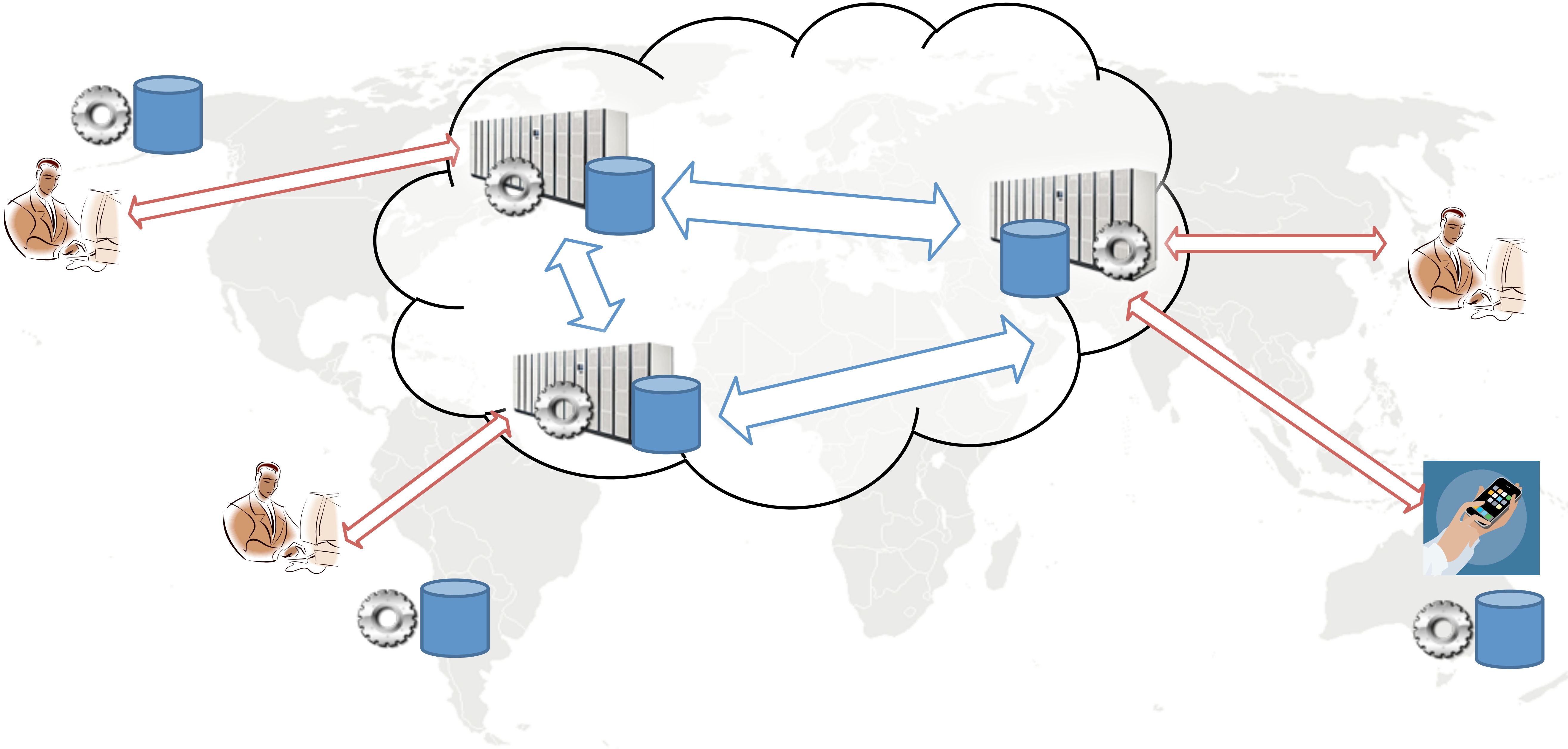
CRDTs in Practice

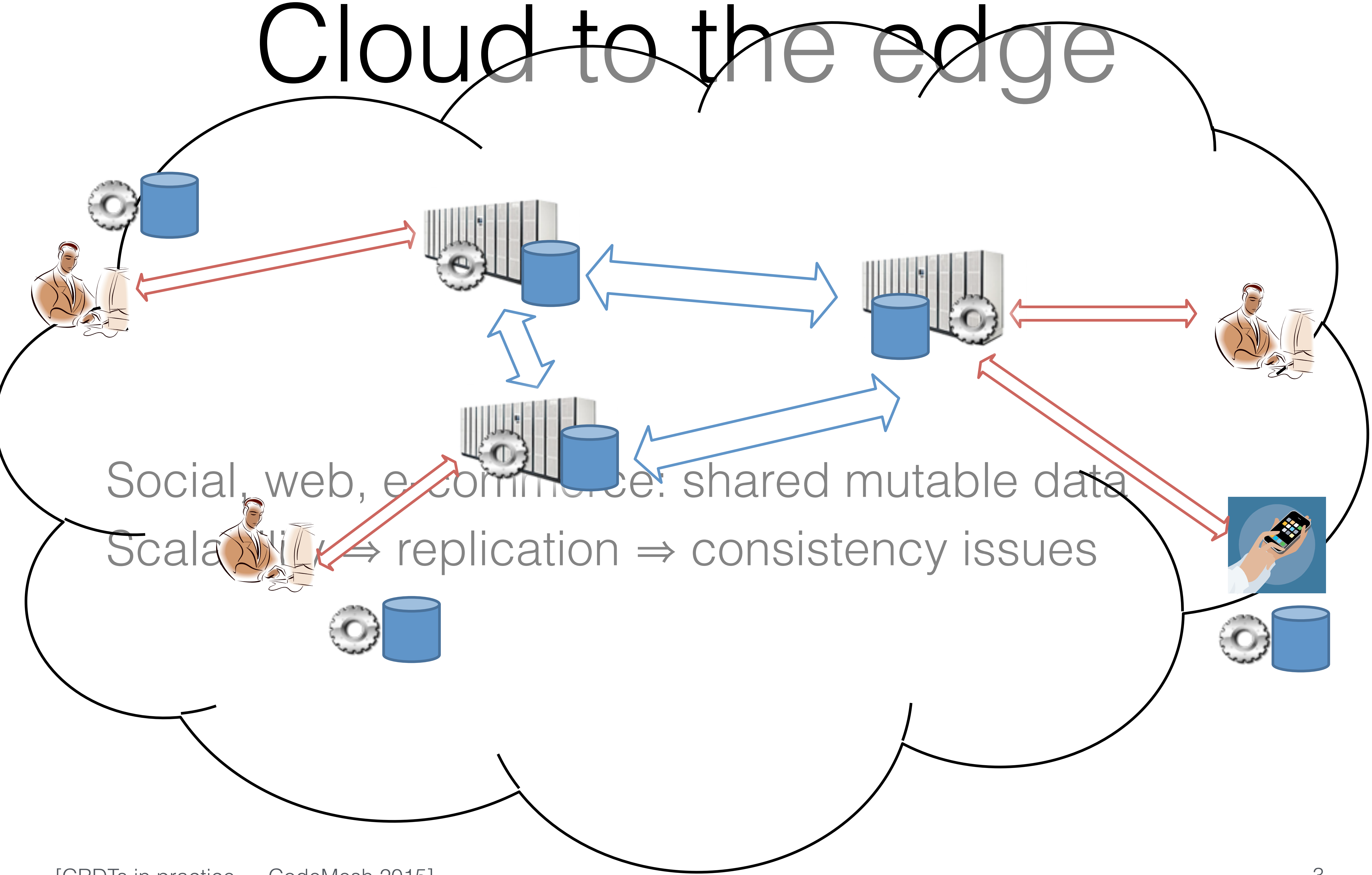
Marc Shapiro – Inria & UPMC

Nuno Preguiça – U. NOVA

Cloud to the edge



Cloud to the edge



Social, web, e-commerce: shared mutable data

Scalability ⇒ replication ⇒ consistency issues

Conflict-free replicated data types

Data type

- Encapsulates issues

Replicated

- At multiple nodes

Available

- Update my replica without coordination
- Convergence guaranteed (by mathematical properties)
- Decentralised, peer-to-peer

Why use CRDTs

Availability is king (otherwise stay away)

⟹ concurrent updates

Fine-grain mutable shared data

- Registers not sufficient

Mobile computing

In DC

Geo-replication

CRDT design concepts

Backward-compatible with sequential datatype

If operations commute, they can be concurrent

- *add(e); rm(f)* ≡ *rm(f); add(e)* ≡ *add(e)* || *rm (f)*

Otherwise, deterministic semantics

- Close to sequential *rm(e);add(e)* or *add(e); rm(e)*
- Don’t lose updates
- Result doesn't depend on order received
- Stable preconditions

TOMTOM®

RIOT GAMES

LEAGUE of LEGENDS®

SOUNDCLOUD

HELIUM

bet365

basho

riak

TRIFORK.
...think software

akka

ROVIO

ANGRY BIRDS

Tapjoy®

PROJECTFIFO

bet365

Largest European on-line betting operator

- Bursty load: 2.5 million simultaneous users
- 1 Tb working set
- 1000s servers
- Slow users: transient inconsistency OK
- Availability, read my writes, monotonic reads
- Transparency

Before: SQLserver, doesn't scale, hours to converge

mid 2013: noSQL riak: available, siblings; ad-hoc merge (hard!)

bet365 CRDT experience

≥ Jan. 2014; in anger ≥ Dec. 2014

ORSWOT add-remove set

- Add, remove element; scan for similar
- 100s Gb

Transformational : “CRDTs saved the day”

- Correct by construction
- Stable; partitions fixed quickly, correctly

Future wish list: “Extra guarantees … without impacting availability.”

CRDT Set design space

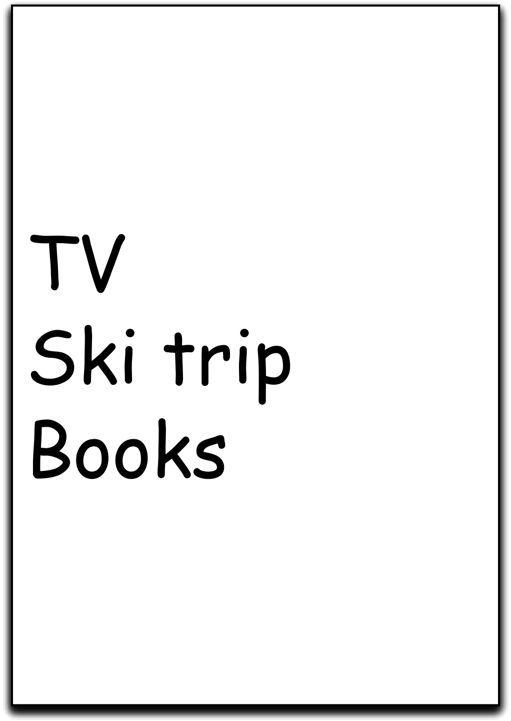
Many Set operations commute: *add(e) / add(f), add(e) / rm(f)*, etc.

Non-commuting pair: *add(e) / rm(e)*

- ~~sequential consistency~~
- last writer wins? $\{ add(e) < rmv(e) \Longrightarrow e \notin S \wedge rmv(e) < add(e) \Longrightarrow e \in S \}$
- error state? $\{\bot_e \in S\}$
- add wins? $\{e \in S\}$
- remove wins? $\{e \notin S\}$

All deterministic, satisfy conditions

Wedding list

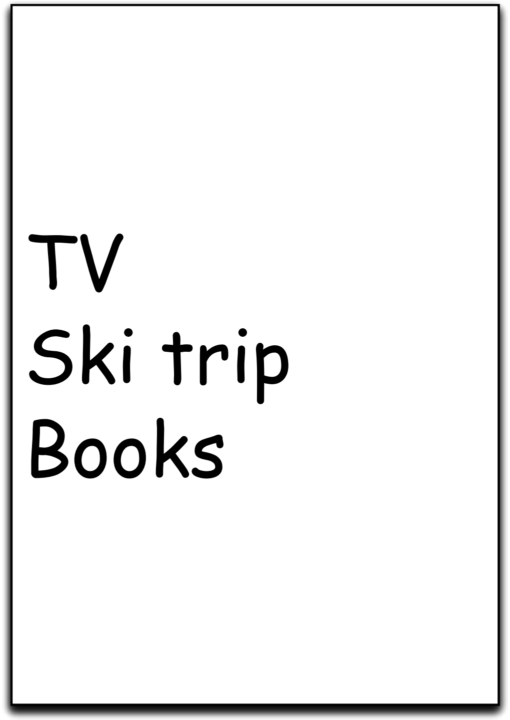


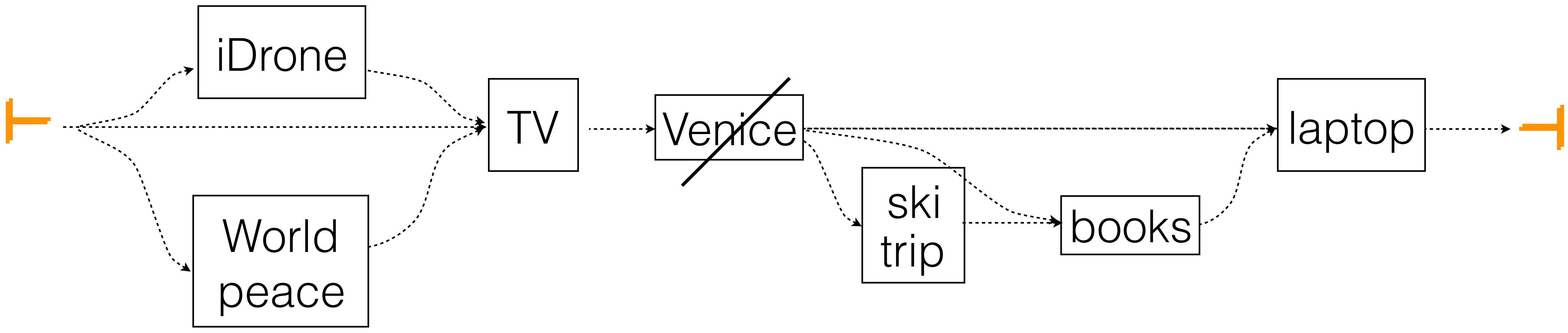
Replicated wedding list

Ordered list of “wishes” (strings)

- *lookup (wish) ⟶ rank*
- *add (position, wish)*
- *rm (position)*

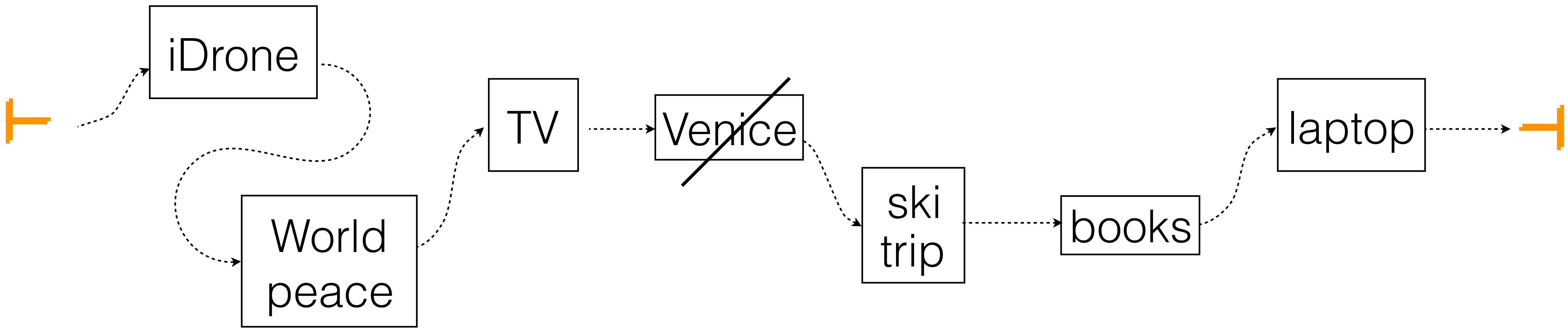
Position: “after item”





Each item points to the next one

- *add (pos, item)*: link *item* after the one at *pos*
- *rm (item)*: mark as tombstone
- *add (pos, item1)* || *add (pos, item2)*: deterministic



Each item points to the next one

- *add (pos, item)*: link *item* after the one at *pos*
- *rm (item)*: mark as tombstone
- *add (pos, item1)* || *add (pos, item2)*: deterministic

Lowering your expectations

World Peace
iDrone
TV
Ski trip
Books
Laptop

- *lookup (wish) ⟶ rank*
- *add (pos, wish)*
- *rm (pos)*
- *mv (wish, pos1, pos2)*

iDrone
TV
Ski trip
Books
Laptop
World Peace

Lowering your expectations

World Peace
iDrone
TV
Ski trip
Books
Laptop
World Peace

- *lookup (wish) ⟶ rank*
- *add (pos, wish)*
- *rm (pos)*
- ~~*mv (wish, pos1, pos2)*~~
  add (…, pos2); rm (pos1)

World Peace
iDrone
TV
Ski trip
Books
Laptop
World Peace

Lowering your expectations

● iDrone
World Peace
TV
Ski trip
Books
Laptop

- *lookup (wish) ⟶ rank*
- *add (pos, wish)*
- *rm (pos)*
- *~~mv (wish, pos1, pos2)~~*
  add (…, pos2); rm (pos1)
- *offer (wish)*

● iDrone
World Peace
TV
Ski trip
Books
Laptop

The problem with invariants

Remove specification

{ true } rm(wish) { tombstone(wish) }

Move, offer: maintain uniqueness invariant

{ ¬offered(wish,_) } offer(wish) { offered(wish, red) }

Precondition *stable* under concurrent updates?

- If so, invariant guaranteed
- Otherwise, all bets are off

Lessons learned

Availability ⟹ concurrent updates

- Mask their undesirable effects

Backwards compatible

- Same sequential semantics
- Commute ⟹ same concurrent semantics
- otherwise, “close enough”

Maintaining invariants

- *Stable preconditions*

Numeric Invariants

Many applications need to enforce conditions like:

$$\text{counter} \geq K$$

E.g.:

- Number of impressions left ≥ 0
- Virtual money in a game ≥ 0

Numeric invariants

$$X \geq 0$$

Given *X = n* , there are n rights to execute *dec()*

Distribute rights among replicas

- Consume rights for *dec()*
- Create rights on *inc()*

CRDT-ish

Execute operations locally without coordination

Peer-to-peer synchronisation

Fail if not enough rights exist

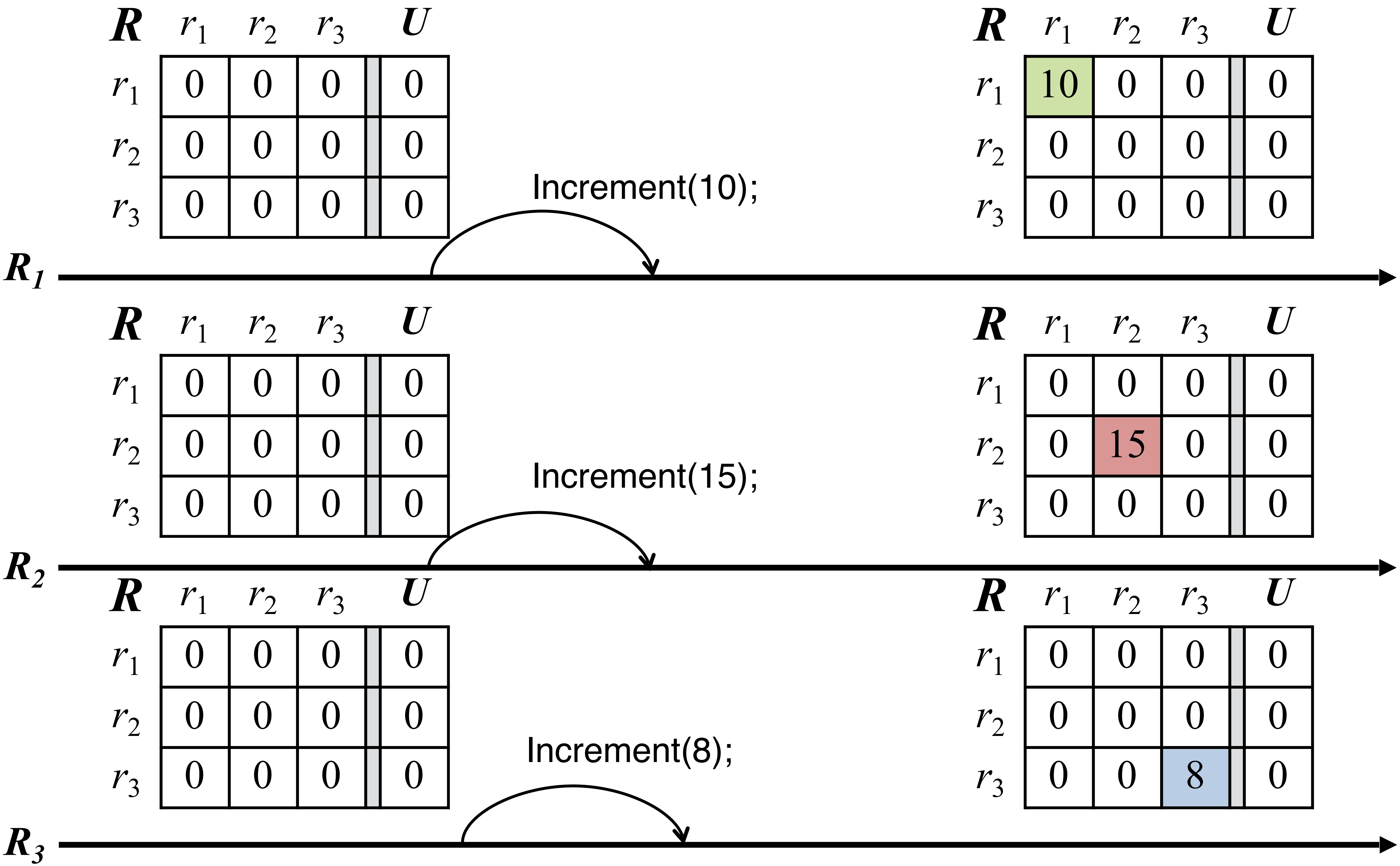
Bounded Counter: API

```
Create(type, value);

Increment(value);
Decrement(value);
Value();

Transfer(to, qty);
```

Bounded Counter: increment



R	r_1	r_2	r_3	*U*
r_1	0	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

Increment(10);

R	r_1	r_2	r_3	*U*
r_1	10	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

R	r_1	r_2	r_3	*U*
r_1	0	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

Increment(15);

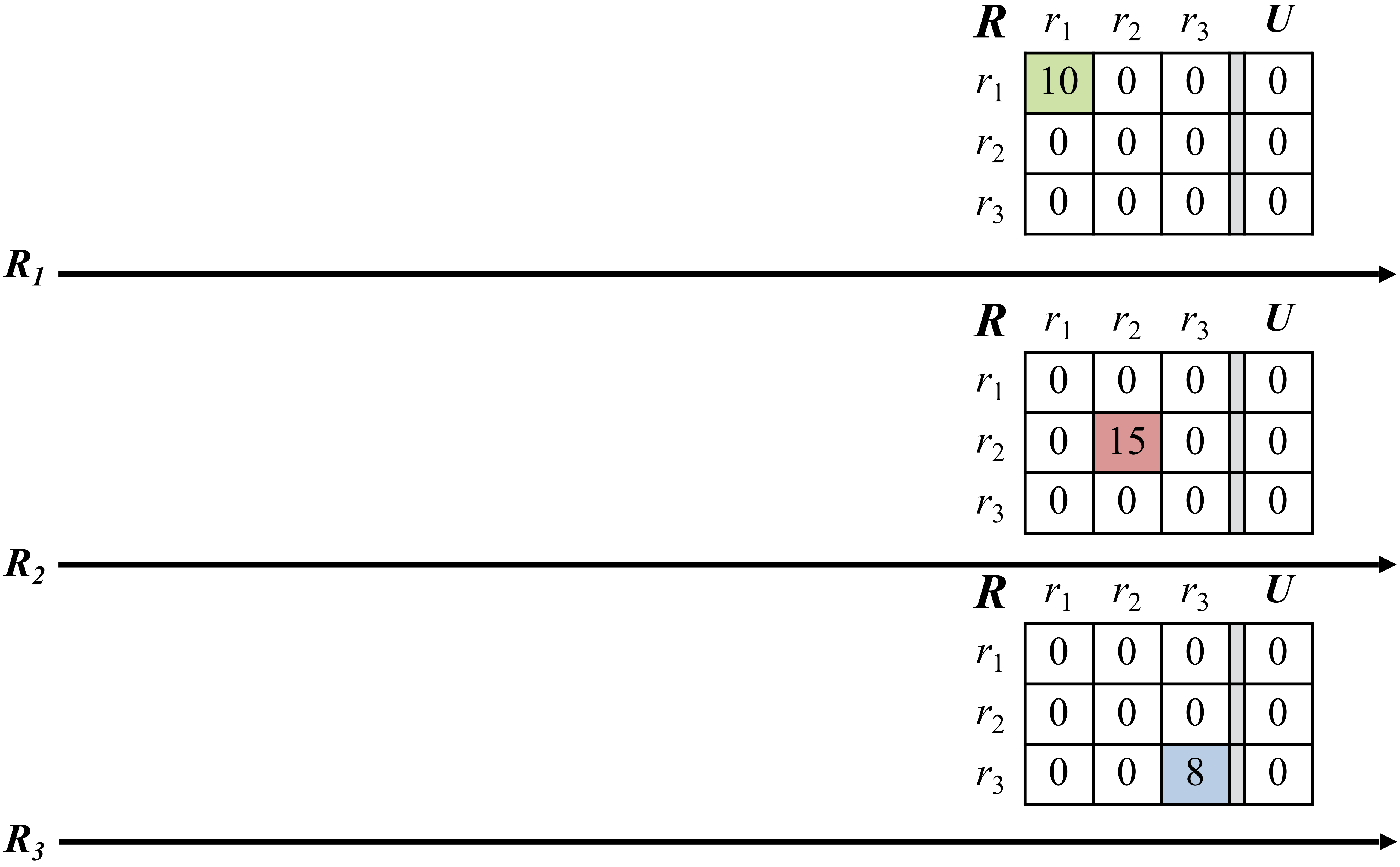
R	r_1	r_2	r_3	*U*
r_1	0	0	0	0
r_2	0	15	0	0
r_3	0	0	0	0

R	r_1	r_2	r_3	*U*
r_1	0	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

Increment(8);

R	r_1	r_2	r_3	*U*
r_1	0	0	0	0
r_2	0	0	0	0
r_3	0	0	8	0

Bounded Counter: increment

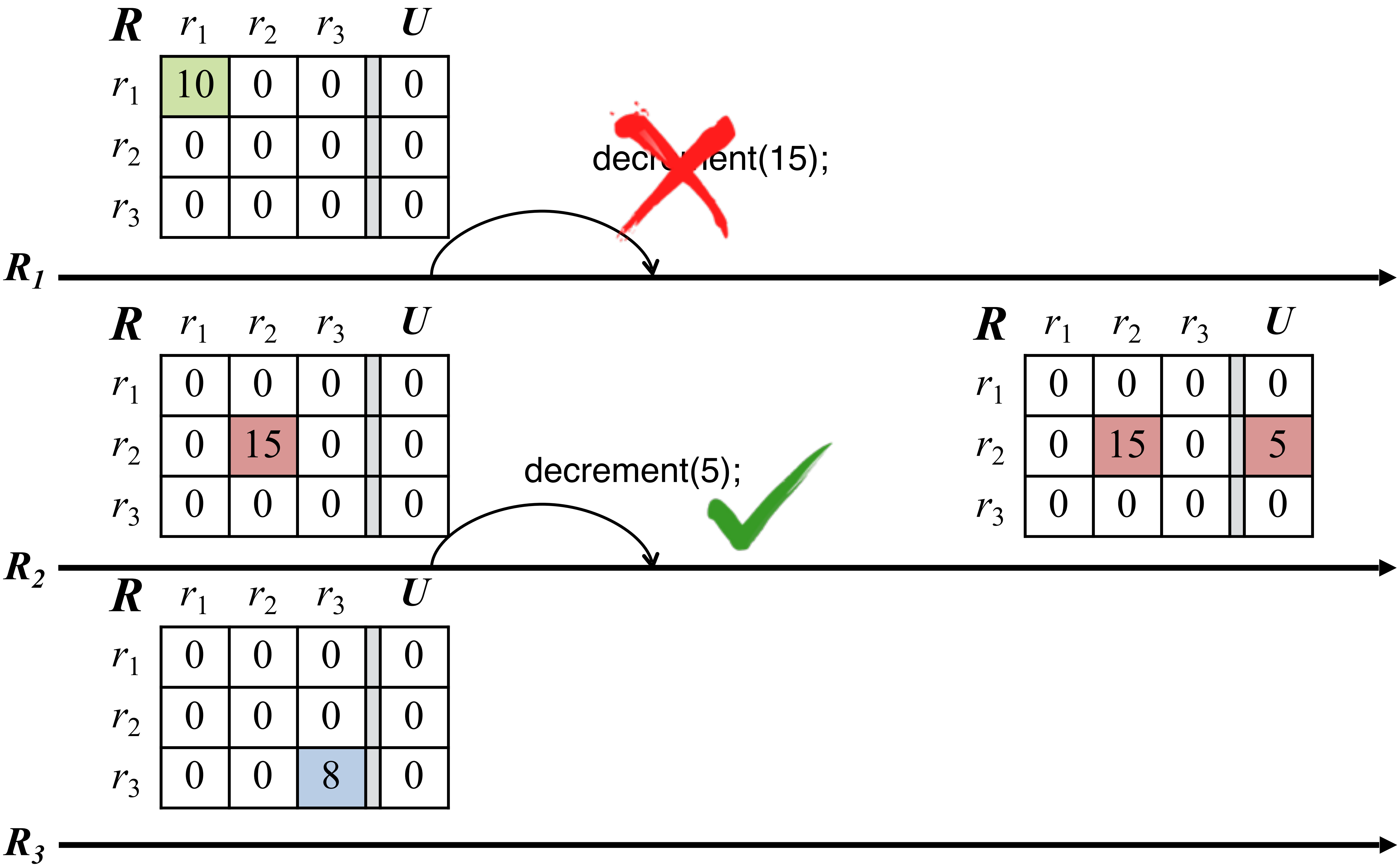


R	r_1	r_2	r_3	***U***
r_1	10	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

R	r_1	r_2	r_3	***U***
r_1	0	0	0	0
r_2	0	15	0	0
r_3	0	0	0	0

R	r_1	r_2	r_3	***U***
r_1	0	0	0	0
r_2	0	0	0	0
r_3	0	0	8	0

Bounded Counter: decrement



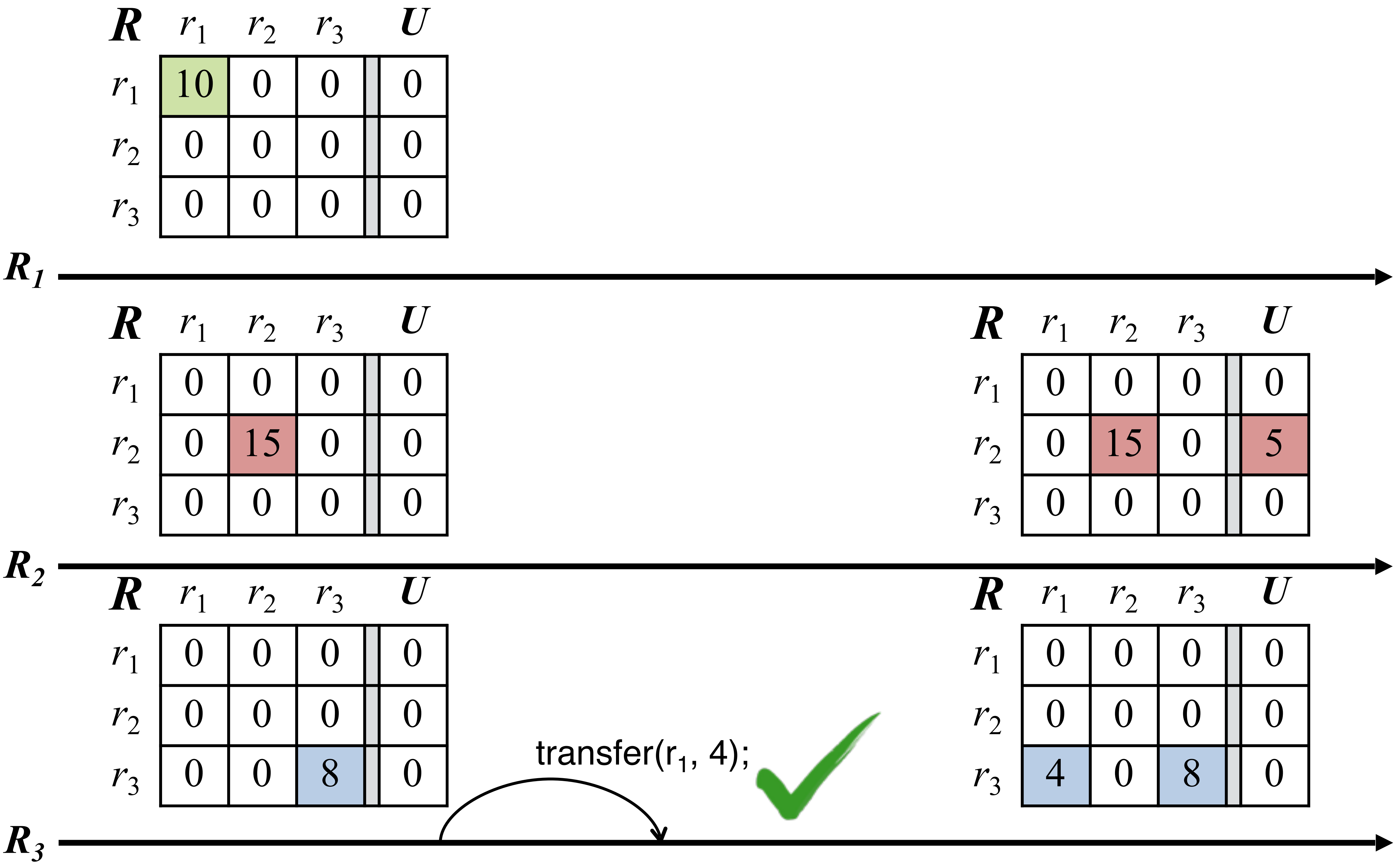
R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	15	0	0
r_3	0	0	0	0

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	15	0	5
r_3	0	0	0	0

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	0	0	0
r_3	0	0	8	0

Bounded Counter: transfer



$\boldsymbol{R}$	r_1	r_2	r_3	$\boldsymbol{U}$
r_1	10	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

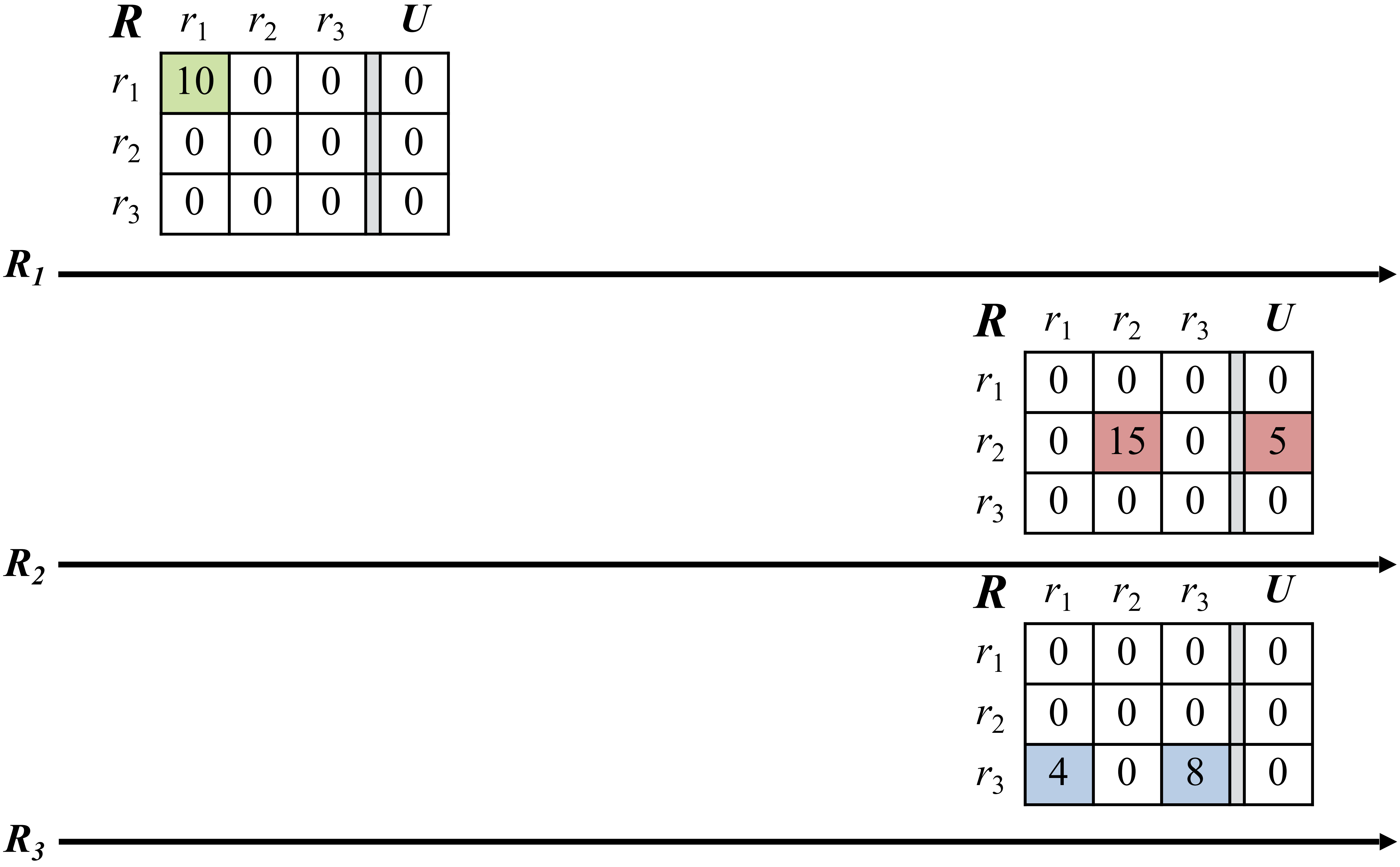
$\boldsymbol{R}$	r_1	r_2	r_3	$\boldsymbol{U}$
r_1	0	0	0	0
r_2	0	15	0	0
r_3	0	0	0	0

$\boldsymbol{R}$	r_1	r_2	r_3	$\boldsymbol{U}$
r_1	0	0	0	0
r_2	0	15	0	5
r_3	0	0	0	0

$\boldsymbol{R}$	r_1	r_2	r_3	$\boldsymbol{U}$
r_1	0	0	0	0
r_2	0	0	0	0
r_3	0	0	8	0

$\boldsymbol{R}$	r_1	r_2	r_3	$\boldsymbol{U}$
r_1	0	0	0	0
r_2	0	0	0	0
r_3	4	0	8	0

Bounded Counter: transfer

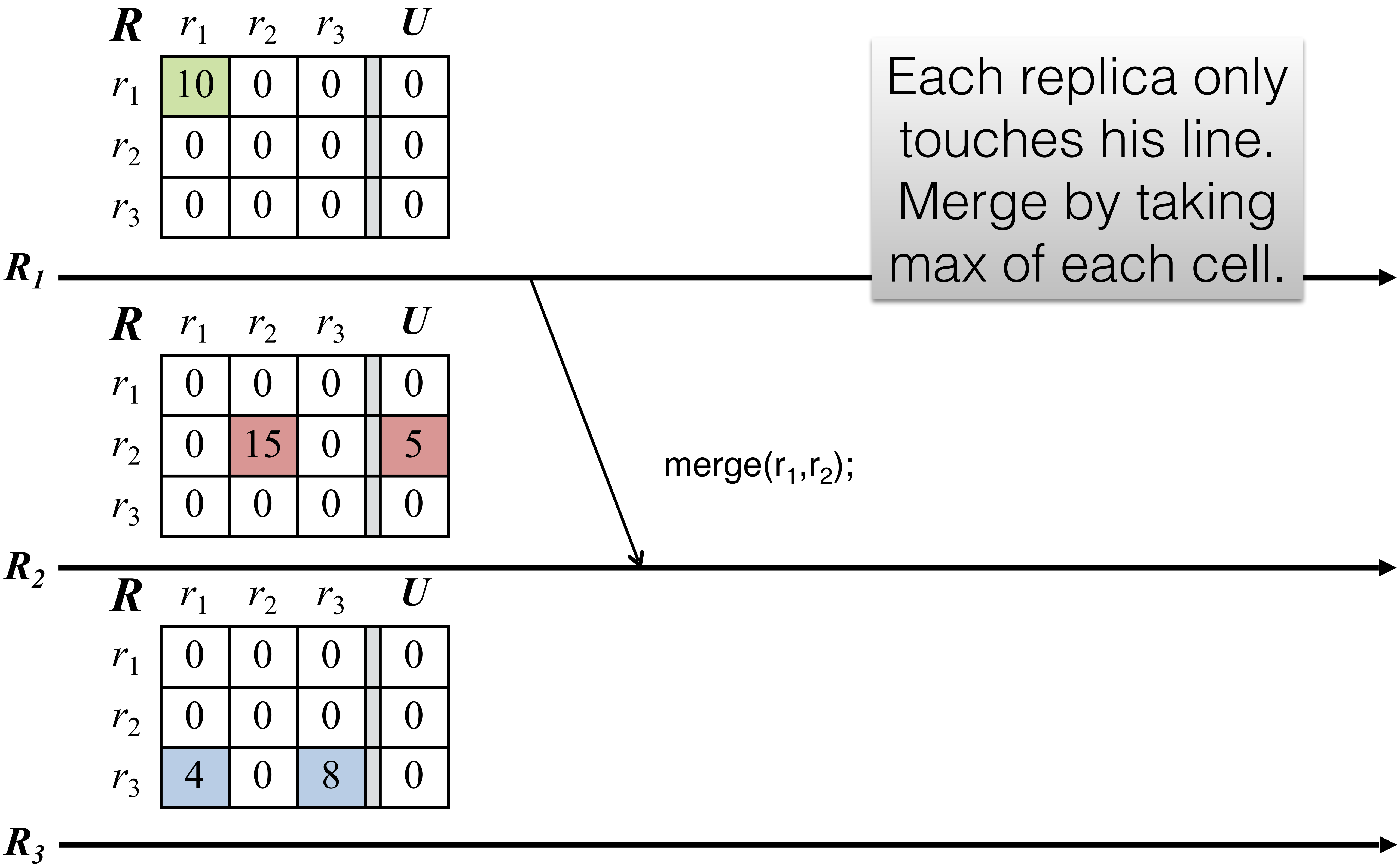


$\boldsymbol{R}$	r_1	r_2	r_3	$\boldsymbol{U}$
r_1	10	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

$\boldsymbol{R}$	r_1	r_2	r_3	$\boldsymbol{U}$
r_1	0	0	0	0
r_2	0	15	0	5
r_3	0	0	0	0

$\boldsymbol{R}$	r_1	r_2	r_3	$\boldsymbol{U}$
r_1	0	0	0	0
r_2	0	0	0	0
r_3	4	0	8	0

Bounded Counter: merge



R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

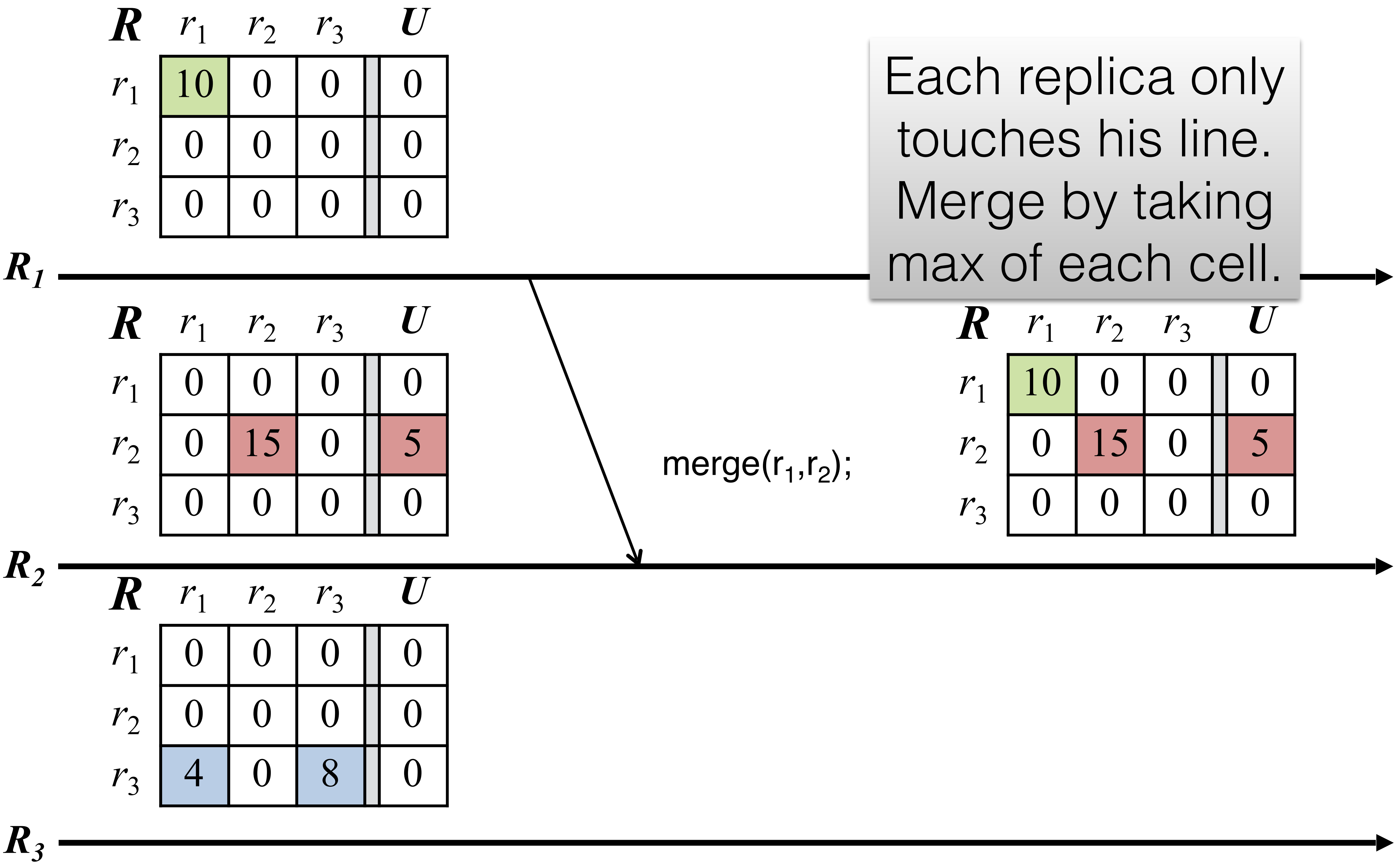
R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	15	0	5
r_3	0	0	0	0

merge(r_1,r_2);

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	0	0	0
r_3	4	0	8	0

Each replica only touches his line. Merge by taking max of each cell.

Bounded Counter: merge



R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	15	0	5
r_3	0	0	0	0

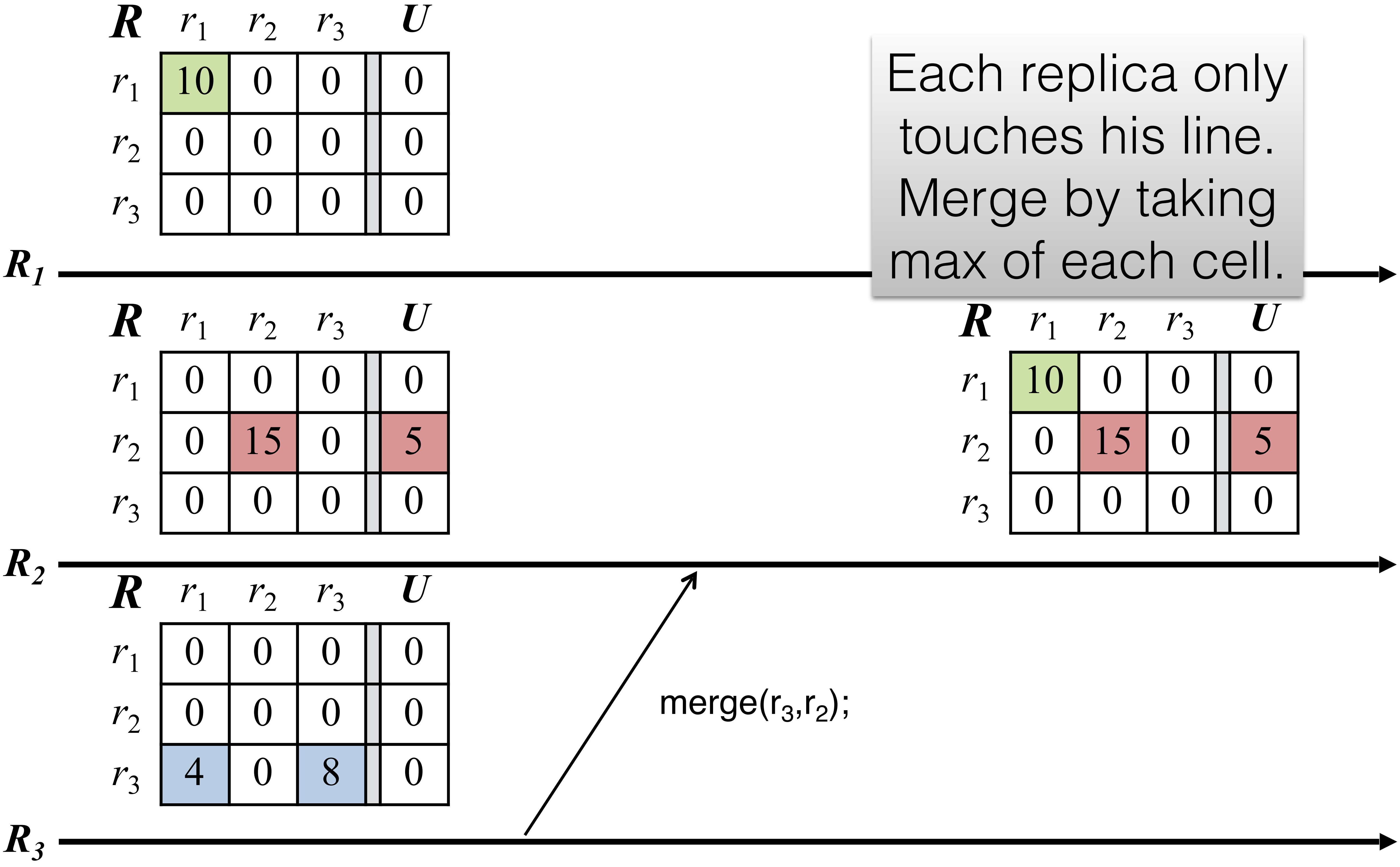
merge(r_1,r_2);

R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	15	0	5
r_3	0	0	0	0

Each replica only touches his line. Merge by taking max of each cell.

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	0	0	0
r_3	4	0	8	0

Bounded Counter: merge



R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

Each replica only touches his line. Merge by taking max of each cell.

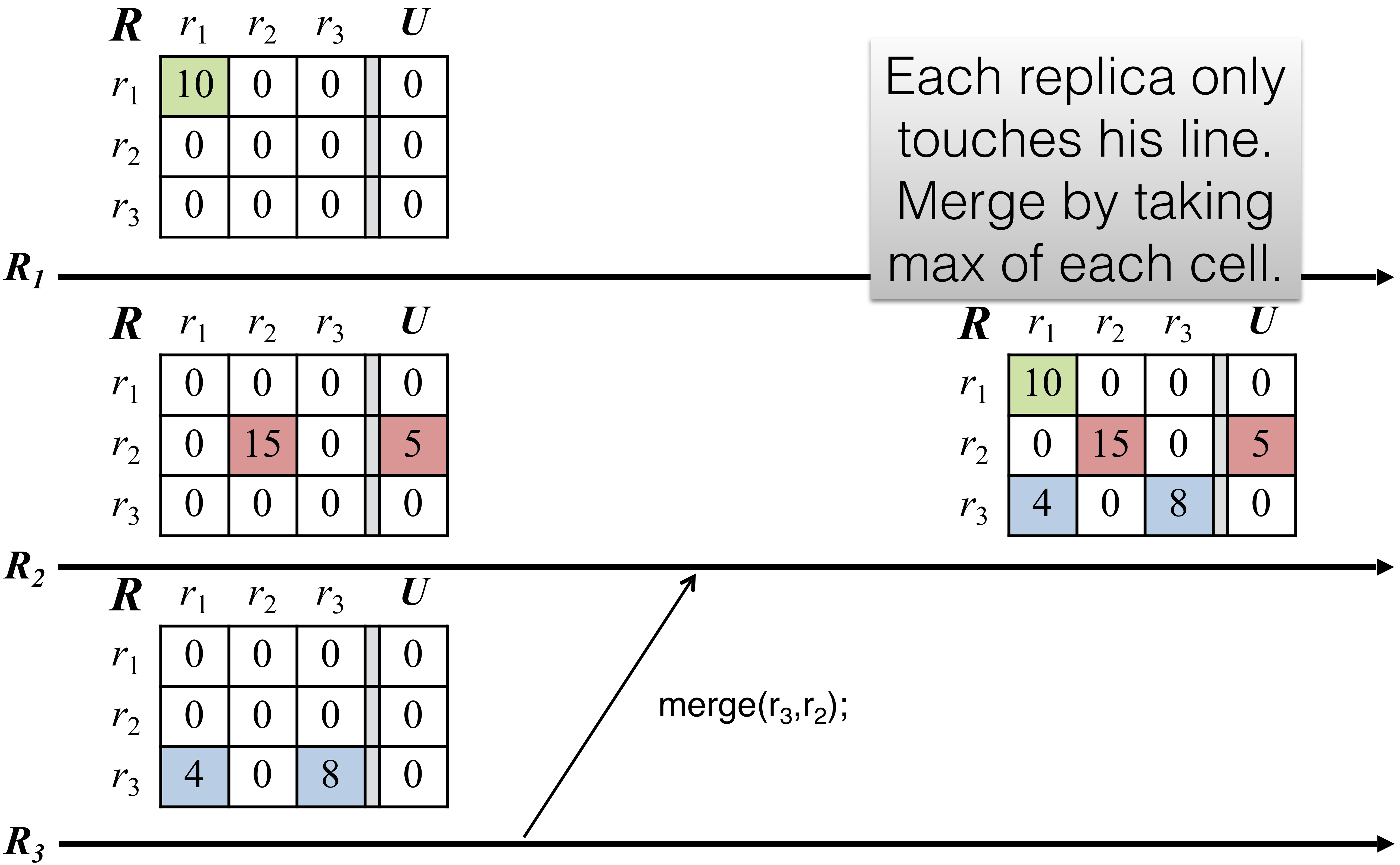
R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	15	0	5
r_3	0	0	0	0

R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	15	0	5
r_3	0	0	0	0

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	0	0	0
r_3	4	0	8	0

merge(r_3,r_2);

Bounded Counter: merge



R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	0	0	0
r_3	0	0	0	0

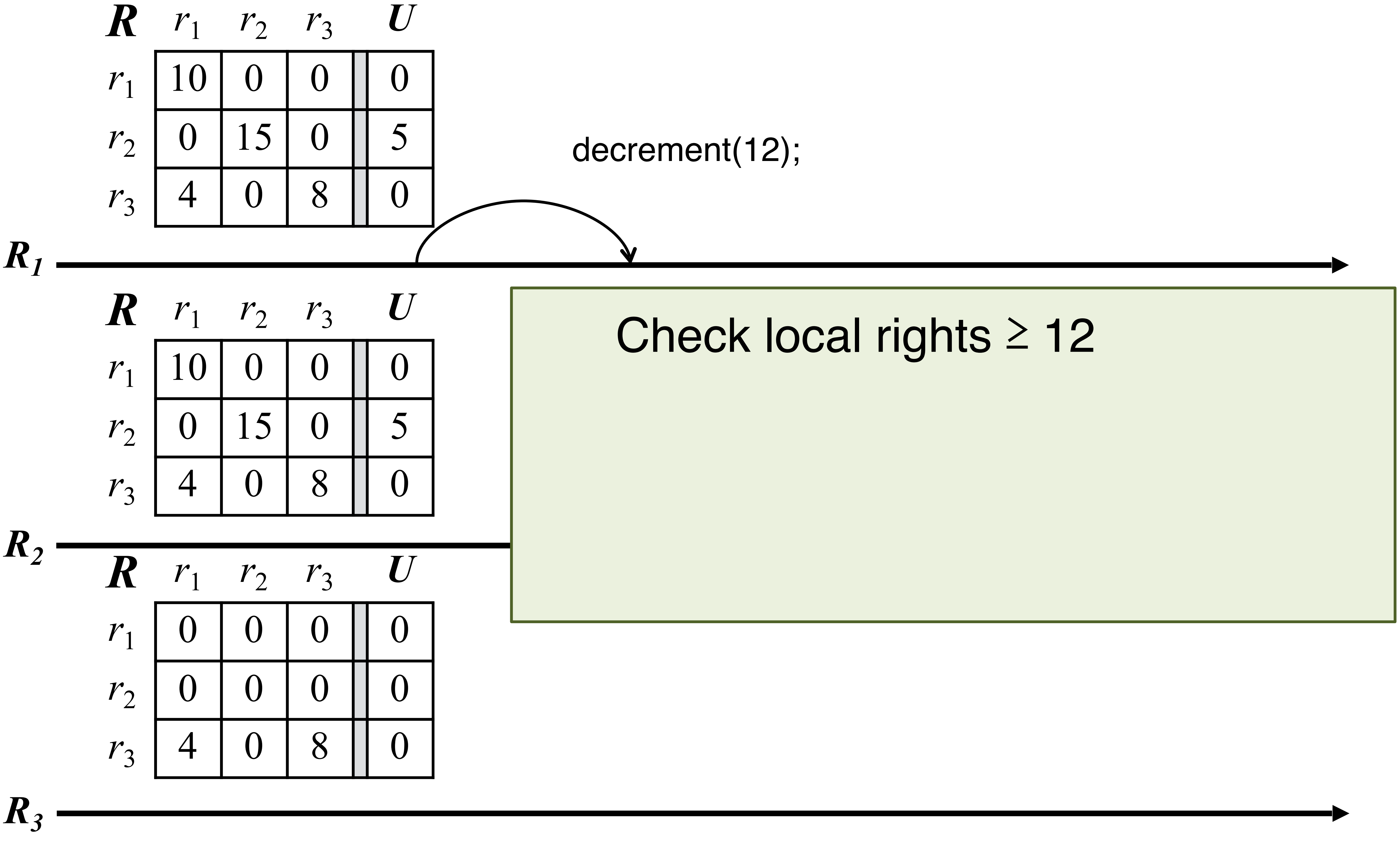
Each replica only touches his line. Merge by taking max of each cell.

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	15	0	5
r_3	0	0	0	0

R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	15	0	5
r_3	4	0	8	0

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	0	0	0
r_3	4	0	8	0

Bounded Counter: decrement



$$\begin{array}{c|ccc|c}
\boldsymbol{R} & r_1 & r_2 & r_3 & \boldsymbol{U} \\
\hline
r_1 & 10 & 0 & 0 & 0 \\
r_2 & 0 & 15 & 0 & 5 \\
r_3 & 4 & 0 & 8 & 0
\end{array}$$

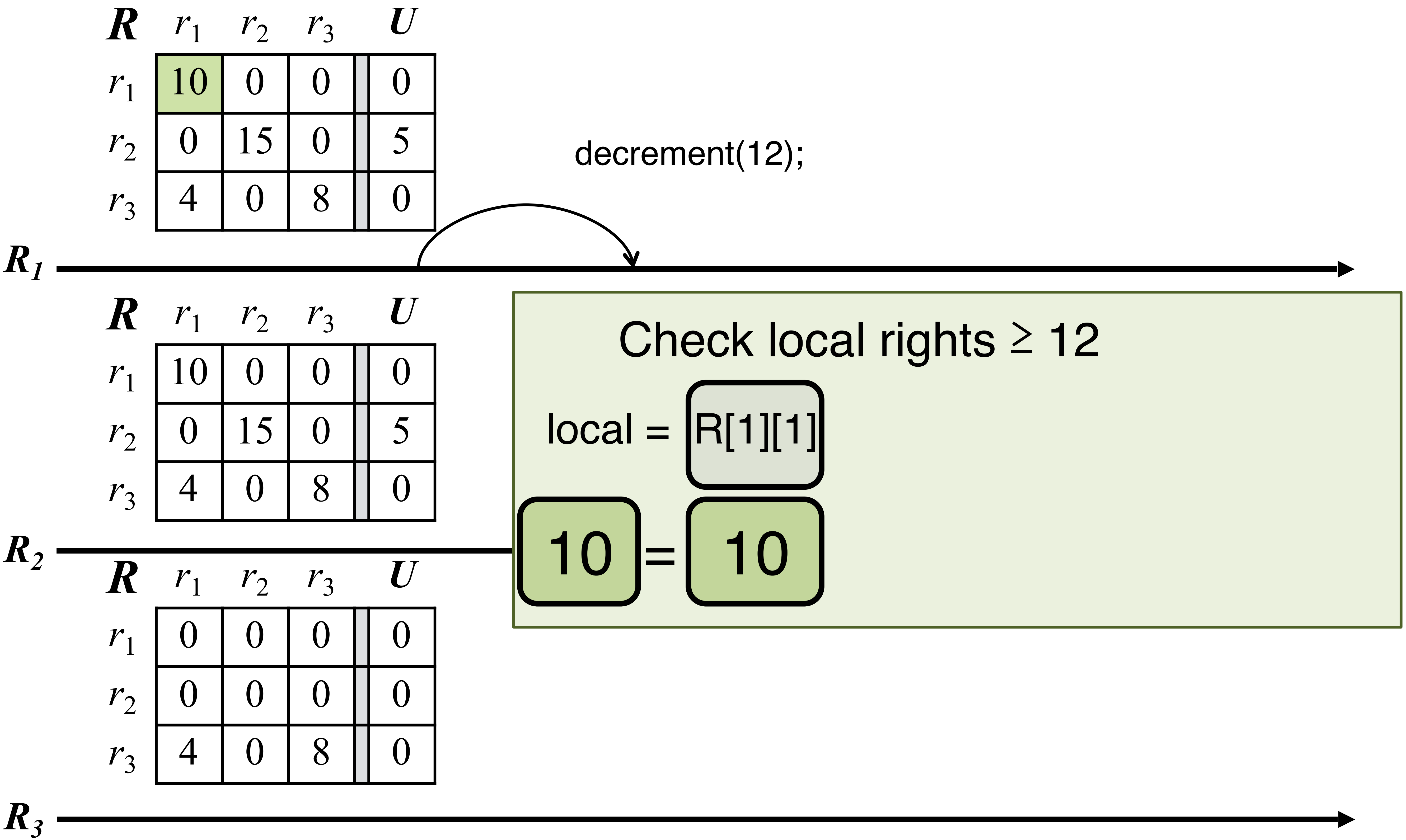
decrement(12);

$$\begin{array}{c|ccc|c}
\boldsymbol{R} & r_1 & r_2 & r_3 & \boldsymbol{U} \\
\hline
r_1 & 10 & 0 & 0 & 0 \\
r_2 & 0 & 15 & 0 & 5 \\
r_3 & 4 & 0 & 8 & 0
\end{array}$$

Check local rights ≥ 12

$$\begin{array}{c|ccc|c}
\boldsymbol{R} & r_1 & r_2 & r_3 & \boldsymbol{U} \\
\hline
r_1 & 0 & 0 & 0 & 0 \\
r_2 & 0 & 0 & 0 & 0 \\
r_3 & 4 & 0 & 8 & 0
\end{array}$$

Bounded Counter: decrement



R	r_1	r_2	r_3	***U***
r_1	10	0	0	0
r_2	0	15	0	5
r_3	4	0	8	0

decrement(12);

R	r_1	r_2	r_3	***U***
r_1	10	0	0	0
r_2	0	15	0	5
r_3	4	0	8	0

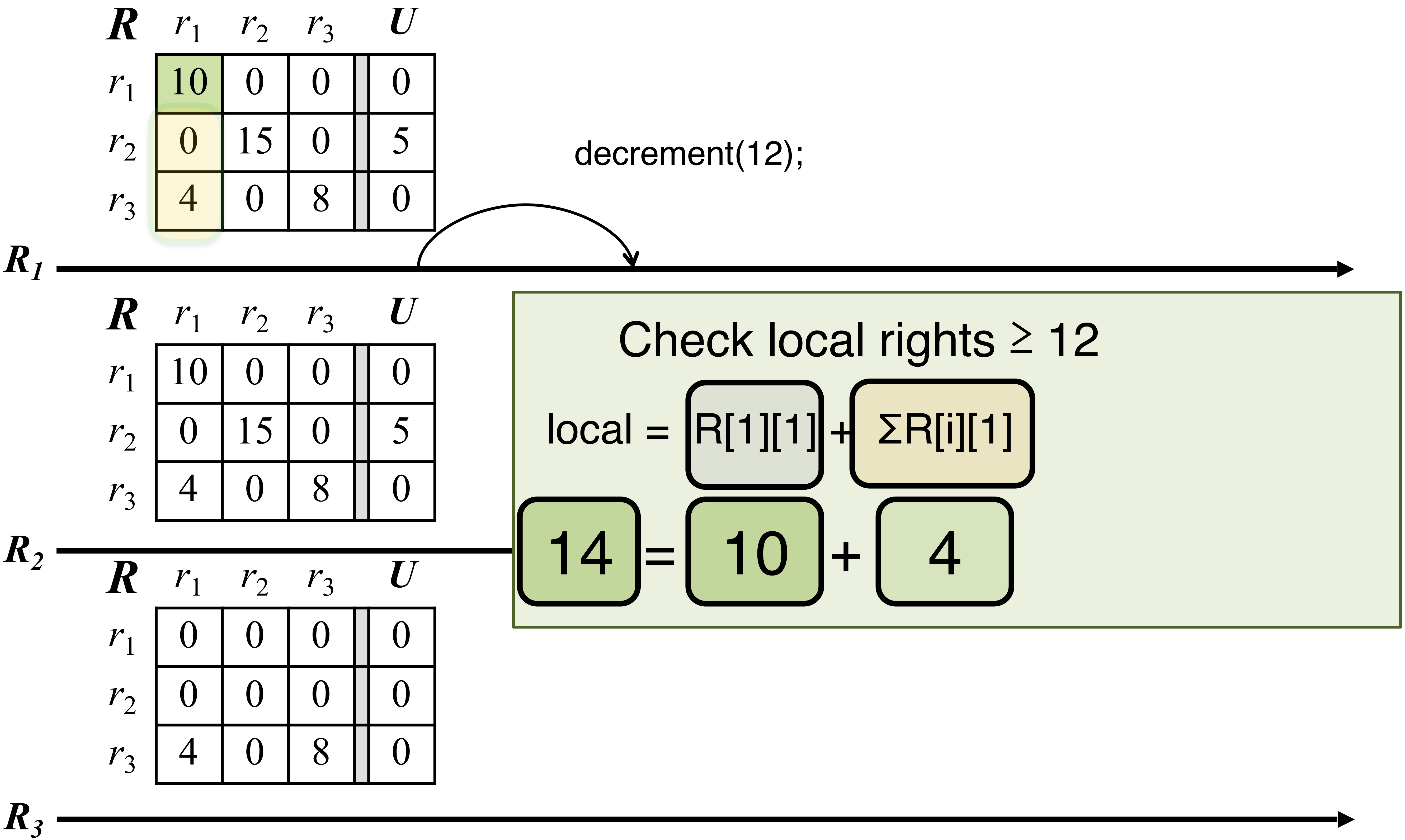
R	r_1	r_2	r_3	***U***
r_1	0	0	0	0
r_2	0	0	0	0
r_3	4	0	8	0

Check local rights ≥ 12

local = R[1][1]

10 = 10

Bounded Counter: decrement



R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	15	0	5
r_3	4	0	8	0

decrement(12);

R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	15	0	5
r_3	4	0	8	0

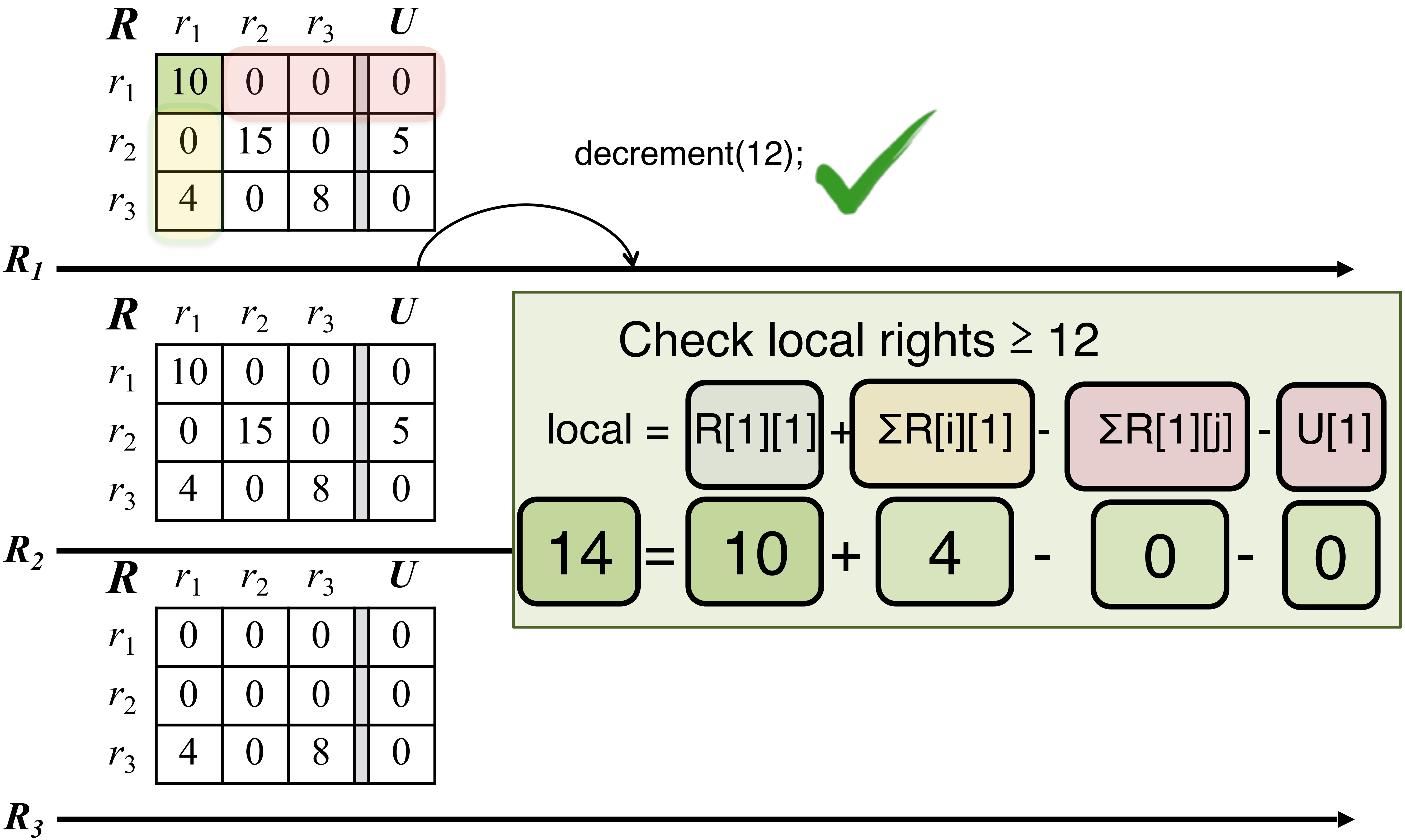
R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	0	0	0
r_3	4	0	8	0

Check local rights ≥ 12

local = R[1][1] + ΣR[i][1]

14 = 10 + 4

Bounded Counter: decrement



R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	15	0	5
r_3	4	0	8	0

decrement(12); ✓

R	r_1	r_2	r_3	U
r_1	10	0	0	0
r_2	0	15	0	5
r_3	4	0	8	0

R	r_1	r_2	r_3	U
r_1	0	0	0	0
r_2	0	0	0	0
r_3	4	0	8	0

Check local rights ≥ 12

local = R[1][1] + ΣR[i][1] - ΣR[1][j] - U[1]

14 = 10 + 4 - 0 - 0

Using Bounded Counter

Operation execute locally; fail if no rights available

Redistribute rights

- On-demand when needed
- Proactive

Peer-to-peer synchronization

Prototype implemented on top of Riak

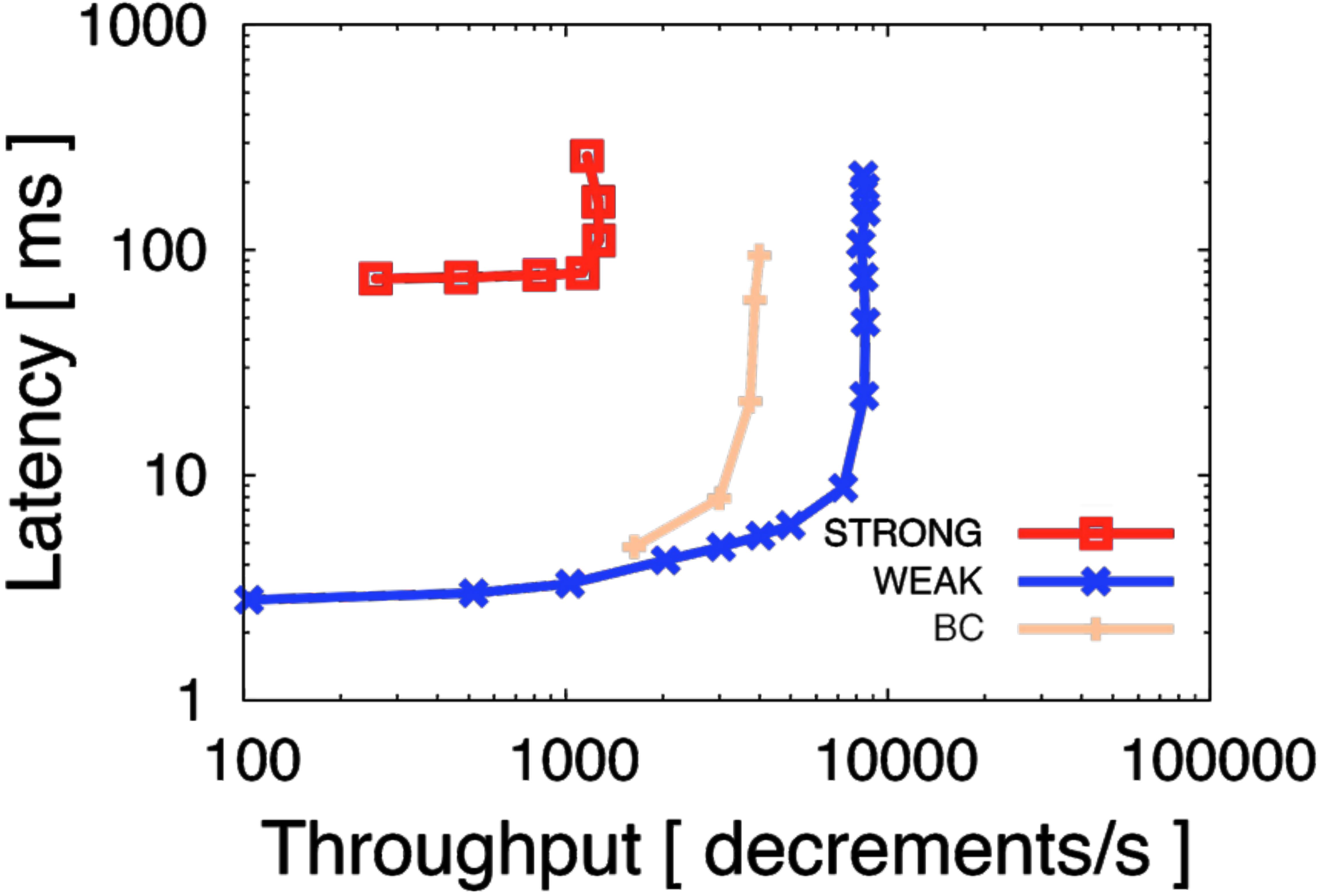
Micro Benchmark

Deployment: 3 Regions on AWS (m1.large)

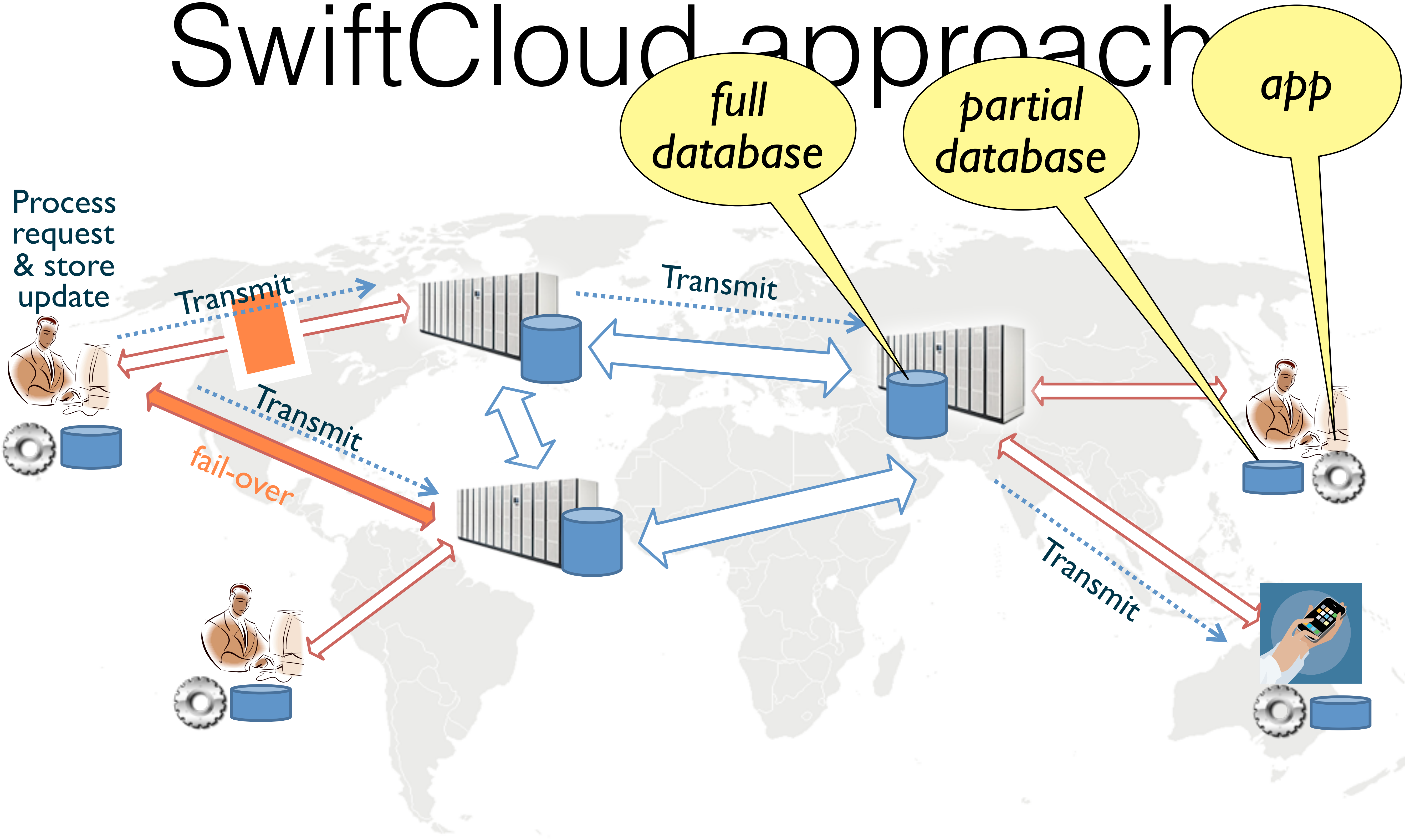
Configurations:

- STRONG - Strong consistency (all writes on 1 DC);
- WEAK- Eventual Consistency (Riak Counters);
- BC - Bounded Counter.

Latency for multiple keys



SwiftCloud approach



SwitftCloud key features

Cache data at clients

- Modify cached data => low latency, high availability

Highly available transactions

- Atomic updates
- Read snapshot
- CRDT rules for margining concurrent updates

Causal consistency

- Write fast, read in the past
- Client-assisted failover

SwiftSocial

High-level operations

- Registering user, Login/Logout
- Post status update; send message
- View wall
- Friendship management

Operations modeled as transactions

State

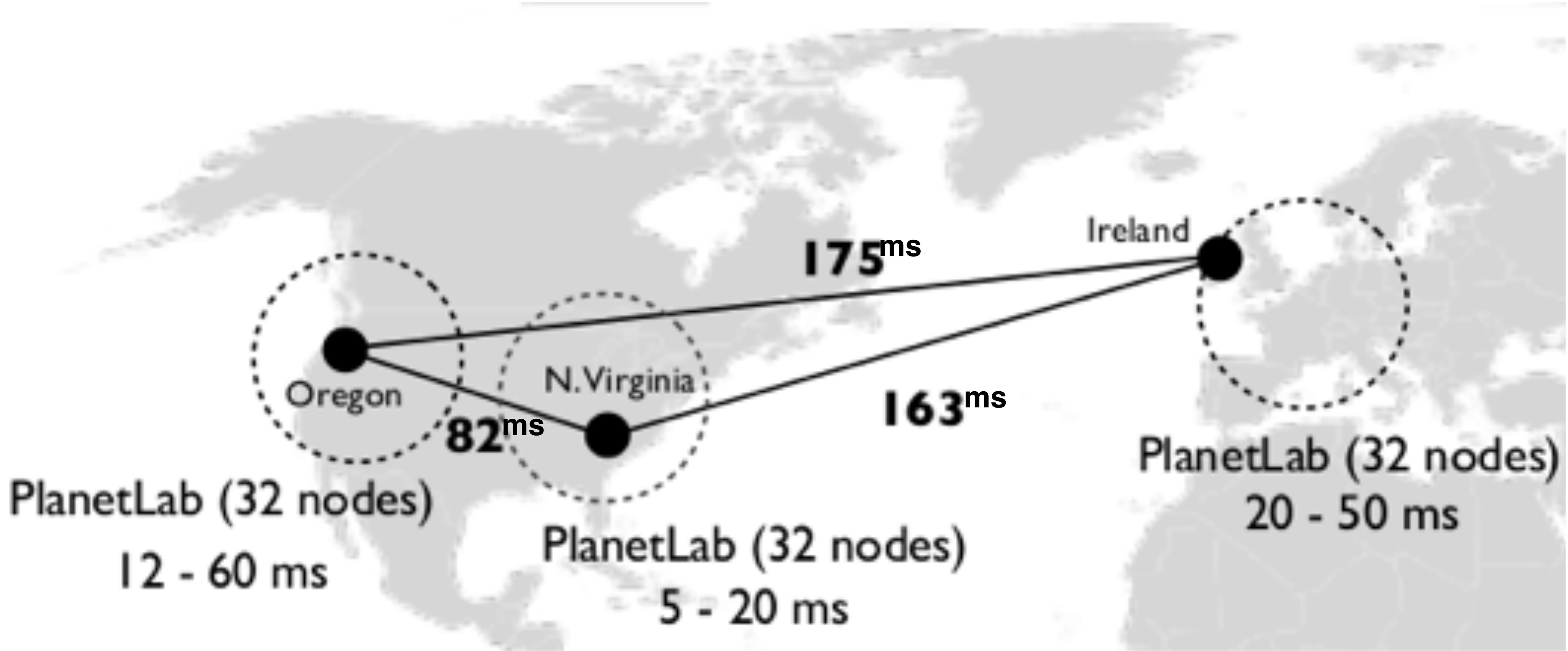
- Set CRDT for messages and friends
- Register CRDT for user data
- Counter CRDT for polls

Swift FS

Directory: (name, type) → object

- Shallow Map
- create (n, t, v) ≈ add
    - Concurrent: merge v recursively
- remove (n,t): whole subtree
    - Concurrent create, edit: re-create
- Object-specific operations (e.g. graph)
- No move => can lead to cycles

Experiments



3 DCs in Amazon EC2

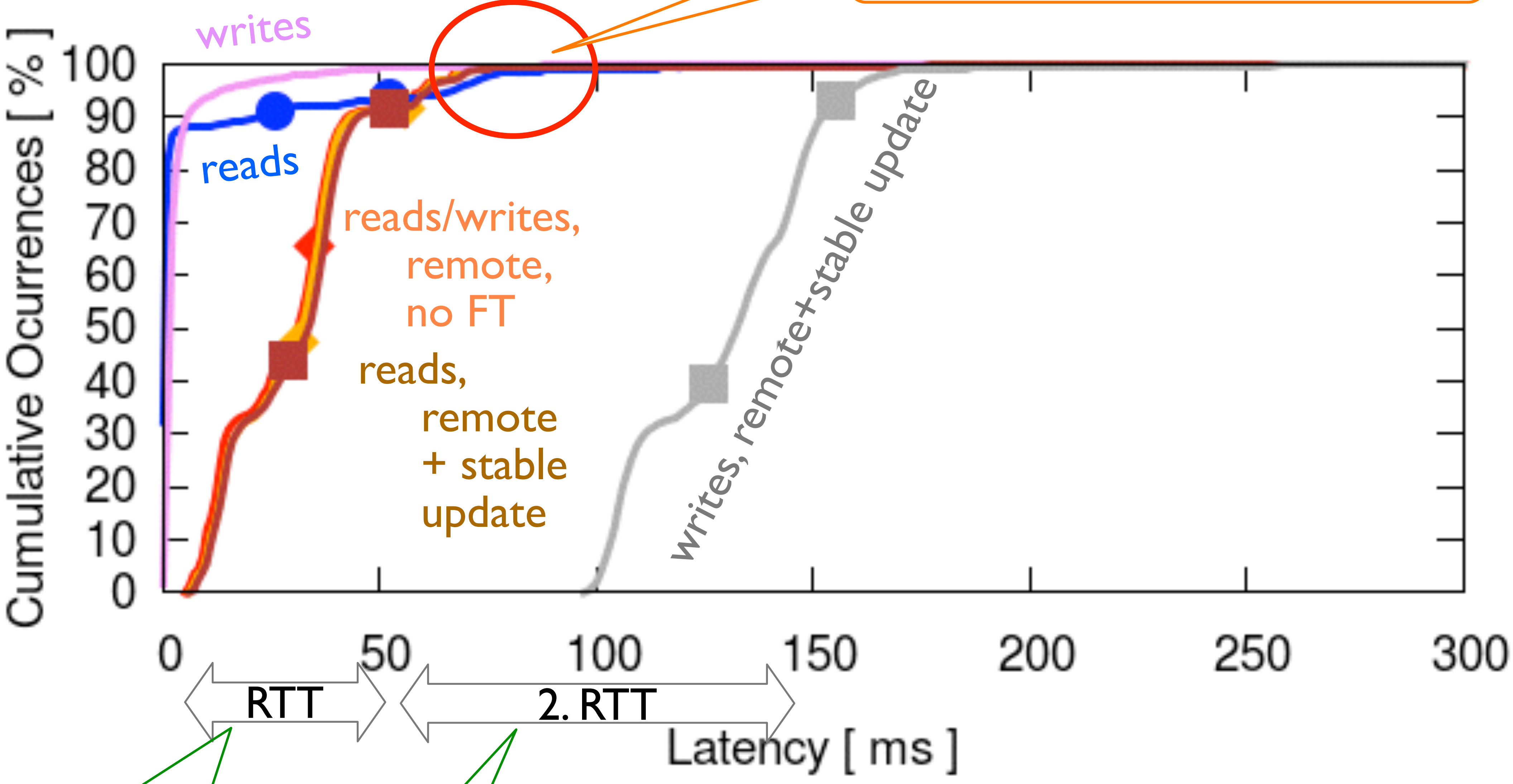
100 client nodes in PlanetLab

Cache size: 512 objects

SwiftSocial: 90% cache hits

Update caching + Read-In-Past minimize l

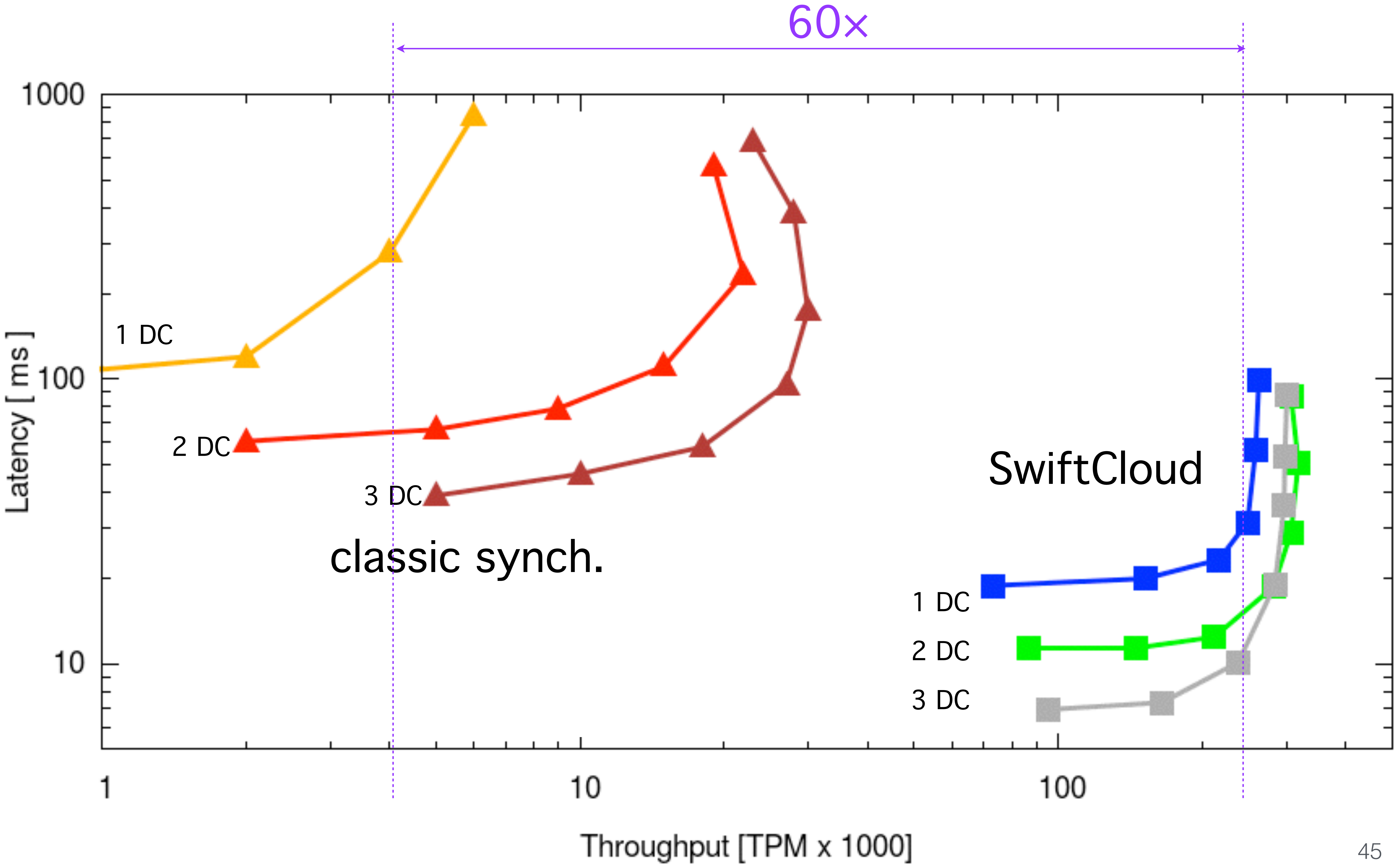
Operations with > 1 cache miss



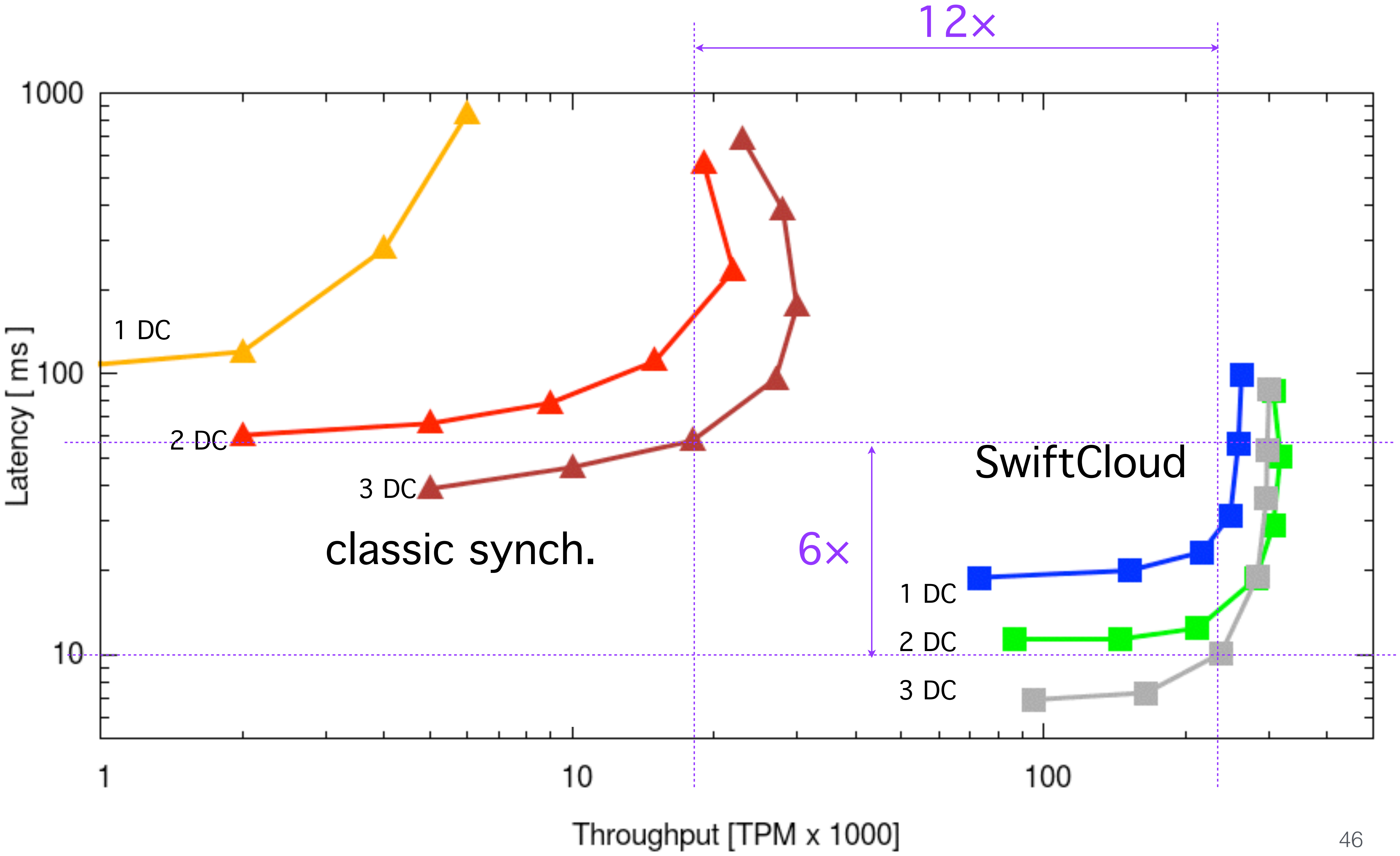
Client-side caching & updates

Read-in-past + client-assisted fault tolerance

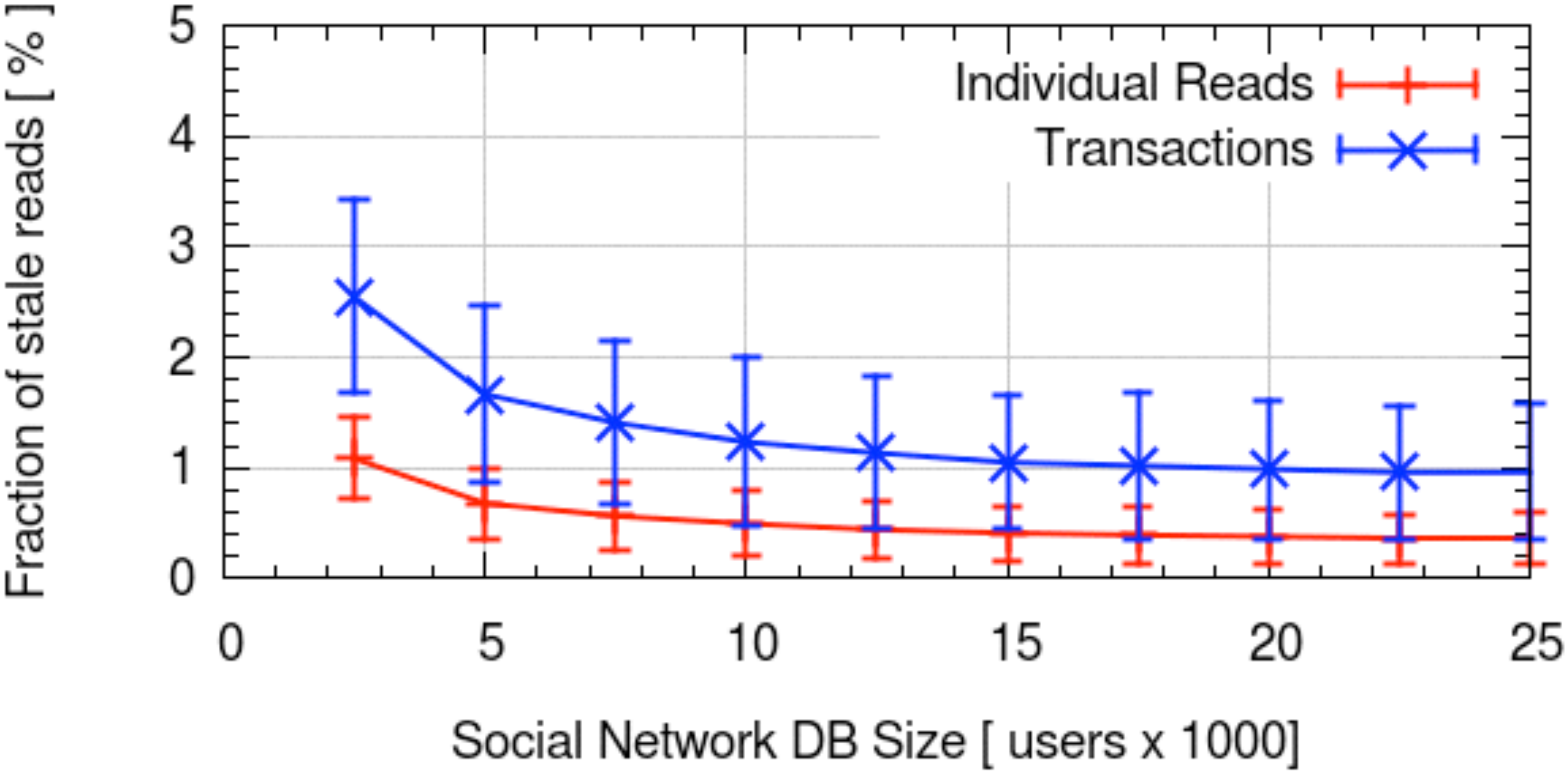
Latency vs. throughput



Latency vs. throughput



Staleness for fault tolerance



Summary

Applications requires multiple CRDTs

- Composition (e.g. Rick Map)

Need to lower expectations…

… but still possible to enforce some invariants

- Multi-key updates: HATs
- Causality
- Numeric invariants
- General invariants: red-blue, just-right consistency

Acknowledgments

SyncFree

European FP7 project #609 551, 2013--2016

Masoud Saeida-Ardakani, Carlos Baquero, Valter Balegas, Annette Bieniusa, Russell Brown, Sérgio Duarte, Carla Ferreira, Alexey Gotsman, Mahsa Najafzadeh, Marek Zawirski, and more.