Technische Universität Kaiserslautern

Weakening consistency for scalable information systems

Arnd Poetzsch-Heffter

Based on work of Annette Bieniusa

together with

Marc Shapiro, Marek Zawirski (INRIA & LIP6)

Nuno Preguiça, Sérgio Duarte (UNL)

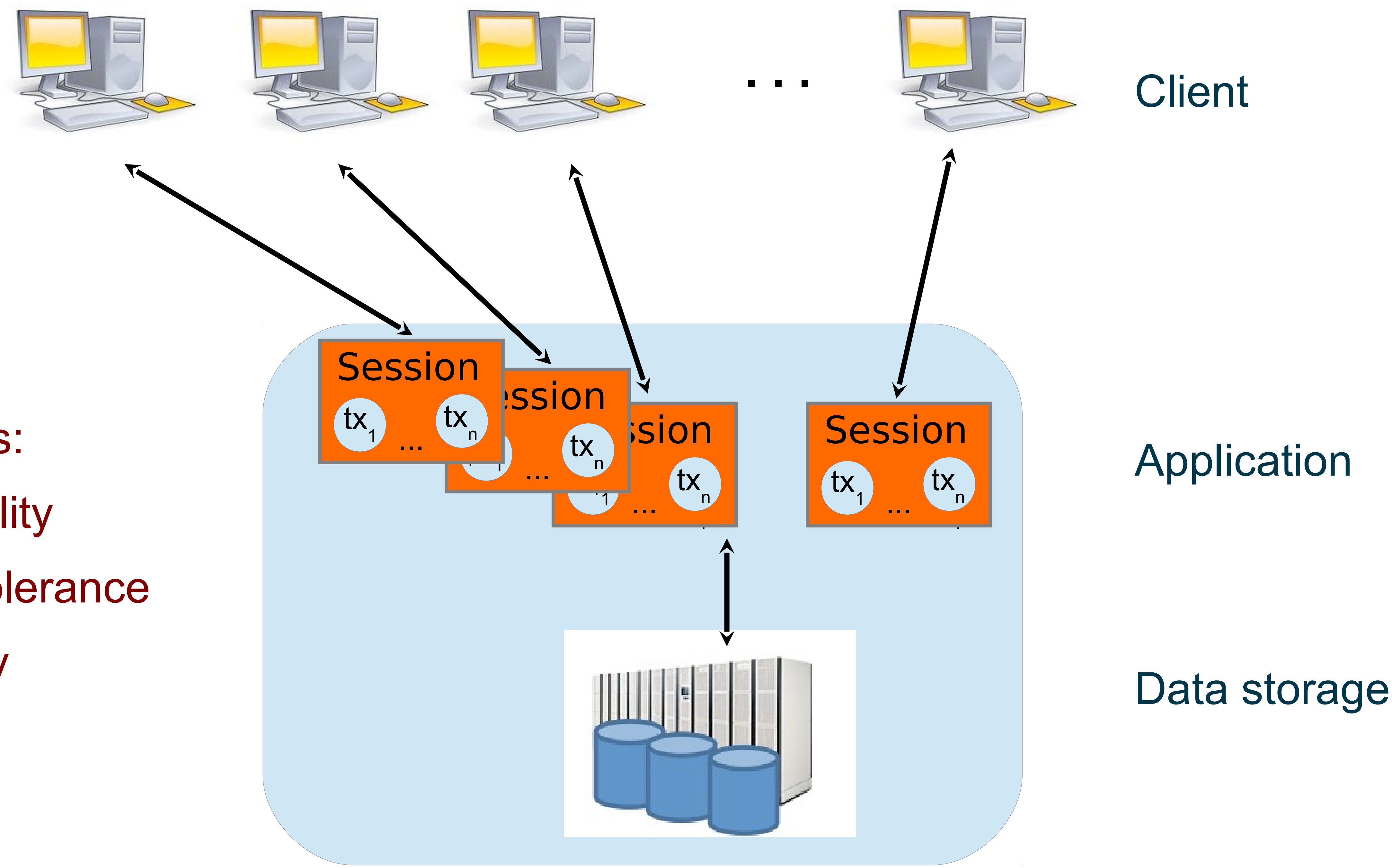
Carlos Baquero (U. Minho)

Overview

- Strong vs. eventual consistency
- Conflict-free replication
- Realizing future information systems

Strong vs. eventual consistency

Centralized information systems



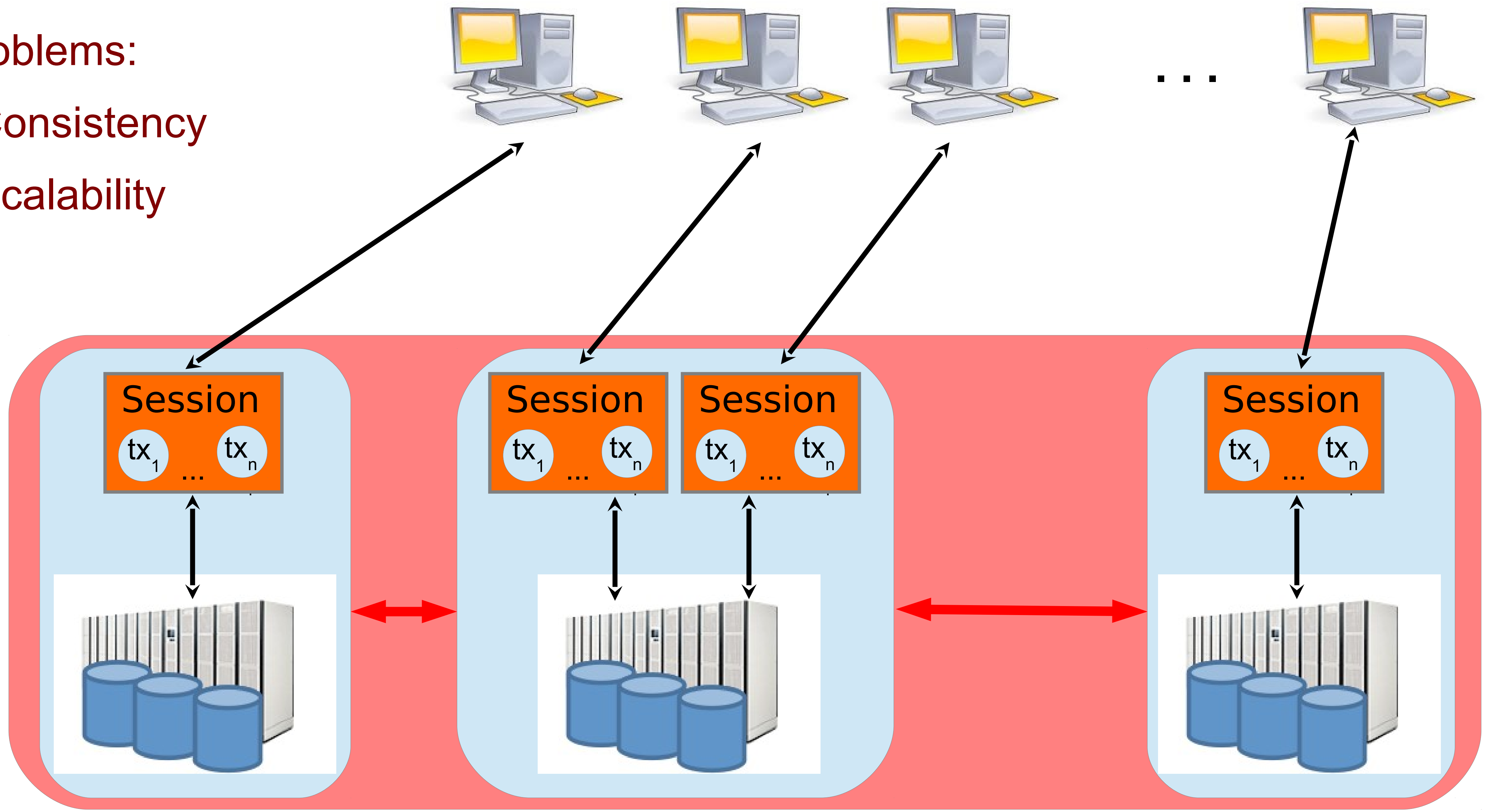
Problems:

- Scalability
- Fault-tolerance
- Latency

Strongly consistent, distributed information systems

Problems:

- Consistency
- Scalability



Discussion

Objectives:

- fault tolerance → redundancy
- low latencies → distribution
- simple to program → strong-consistency

CAP (Brewer `00; Gilbert & Lynch `06)

Strongly-Consistent ∩ *Available* ∩ *Partition-Tolerant* = ∅

Way out: Give up strong consistency

Eventual consistency

<u>Basic ideas:</u>

- Clients can live with weaker forms of consistency
- Update each replica independently
  - transport changes to other replicas
  - replay or merge
- Guaranteed delivery:
  - eventually, all replicas receive all updates
  - hopefully they converge... (otherwise: conflicts)
  - but order of updates differs!

Using eventual consistency

Different approaches:

- Application-specific vs. general approaches
- Conflict resolution:
  - manual
  - automatic
  - no conflicts
- Convergence:
  - ad hoc / programmed
  - guaranteed

Conflict-free replication

Strong eventual Consistency

Update local + propagate:

- Update is durable
- Broadcast
- No synchronization

No conflict:

- Unique outcome of
  updates (& propagations)

Assumptions for strong eventual consistency

Eventual delivery:

Every update eventually executes at all replicas.

Termination:

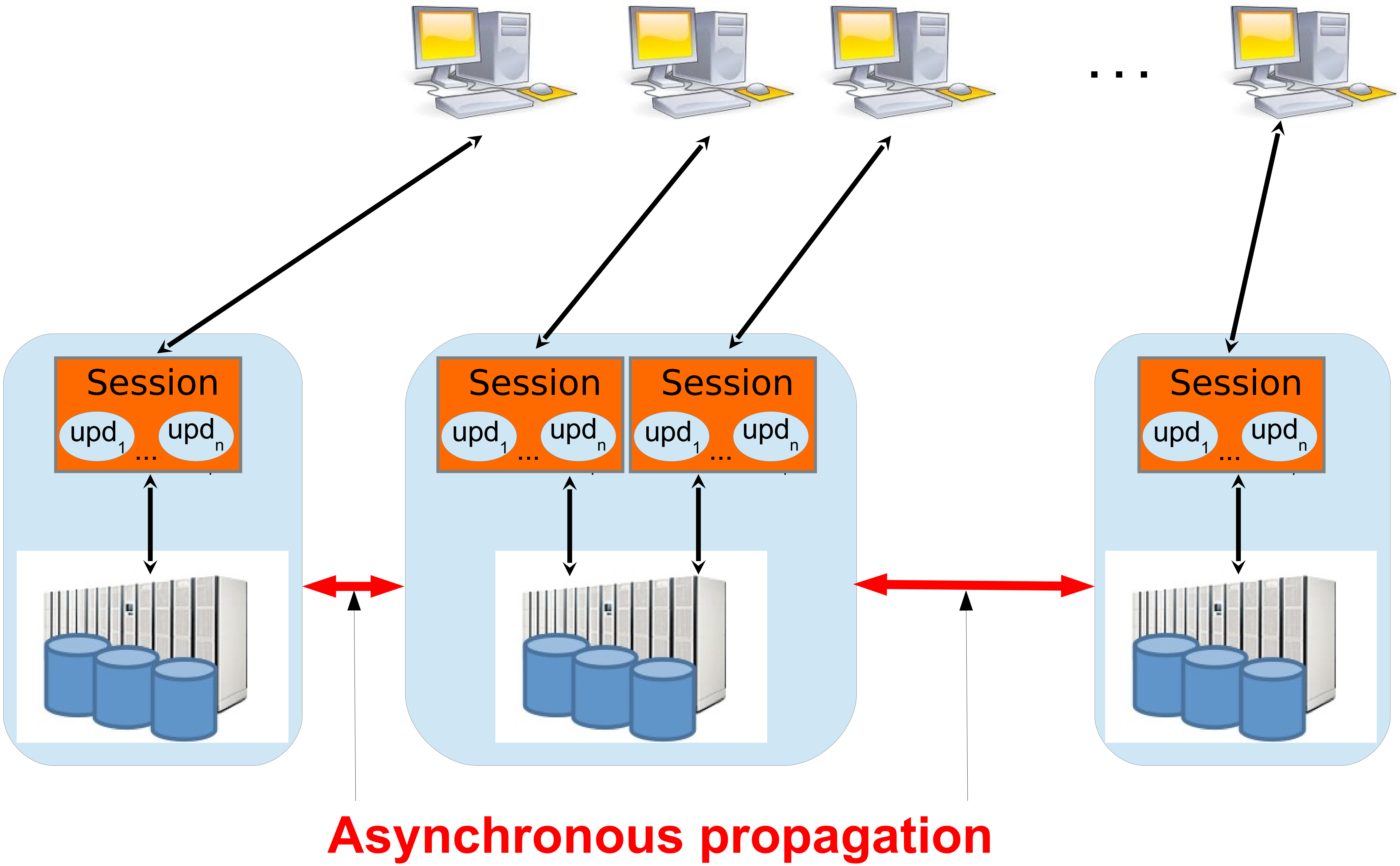
Every update terminates.

***Strong** convergence*:

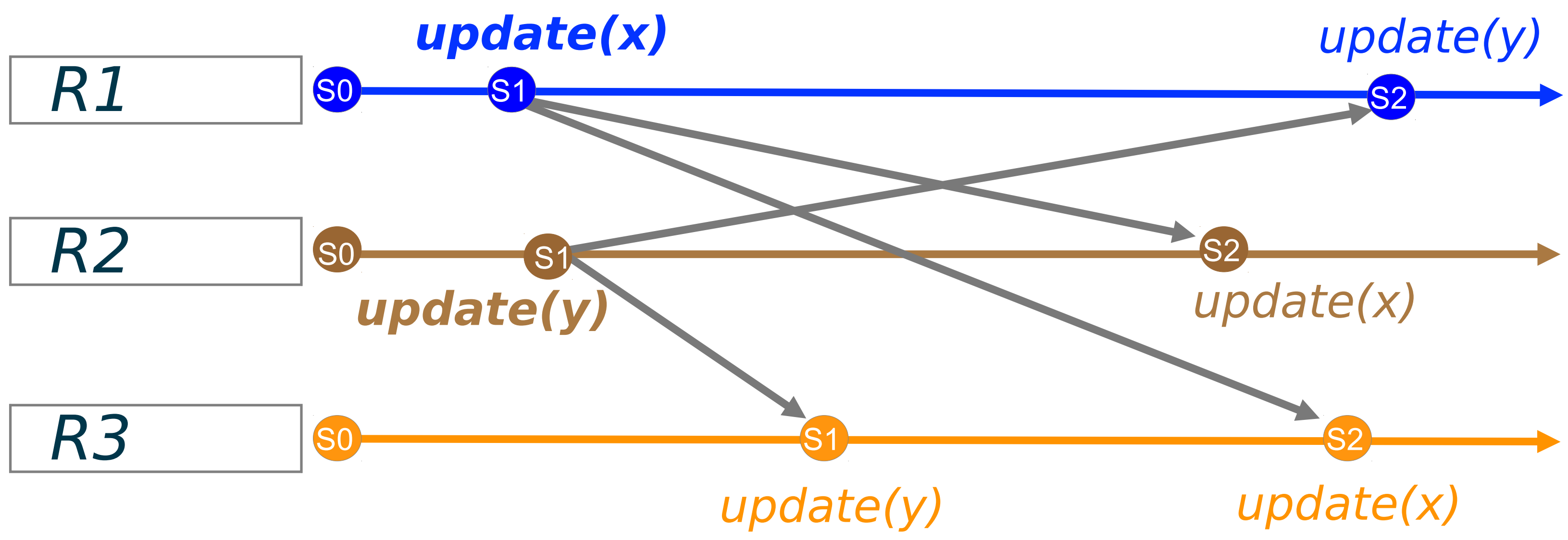
Correct replicas that have executed the same updates have equivalent state.

Deterministic, context-independent outcome to concurrent updates

Conflict-free replicated data types (CRDTs)

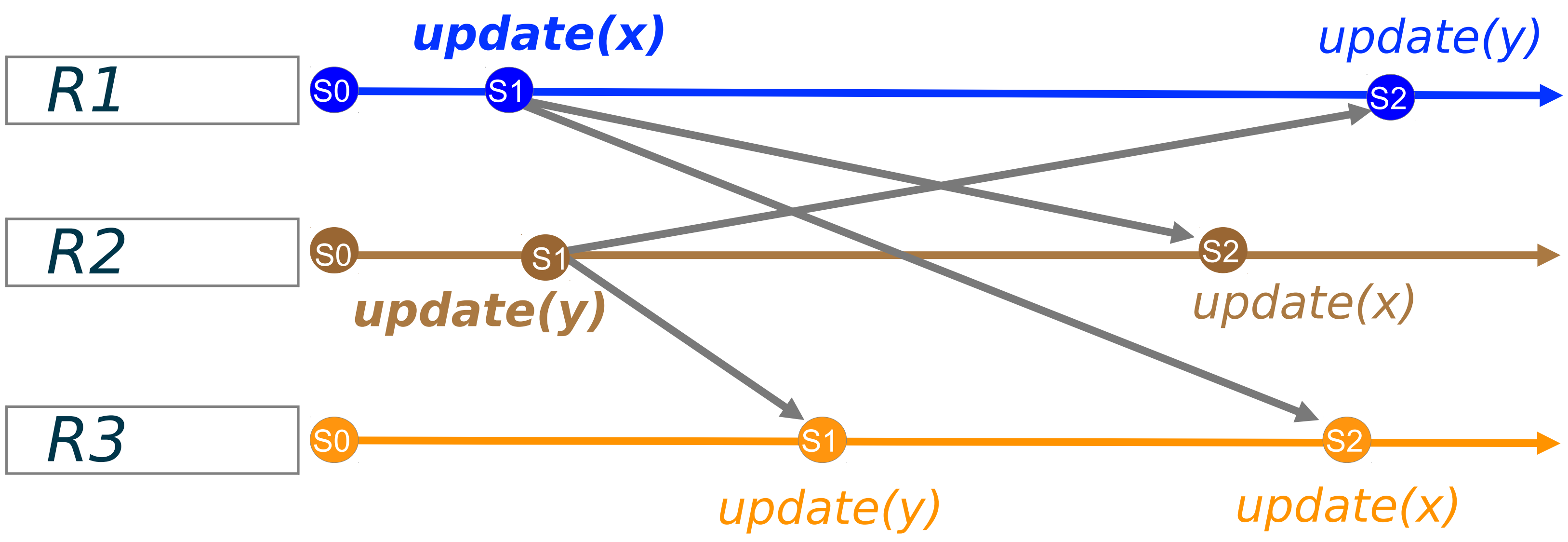


Operation-based updates



- Small messages, no information duplication
- Uses **causal broadcast**
  - Vector clock counts messages received / node
  - Size of vector clock ~ number of replicas
- Consensus not required

Operation-based CRDTs

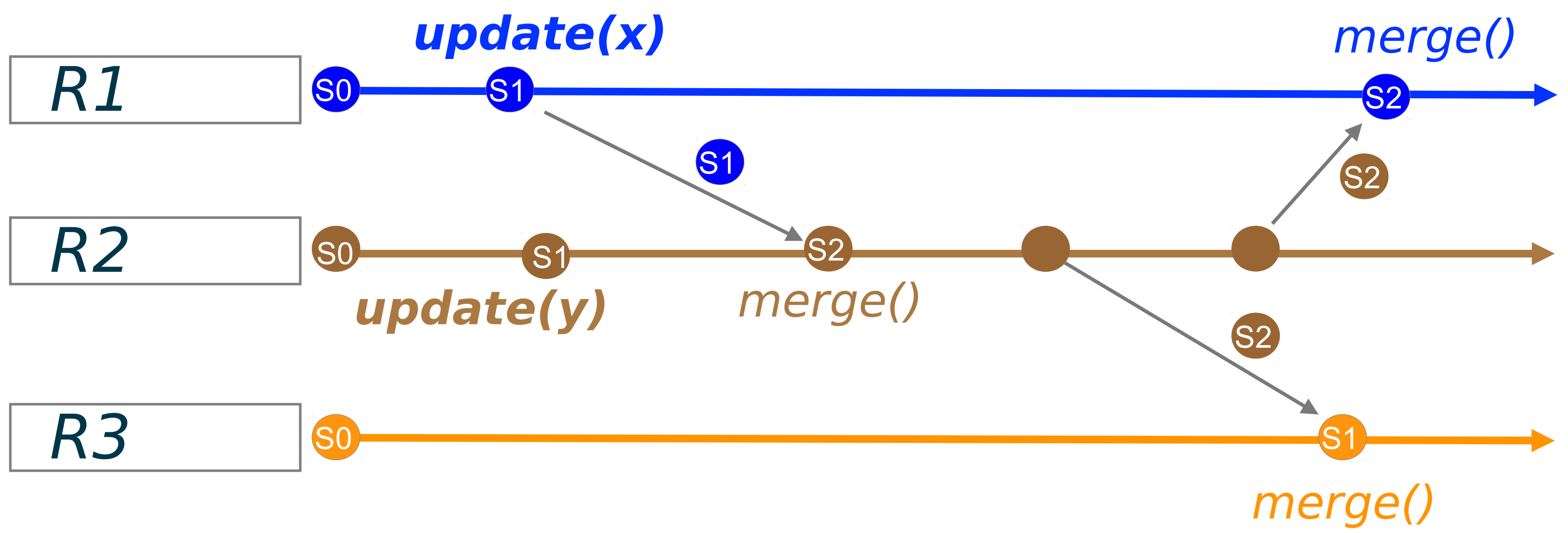


- Example: Counter with *incr* and *decr*
- All replicas have equivalent state in the end
- Sufficient condition:
  - Reliable causal delivery of vector clocks
  - Concurrent operations commute

Operation-based specification

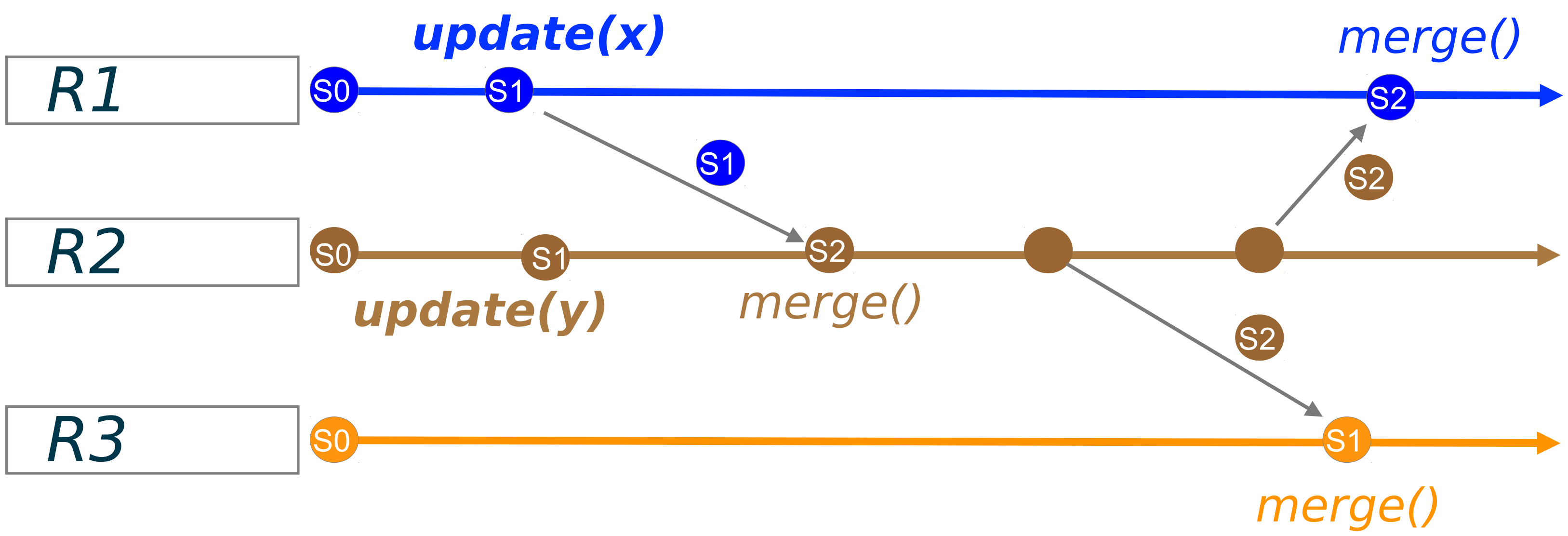
payload *Payload type; instantiated at all replicas*
  initial *Initial value*
query *Source-local operation (arguments) : returns*
  pre *Precondition*
  let *Execute at source, synchronously, no side effects*
update *Global update (arguments) : returns*
  prepare *(arguments) : returns*
    pre *Precondition at source*
    let *1st phase: synchronous, at source, no side effects*
  effect *(arguments passed downstream)*
    pre *Precondition against downstream state*
    2nd phase, asynchronous, side-effects to downstream state

State-based / data shipping



- Epidemic propagation
- Eventual delivery
- Consensus not required
- Inefficient for large payload
- Convergence

State-based CRDTs



- All replicas have equivalent state in the end
- Sufficient condition: **monotonic semi-lattice**
  - partial order
  - monotonic
  - *merge* computes LUB
  - *merge* eventually delivered

State-based specification

payload *Payload type; instantiated at all replicas*
  initial *Initial value*

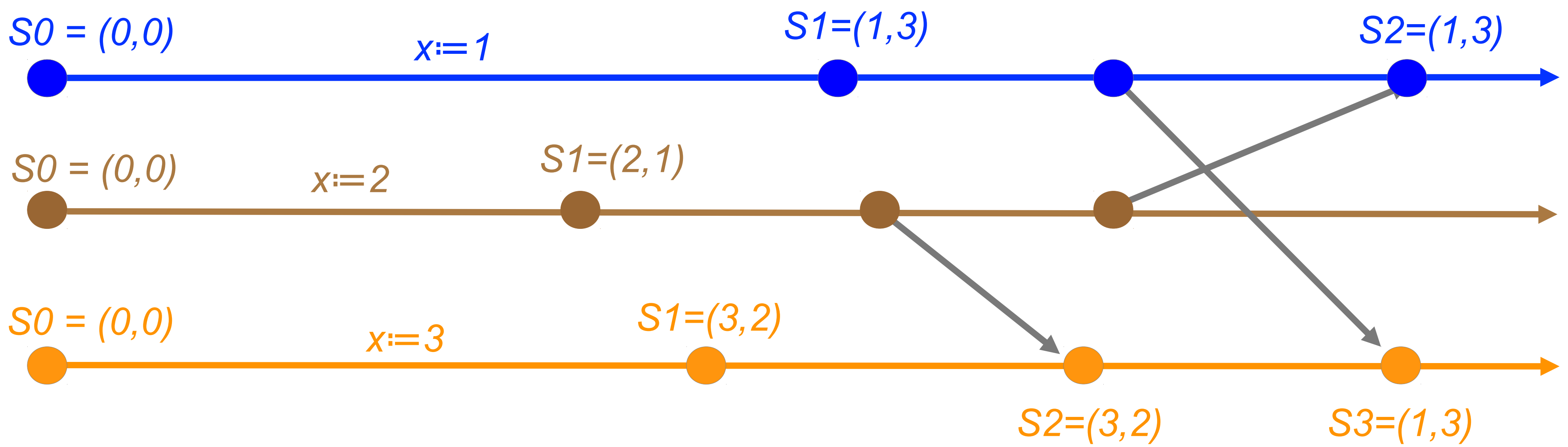
query *Query* (*arguments*) : *returns*
  pre *Precondition*
  let *Evaluate synchronously, no side effects*

update *Source-local operation* (*arguments*) : *returns*
  pre *Precondition*
  let *Evaluate at source, synchronously*
  Side-effects at source to execute synchronously

compare (value1, value2) : boolean b
  Is value1 $\leq$ *value2 in semilattice?*

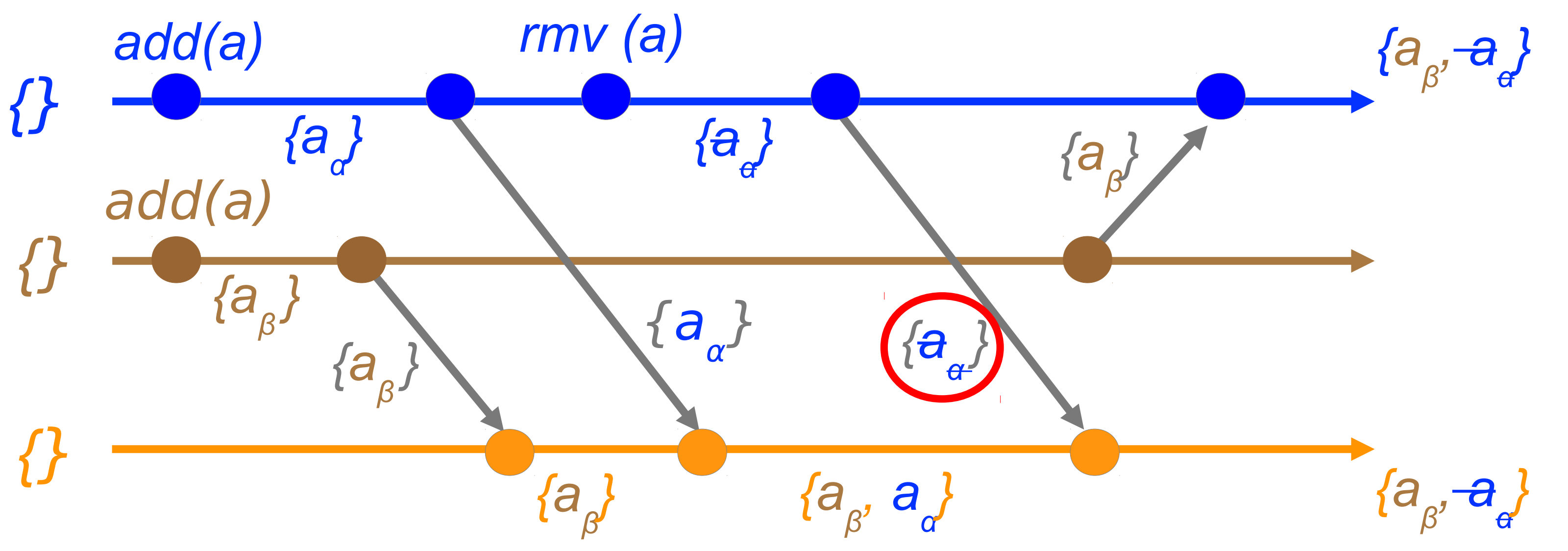
merge (value1, value2) : payload mergedValue
  LUB merge of value1 and value2, at any replica

Last-writer-wins register



Payload		$S \stackrel{\text{def}}{=}$	*(value v, timestamp ts)*
Update	$S \bullet [x \coloneqq v]$	$\stackrel{\text{def}}{=}$	$(v, \text{now}())$
Merge	$S \bullet merge(S')$	$\stackrel{\text{def}}{=}$	$S.ts < S'.ts\ ?\ S' : S$
Compare	$S \leq S'$	$\stackrel{\text{def}}{=}$	$S.ts \leq S'.ts$

Observed-remove Set



Payload $S \stackrel{\text{def}}{=} (\, A = \{(e, uid), \ldots\},\ R = \{(e', uid'), \ldots\})$

Update $S \bullet add(e) \stackrel{\text{def}}{=} (\, A \cup \{(e,uid)\}, R\,)$

$S \bullet rmv(e) \stackrel{\text{def}}{=} (A \setminus T, R \cup T)\ \text{ with }\ T = \{\, (e, \_\,) \in A \,\}$

Lookup $S \bullet lookup(e) \stackrel{\text{def}}{=} e \in A$

Merge $S \bullet merge(S') \stackrel{\text{def}}{=} (A \setminus R' \cup A' \setminus R,\ R \cup R')$

Compare $S \leq S' \stackrel{\text{def}}{=} A \cup R \subseteq A' \cup R' \wedge R \subseteq R'$

Further examples of CRDTs

Register
- Last-Writer Wins
- Multi-Value

Set
- Grow-Only
- 2P
- Observed-Remove

Map

Counter
- Unlimited
- Non-negative

Graph
- Directed
- Monotonic DAG
- Edit graph

Sequence

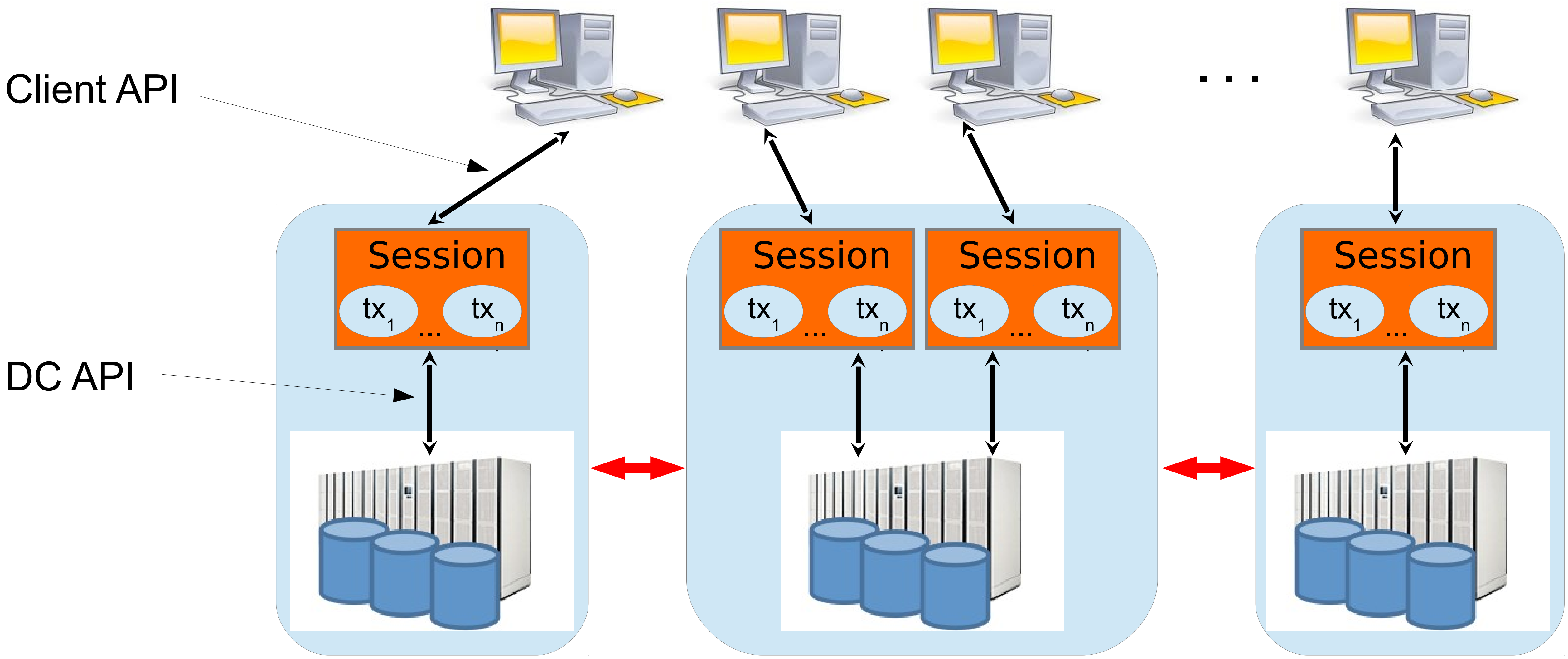
Summary: CRDT

- Concurrent updates have deterministic outcome
- Sufficient conditions:
  - State-based: epidemic, monotonic semi-lattice
  - Op-based: causal, concurrent ⇒ commute

- CRDTs
  - don‘t lose updates
  - converge eventually
  - have durable updates, no rollbacks
  - support unlimited (crash-recovery) failures

Realizing future information systems

Programming model



Central questions:

- **What is the application-independent API of data store?**
- **How can CRDTs be combined to realize client API?**
- **What is needed in addition to CRDTs?**

Further challenges

- More complex architectures:
  - client state
  - DC hopping
- Global state guarantees:
  - support of reservations
  - stable preconditions
- Transactions
- Verification techniques
- Using CRDTs for concurrent programming