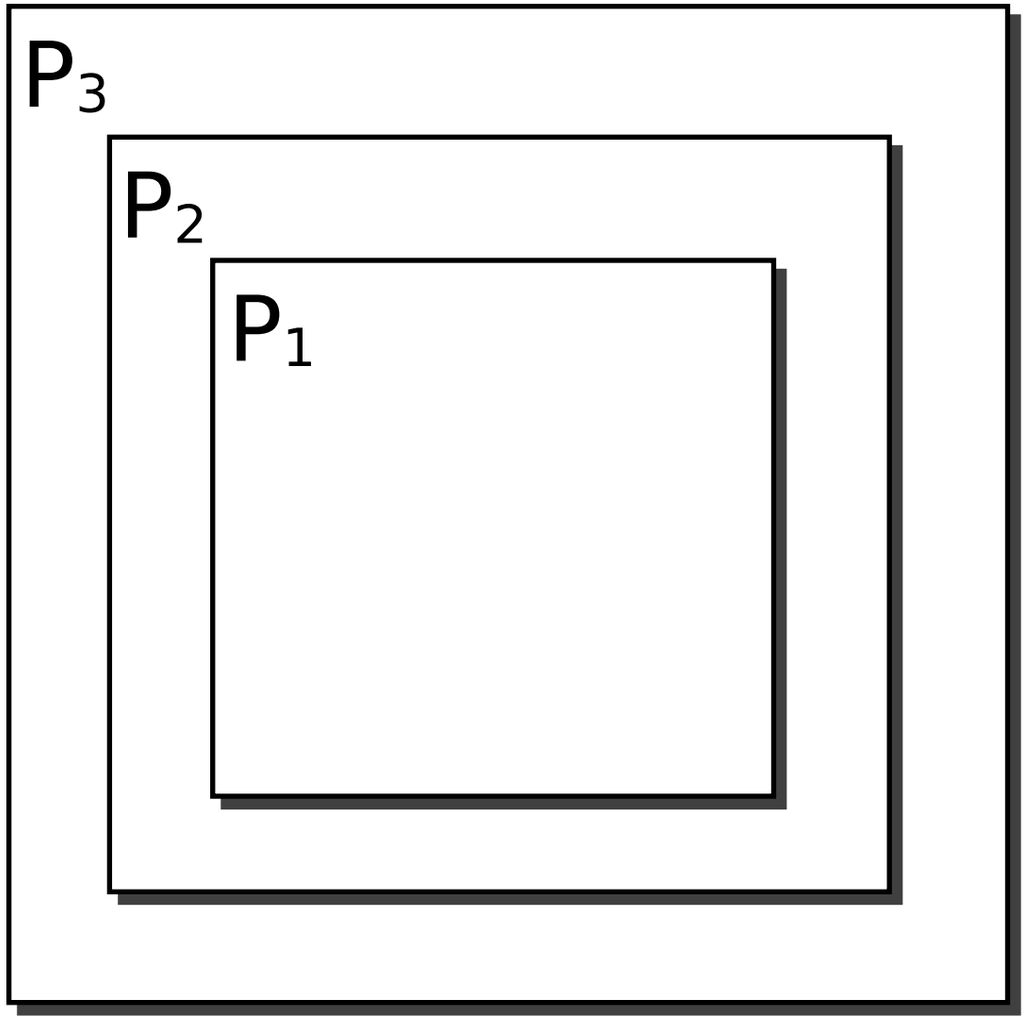
- Develop methods for proving protocols correct, including realistic deployments
- Get insights into their structure
- Focus on single-decree Paxos and Multi-Paxos

Approach

- Modular reasoning: verify parts of the protocol separately instead of the whole thing

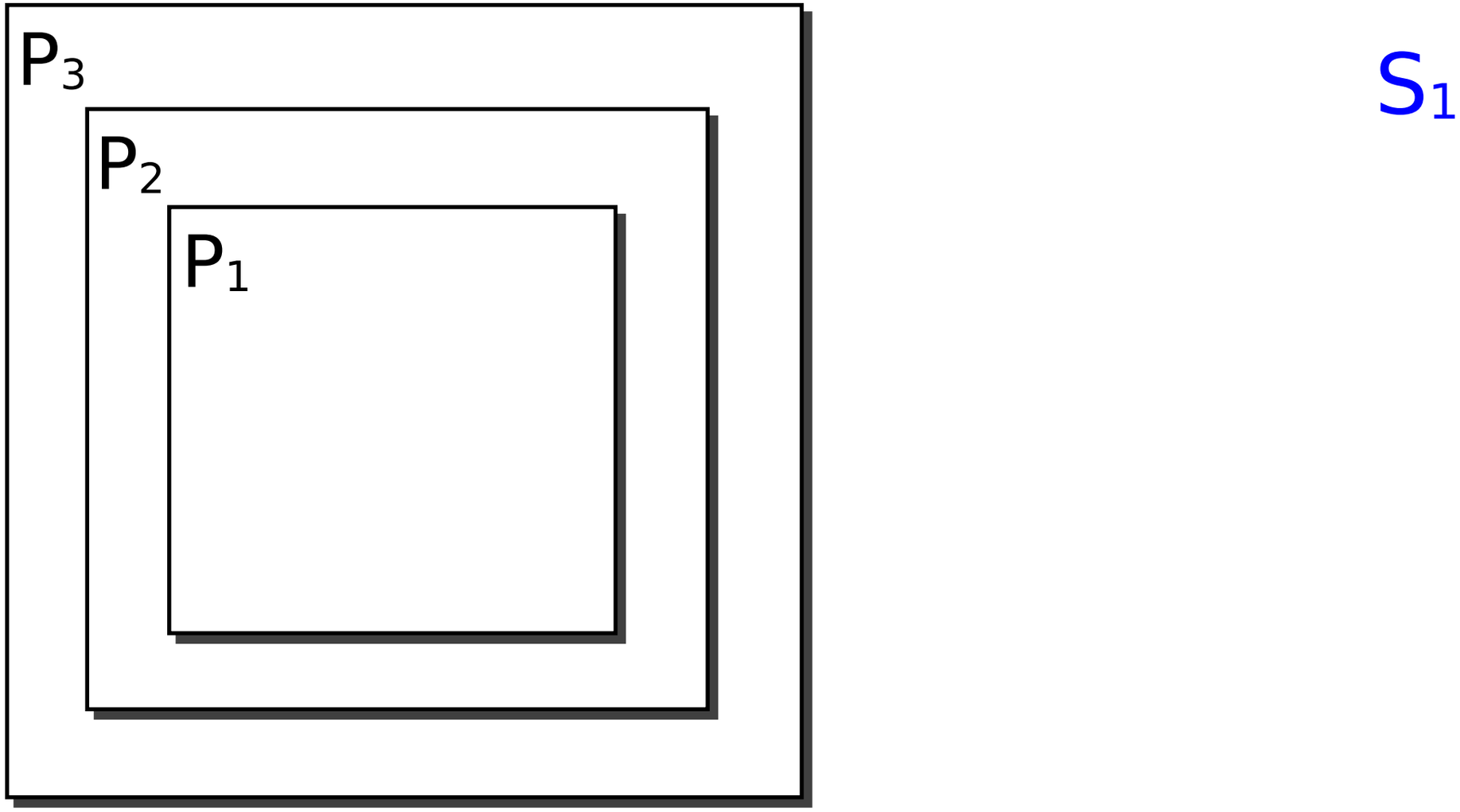
Approach

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



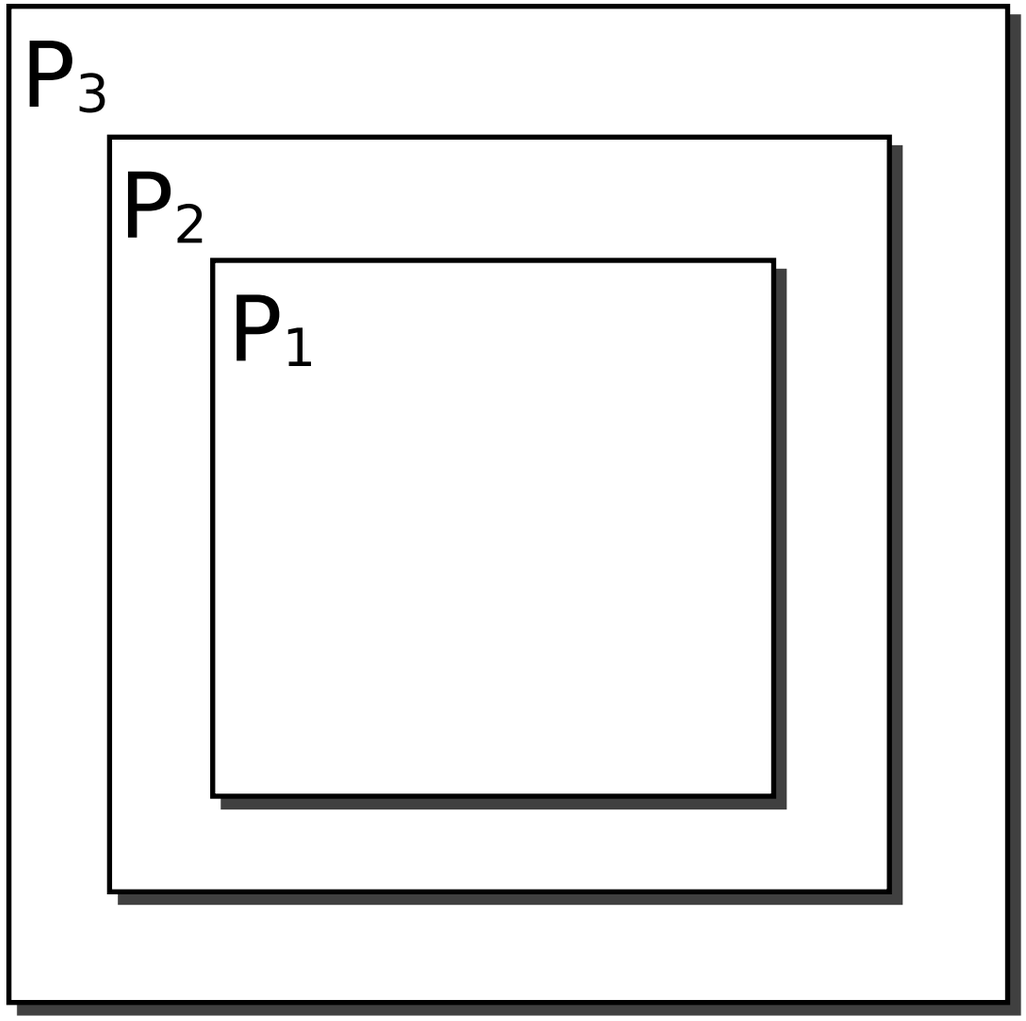
Approach

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



Approach

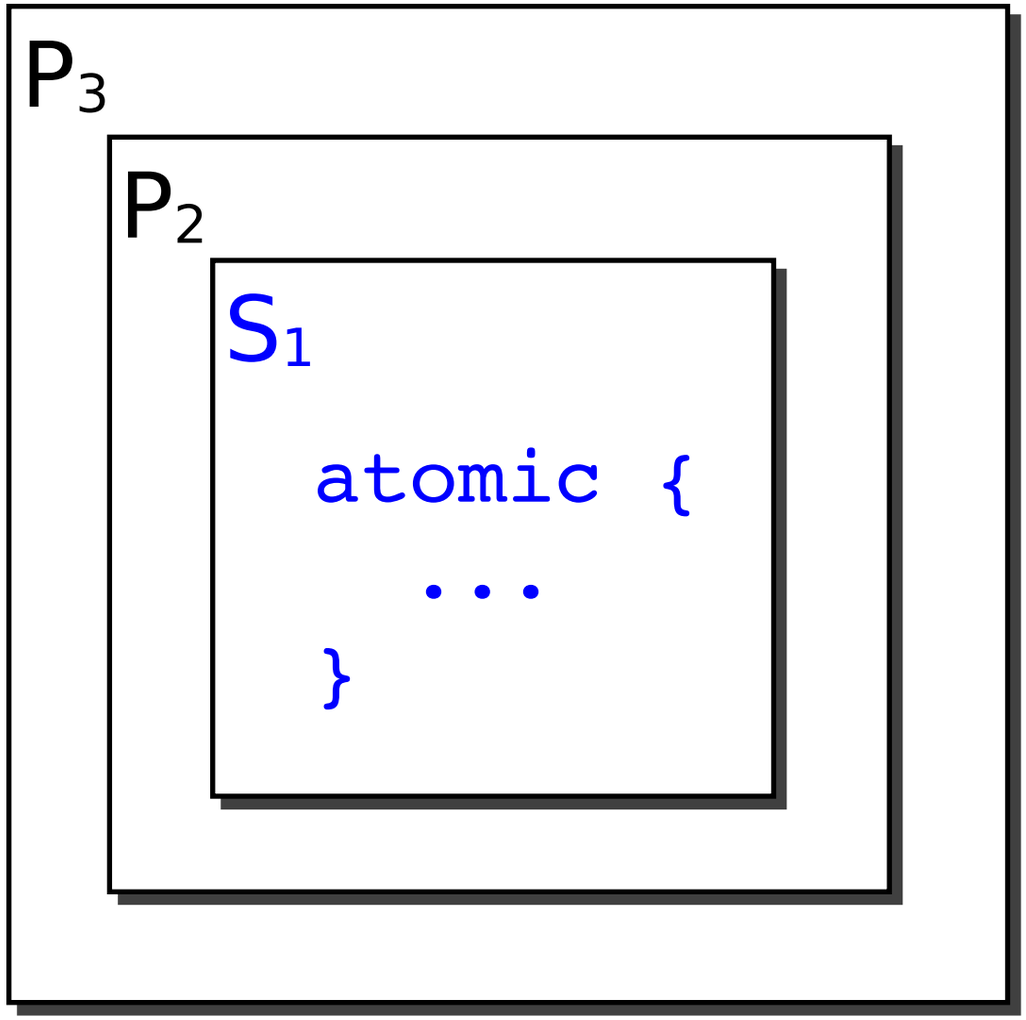
- Modular reasoning: verify parts of the protocol separately instead of the whole thing
  - Linearizability implies refinement [Filipovic+ 2009]



$P_1 \sqsubseteq S_1$

Approach

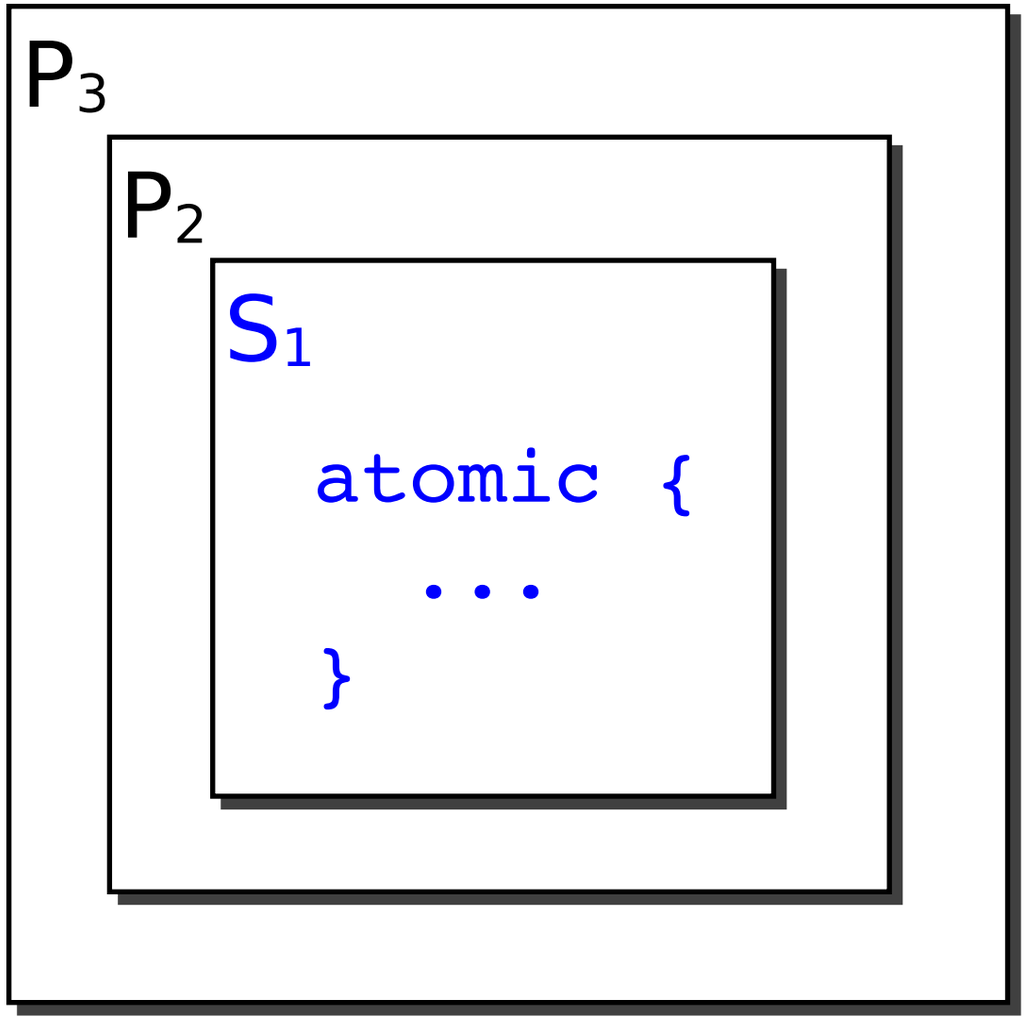
- Modular reasoning: verify parts of the protocol separately instead of the whole thing
  - Linearizability implies refinement [Filipovic+ 2009]



$P_1 \sqsubseteq S_1$

Approach

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

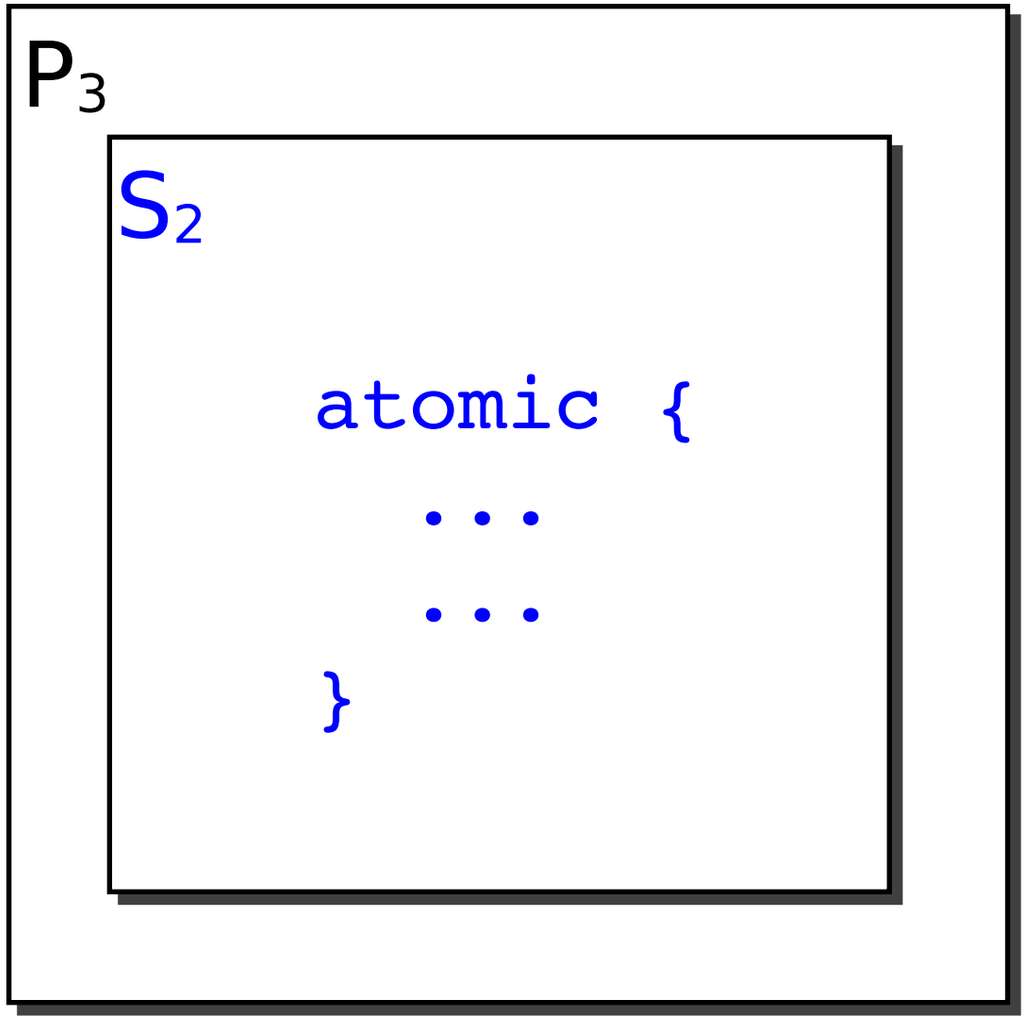


$P_1 \sqsubseteq S_1$

$P_2(S_1) \sqsubseteq S_2$

Approach

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

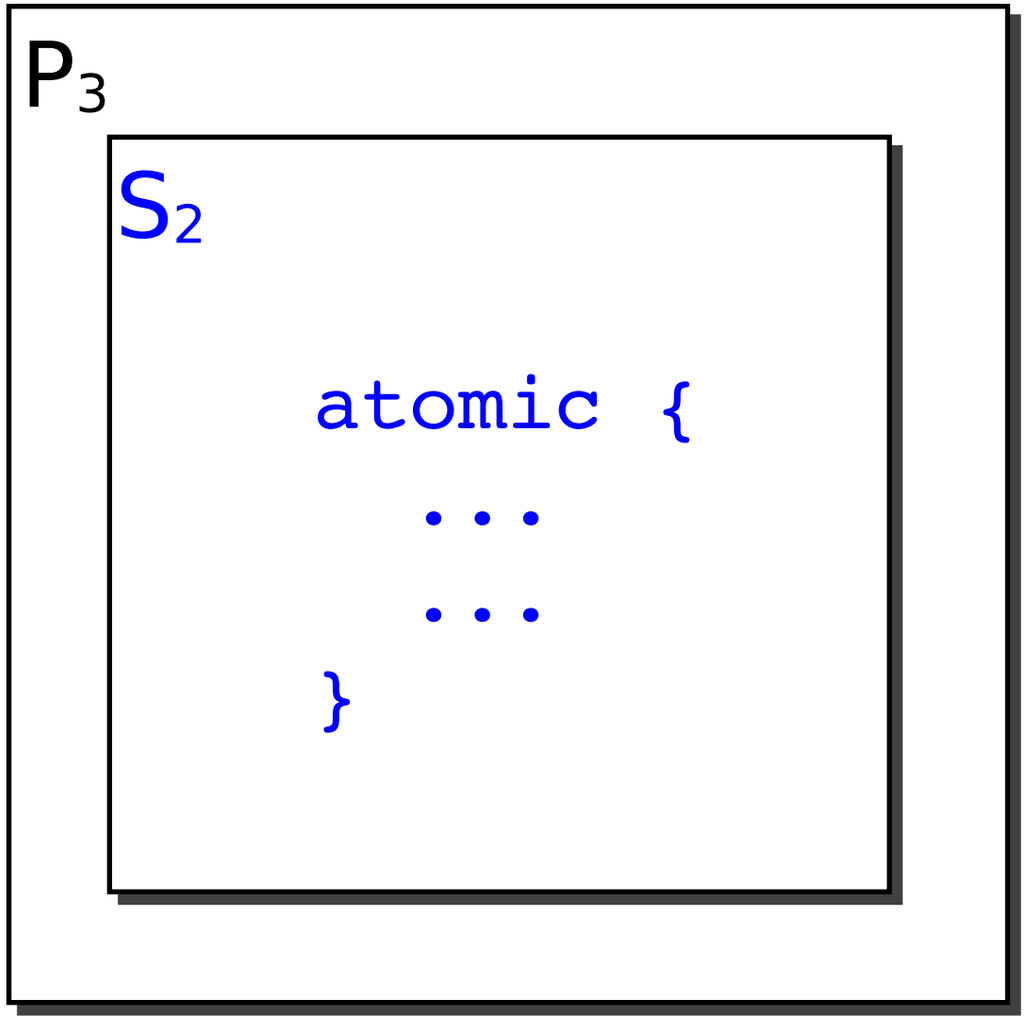


$P_1 \sqsubseteq S_1$

$P_2(S_1) \sqsubseteq S_2$

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

$P_1 \sqsubseteq S_1$

$P_2(S_1) \sqsubseteq S_2$

$P_3(S_2) \sqsubseteq S_3$

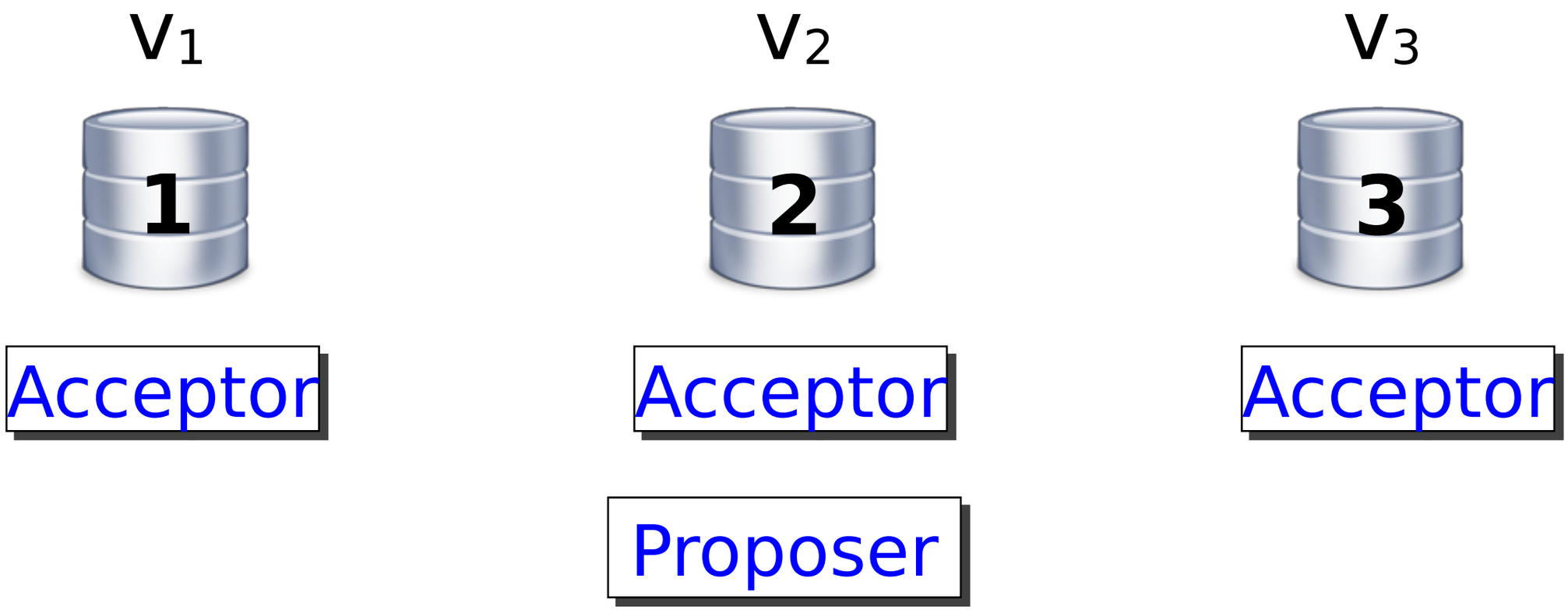
Approach

- Modular reasoning: verify parts of the protocol separately instead of the whole thing
  - Linearizability implies refinement [Filipovic+ 2009]
  - Transformations of the network semantics, à la *Verified System Transformers* of the Verdi framework [Wilcox+ 2015]

Approach

- Modular reasoning: verify parts of the protocol separately instead of the whole thing
  - Linearizability implies refinement [Filipovic+ 2009]
  - Transformations of the network semantics, à la *Verified System Transformers* of the Verdi framework [Wilcox+ 2015]

Prove one variant of the protocol without unpacking the proof of a simpler variant



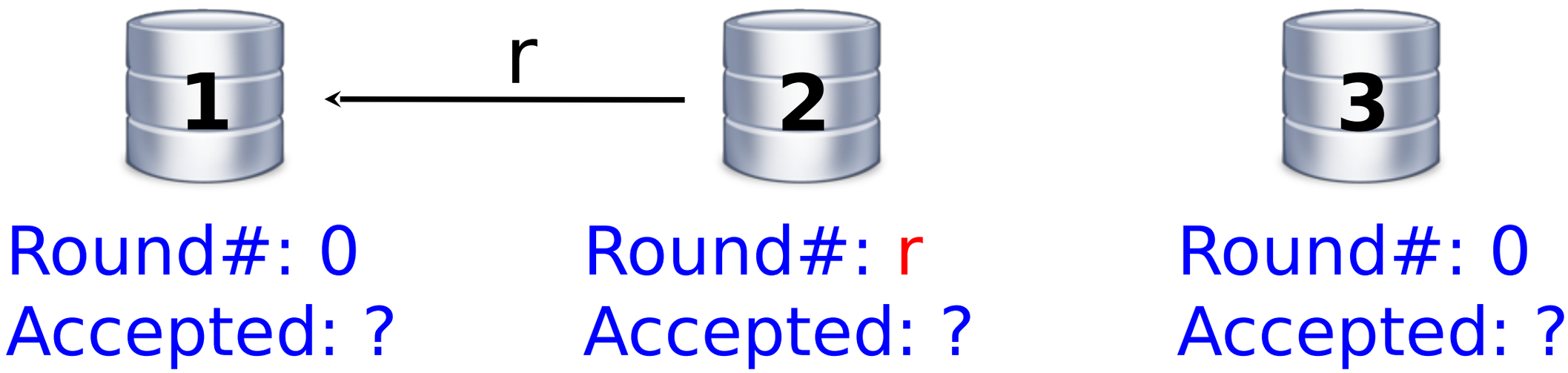
- Acceptors = members of parliament:
  can vote to accept a value,majority wins
- Proposer = parliament speaker:
  proposes its value to vote on

Round#: 0
Accepted: ?

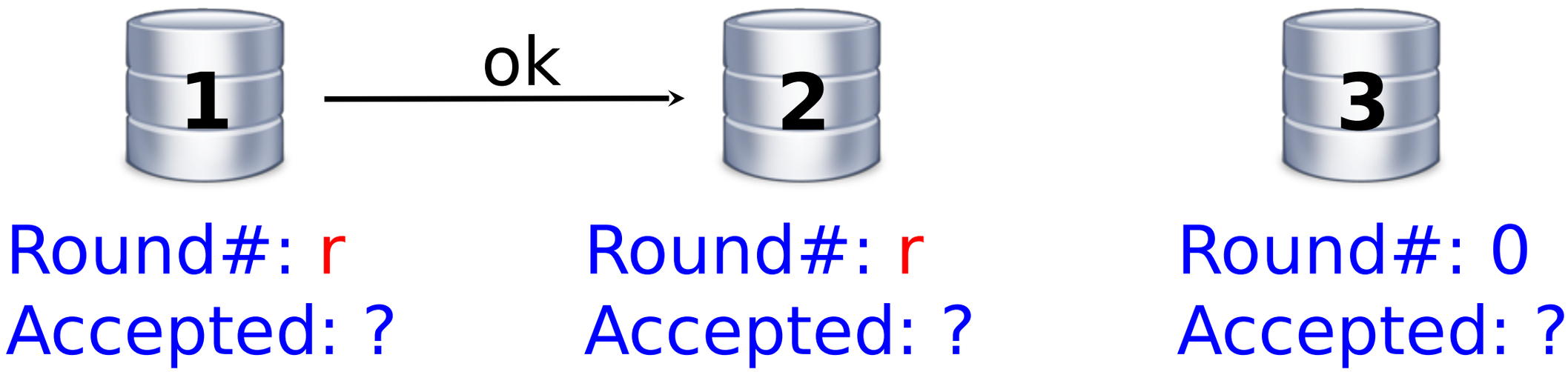
Round#: 0
Accepted: ?

Round#: 0
Accepted: ?

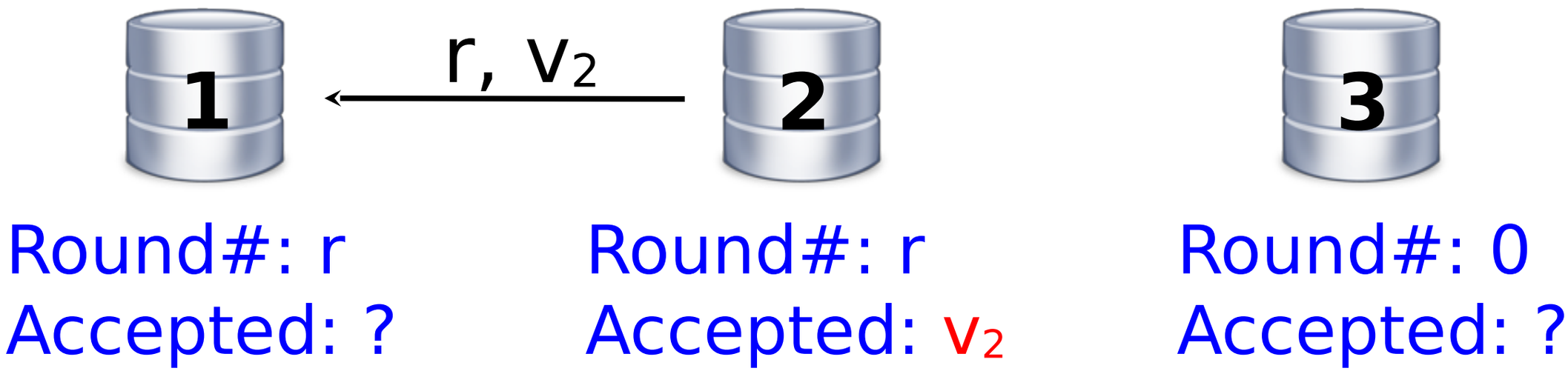
- Phase 1: a proposer choses a round r and convinces a majority of acceptors to switch to r
- Acceptor switches only if it’s current round is less



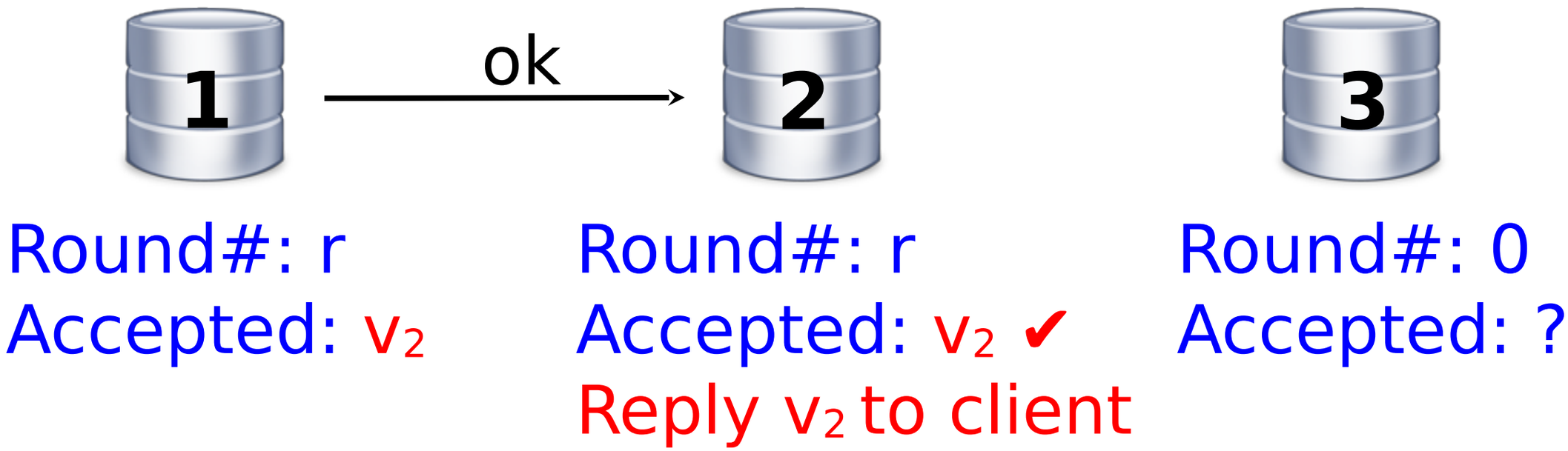
- Phase 1: a proposer choses a round r and convinces a majority of acceptors to switch to r
- Acceptor switches only if it’s current round is less



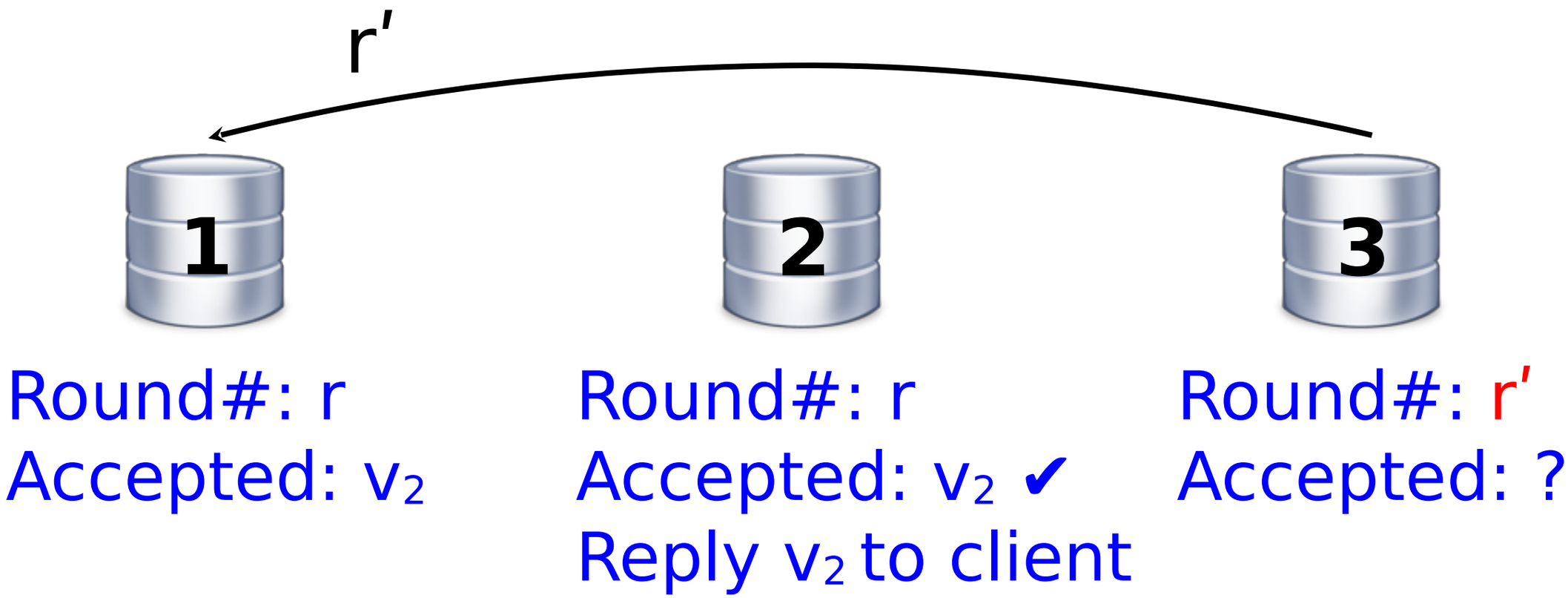
- Phase 1: a proposer choses a round r and convinces a majority of acceptors to switch to r
- Acceptor switches only if it’s current round is less



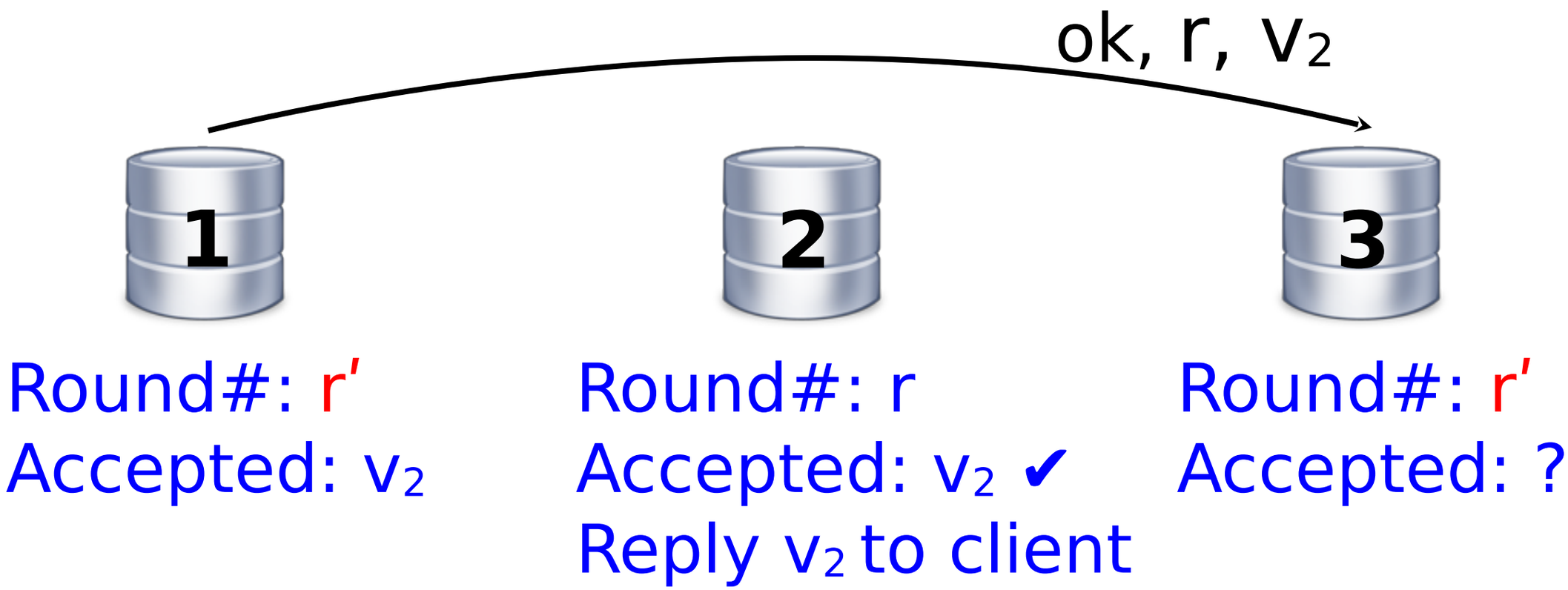
- Phase 2: the proposer sends its value tagged with the round number
- Acceptor only accepts a value tagged with the round it is in



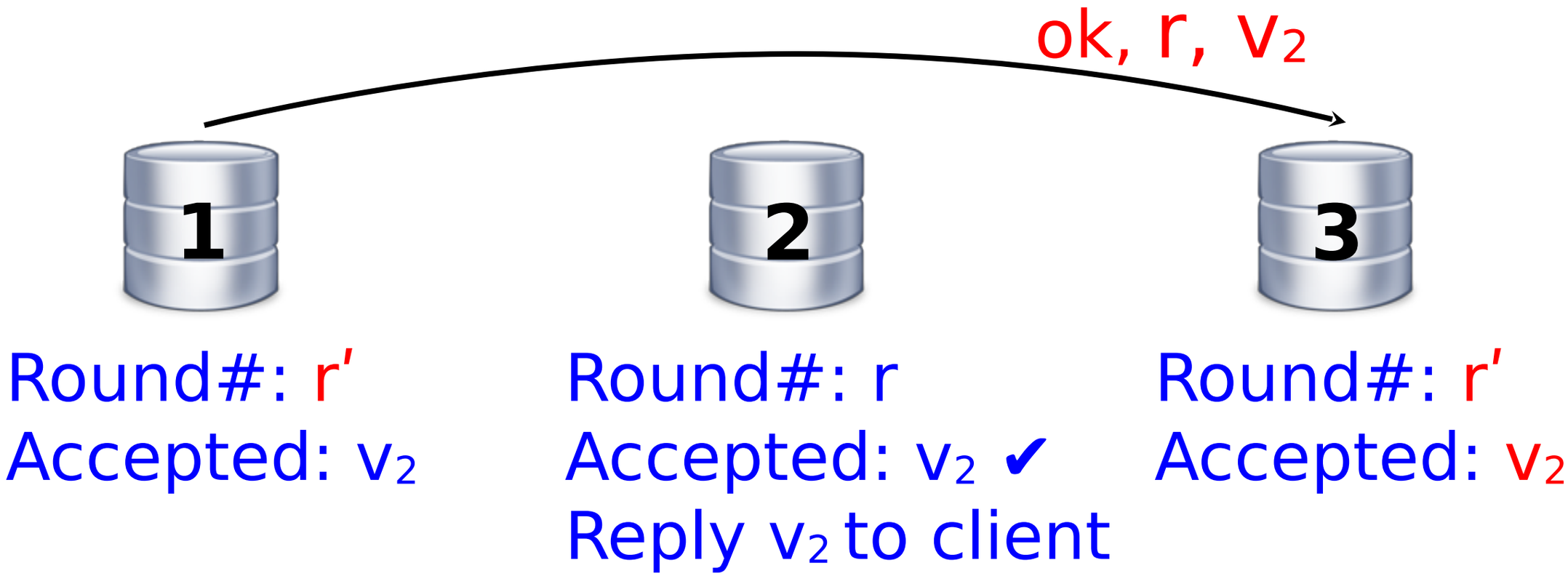
- Phase 2: the proposer sends its value tagged with the round number
- Acceptor only accepts a value tagged with the round it is in



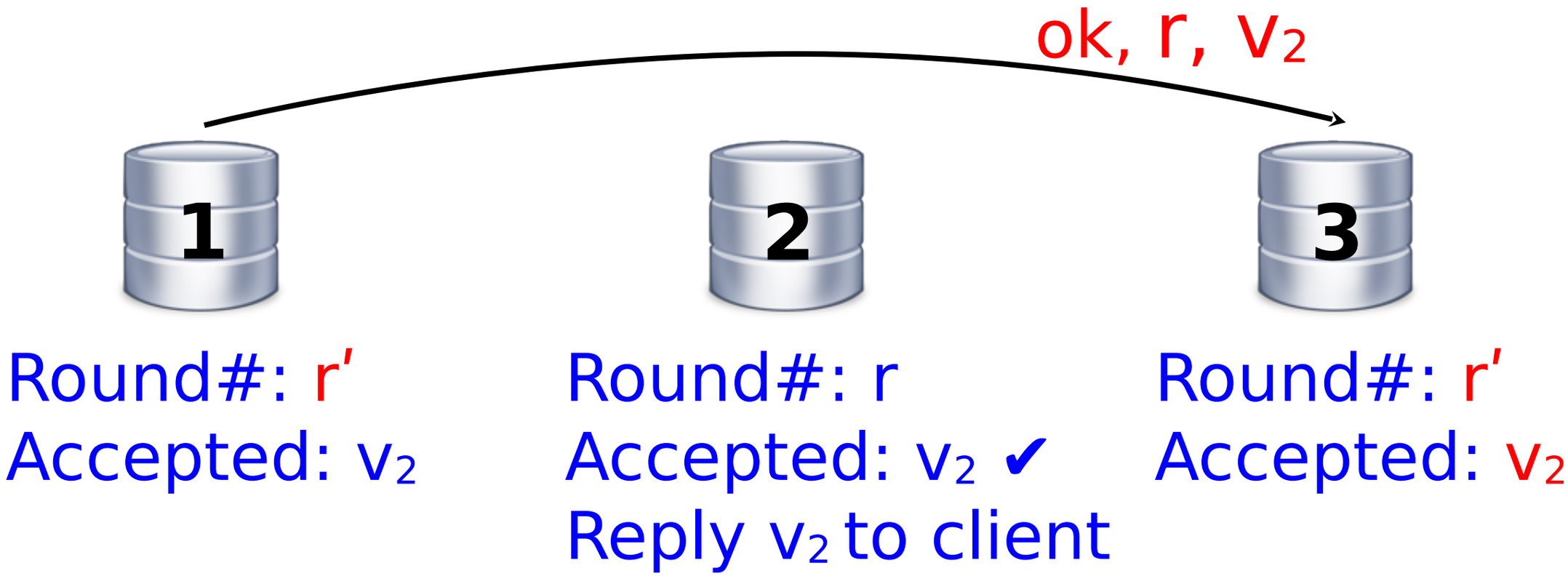
- Phase 1: a proposer choses a round r’ and convinces a majority of acceptors to switch to r’



- Phase 1: a proposer choses a round r’ and convinces a majority of acceptors to switch to r’
- Acceptor sends to the proposer its round number and value



- Phase 1: a proposer choses a round r’ and convinces a majority of acceptors to switch to r’
- Acceptor sends to the proposer its round number and value
- If some acceptor has accepted a value, the proposer proposes the value with the highest round number



- Phase 1: a proposer choses a round r' and co
sw
- Ac nd number and value
- If some acceptor has accepted a value, the proposer proposes the value with the highest round number

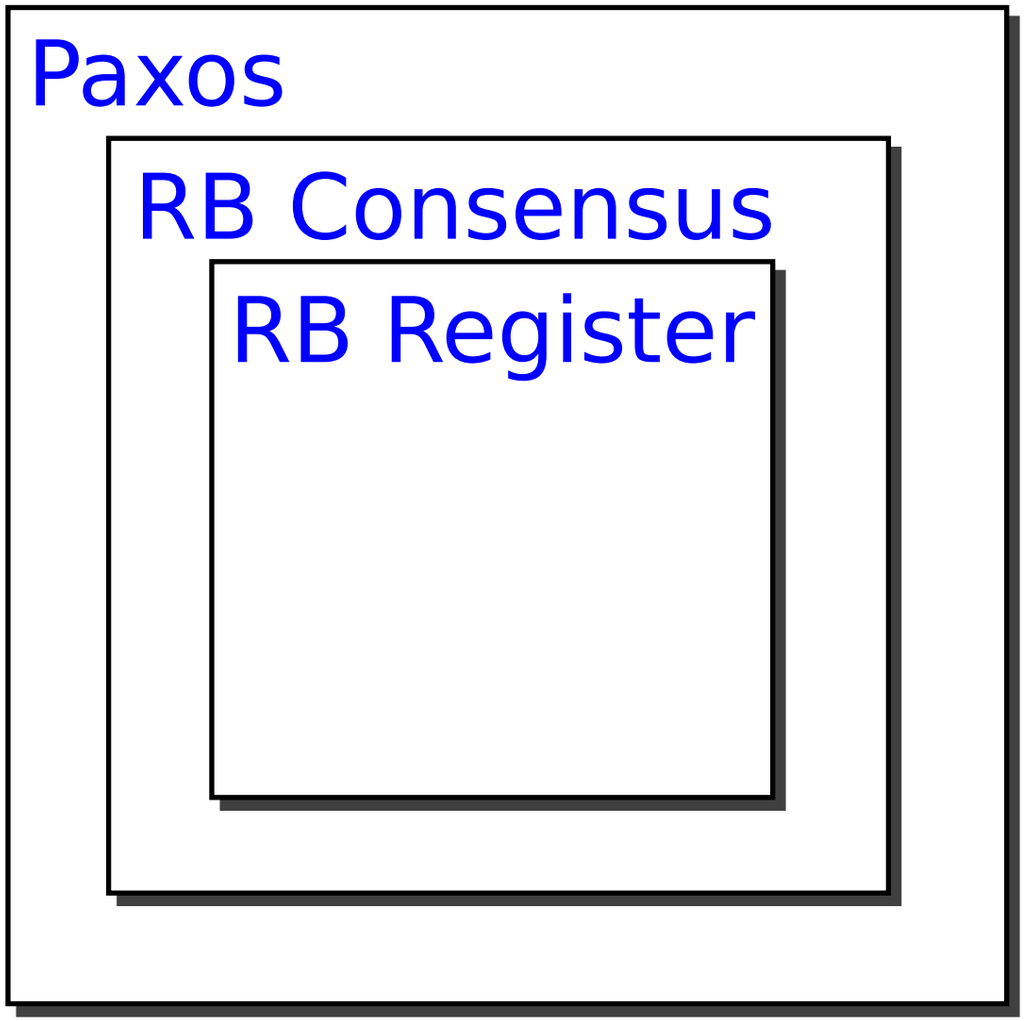
Ensures that the chosen value v_2 will not be changed

Modular structure in single-decree Paxos

- Steal abstractions from an existing analysis of Paxos [Boichat+ 2003]
- Show their linearizability → modular proof of Paxos

Round Based Register

[Boichat+ 2003]



- Data type encapsulating the state of acceptors
- `read(int k)`

  Phase 1 of Paxos
- `write(int k, val v)`

  Phase 2 of Paxos

Round Based Register - read

```
read(int k) {
  query acceptors and switch them to round k;
  if (majority of acceptors acknowledge) {
    if (no acceptor has accepted a value) {
      return (false, undef);
    }
    else {
      v ::= value at acceptor with highest round;
      return (true, v);
    }
  }
  else { return (false, undef); }
}
```

Round Based Register - read

```
read(int k) {
  query acceptors and switch them to round k;
  if (majority of acceptors acknowledge) {
    if (no acceptor has accepted a value) {
      return (false, undef);
    }
    else {
      v ::= value at acceptor with highest round;
      return (true, v);
    }
  }
  else { return (false, undef); }
}
```

Round Based Register - read

```
read(int k) {
  query acceptors and switch them to round k;
  if (majority of acceptors acknowledge) {
    if (no acceptor has accepted a value) {
      return (false, undef);
    }
    else {
      v ::= value at acceptor with highest round;
      return (true, v);
    }
  }
  else { return (false, undef); }
}
```

Round Based Register - write

```
write(int k, val v) {
  update acceptors at round k with value v;
  if (majority of acceptors acknowledges) {
    return true;
  }
  else {
    return false;
  }
}
```

Round Based Register - write

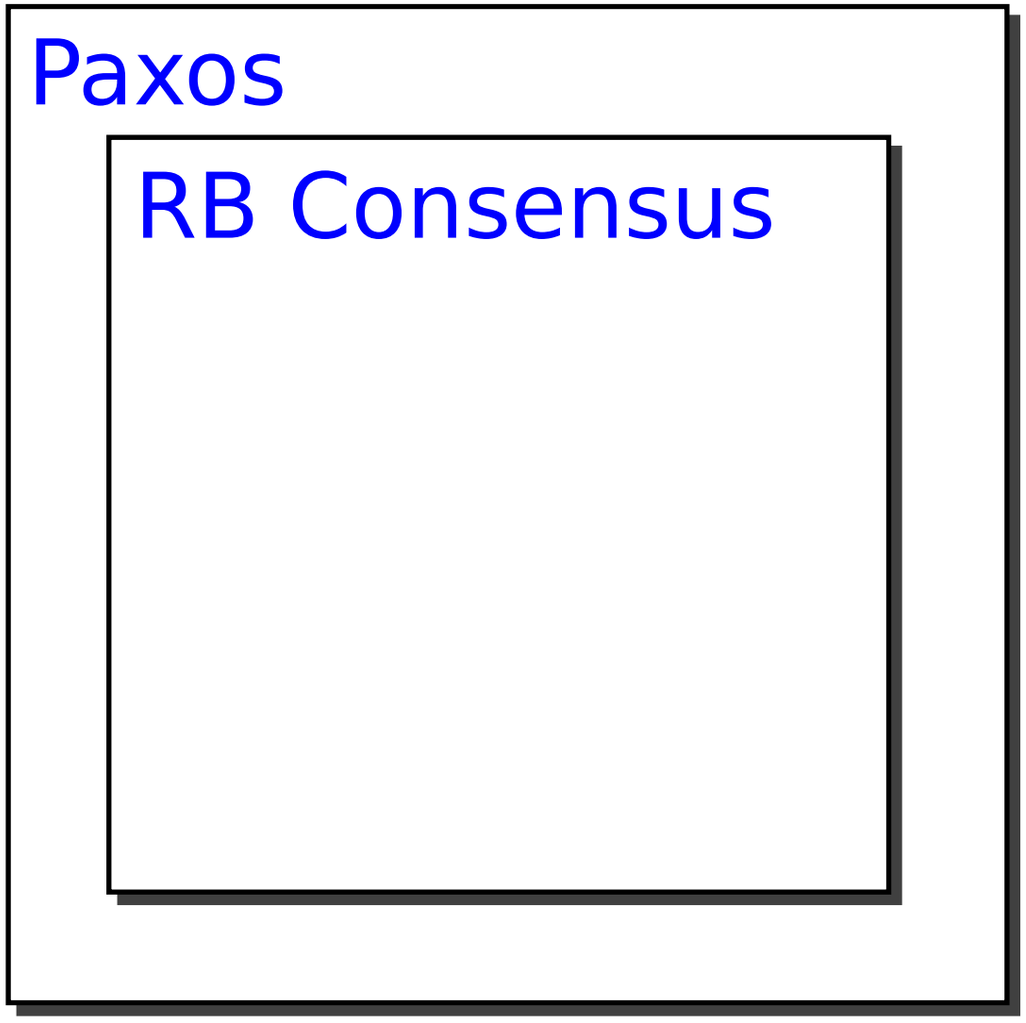
```
write(int k, val v) {
  update acceptors at round k with value v;
  if (majority of acceptors acknowledges) {
    return true;
  }
  else {
    return false;
  }
}
```

Round Based Register - write

```
write(int k, val v) {
  update acceptors at round k with value v;
  if (majority of acceptors acknowledges) {
    return true;
  }
  else {
    return false;
  }
}
```

Round Based Consensus

[Boichat+ 2003]



- Routine leading Phase 1 and Phase 2 in the Paxos algorithm
- `proposeRC(int k, val v)`

Round Based Consensus - proposeRC

```
proposeRC(int k, val v0) {
  (res, v) := read(k);
  if (res) {
    if (v = undef) {
      v := v0;
    }
    res := write(k, v);
    if (res) { return (true, v); }
  }
  return (false, undef);
}
```

Round Based Consensus - proposeRC

```
proposeRC(int k, val v0) {
  (res, v) := read(k);
  if (res) {
    if (v = undef) {
      v := v0;
    }
    res := write(k, v);
    if (res) { return (true, v); }
  }
  return (false, undef);
}
```

Paxos

- Entry module, encapsulates rounds
- `proposeP(val v)`

Paxos - proposeP

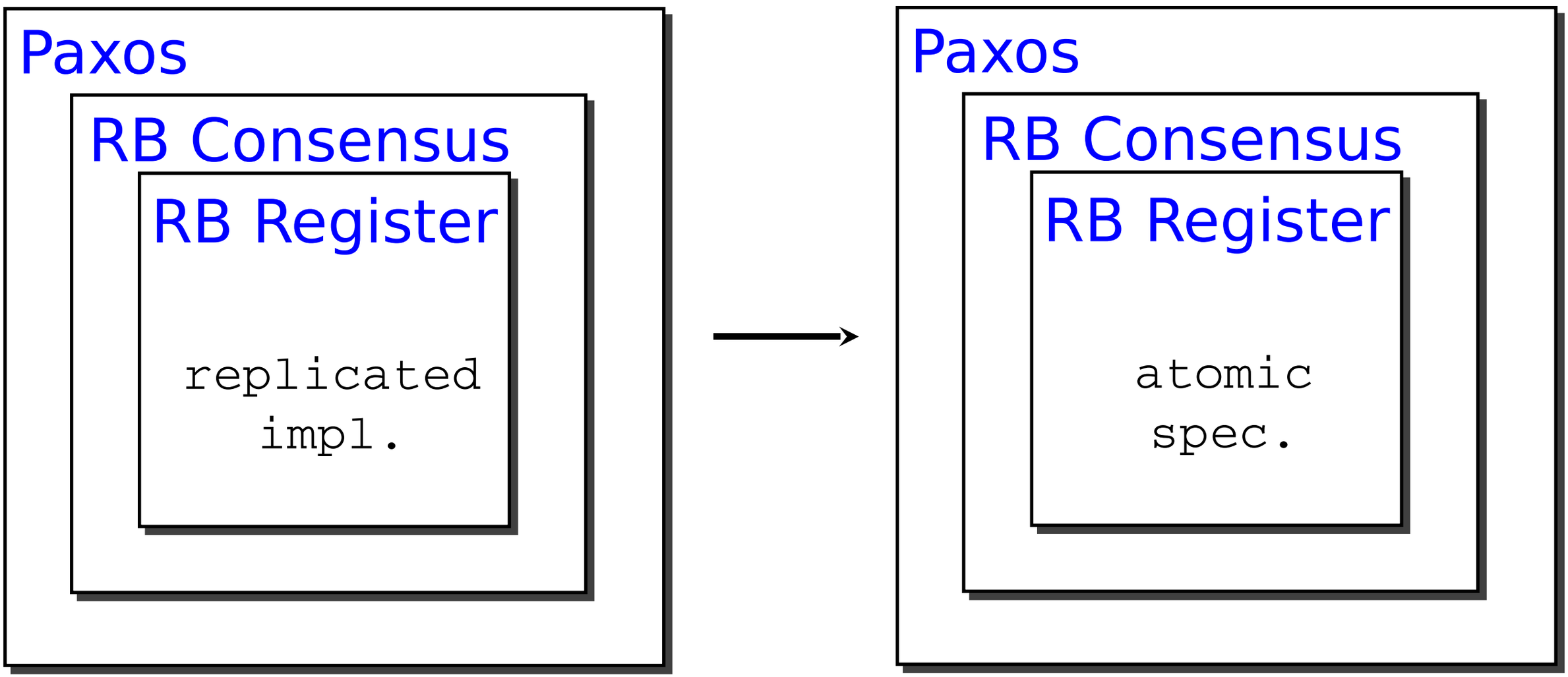
```
proposeP(val v0) {
  pick a round k;
  do {
    (res, v) := proposeRC(k, v0);
    increase round k;
  } while (!res);
  return v;
}
```

Paxos - proposeP

```
proposeP(val v0) {
  pick a round k;
  do {
    (res, v) := proposeRC(k, v0);
    increase round k;
  } while (!res);
  return v;
}
```

Contribution

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



```
abs_v := undef;
abs_round := 0;
vals := {undef};
```

```
abs_v := undef;
abs_round := 0;
vals := {undef};
```

Atomic, non-deterministic methods

```
read(int k) {
  atomic {
    pick random vR from vals;
    pick random boolean b;
    if (b) {
      if (k >= abs_round) {
        abs_round := k;
        if (!(abs_v = undef)) {
          v := abs_v;
        }
        else { v := vR; }
      }
      else { v := vR; }
      return (true, v);
    }
    else {
      return (false, undef);
    }
  }
}
```

```
write(int k, val vW) {
  atomic {
    pick random boolean b;
    vals := vals U {vW};
    if (b && (k >= abs_round)) {
      abs_round := k;
      abs_v := vW;
      return true;
    }
    else {
      return false;
    }
  }
}
```

```
abs_v := undef;
abs_round := 0;
vals := {undef};
```

Every proposed value vW is stored in vals, whether write fails or no

```
read(int k) {
  atomic {
    pick random vR from vals;
    pick random boolean b;
    if (b) {
      if (k >= abs_round) {
        abs_round := k;
        if (!(abs_v = undef)) {
          v := abs_v;
        }
        else { v := vR; }
      }
      else { v := vR; }
      return (true, v);
    }
    else {
      return (false, undef);
    }
  }
}
```

```
write(int k, val vW) {
  atomic {
    pick random boolean b;
    vals := vals U {vW};
    if (b && (k >= abs_round)) {
      abs_round := k;
      abs_v := vW;
      return true;
    }
    else {
      return false;
    }
  }
}
```

```
abs_v := undef;
abs_round := 0;
vals := {undef};
```

Write can fail even if `k` is higher or equal than the current round
➜ a round will be “stolen” by posterior write, modeled by boolean `b`

```
read(int k) {
  atomic {
    pick random vR from vals;
    pick random boolean b;
    if (b) {
      if (k >= abs_round) {
        abs_round := k;
        if (!(abs_v = undef)) {
          v := abs_v;
        }
        else { v := vR; }
      }
      else { v := vR; }
      return (true, v);
    }
    else {
      return (false, undef);
    }
  }
}
```

```
write(int k, val vW) {
  atomic {
    pick random boolean b;
    vals := vals U {vW};
    if (b && (k >= abs_round)) {
      abs_round := k;
      abs_v := vW;
      return true;
    }
    else {
      return false;
    }
  }
}
```

```
abs_v := undef;
abs_round := 0;
vals := {undef};
```

Read may succeed even if `k` is lower than the current round
➜ a failing write "contaminates" some acceptor, modeled by value `vR` and boolean `b`

```
read(int k) {
  atomic {
    pick random vR from vals;
    pick random boolean b;
    if (b) {
      if (k >= abs_round) {
        abs_round := k;
        if (!(abs_v = undef)) {
          v := abs_v;
        }
        else { v := vR; }
      }
      else { v := vR; }
      return (true, v);
    }
    else {
      return (false, undef);
    }
  }
}
```

```
write(int k, val vW) {
  atomic {
    pick random boolean b;
    vals := vals U {vW};
    if (b && (k >= abs_round)) {
      abs_round := k;
      abs_v := vW;
      return true;
    }
    else {
      return false;
    }
  }
}
```

```
abs_v := undef;
abs_round := 0;
vals := {undef};
```

If a write succeeds, a succeeding read will pick the written value
➜ a decision taken in consensus cannot be changed

```
read(int k) {
  atomic {
    pick random vR from vals;
    pick random boolean b;
    if (b) {
      if (k >= abs_round) {
        abs_round := k;
        if (!(abs_v = undef)) {
          v := abs_v;
        }
        else { v := vR; }
      }
      else { v := vR; }
      return (true, v);
    }
    else {
      return (false, undef);
    }
  }
}
```

```
write(int k, val vW) {
  atomic {
    pick random boolean b;
    vals := vals U {vW};
    if (b && (k >= abs_round)) {
      abs_round := k;
      abs_v := vW;
      return true;
    }
    else {
      return false;
    }
  }
}
```

```
abs_v := undef;
abs_round := 0;
vals := {undef};
```

If a write succeeds, a succeeding read will pick the written value
➜ a decision taken in consensus cannot be changed

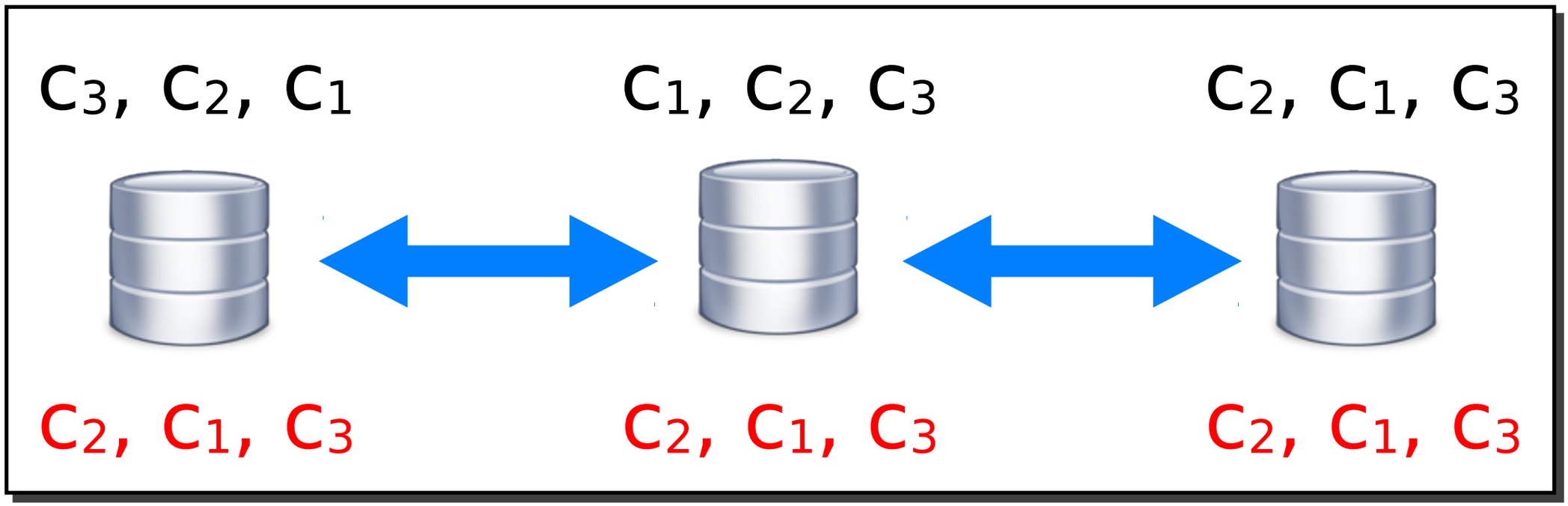
```
read(int k) {
  atomic {
    pick random vR from vals;
    pick random boolean b;
    if (b) {
      if (k >= abs_rou
        abs_round := k
        if (!(abs_v =
          v := abs_v;
        }
        else { v := vR
      }
      else { v := vR;
      return (true, v)
    }
    else {
      return (false, u
    }
  }
}
```

```
write(int k, val vW) {
  atomic {
    pick random boolean b;
    vals := vals U {vW};
    >= abs_round)) {
    := k;
    vW;
    ue;
    lse;
```

```
proposeRC(int k, val v0) {
  (res, v) := read(k);
  if (res) {
    if (v = undef) {
      v := v0;
    }
    res := write(k, v);
    if (res) {
      return (true, v);
    }
  }
  return (false, undef);
}
```

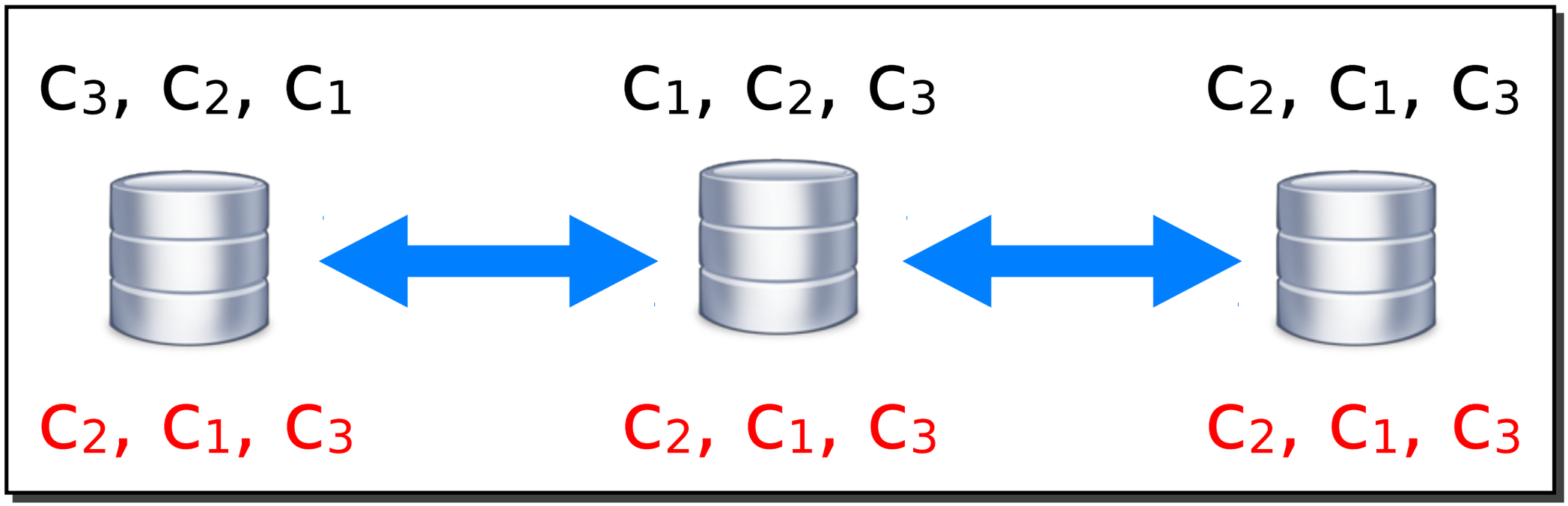
Multi-Paxos

State machine replication requires solving a sequence of consensus instances



Multi-Paxos

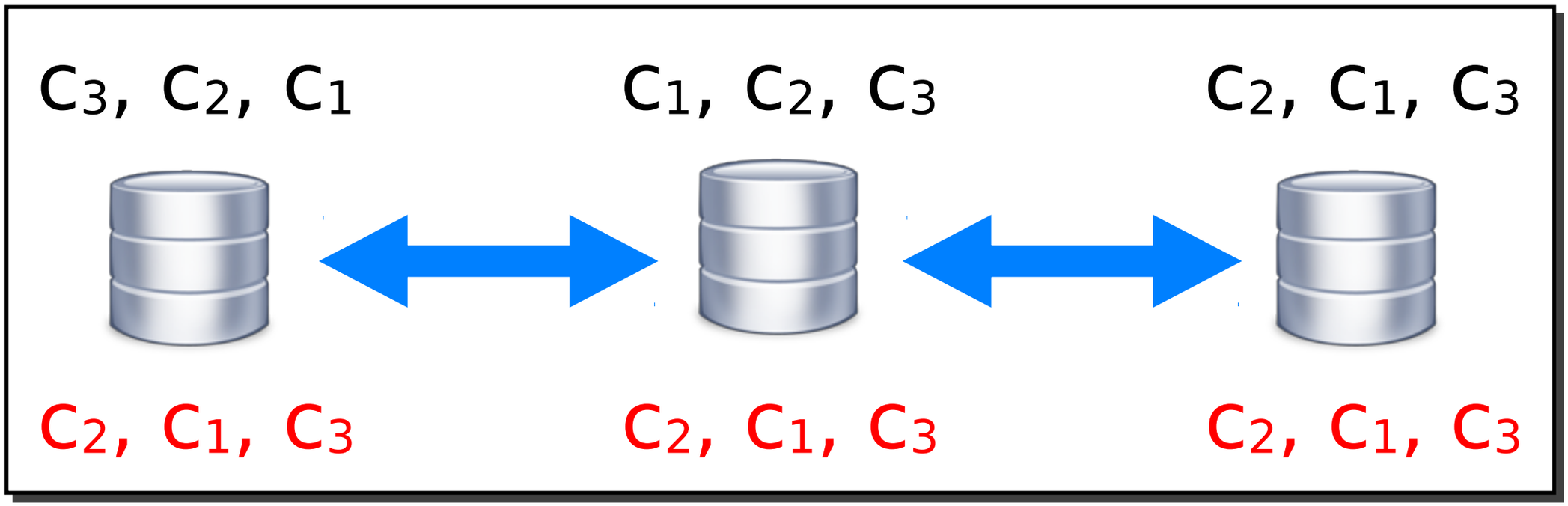
State machine replication requires solving a sequence of consensus instances



- Naive solution: execute a separate Paxos instance for each sequence element

Multi-Paxos

State machine replication requires solving a sequence of consensus instances

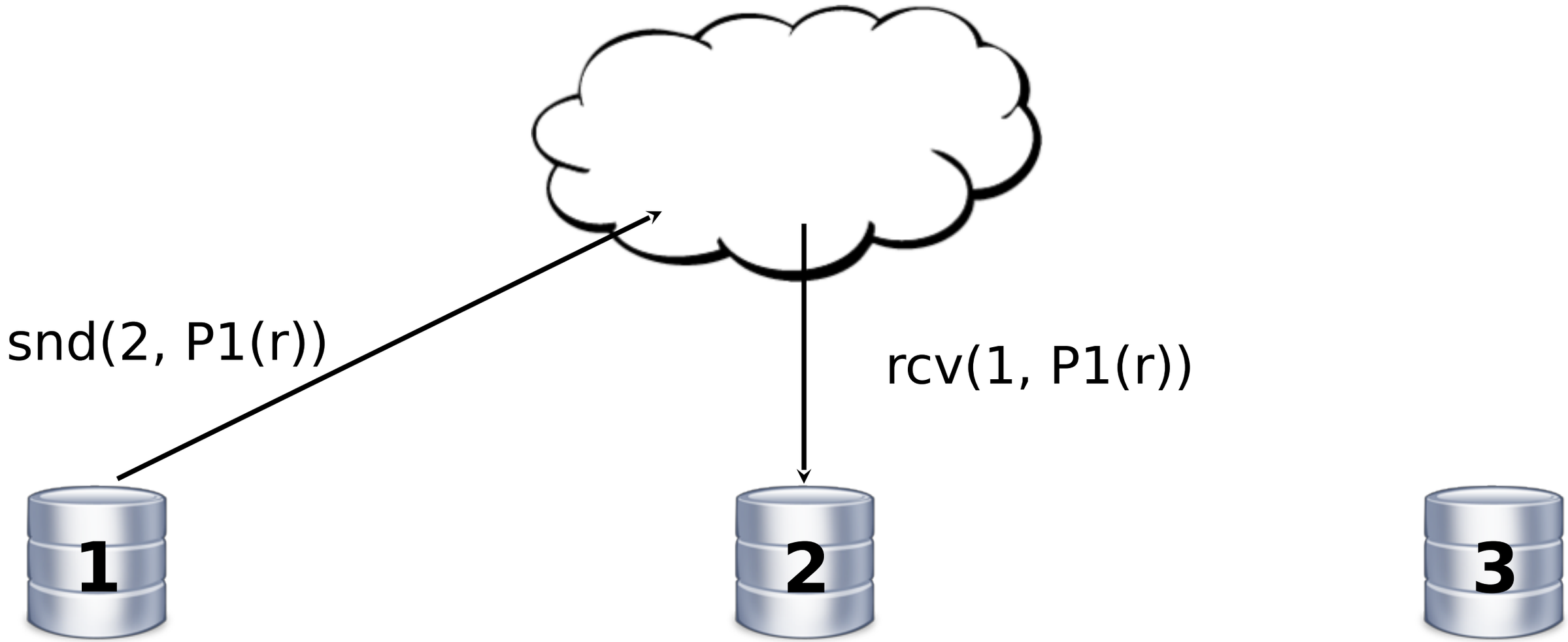


- Naive solution: execute a separate Paxos instance for each sequence element
- Multi-Paxos: execute Phase 1 once for multiple sequence elements

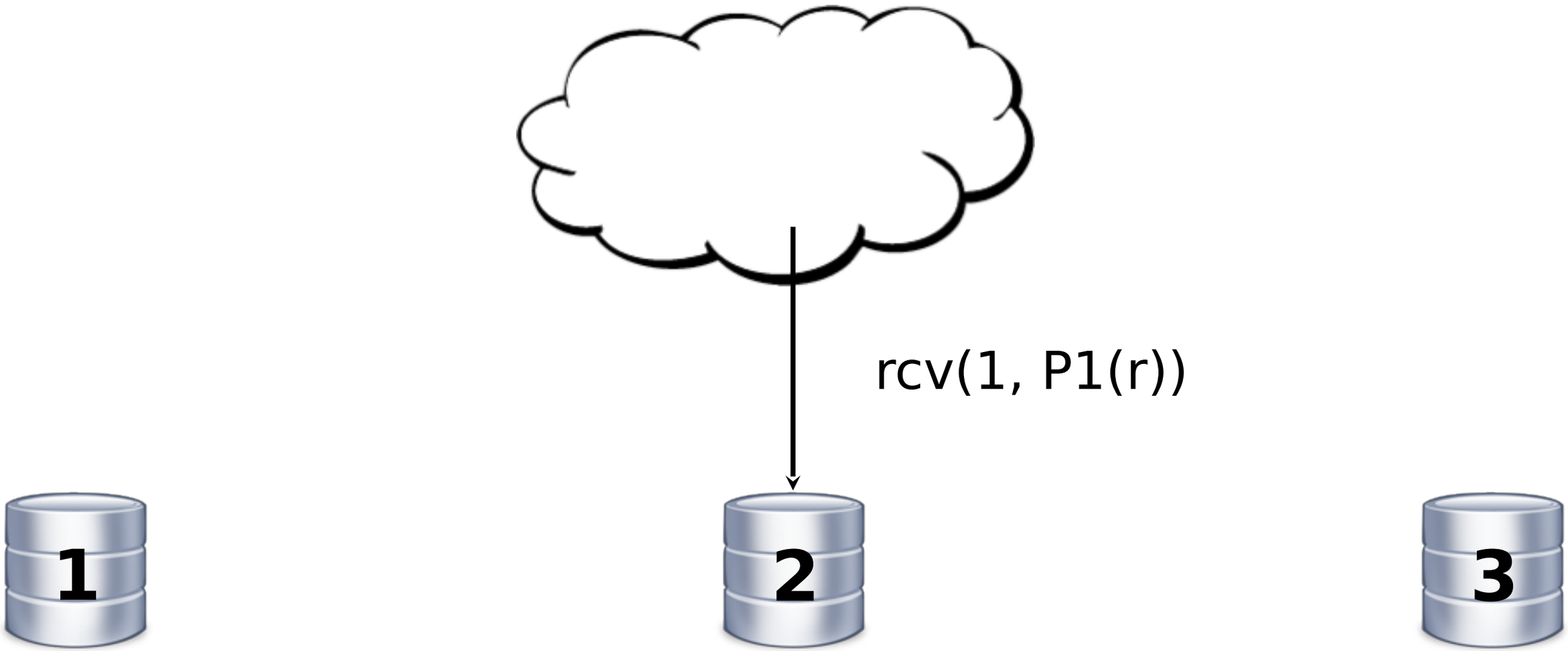
Contribution

Multi-Paxos refines the naive solution, shown by transformations of the network semantics à la Verdi [Wilcox+ 2015]

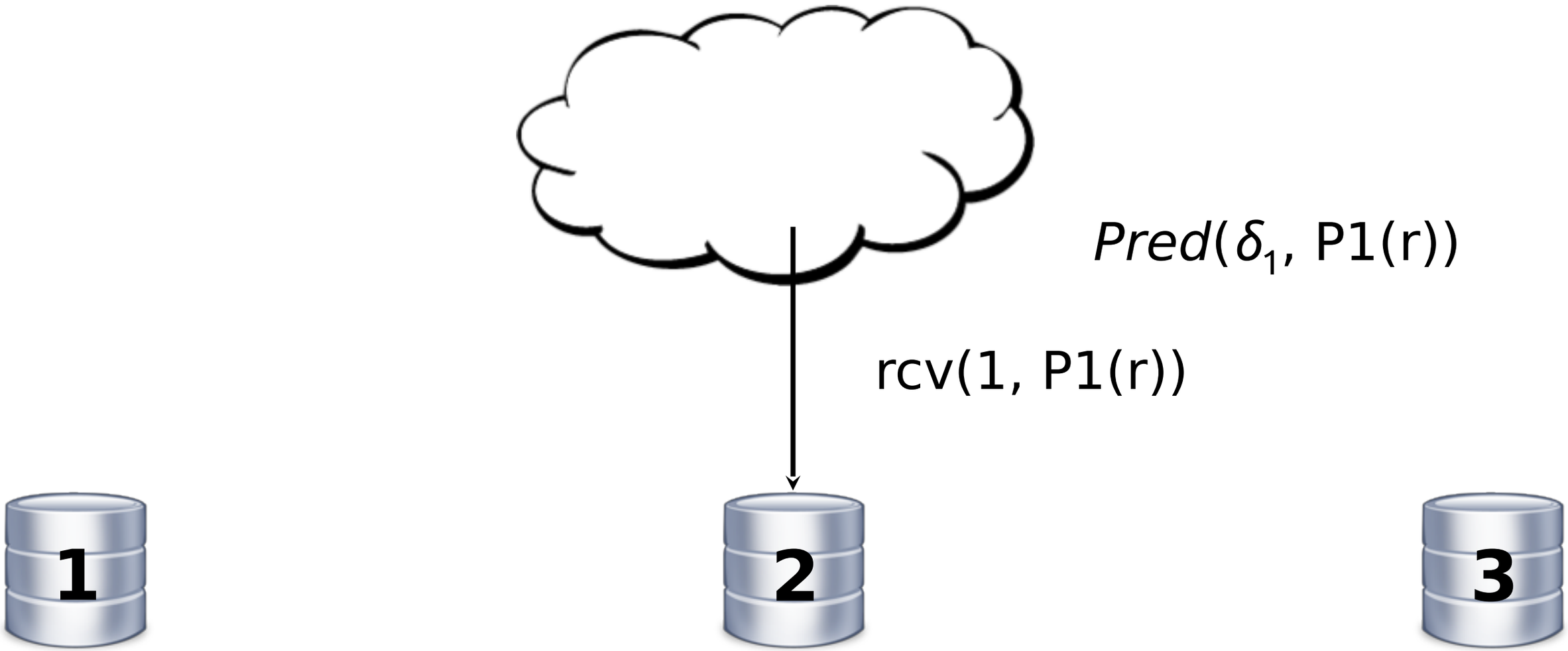
Simple Semantics



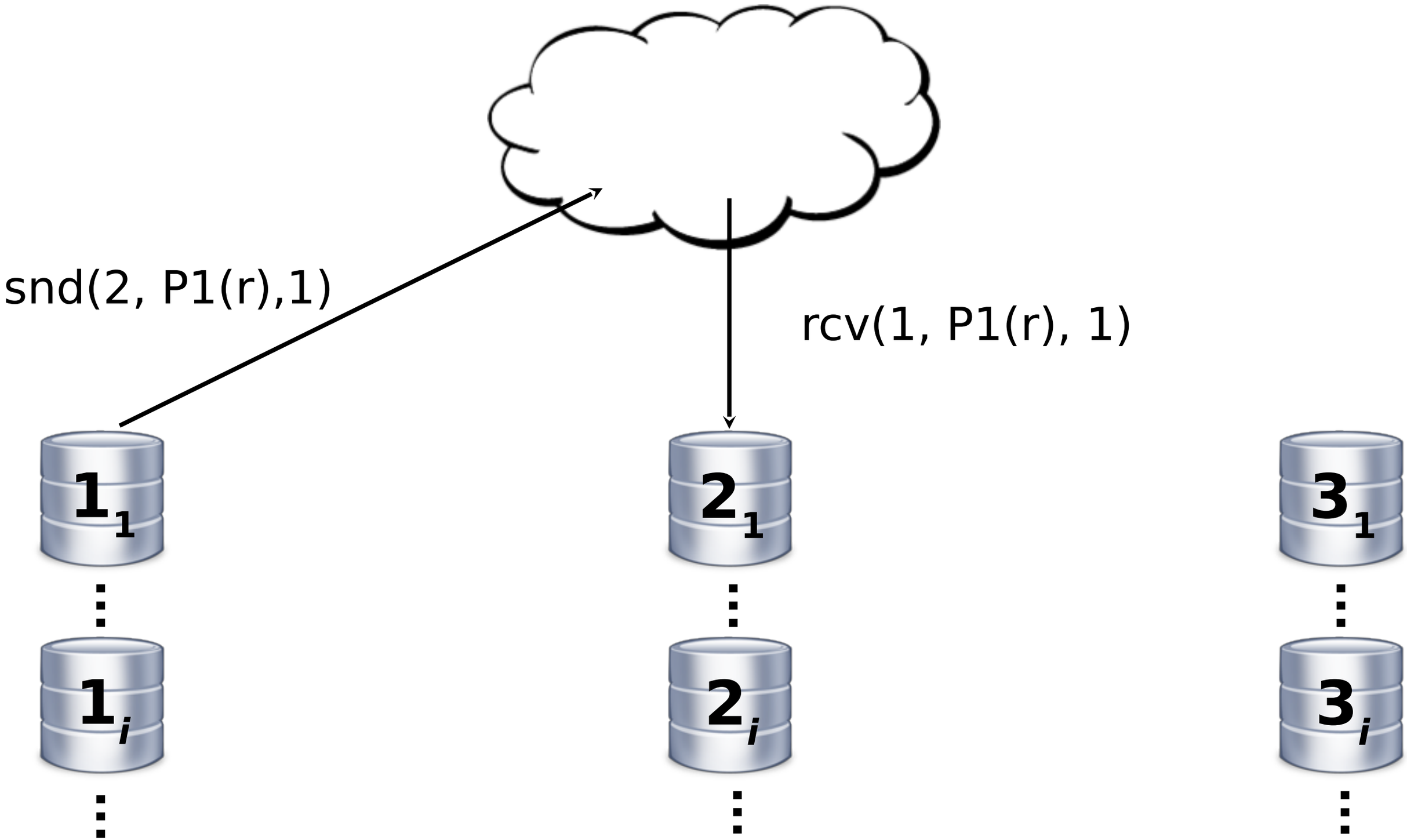
Out-of-Thin-Air Semantics



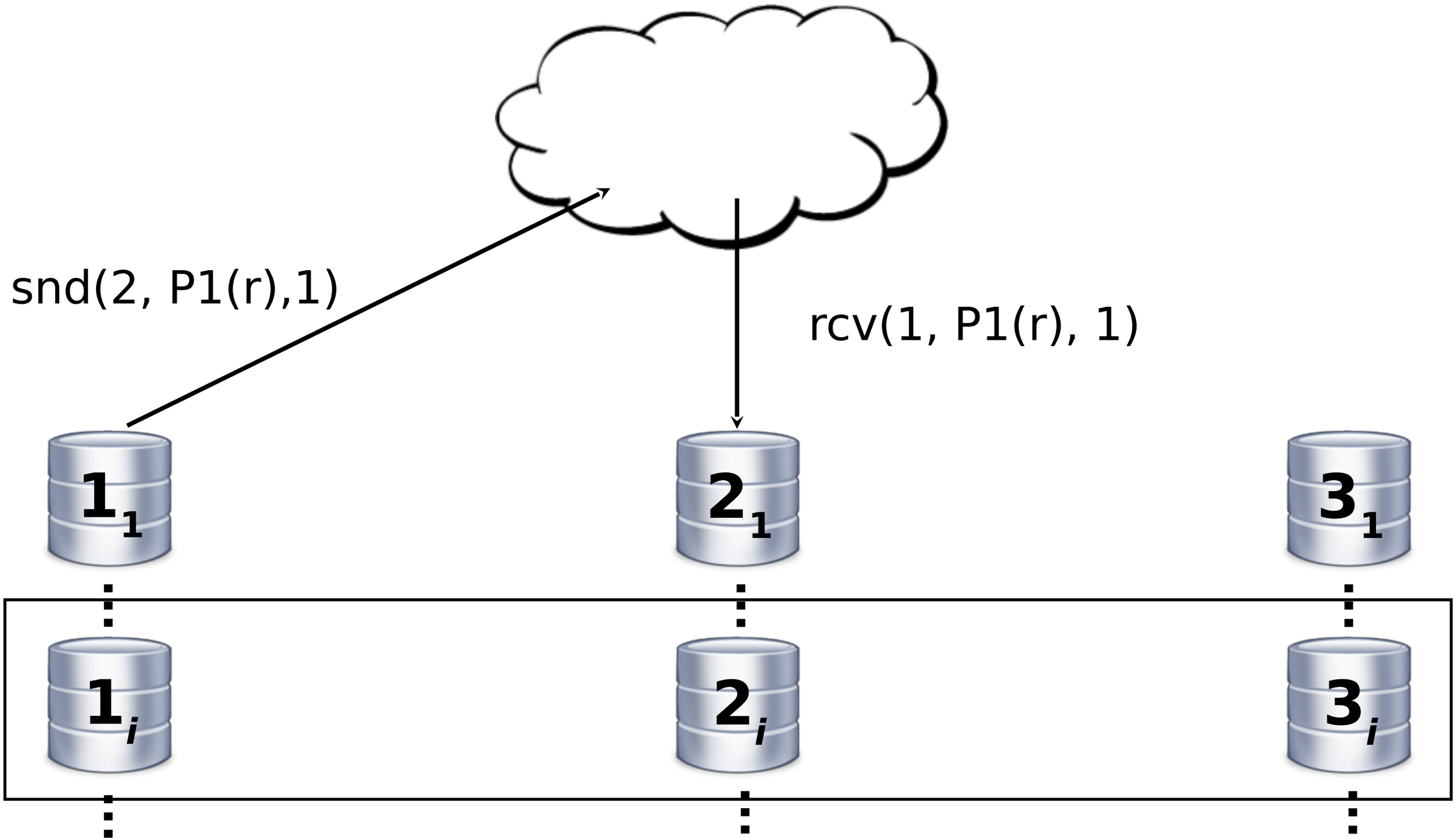
Out-of-Thin-Air Semantics



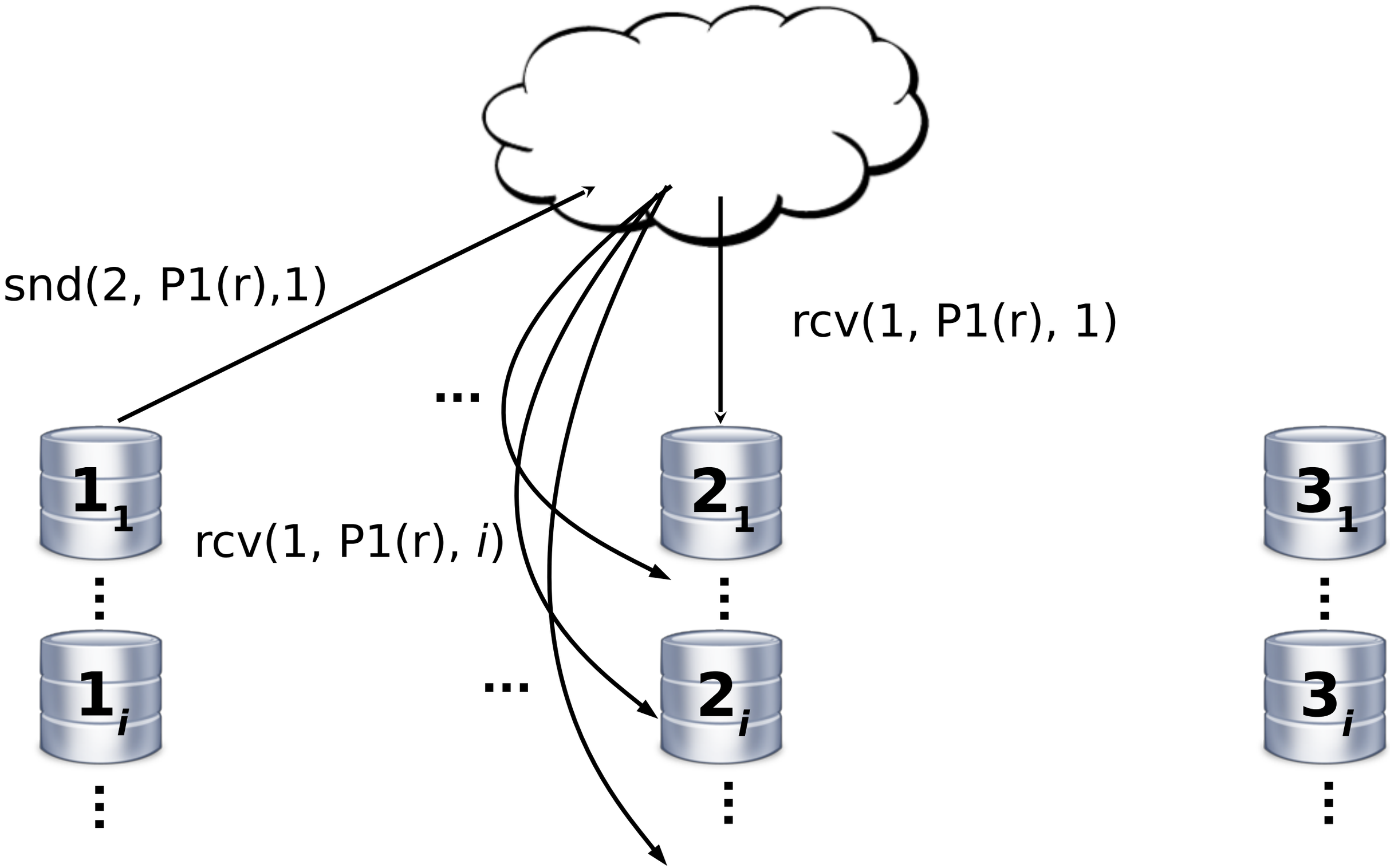
Slot-Replicating Semantics



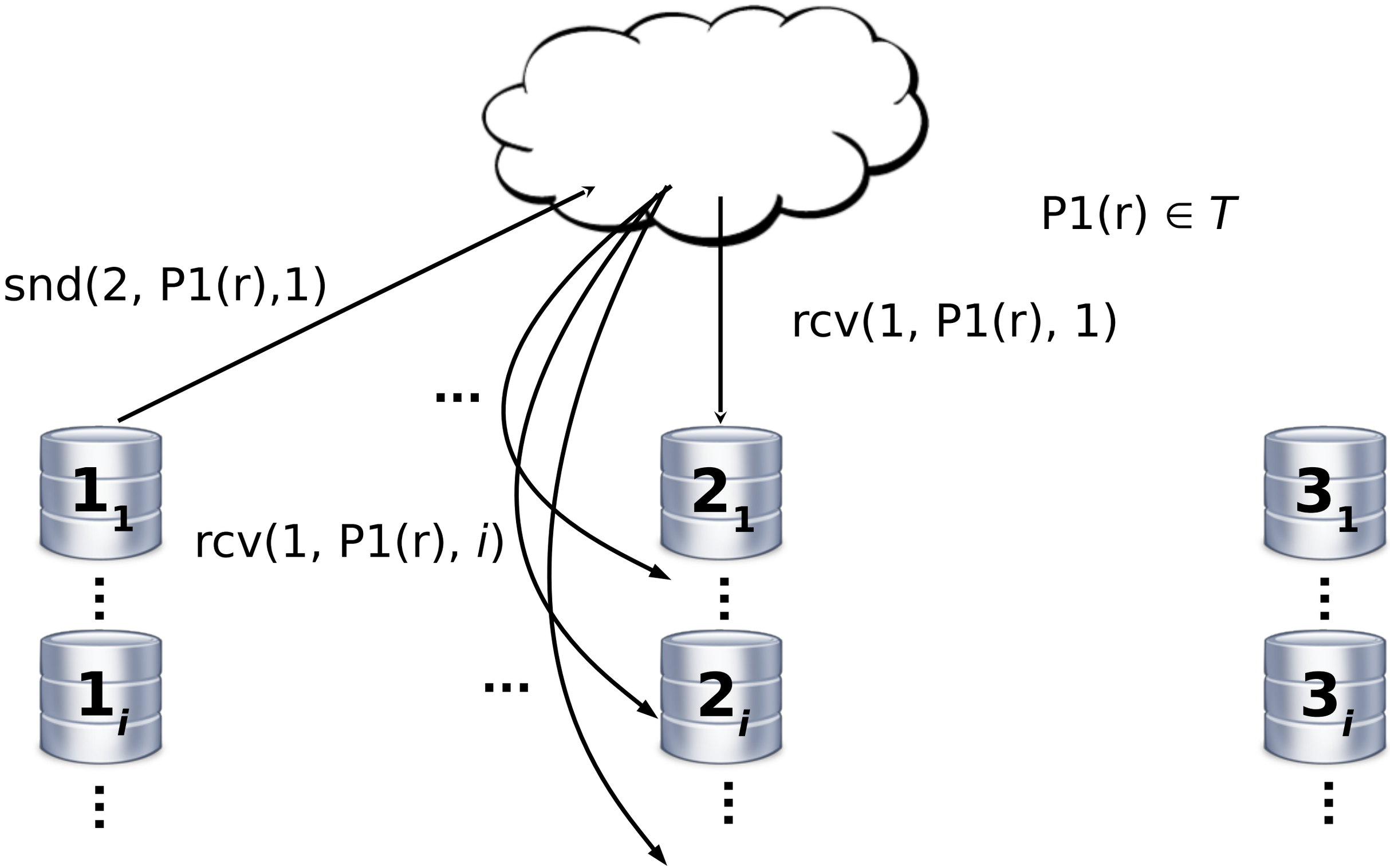
Slot-Replicating Semantics



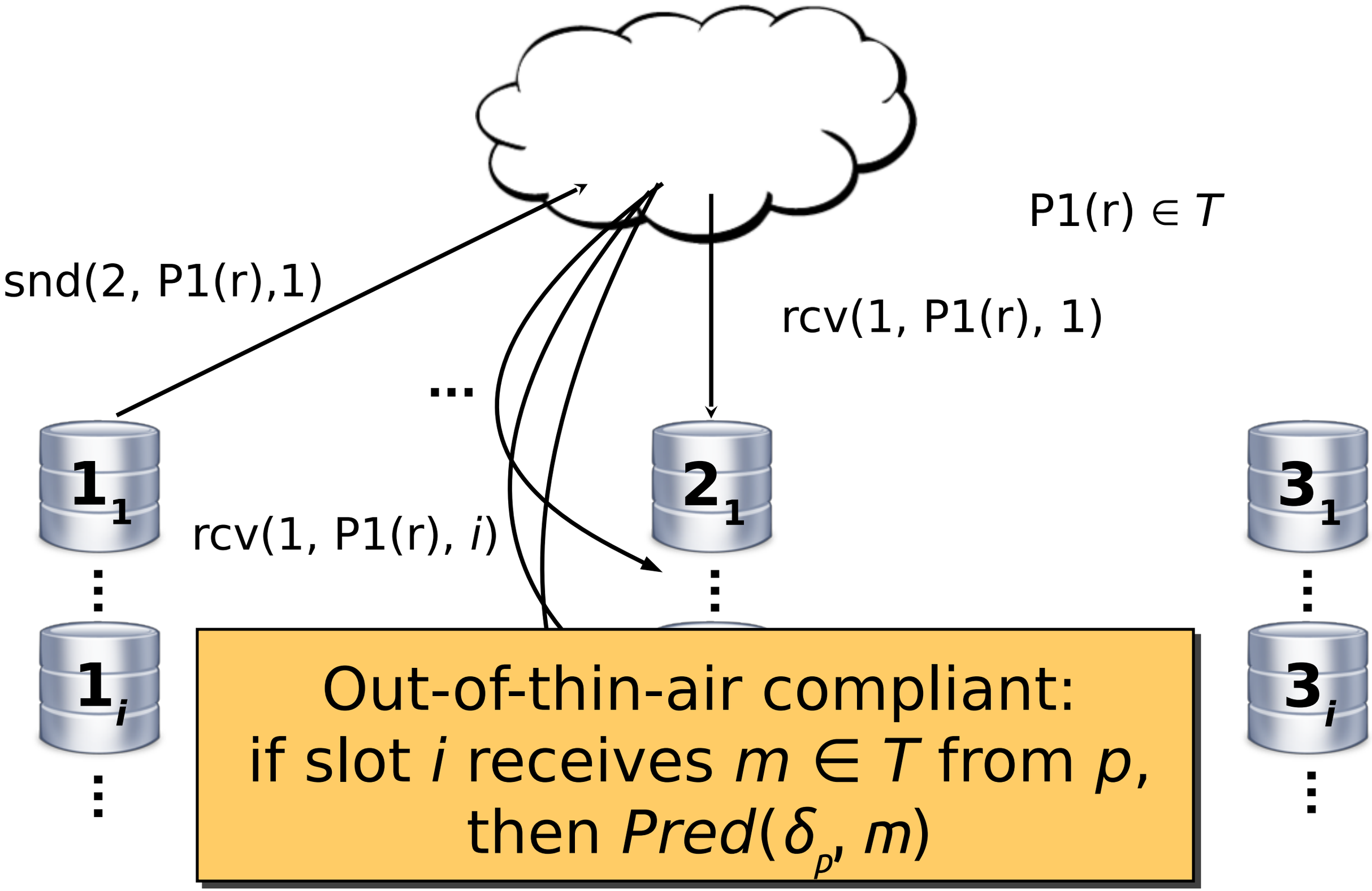
Widening Semantics



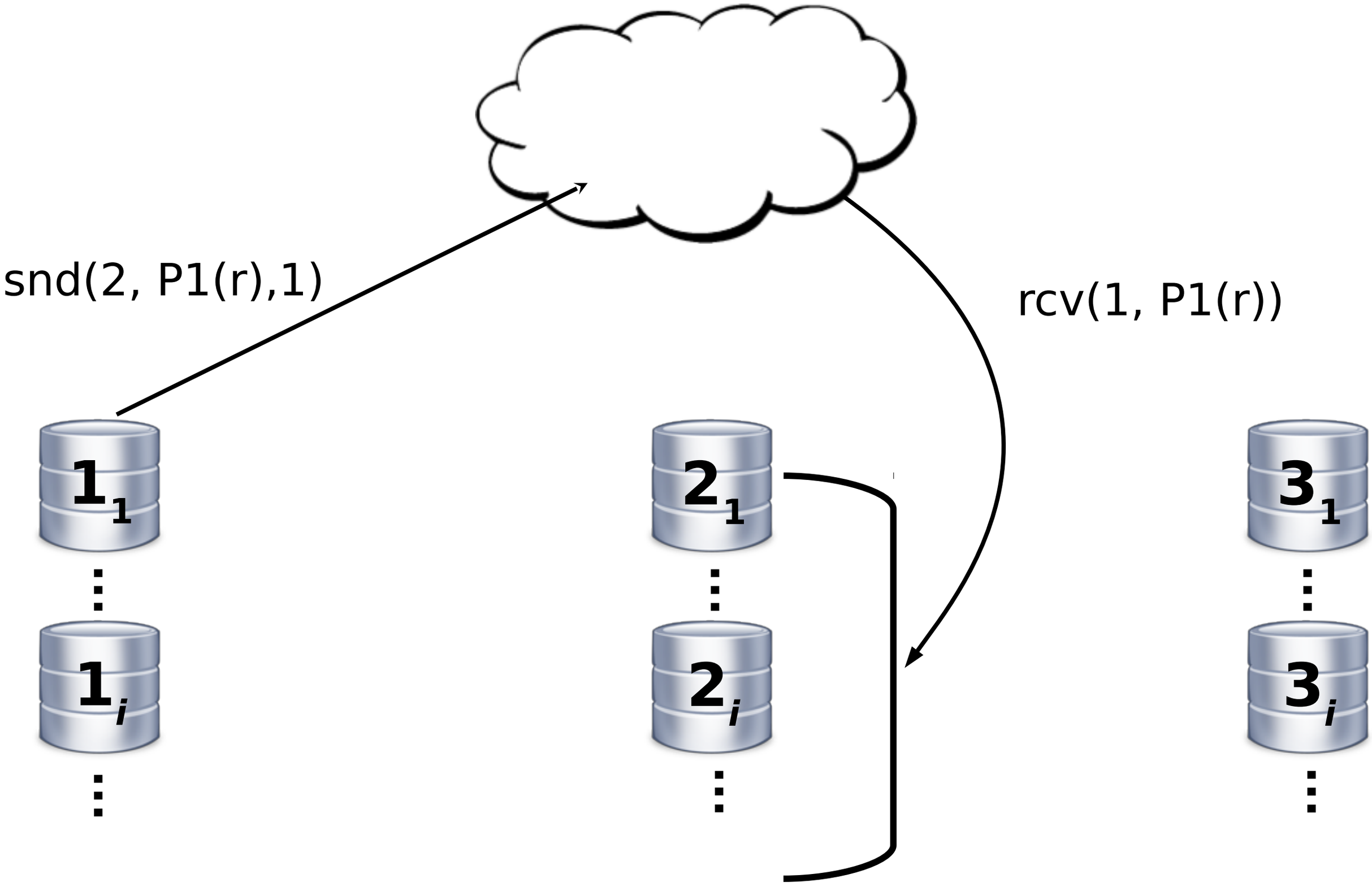
Widening Semantics



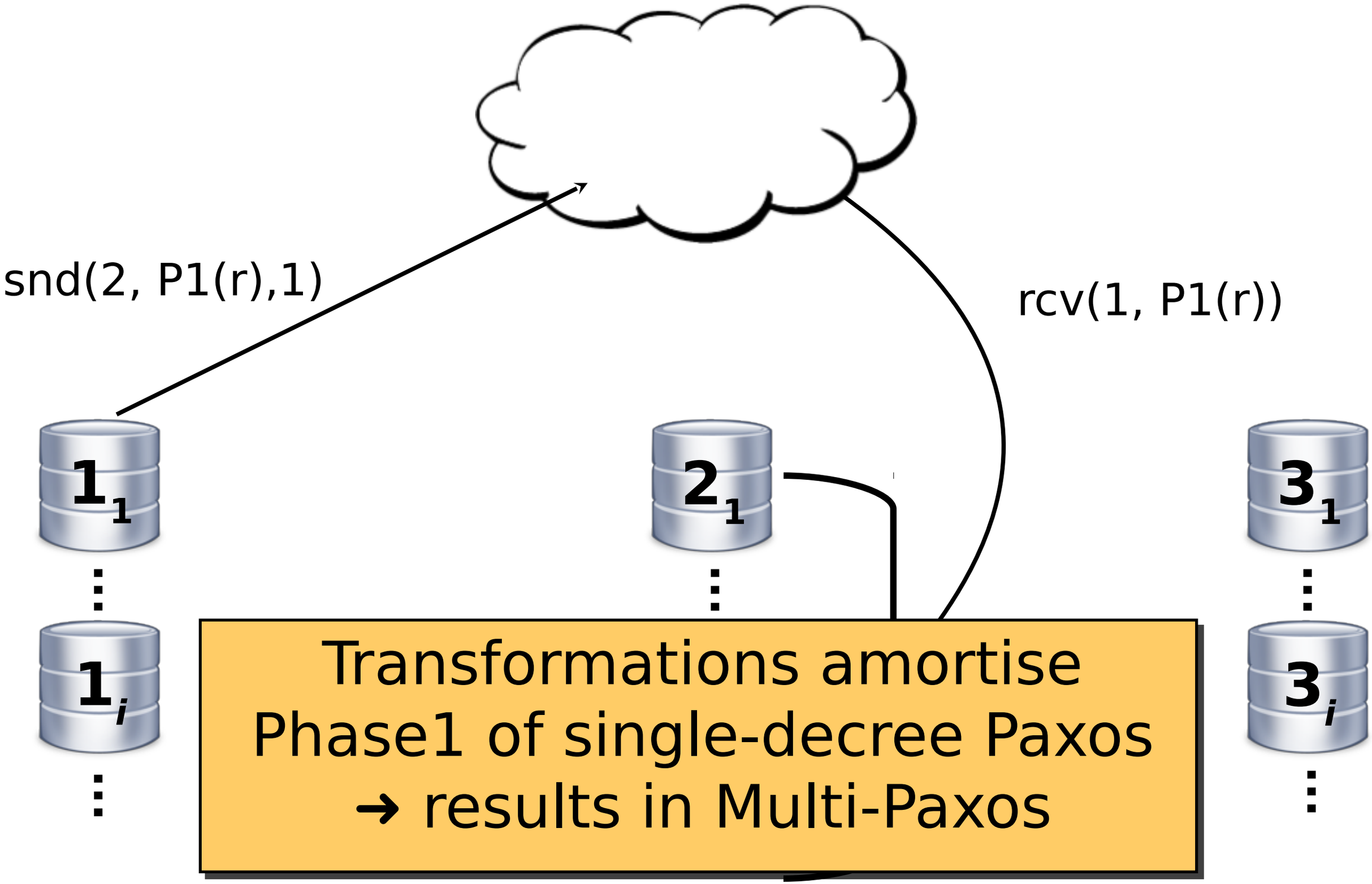
Widening Semantics



Optimised Widening Semantics



Optimised Widening Semantics



Summary

- Modular reasoning to verify each component separately
- Linearisability as a correctness criterium for refinement
- Deconstruction of single-decree Paxos by [Boichat+ 2003] linearises wrt non-deterministic specifications
- Behaviour-preserving transformations of the network semantics à la Verdi [Wilcox+ 2015]
- Multi-Paxos can be verified without unpacking the correctness proof of single-decree Paxos

Summary

- Modular reasoning to verify each component separately
- Linearisability as a correctness criterium for refinement
- Deconstruction of single-decree Paxos by [Boichat+ 2003] linearises wrt non-deterministic specifications
- Behaviour-preserving transformations of the network semantics à la Verdi [Wilcox+ 2015]
- Multi-Paxos can be verified without unpacking the correctness proof of single-decree Paxos

Thanks!