Distributed Software Transactional Memories

Foundations, Algorithms and Tools

Maria Couceiro
(with Paolo Romano and Luís Rodrigues)
IST/ INESC-ID

Contents

- Part I: (Non-Distributed) STMs
- Part II: Distributed STMs
- Part III: Case-studies
- Part IV: Conclusions

Contents

- **Part I: (Non-Distributed) STMs**
- Part II: Distributed STMs
- Part III: Case-studies
- Part IV: Conclusions

(Non-Distributed) STMs

- Basic Concepts
- Example Algorithms

Basic Concepts

- Concurrent programming has always been a challenge
- One needs to control the concurrent access to shared data by multiple threads
- This is hard for most programmers.
- Concurrent programming has been a "niche"

Basic Concepts

- In the past:
  - More performance via faster CPUs
- Now:
  - More performance via more CPUs
  - Concurrent programming has to become mainstream

Basic Concepts

- Ideally
  - Performance would scale linearly with the number of cores
  - (with 8 cores we would have a program 8 times faster)
- Reality:
  - Speed up limited by % serial code
    - Small % can kill performance (Amdahl's Law)
  - Say 25% of the program is serial
  - 8 cores = 2.9 speedup.

Basic Concepts

- Ideally
  - Performance would scale linearly with the number of cores
  - (with 8 cores we would have a program 8 times faster)
- Reality:
  - Small % of serial code can kill performance (Amdahl's Law)
  - Say 25% of the program is serial
  - 32 cores = 3.7 speedup.

Basic Concepts

- Ideally
  - Performance would scale linearly with the number of cores
  - (with 8 cores we would have a program 8 times faster)
- Reality:
  - Small % of serial code can kill performance (Amdahl's Law)
  - Say 25% of the program is serial
  - 128 cores = 3.9 speedup.

Basic Concepts

- It is hard or impossible to structure a program in a set of parallel independent tasks.
- We need efficient and simple mechanisms to manage concurrency.

Explicit synchronization

- One of the most fundamental and simple synchronization primitive is the lock

```
non-synchronized code;
lock ();
do stuff on shared data;
unlock ();
more non-synchronized code;
```

Many problems with locks

- Deadlock:
  - locks acquired in “wrong” order.
- Races:
  - due to forgotten locks
- Error recovery tricky:
  - need to restore invariants and release locks in exception handlers

Fine Grained Parallelism?

- Very complex:
  - Need to reason about deadlocks, livelocks, priority inversions.
  - Verification nightmare as bugs may be hard to reproduce.
- Make parallel programming accessible to the masses!!!

Concurrent Programming Without Locks

- Lock-free algorithms.
- Hard to design and prove correct.
- Only for very specialized applications.
- Designed and implemented by top experts.

Abstractions for simplifying concurrent programming…

Atomic transactions

```
atomic {
    access object 1;
    access object 2;
}
```

Transactional Memories

- Hide away synchronization issues from the programmer.
- Advantages:
  - avoid deadlocks, priority inversions, convoying;
  - simpler to reason about, verify, compose.

TMs: where we are, challenges, trends

- Theoretical Aspects
  - Formalization of adequate consistency guarantees, performance bounds.
- Hardware support
  - Very promising simulation-based results, but no support in commercial processors.

TMs: where we are, challenges, trends

- Software-based implementations (STM)
  - Performance/scalability improving, but overhead still not satisfactory.
- Language integration
  - Advanced supports (parallel nesting, conditional synchronization) are appearing...
  - ...but lack of standard APIs & tools hampers industrial penetration.

TMs: where we are, challenges, trends

- Operating system support
  - Still in its infancy, but badly needed (conflict aware scheduling, transactional I/O).
- Recent trends:
  - Shift towards **distributed** environments to enhance scalability & dependability.

Run-time

- How does it work?
  - The run time implements concurrency control in an automated manner.
- Two main approaches:
  - Pessimistic concurrency control (locking).
  - Optimistic concurrency control.

Example of pessimistic concurrency control

- Each item has a read/write lock.
- When an object is read, get the read lock.
  - Block if write lock is taken.
- When an object is written, get the write lock.
  - Block if read or write lock is taken.
- Upon commit/abort:
  - Release all locks.

Example of optimistic concurrency control

- Each item has a version number.
- Read items and store read version.
- Write local copy of items.
- Upon commit do atomically:
  - If all read items still have the read version (no other concurrent transaction updated the items)
    - then apply all writes (increasing the version number of written items).
  - Else,
    - abort.

Many, many, variants exist

- For instance, assume that two phase locking is used and a deadlock is detected. It is possible:
  - Abort both transactions.
  - Abort the oldest transaction.
  - Abort the newest transaction.
  - Abort the transaction that did less work.

Many, many, variants exist

- For instance, assume that two phase locking is used and a deadlock is detected. It is possible:
  - Abort both transactions
  - Abort the oldest transaction
  - Abort the newest transaction
  - Abort the transaction that did less work

Each alternative offers different performance with different workloads.

How to choose?

- What is a correct behavior?
- Which safety properties should be preserved?
- Which liveness properties should be preserved?

How to choose?

- What is a correct behavior?
- Which safety properties should be preserved?
- Which liveness properties should be preserved?

To answer these questions we need a bit of theory.

Theoretical Foundations

- Safety:
  - What schedules are acceptable by an STM?
  - Is classic atomicity property appropriate?
- Liveness:
  - What progress guarantees can we expect from an STM?

Theoretical Foundations

- Safety:
  - What schedules are acceptable by an STM?
  - Is classic atomicity property appropriate?
- Liveness:
  - What progress guarantees can we expect from an STM?

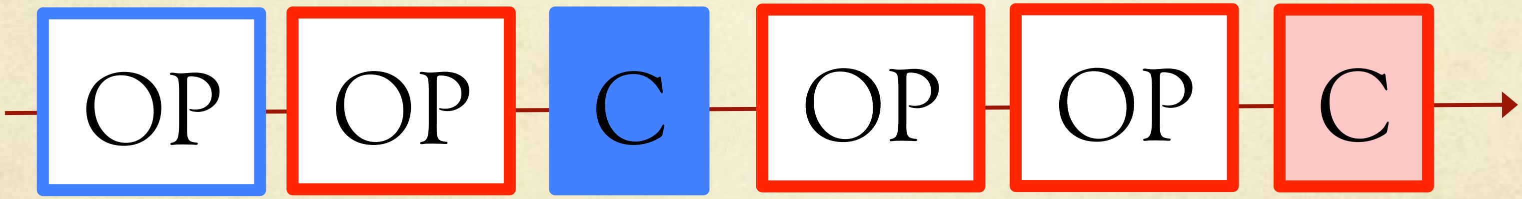
Classic atomicity property

- A transaction is a sequence of read/write operations on variables:
  - sequence unknown a priori (otherwise called static transactions).
  - asynchronous (we do not know a priori how long it takes to execute each operation).
- Every operation is expected to complete.
- Every transaction is expected to abort or commit.

Histories

- The execution of a set of transactions on a set of objects is modeled by a **history**

- A history is a total order of operation, commit and abort events

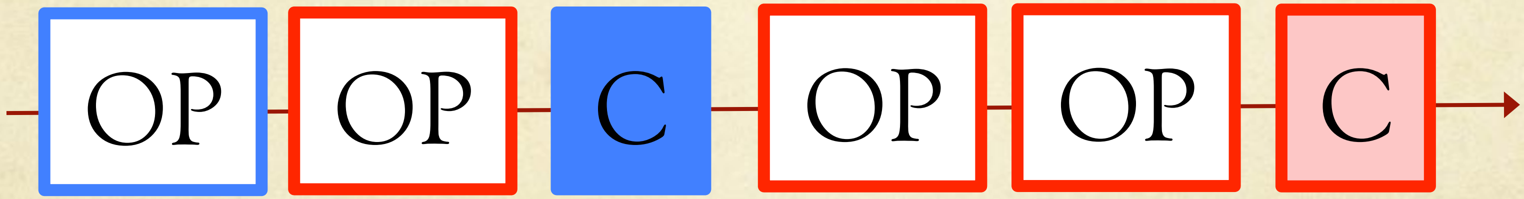


Histories

- Two transactions are **sequential** (in a history) if one invokes its first operation after the other one commits or aborts; they are **concurrent** otherwise.

Histories

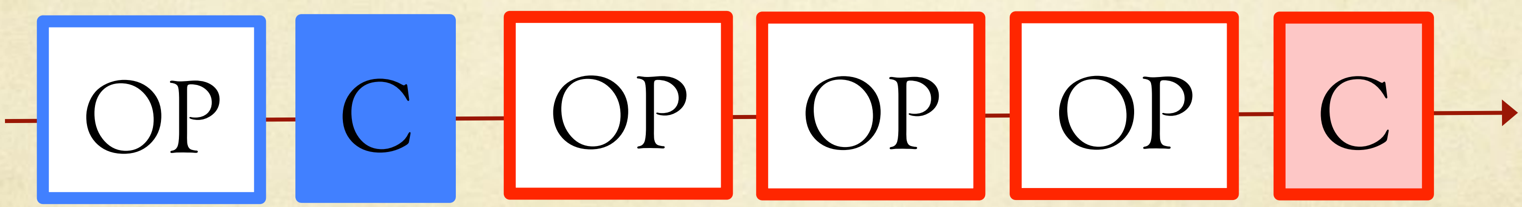
- Two transactions are **sequential** (in a history) if one invokes its first operation after the other one commits or aborts; they are **concurrent** otherwise.
- Non-sequential:



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Histories

- Two transactions are **sequential** (in a history) if one invokes its first operation after the other one commits or aborts; they are **concurrent** otherwise.
- Sequential:

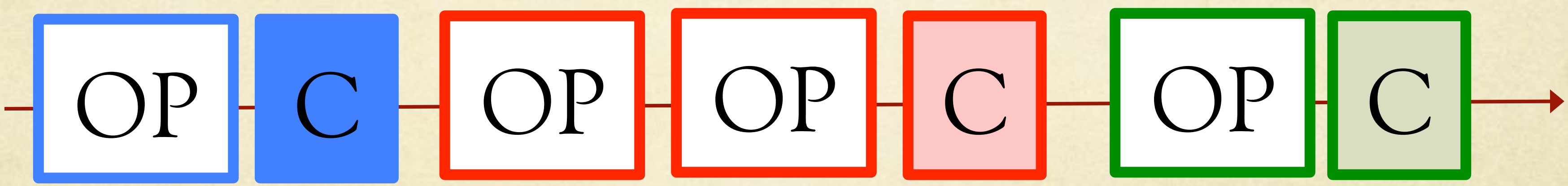


Histories

- A history is sequential if it has only sequential transactions; it is concurrent otherwise

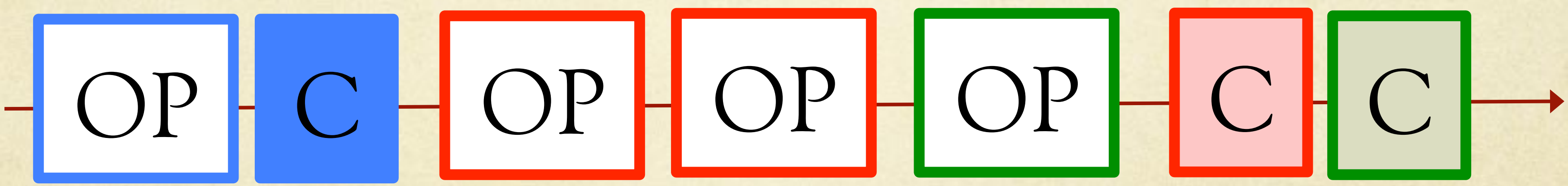
Histories

- A history is sequential if it has only sequential transactions; it is concurrent otherwise.
- Sequential:



Histories

- A history is sequential if it has only sequential transactions; it is concurrent otherwise.
- Non-sequential:



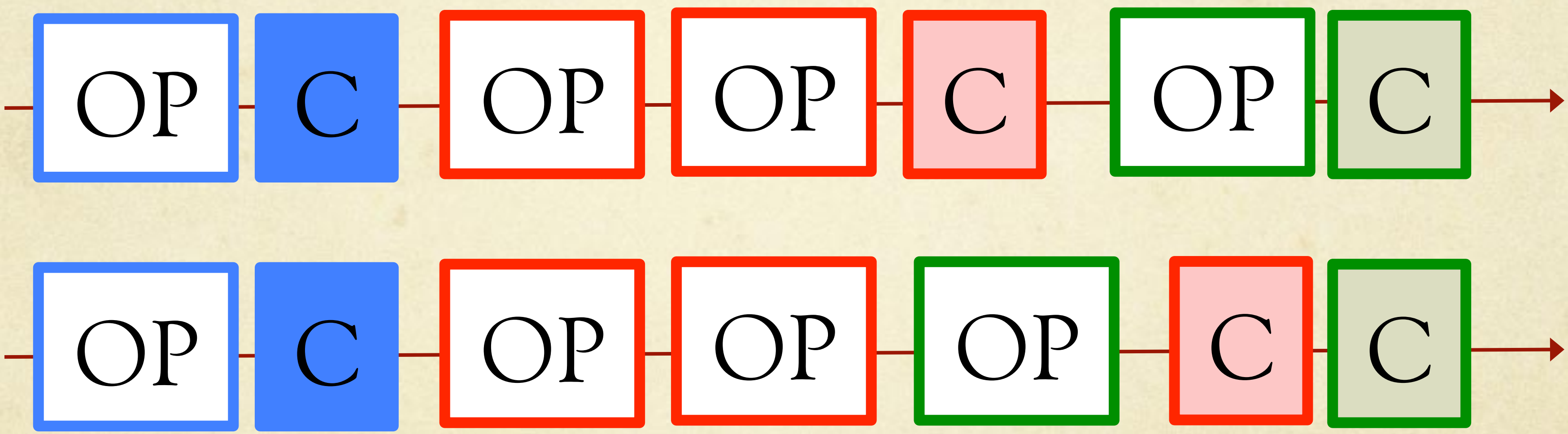
M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Histories

- Two histories are **equivalent** if they have the same transactions.

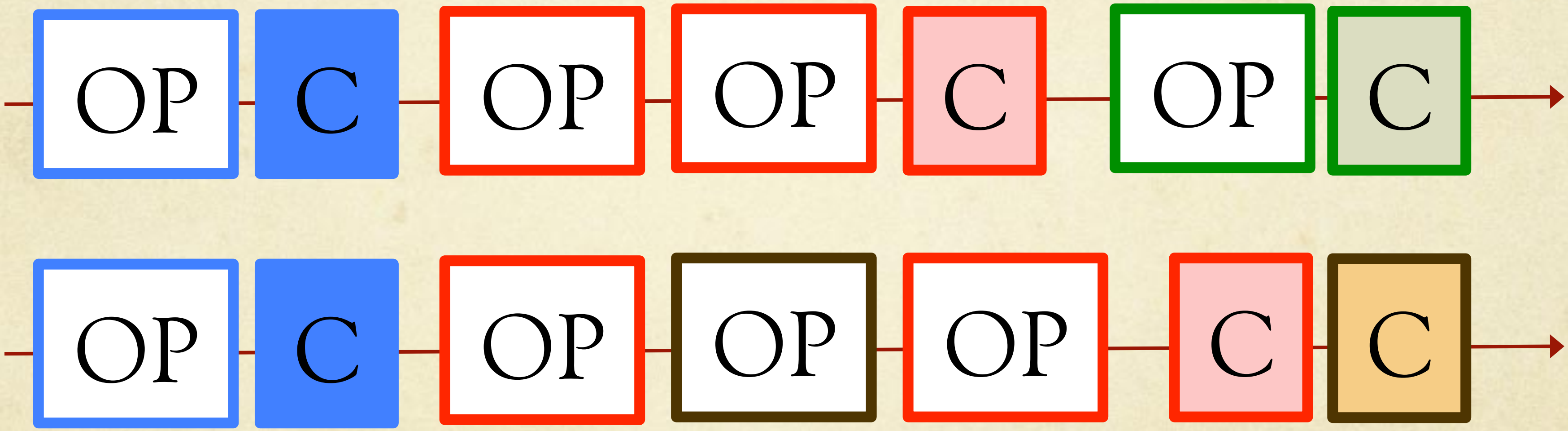
Histories

- Two histories are **equivalent** if they have the same transactions
- **Equivalent**:



Histories

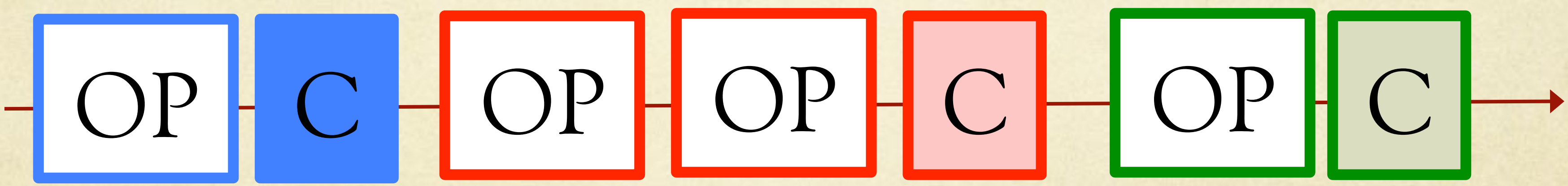
- Two histories are **equivalent** if they have the same transactions
- **Non-equivalent:**



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

What the programmer wants?

- Programmer does not want to be concerned about concurrency issues.
- Execute transactions "as if" they were serial
- No need to be "serially executed" as long as results are the same



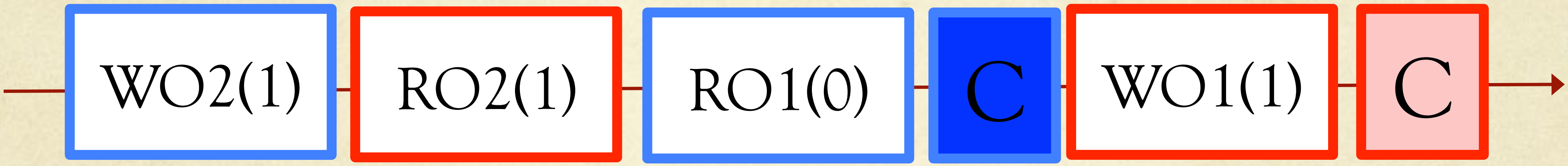
M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Serializability's definition (Papa79 - View Serializability)

- A history H of **committed** transactions is **serializable** if there is a history S(H) that is:
  - equivalent to H
  - sequential
  - **every read returns the last value written**

Serializability

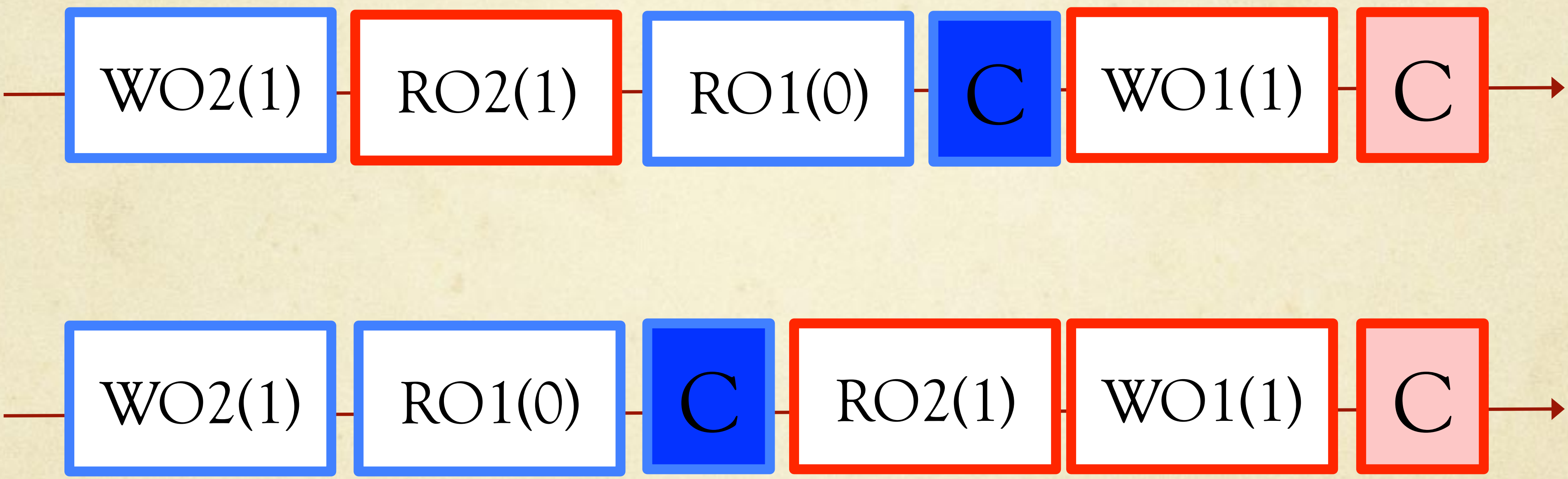
- Serializable?



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Serializability

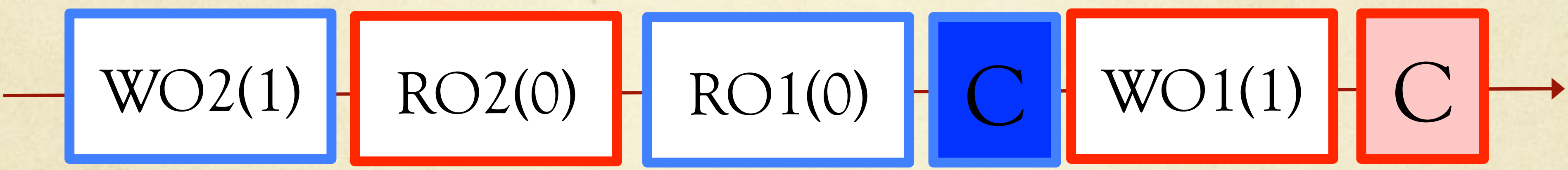
- Serializable!



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

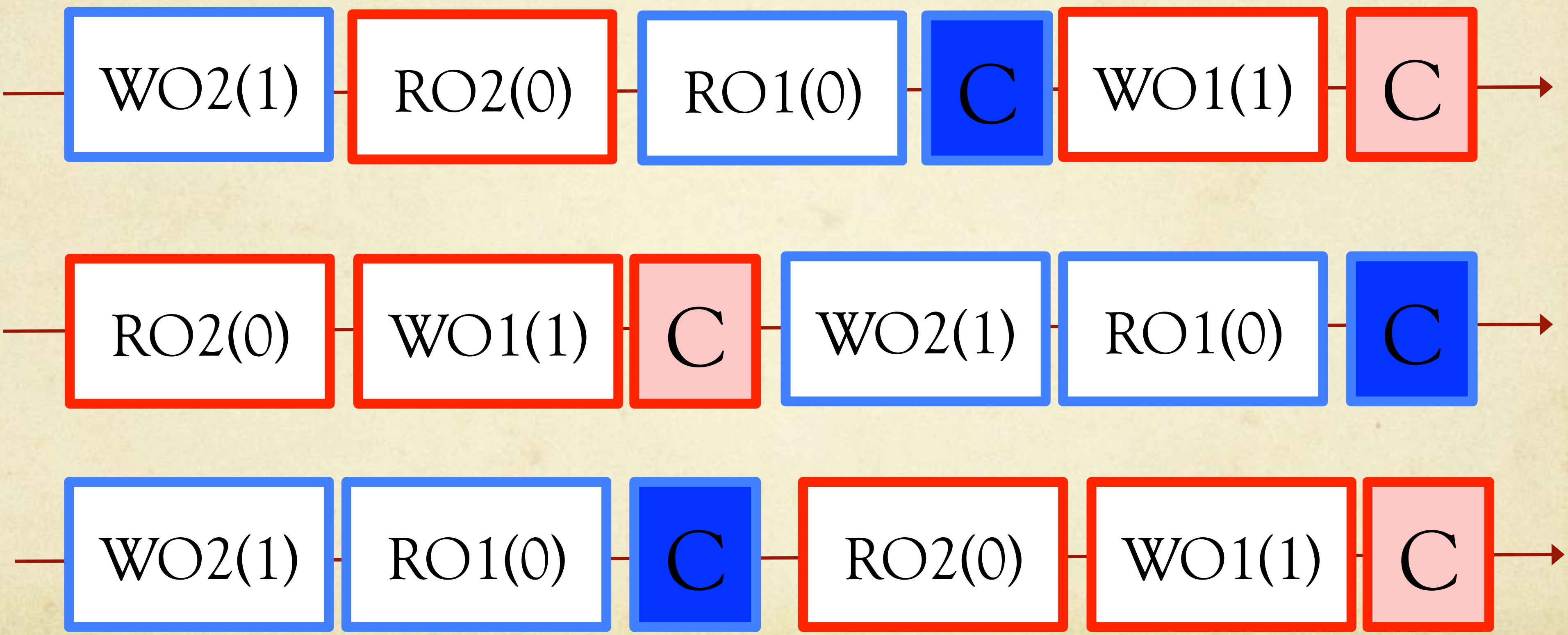
Serializability

- Serializable?



Serializability

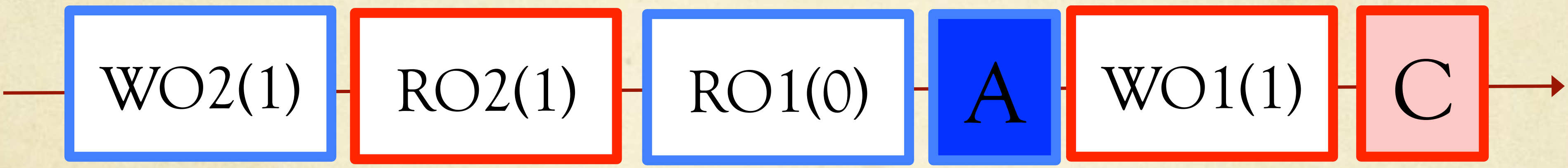
- Non-serializable!



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

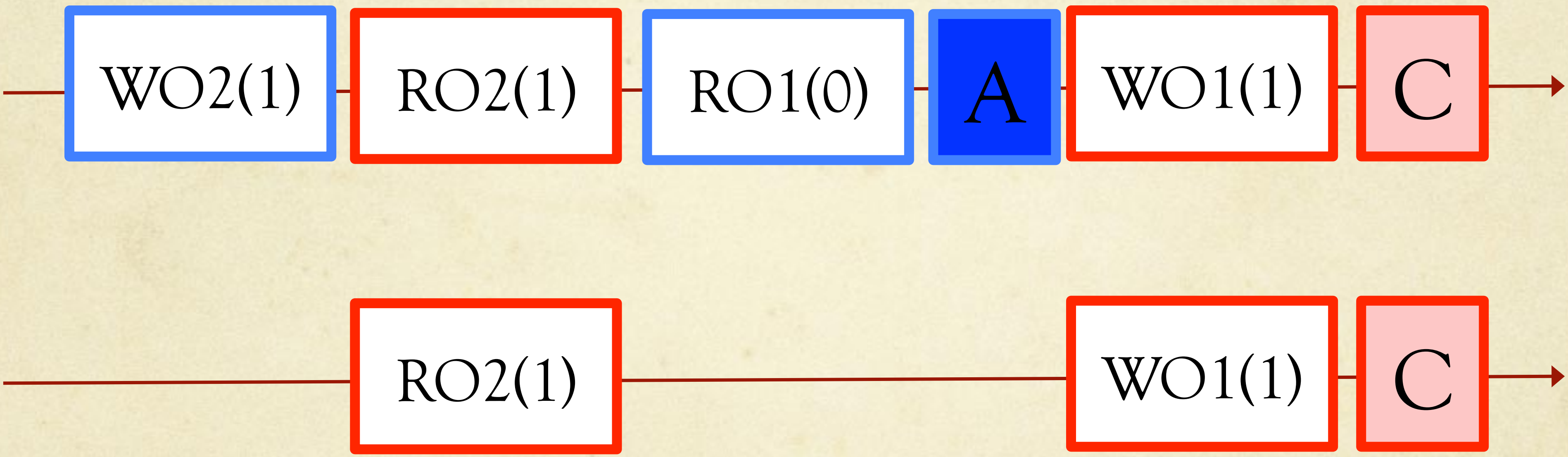
Opacity

- Serializable (blue aborts)?



Opacity

- Serializable: only committed transactions matter!



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Opacity

- In a database environment, transactions run SQL:
  - no harm if inconsistent values are read as long as the transaction aborts.
- This is not the same in a general programming language:
  - observing inconsistent values may crash or hang an otherwise correct program!

Opacity: example

Initially: x:=1; y:=2

- T1: x := x+1; y := y+1
- T2: z:= 1 / (y-x);

If T1 and T2 are atomic, the program is correct.

Opacity: example

Initially: x:=1; y:=2

- T1: x := x+1; y := y+1
- T2: z:= 1 / (y-x);

Otherwise…

Opacity: example

Initially: x:=1; y:=2

- T1: x := x+1; y := y+1
- T2: z:= 1 / (y-**x**);

Otherwise...

Opacity: example

Initially: x:=1; y:=2

- T1: x := x+1; y := y+1
- T2: z:= 1 / (2-**x**);

Otherwise…

Opacity: example

Initially: x:=1; y:=2

- T1: **x := x+1**; **y := y+1**
- T2: z:= 1 / (2-x);

Otherwise…

Opacity: example

Initially: x:=1; y:=2

After T1: x:=2; y:=3

- **T1: x := x+1; y := y+1**

- T2: z:= 1 / (2-x);

Otherwise...

Opacity: example

Initially: x:=1; y:=2

After T1: x:=2; y:=3

- **T1: x := x+1; y := y+1**
- T2: z:= 1 / (2-2);

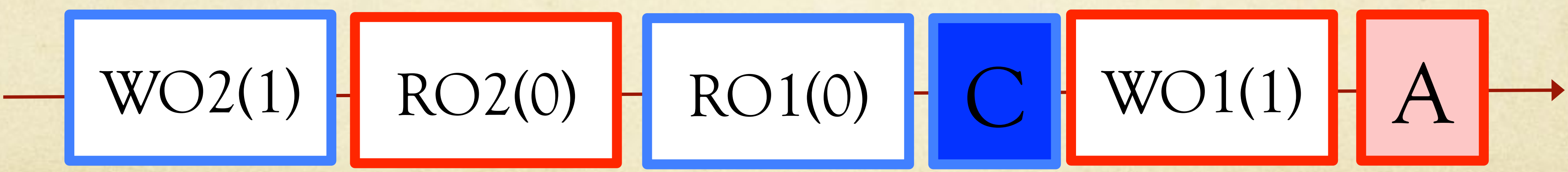
Otherwise…divide by zero!

Opacity [GK08]

- Intuitive definition:
  - every operation sees a consistent state (*even if the transaction ends up aborting*)

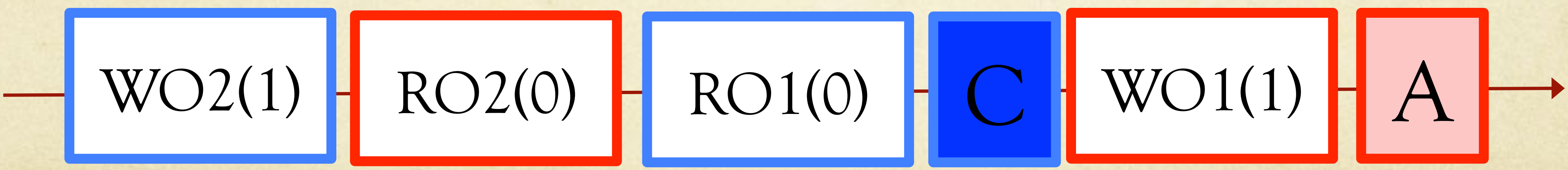
Opacity
[GK08]

- Intuitive definition:
  - every operation sees a consistent state (*even if the transaction ends up aborting*)
- Following history is serializable but violates opacity!



Does classic optimistic concurrency control guarantee opacity?

- Writes are buffered to private workspace and applied atomically at commit time
- Reads are optimistic and the transaction is validated at commit time.
- Opacity is not guaranteed!



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Theoretical Foundations

- Safety:
  - What schedules are acceptable by an STM?
  - Is classic atomicity property appropriate?

- **Liveness:**
  - **What progress guarantees can we expect from an STM?**

Progress

- STMs can abort transactions or block operations…
- But we want to avoid implementations that abort all transactions!
- We want operations to return and transactions to commit!

Requirements

- **Correct** transactions:
  - **commit** is invoked after a finite number of operations
  - either **commit** or perform an infinite number of (low-level) steps
- **Well-formed** histories:
  - every transaction that aborts is immediately repeated until it commits

Conditional progress: obstruction freedom

- A correct transaction that eventually does not encounter contention eventually commits

- …but what to do upon contention?

Contention-managers

- Abort is unavoidable
- But want to maximize the number of commits
- Obstruction freedom property: progress and correctness are addressed by different modules.

Contention-managers encapsulate policies for dealing with contention scenarios.

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Contention-managers

Let TA be executing and TB a new transaction that arrives and creates a conflict with TA.

CM: Aggressive

Let TA be executing and TB a new transaction that arrives and creates a conflict with TA.

- **Aggressive contention manager:**
    - always aborts TA

CM: Backoff

Let TA be executing and TB a new transaction that arrives and creates a conflict with TA.

- **Backoff contention manager:**
  - TB waits an exponential backoff time
  - If conflict persists, abort TA

CM: Karma

Let TA be executing and TB a new transaction that arrives and creates a conflict with TA.

- **Karma contention manager:**
  - Assign priority to TA and TB
    - Priority proportional to work already performed
  - Let Ba be how many times TB has been aborted
  - Abort TA if Ba > (TA-TB)

CM: Greedy

Let TA be executing and TB a new transaction that arrives and creates a conflict with TA.

- **Greedy contention manager:**
  - Assign priority to TA and TB based on start time
  - If TB<TA and TA not blocked then wait
  - Otherwise abort TA

(Non-Distributed) STMs

- Basic Concepts
- Example Algorithms
  - DSTM
  - JVSTM

(Non-Distributed) STMs

- Basic Concepts
- **Example Algorithms**
  - **DSTM**
  - **JVSTM**

DSTM

- Software transactional memory for dynamic-sized data structures.
- Herlihy, Luchangco, Moir, and Scherer, 2003.
- Prior designs: static transactions.
- DSTM: dynamic creation of transactional objects.

DSTM

- Killer write:
  - Ownership.
- Careful read:
  - Validation.

DSTM - Writes

- To write o, T requires a write-lock on o.
- T aborts T' if some T' acquired a write-lock on o:
  - Locks implemented via Compare & Swap.
- Contention manager can be used to reduce aborts.

DSTM – Reads and Validation

- Concurrent reads do not conflict.
- To read o, T checks if *all* objects read remain valid;
  - else abort T.
- Before committing, T checks if all objects read remain valid and releases all its locks.
  - Make sure that the transaction observes a consistent state.
  - If the validation fails, transaction is restarted.

DSTM - Why is careful read needed?

- No lock is acquired upon a read:
  - **invisible reads**
  - visible read invalidate cache lines
  - bad performance with read-dominate workloads due to high bus contention

- What if we validated only at commit time?

Serializability? Opacity?

DSTM - Why is careful read needed?

- No lock is acquired upon a read:
  - **invisible reads**
  - visible read invalidate cache lines
  - bad performance with read-dominate workloads due to high bus contention

- What if we validated only at commit time?

Serializability? Y

Opacity? N

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

(Non-Distributed) STMs

- Basic Concepts
- **Example Algorithms**
  - **DSTM**
  - **JVSTM**

JVSTM

- Java Versioned Software Transactional Memory.
- Cachopo and Rito-Silva. 2006.
- Versioned boxes as the basis for memory transactions.

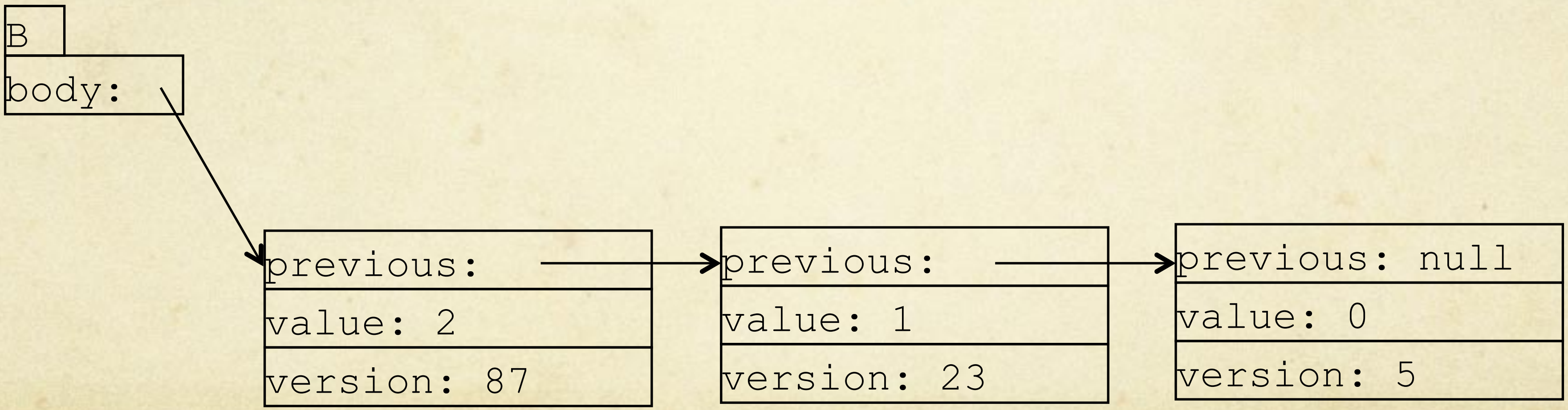
JVSTM

- Optimized for read-only transactions:
    - Never aborted or blocked;
    - No overhead associated with readset tracking.
- How?
    - Multi-version concurrency control.
    - Local writes (no locking, optimistic approach)
    - Commit phase in global mutual exclusion.
        - Recently introduced a parallel commit version [FC09].
    - Global version number (GVN)

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

JVSTM - Versioned boxes

- Versioned boxes
  - Each transactional location uses a versioned box to hold the history of values for that location.



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

JVSTM - Algorithm

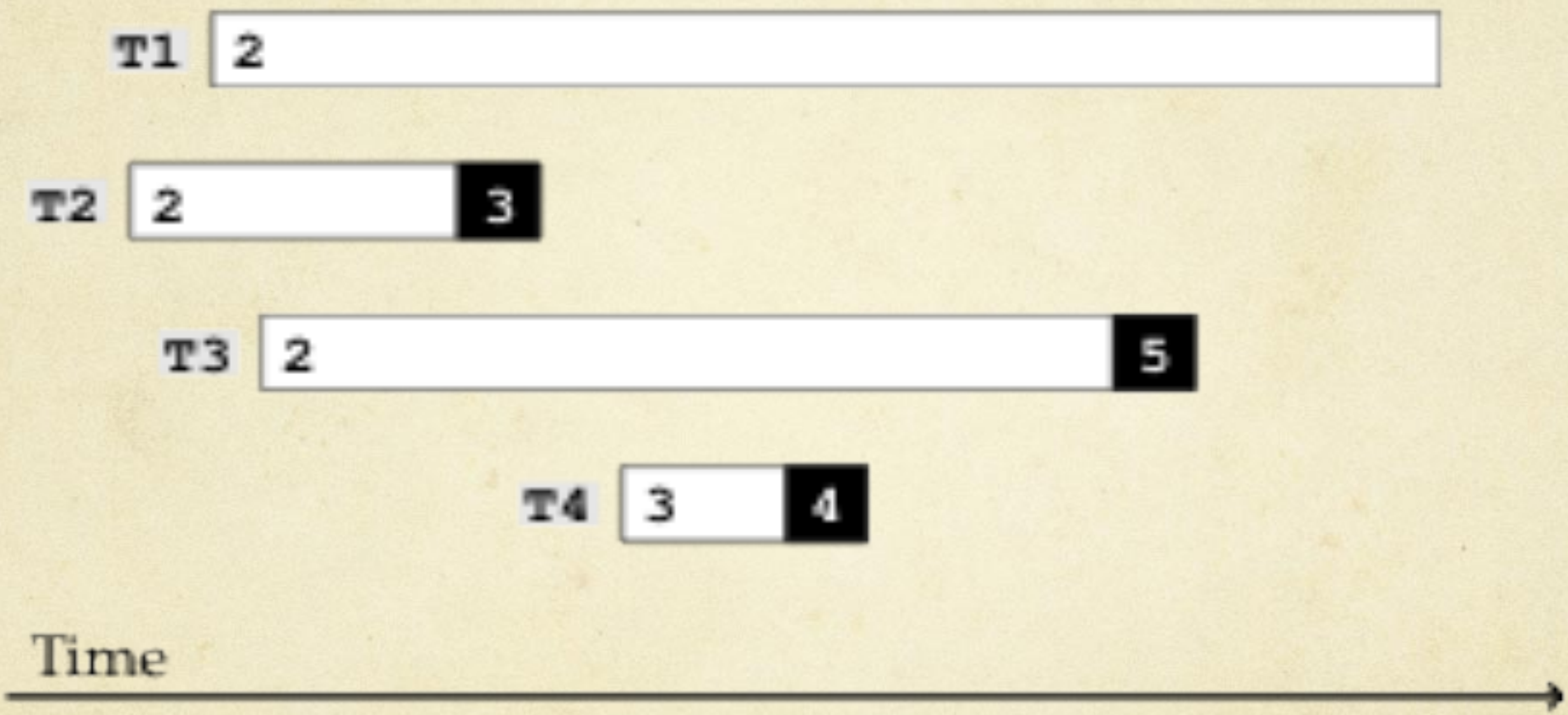
- Upon begin T, read GVN and assigned it to T snapshot ID (sID).
- Upon read on object o:
  - If o is in T's writeset, return last value written,
  - else return the version of the data item whose sID is "the largest sID to be smaller than the T' sID".
  - If T is not read-only, add o to readset.

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

JVSTM - Algorithm

- Upon write, just add to the writeset.
  - No early conflict detection.
- Upon commit:
  - Validate readset:
    - Abort if any object read has changed.
  - Acquire new sID (atomic increase of GVN).
  - Apply writeset: add new version in each written VBox .

JVSTM - Execution



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Contents

- Part I: (Non-Distributed) STMs
- **Part II: Distributed STMs**
- Part III: Case-studies
- Part IV: Conclusions

Distributed STMs

- Origins
- Goals
- Distribution Strategies
- Programming Models
- Toolbox

Origins

- Convergence of two main areas:
  - Distributed Shared Memory
  - Database Replication

Distributed Shared Memory

- DSM aims at providing a single system image
  - Fault-tolerance via checkpointing
- Strong consistency performs poorly
  - Myriad of weak-consistency models
  - Programming more complex
- Explicit synchronization
  - Locks, barriers, etc

DSTMs vs DSM

- DSTMs are simpler to program
- Transactions introduce boundaries where synchronization is required
- By avoiding to keep memory consistency at every (page) access or at the level of fine-grain locks, it may be possible to achieve more efficient implementations

Database Replication

- Databases use transactions
  - Constrained programming model
  - Durability is typically a must
- Database replication was considered too slow
- In the last 10 years new database replication schemes have emerged
  - Based on atomic broadcast and on a single coordination phase at the beginning/ end of the transaction.

DSTMs vs DBMS

- Transactions are often much shorter in the STM world
  - This makes coordination comparatively more costly
- Durability is often not an issue
  - This makes coordination comparatively more costly
- Database replication techniques can be used as a source of inspiration to build fault-tolerant DSTMs

Distributed STMs

- Origins
- Goals
- Distribution Strategies
- Programming Models
- Toolbox

Goals

- Better performance:
  - Doing reads in parallel on different nodes.
  - Computing writes in parallel on different items.
- Fault-tolerance:
  - Replication the memory state so that it survives the failure of a subset of nodes.

Distributed STMs

- Origins
- Goals
- Distribution Strategies
- Potential Problems
- Toolbox

Distribution Strategies

- Single System Image
  - Distribution is hidden
  - Easier when full replication is implemented
  - No control of the data locality
- Partitioned Global Address Space
  - Different nodes have different data
  - Distribution is visible to the programmer
  - Programmer has fine control of data locality
  - Complex programming model

Distribution Strategies

- Partitioned non-replicated
  - Max capacity
  - No fault-tolerance
  - No load balancing for reads on multiple nodes
- Full replication
  - No extra capacity
  - Max fault-tolerance
  - Max potential load balancing for reads

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Distributed STMs

- Origins
- Goals
- Distribution Strategies
- Programming Models
- Toolbox

Dataflow Model

- Transactions are immobile and objects move through the network.
- Write: processor locates the object and acquires ownership.
- Read: processor locates the object and acquires a read-only copy.
- Avoids distributed coordination.
- Locating objects can be very expensive.

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Control Flow Model

- Data is statically assigned to a home node and does not change over time.
- Manipulating objects:
  - In the node (via RPC);
  - First data is copied from the node then the are changes written back.
- Relies on fast data location mechanism.
- Static data placement may lead to poor data locality.

Distributed STMs

- Origins
- Goals
- Distribution Strategies
- Programming Models
- Toolbox

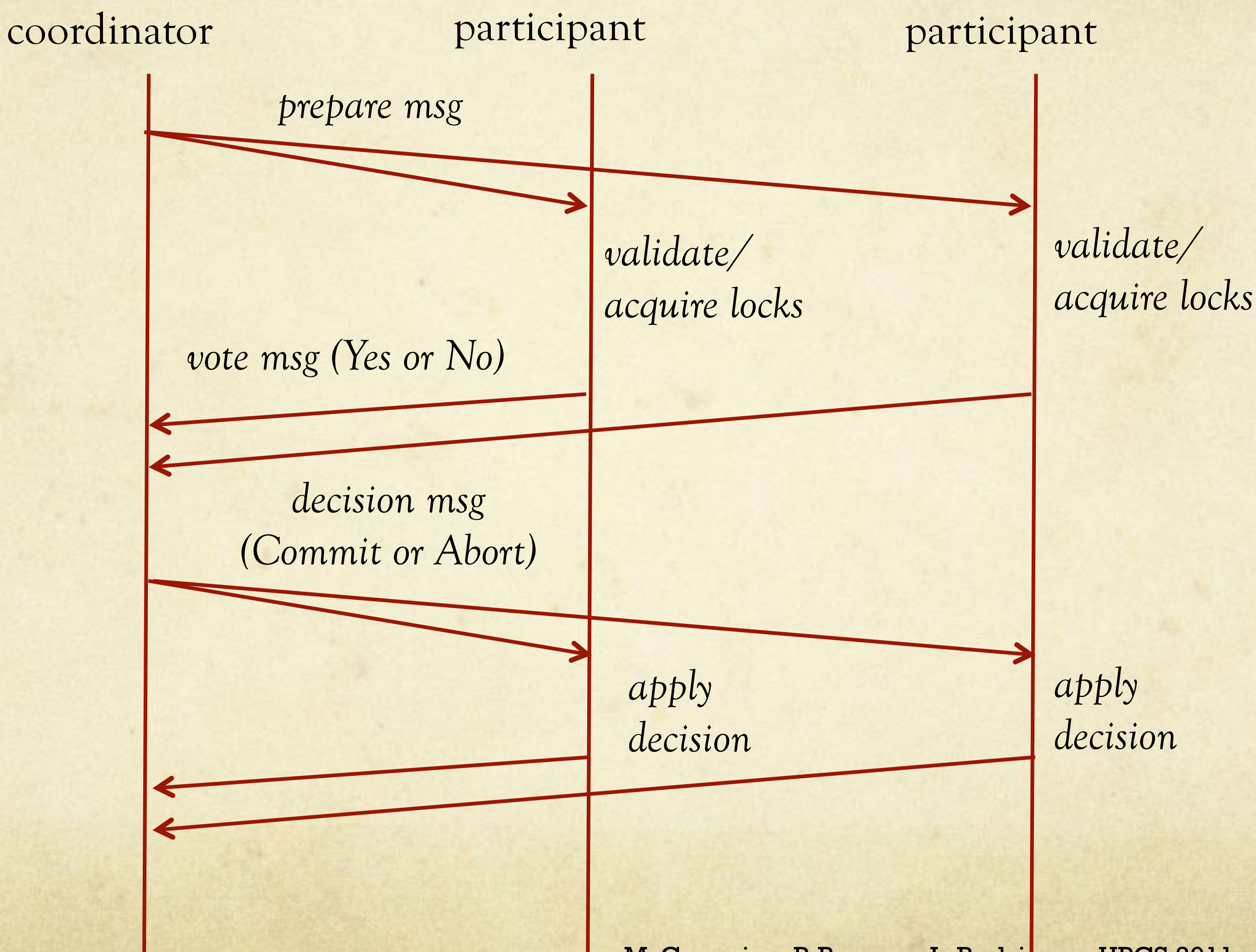
Toolbox

- **Atomic Commitment**
- Uniform Reliable Broadcast (URB)
- Atomic Broadcast (AB)
- Replication Strategies

Atomic Commitment

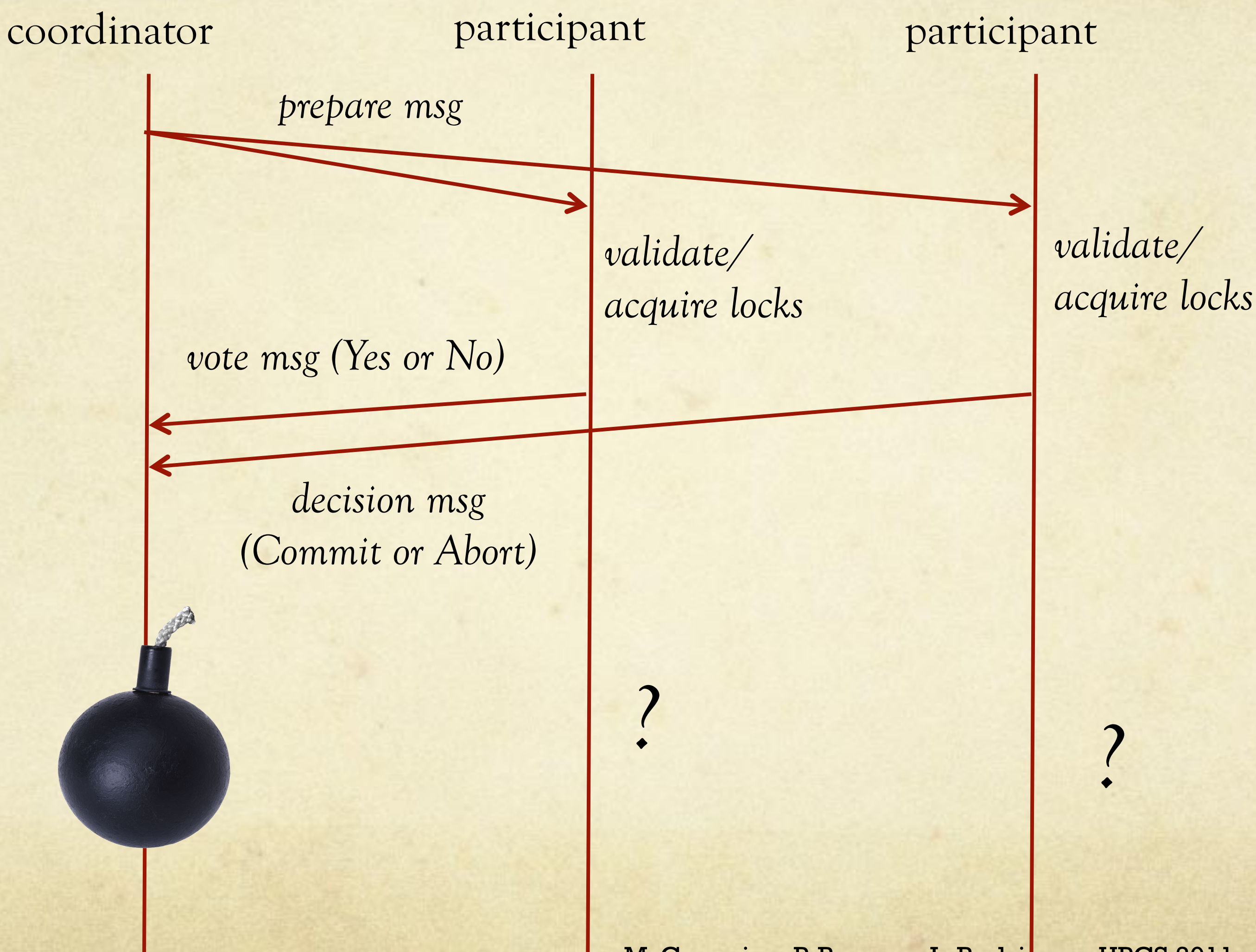
- Atomicity: all nodes either commit or abort the entire transaction.
- Set of nodes, each node has input:
  - CanCommit
  - MustAbort
- All nodes output same value
  - Commit
  - Abort
- Commit is only output if all nodes CanCommit

2-phase commit



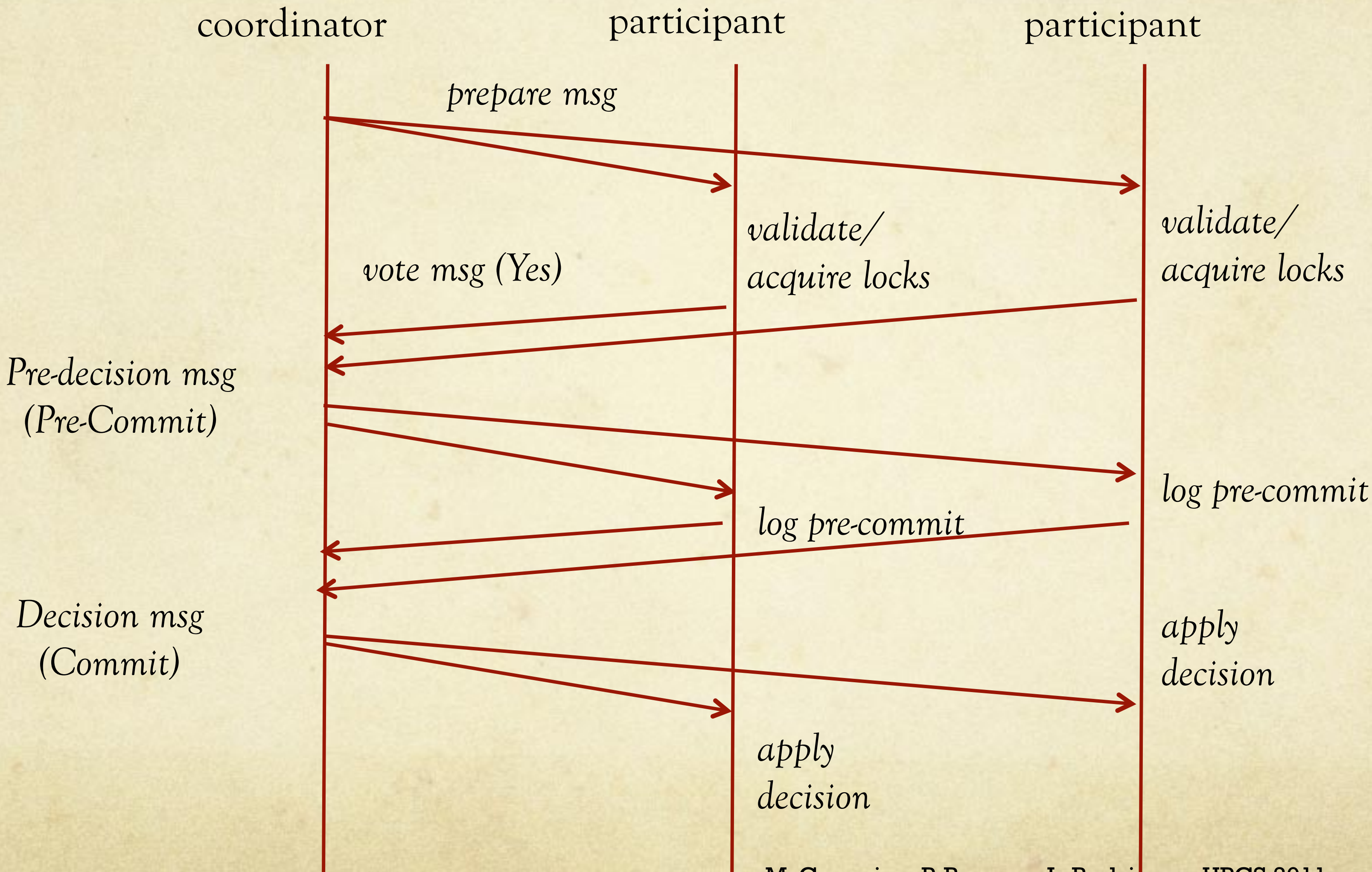
M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

2PC is blocking



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

3PC



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Toolbox

- Atomic Commitment
- Uniform Reliable Broadcast (URB)
- Atomic Broadcast (AB)
- Replication Strategies

Uniform Reliable Broadcast

- Allows to broadcast a message **m** to all replicas
- If a node delivers **m**, every correct node will deliver **m**
- Useful to propagate updates

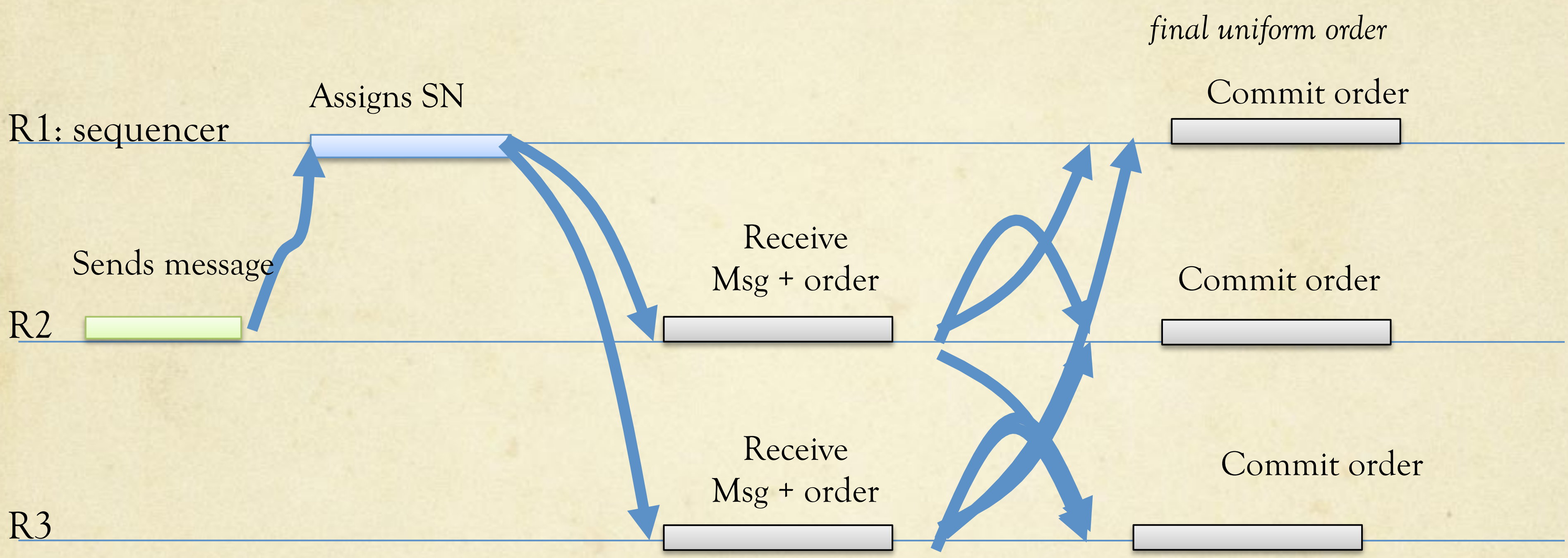
Toolbox

- Atomic Commitment
- Uniform Reliable Broadcast (URB)
- **Atomic Broadcast (AB)**
- Replication Strategies

Atomic Broadcast

- Reliable broadcast with total order
- If replica R1 receives **m1** before **m2**, any other correct replica Ri also receives **m1** before **m2**
- Can be used to allow different nodes to obtain locks in the same order.

Sequencer-based ABcast

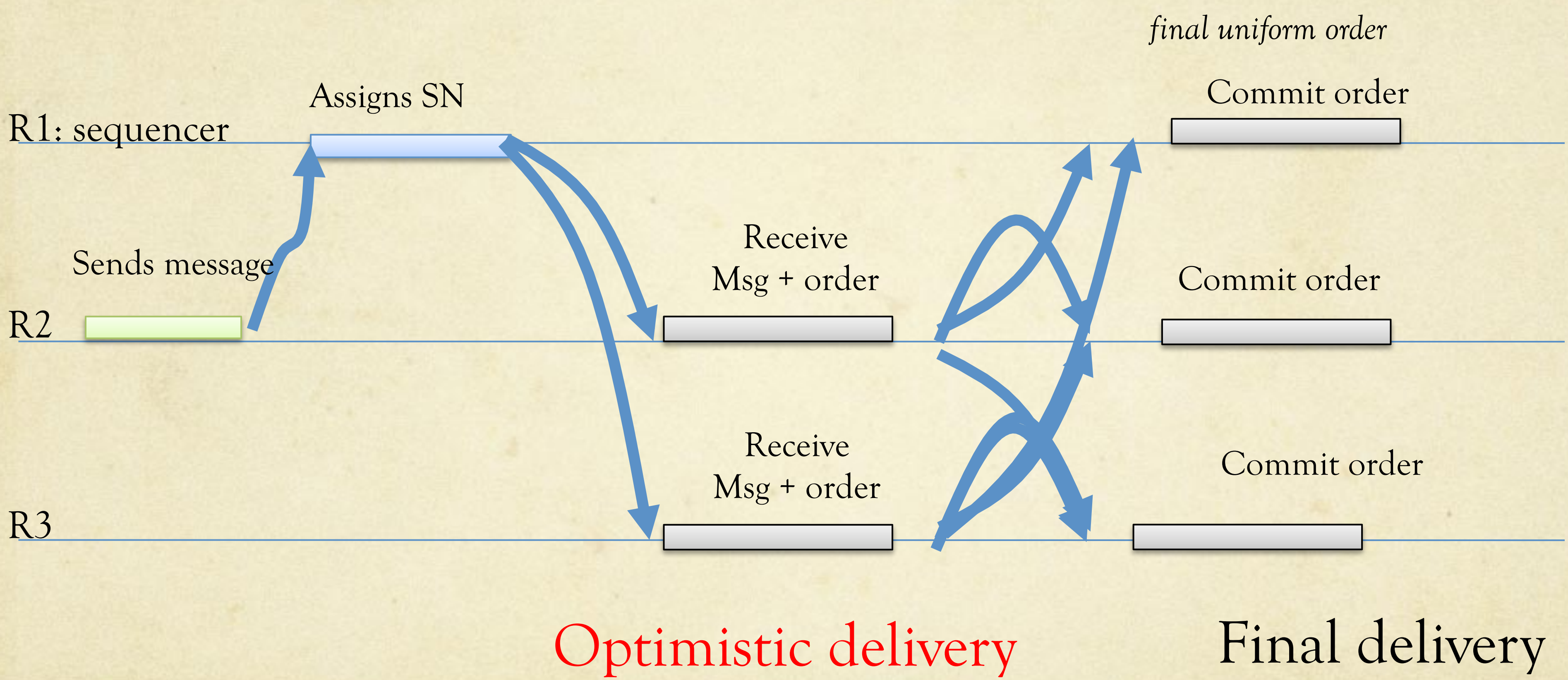


M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Abcast with optimistic delivery

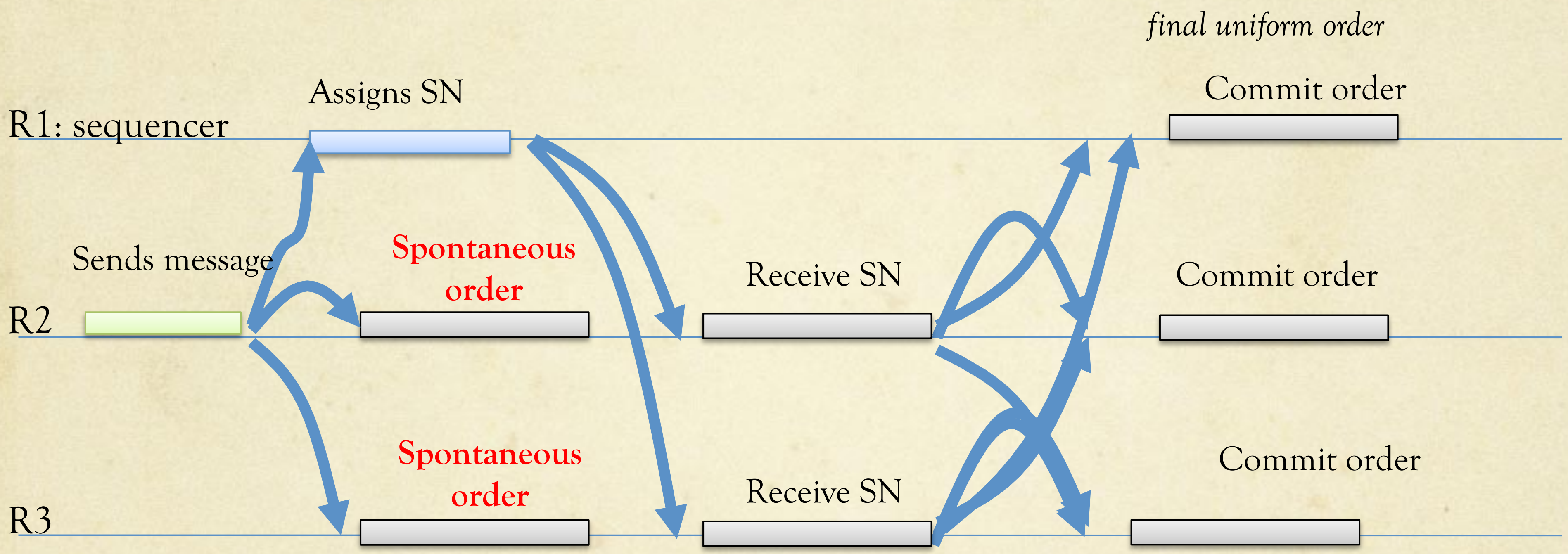
- Total order with optimistic delivery.
- Unless the sequencer node crashes, final uniform total order is the same as regular total order.
- Application may start certificating the transaction locally based on optimistic total order delivery.

ABcast with optimistic delivery



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

ABcast with optimistic delivery



Spontaneous order delivery

Final delivery

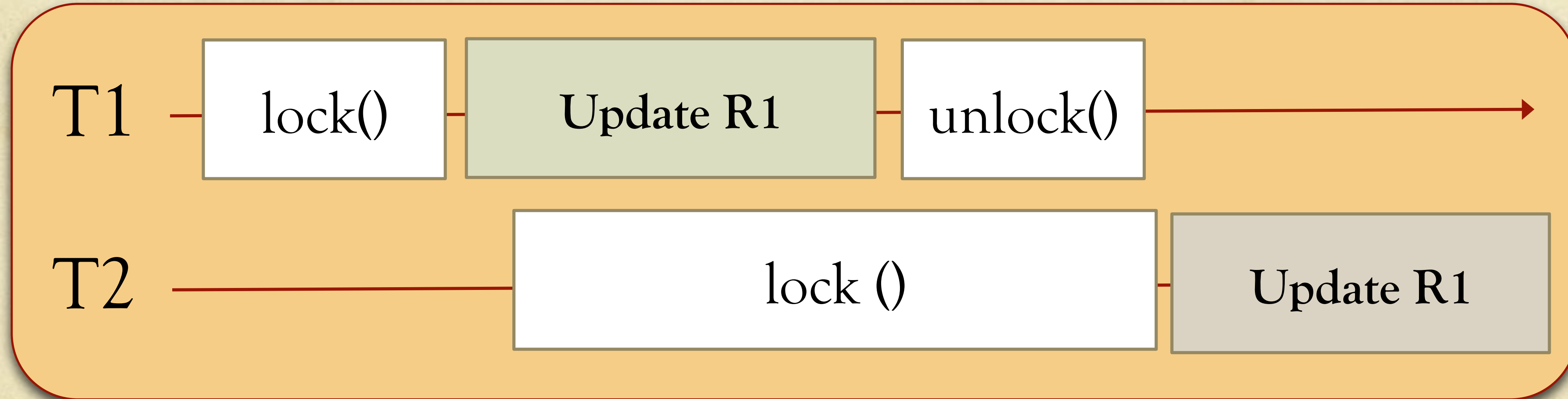
M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Toolbox

- Atomic Commitment
- Uniform Reliable Broadcast (URB)
- Atomic Broadcast (AB)
- **Replication Strategies**

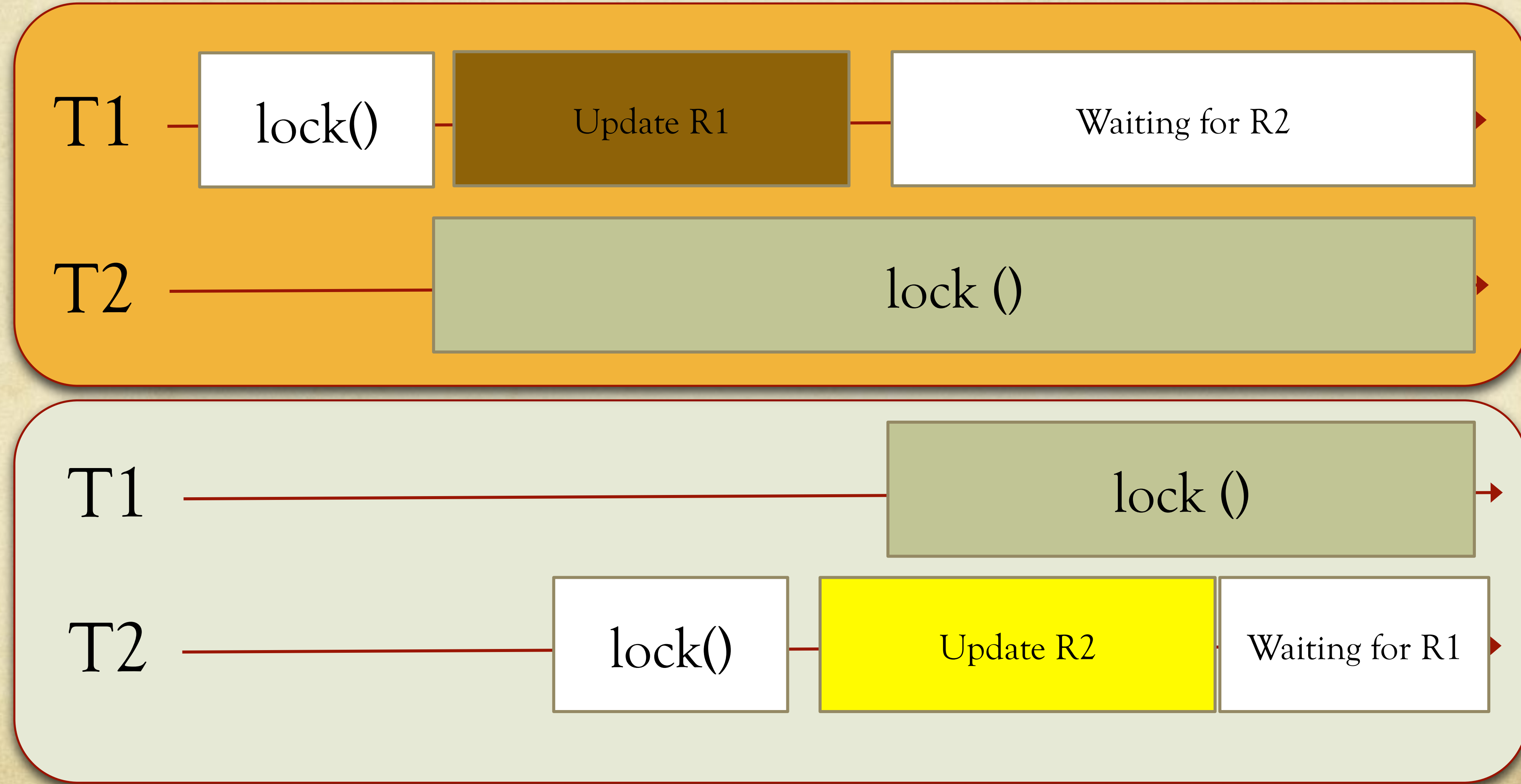
Replicating a single lock

- In absence of replication, there's no chance to fall into deadlocks with a single lock... what if we add replication?



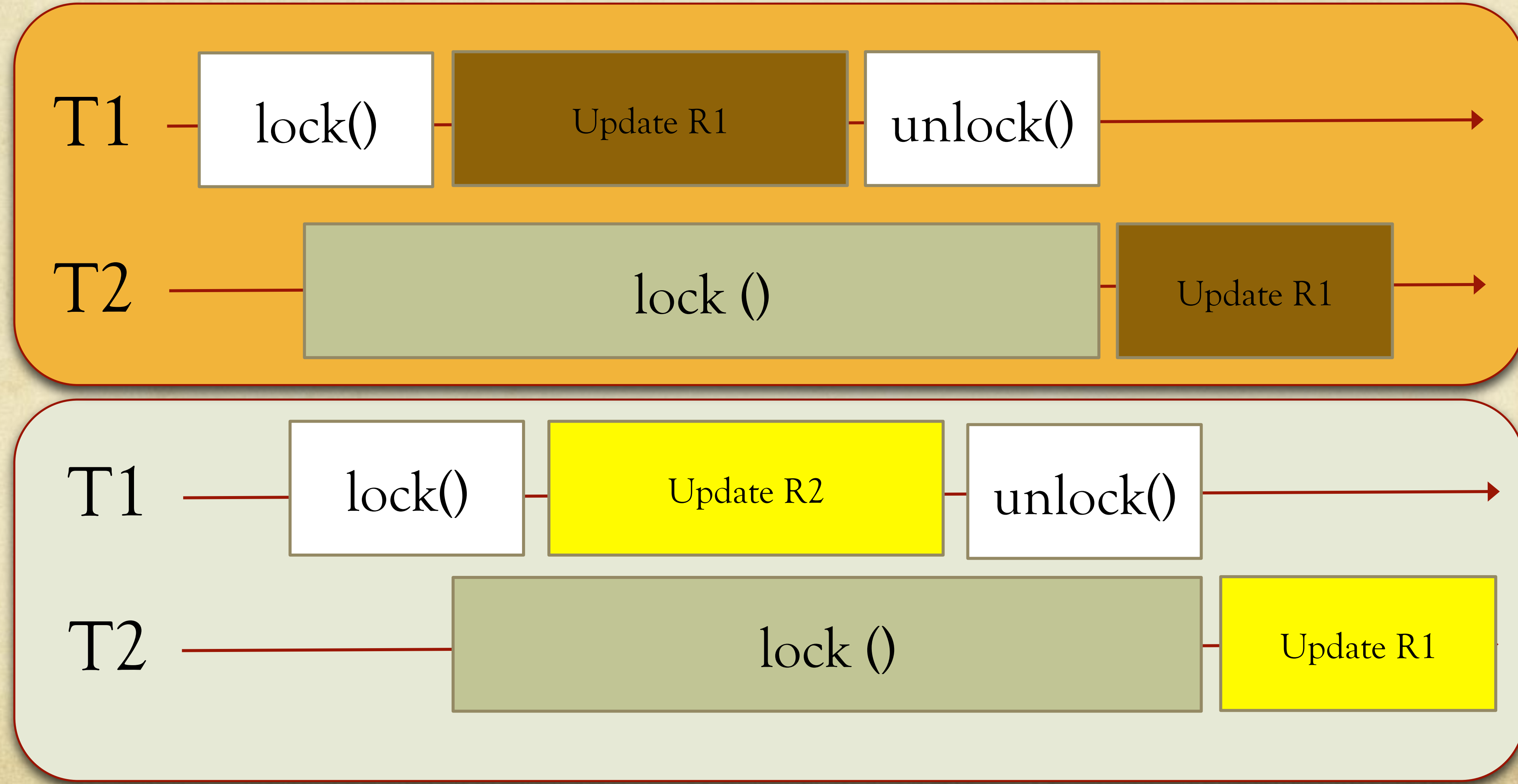
M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Replicating a single lock



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Replicating a single lock



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Coordination is slow

- Drawback of previous approach:
  - Coordination among replicas needs to be executed at every lock operation.
  - Atomic broadcast is an expensive primitive.
  - The system becomes too slow.
- Solution:
  - Limit the coordination among replicas **to a single phase**, at the beginning of the transaction or commit time.

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

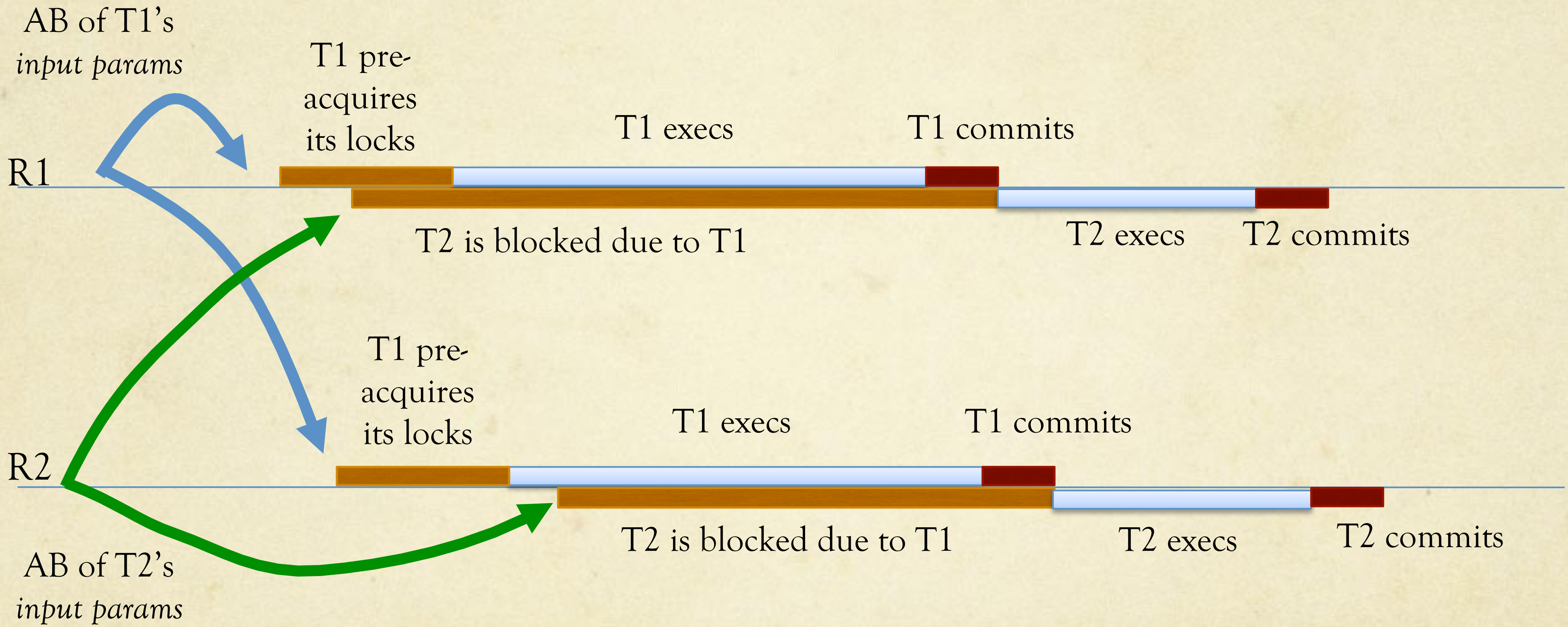
Single-phase schemes

- State machine replication
- Single master (primary-backup)
- Multiple master (certification)
  - Non-voting
  - Voting

State-machine replication

- All replicas execute the same set of transactions, in the same order.
- Transactions are shipped to all replicas using atomic broadcast.
- Replicas receive transactions in the same order.
- Replicas execute transaction by that order.
  - Transactions need to be deterministic!

State-machine replication



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Single-phase schemes

- State machine replication
- **Single master (primary-backup)**
- Multiple master (certification)
  - **Non-voting**
  - Voting

Primary-backup

- Write transactions are executed entirely in a single replica (the primary)
- If the transaction aborts, no coordination is required.
- If the transaction is ready to commit, coordination is required to update all the other replicas (backups).
  - Reliable broadcast primitive.
- Read transactions may be executed on backup replicas.
  - Works fine for workloads with very few update transactions.
  - Otherwise the primary becomes a bottleneck.

Primary-backup

- Synchronous updates:
  - Updates are propagated during the commit phase:
    - Data is replicated immediately
    - Read transactions observe up to date data in backup replicas
    - Commit must wait for reliable broadcast to finish
- Asynchronous updates:
  - The propagation of updates happens in the background:
    - Multiple updates may be batched
    - Commit is faster
    - There is a window where a single failure may cause data to be lost
    - Read transactions may read stale data

Single-phase schemes

- State machine replication
- Single master (primary-backup)
- **Multiple master (certification)**
  - Non-voting
  - Voting

Multi-master

- A transaction is executed entirely in a single replica.
- Different transactions may be executed on different replicas.
- If the transaction aborts, no coordination is required.
- If the transaction is ready to commit, coordination is required:
  - To ensure serializability
  - To propagate the updates

Multi-master

- Two transactions may update concurrently the same data in different replicas.
- Coordination must detect this situation and abort at least one of the transactions.
- Two main alternatives:
  - Non-voting algorithm
  - Voting algorithm

Single-phase schemes

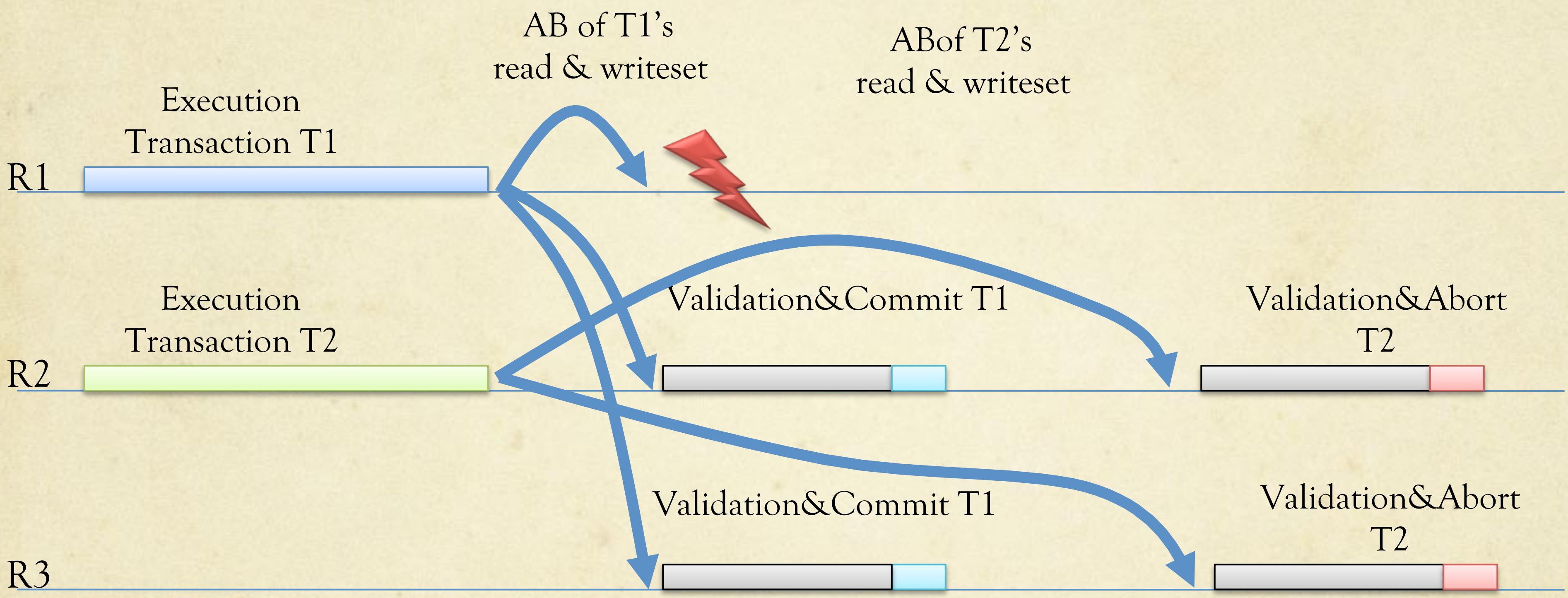
- State machine replication
- Single master (primary-backup)
- Multiple master (certification)
  - Non-voting
  - Voting

Non-voting

- The transaction executes locally.
- When the transaction is ready to commit, the **read and write set** are sent to all replicas using atomic broadcast.
- Transactions are certified in total order.
- A transaction may commit if its read set is still valid (i.e., no other transaction has updated the read set).

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Non-voting



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

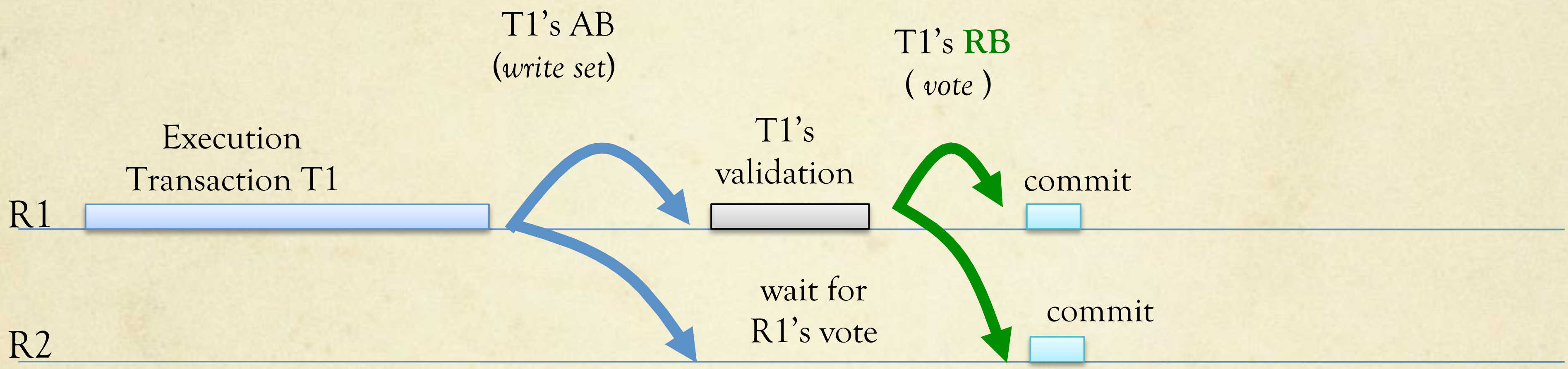
Single-phase schemes

- State machine replication
- Single master (primary-backup)
- Multiple master (certification)
  - Non-voting
  - Voting

Voting

- The transaction executes locally at replica R
- When the transaction is ready to commit, **only the write set** is sent to all replicas using atomic broadcast
- Transactions' commit requests are processed in total order
- A transaction may commit if its read set is still valid (i.e., no other transaction has updated the read set):
  - **Only R can certify the transaction!**
- R send the outcome of the transaction to all replicas:
  - Reliable broadcast

Voting



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Contents

Part III: Case-Studies

- Partitioned Non-Replicated
  - STM for clusters (Cluster-STM)
- Partitioned (Replicated)
  - Static Transactions (Sinfonia)
- Replicated Non-Partitioned
  - Certification-based with Bloom Filters (D^2STM)
  - Certification with Leases (ALC)
  - Active Replication with Speculation (AGGRO)

Part III: Case-Studies

- Partitioned Non-Replicated
  - **STM for clusters (Cluster-STM)**
- Partitioned (Replicated)
  - Static Transactions (Sinfonia)
- Replicated Non-Partitioned
  - Certification-based with Bloom Filters (D^2STM)
  - Certification with Leases (ALC)
  - Active Replication with Speculation (AGGRO)

Cluster-STM

- Software Transactional Memory for Large Scale Clusters
- Bocchino, Adve, and Chamberlain. 2008
- Partitioned (word-based) address space
- No persistency, no replication, no caching
- Supports only single thread per node
- Various lock acquisition schemes + 2PC

Cluster-STM

- Various methods for dealing with partitioned space
- Data movement (Dataflow model):
  - **stm get**(src proc, dest, work proc, src, size, open)
  - **stm put**(src proc, work proc, dest, src, size, open)
- Remote execution (Control flow model):
  - **stm on**(src proc, work proc, function, arg buf, arg buf size, result buf, result buf size)

Cluster-STM

```
increment(proc_t proc, int *addr) {
  atomic {
    on(proc) {
      ++*addr
    }
  }
}
```

Cluster-STM

```
increment(proc_t proc, int* addr) {
  stm_start(MY_ID)
  stm_on(MY_ID, proc, increment_local,
         addr, sizeof(int*), 0, 0)
  stm_commit(MY_ID)
}
```

Cluster-STM

```
increment_local(proc_t src_proc,
                void* arg,
                size_t arg_size,
                void *result,
                size_t result_size) {
  int *addr = *((int*) arg);
  int tmp;
  stm_open_read(src_proc, addr, sizeof(int))
  stm_read(src_proc, &tmp, addr, sizeof(int))
  ++tmp;
  stm_open_write(src_proc, addr, sizeof(int))
  stm_write(src_proc, addr, &tmp, sizeof(int))
}
```

Cluster-STM

- Read locks (RL) vs. read validation (RV)
- RL:
  - immediately acquire a lock as a read (local or remote) is issued
  - abort upon contention (avoid deadlock)
  - as coordinator ends transaction, it can be committed w/o 2PC
- Note: distributed model w/o caching:
  - each access to non local data implies remote access:
  - eager locking is for free
  - with caching only RV could be employable

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Cluster-STM

- Read locks (RL) vs. read validation (RV)
- RV:
  - commit time validation (not opaque)
  - validity check requires 2PC

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Cluster-STM

- Write buffering schemes
- UL undo log:
  - write is applied and an undo log is maintained
  - forced sync upon each write
- WB write buffering:
  - writes applied in local buffer
  - avoid communications for writes during exec phase
  - requires additional communication at commit time

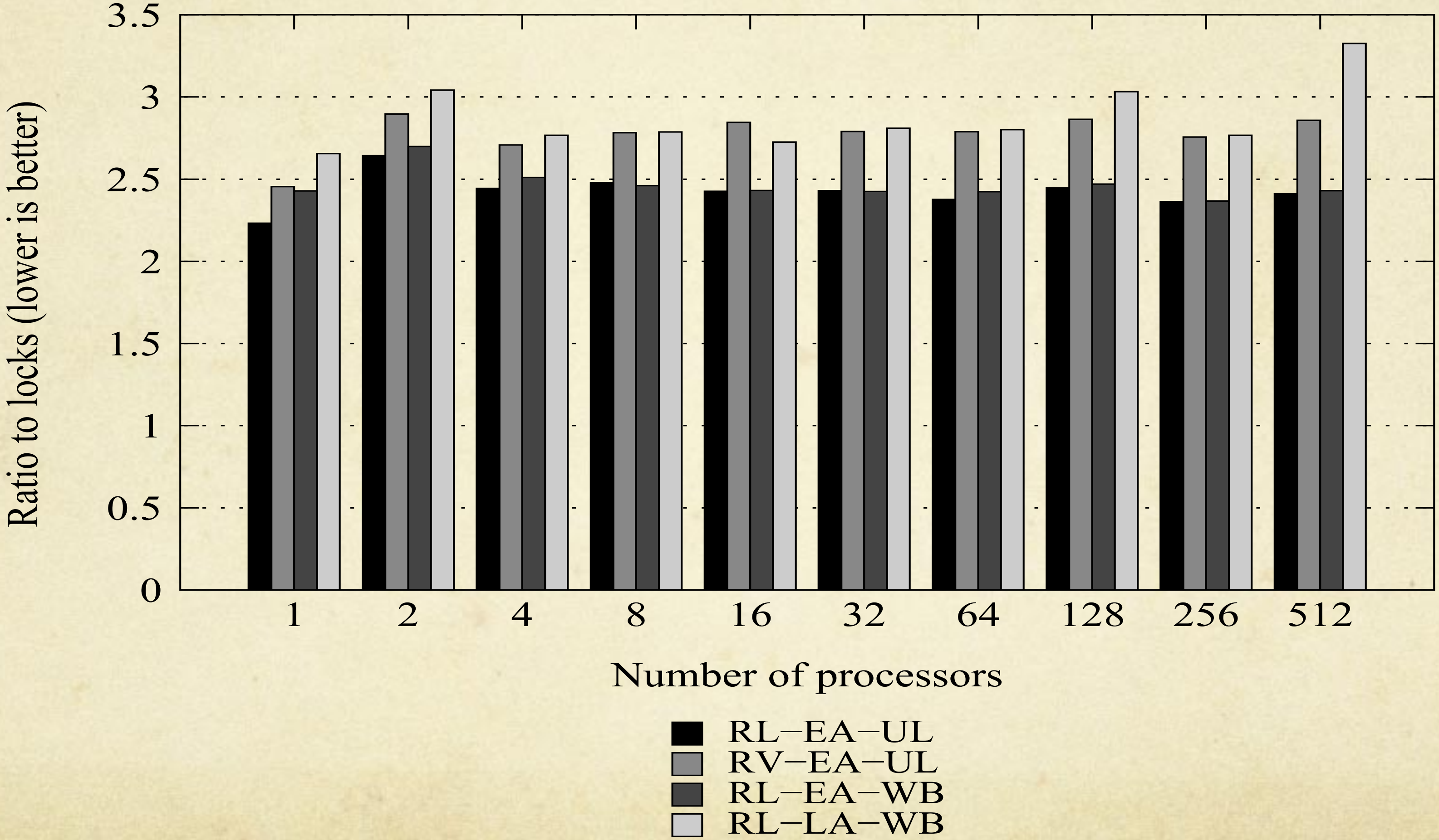
M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Cluster-STM

- Write buffering: two lock acquisition schemes
- LA: Late acquire
  - at commit time.
  - may allow for more concurrency
- EA: Early acquire
  - as the write is issued
  - may avoid wasted work by doomed transactions

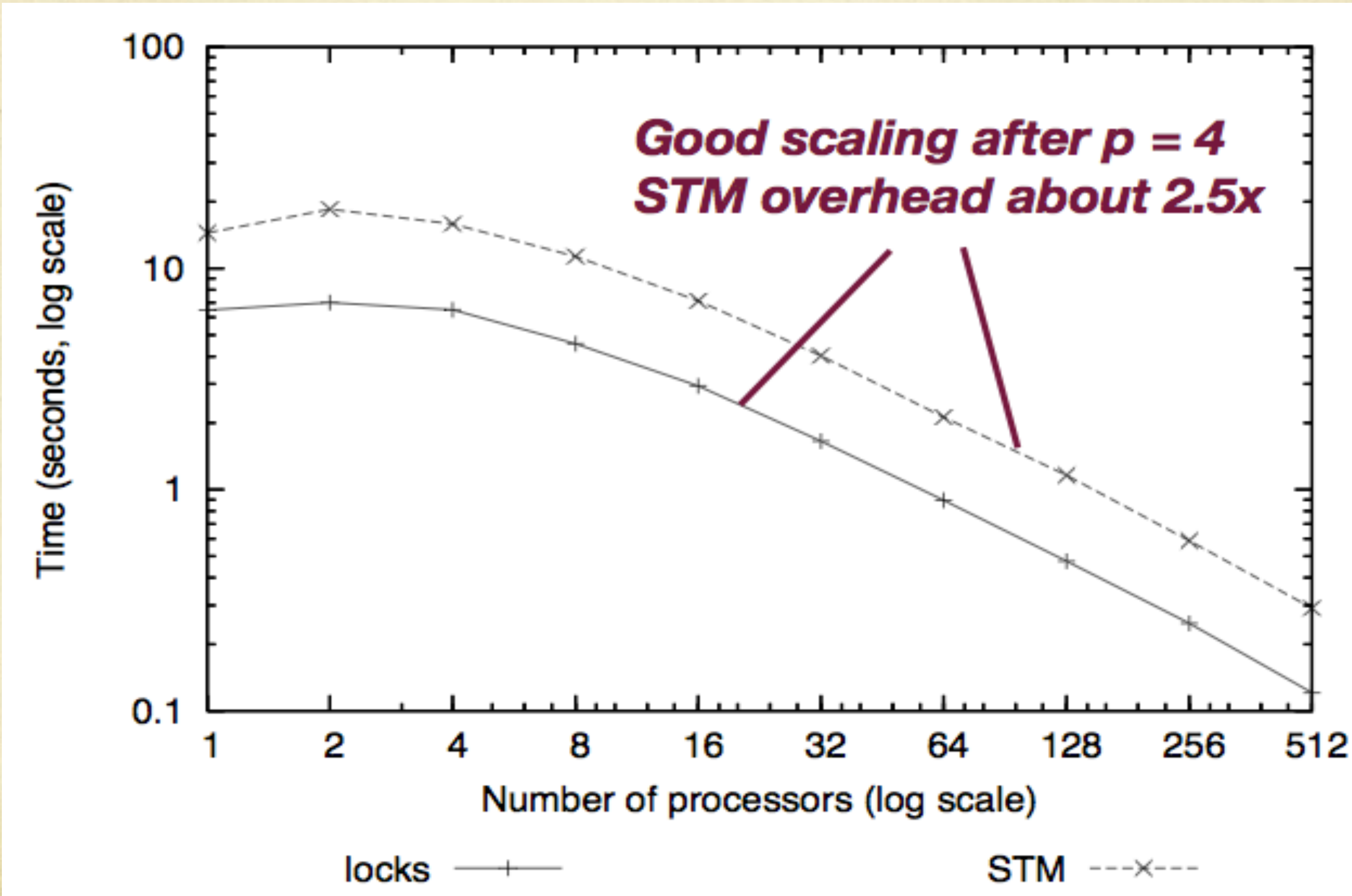
M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Cluster-STM: Graph Analysis (SSCA2)



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Cluster-STM: Graph Analysis (SSCA2)



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Part III: Case-Studies

- Partitioned Non-Replicated
  - STM for clusters (Cluster-STM)
- Partitioned (Replicated)
  - **Static Transactions (Sinfonia)**
- Replicated Non-Partitioned
  - Certification-based with Bloom Filters (D^2STM)
  - Certification with Leases (ALC)
  - Active Replication with Speculation (AGGRO)

Sinfonia

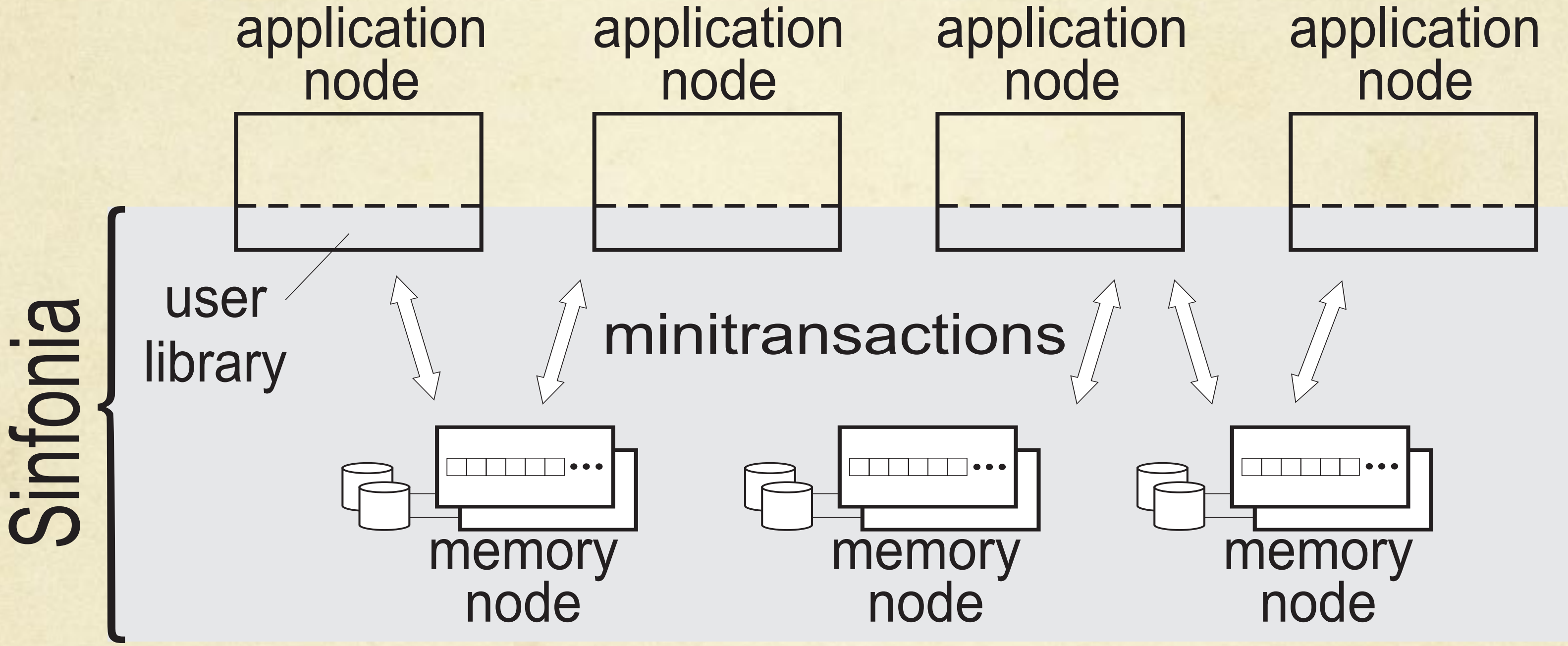
- Sinfonia: A new paradigm for building scalable distributed systems.
- Aguilera, Merchant, Shah, Veitch, and Karamanolis, 2009.
- Partitioned global (linear) address space
- Optimized for **static** transactions

Sinfonia

- Mini-transactions:
  - A-priori knowledge on the data to be accessed
- Two types of nodes:
  - Application nodes
  - Memory nodes
- Fault-tolerance via:
  - In-memory replication
  - Sync (log) + async checkpoint for persistency on memory nodes

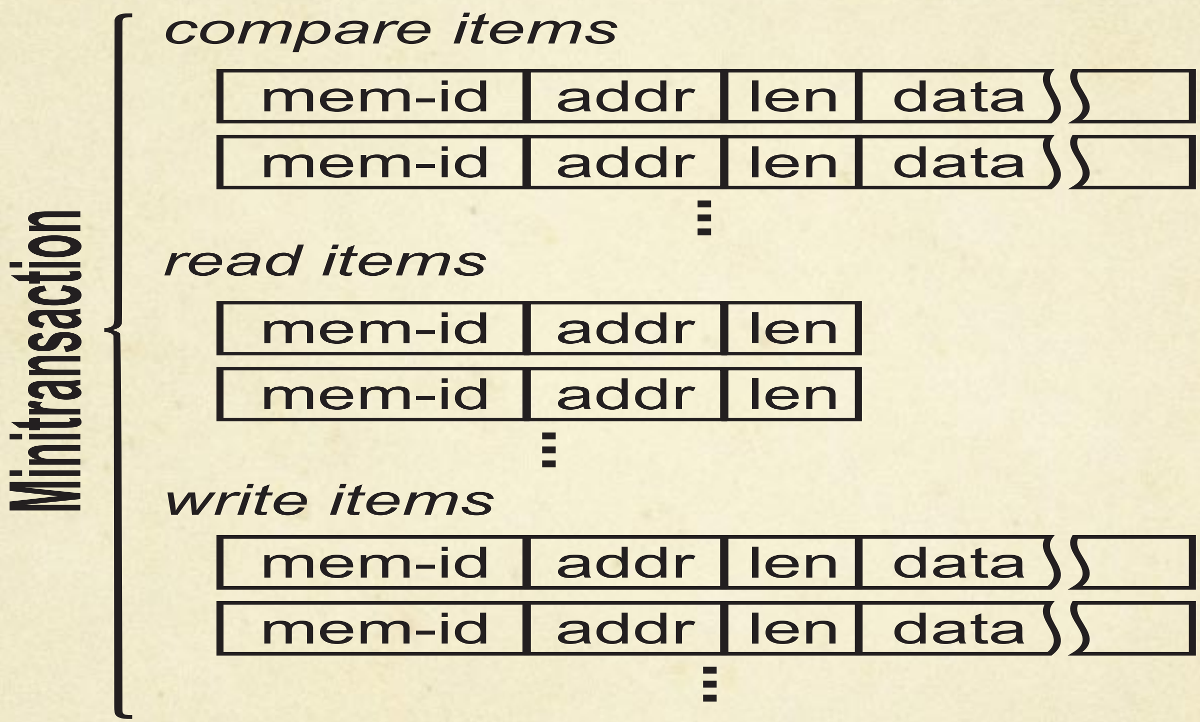
M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Sinfonia



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Sinfonia



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Sinfonia

API

```
class Minitransaction {
 public:
  void cmp(memid,addr,len,data); // add cmp item
  void read(memid,addr,len,buf); // add read item
  void write(memid,addr,len,data); // add write item
  int exec_and_commit();     // execute and commit
};
```

Example

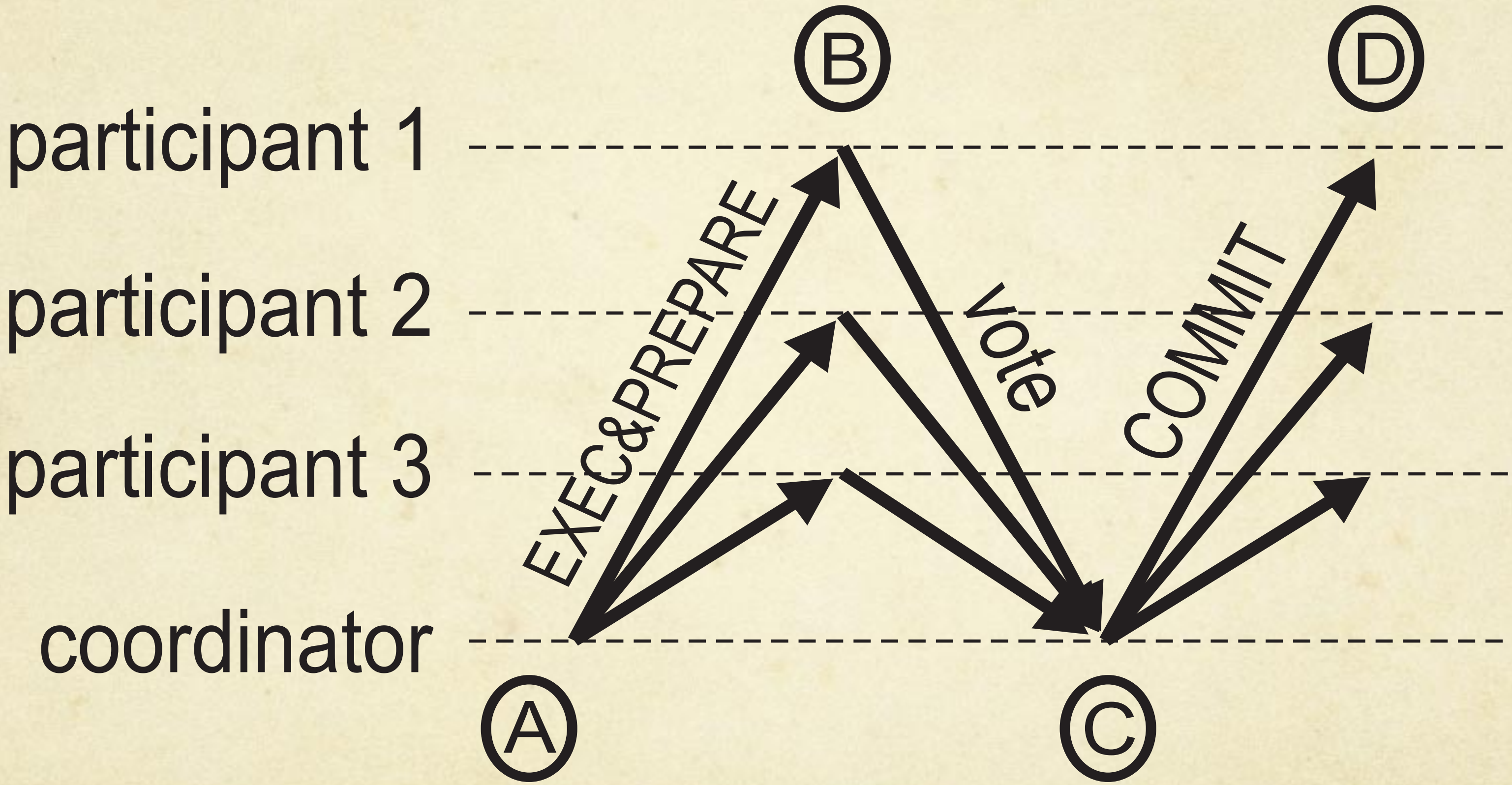
```
...
t = new Minitransaction;
t->cmp(memid, addr, len, data);
t->write(memid, addr, len, newdata);
status = t->exec_and_commit();
...
```

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Sinfonia

- Global space is partitioned
  - Transaction may need to access different memory nodes
  - It can only commit if it can commit at all memory nodes
  - 2-phase commit

Sinfonia



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Sinfonia

- No support for caching:
  - delegated to application level
  - same applies for load balancing
- Replication:
  - aimed at fault-tolerance, not enhancing performance
  - fixed number of replicas per memory node
  - primary-backup scheme ran within first phase of 2PC

Part III: Case-Studies

- Partitioned Non-Replicated
  - STM for clusters (Cluster-STM)
- Partitioned (Replicated)
  - Static Transactions (Sinfonia)
- Replicated Non-Partitioned
  - **Certification-based with Bloom Filters (D^2STM)**
  - Certification with Leases (ALC)
  - Active Replication with Speculation (AGGRO)

D^{2}STM

- D^{2}STM: Dependable Distributed STM
- Couceiro, Romano, Rodrigues, Carvalho, 2009
- Single-image system
  - Full replication
  - Strong consistency
- Certification-based replication scheme
  - Based on Atomic Broadcast
  - Built on top of JVSTM

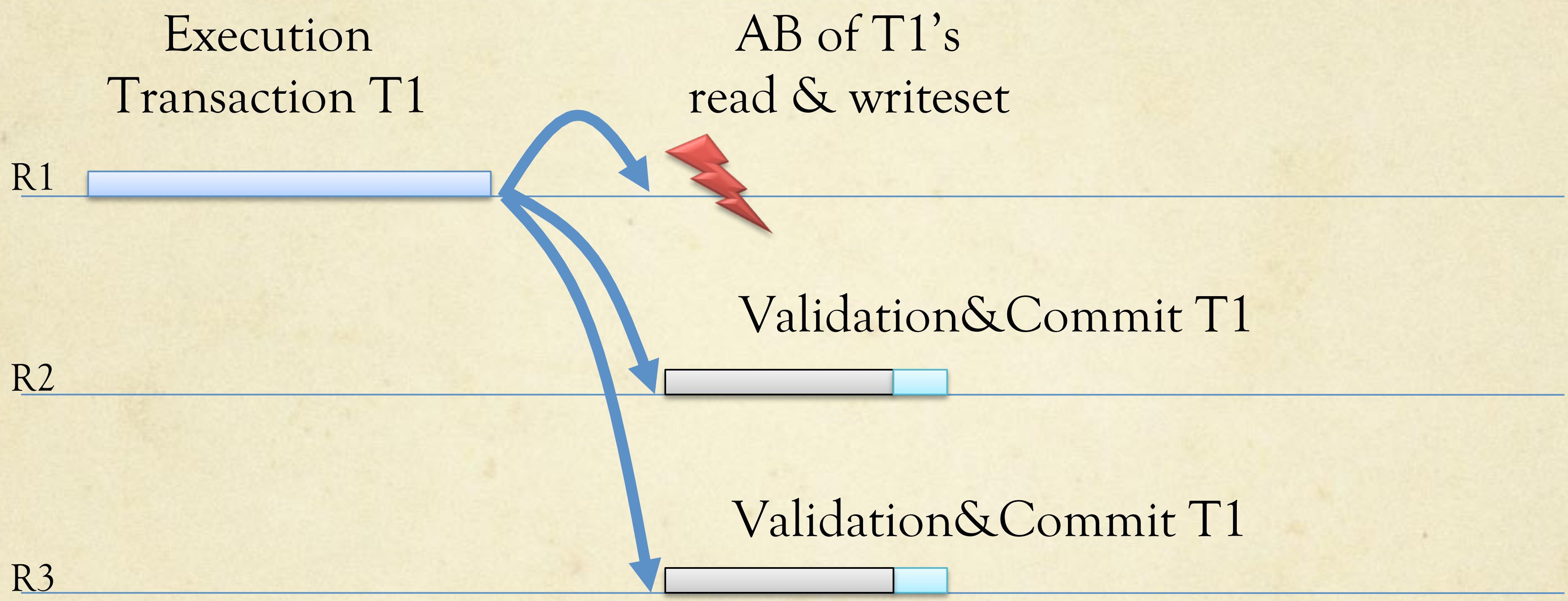
M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

D^{2}STM

- Non-voting replication scheme
- Transactions execute in a single replica
- No communication during the execution
- Writeset **and** readset AB at commit time
- Deterministic certification executed in total order by all replicas
- No distributed deadlocks

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

D²STM



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

D^2STM

AB of both T1's **readset** & writeset

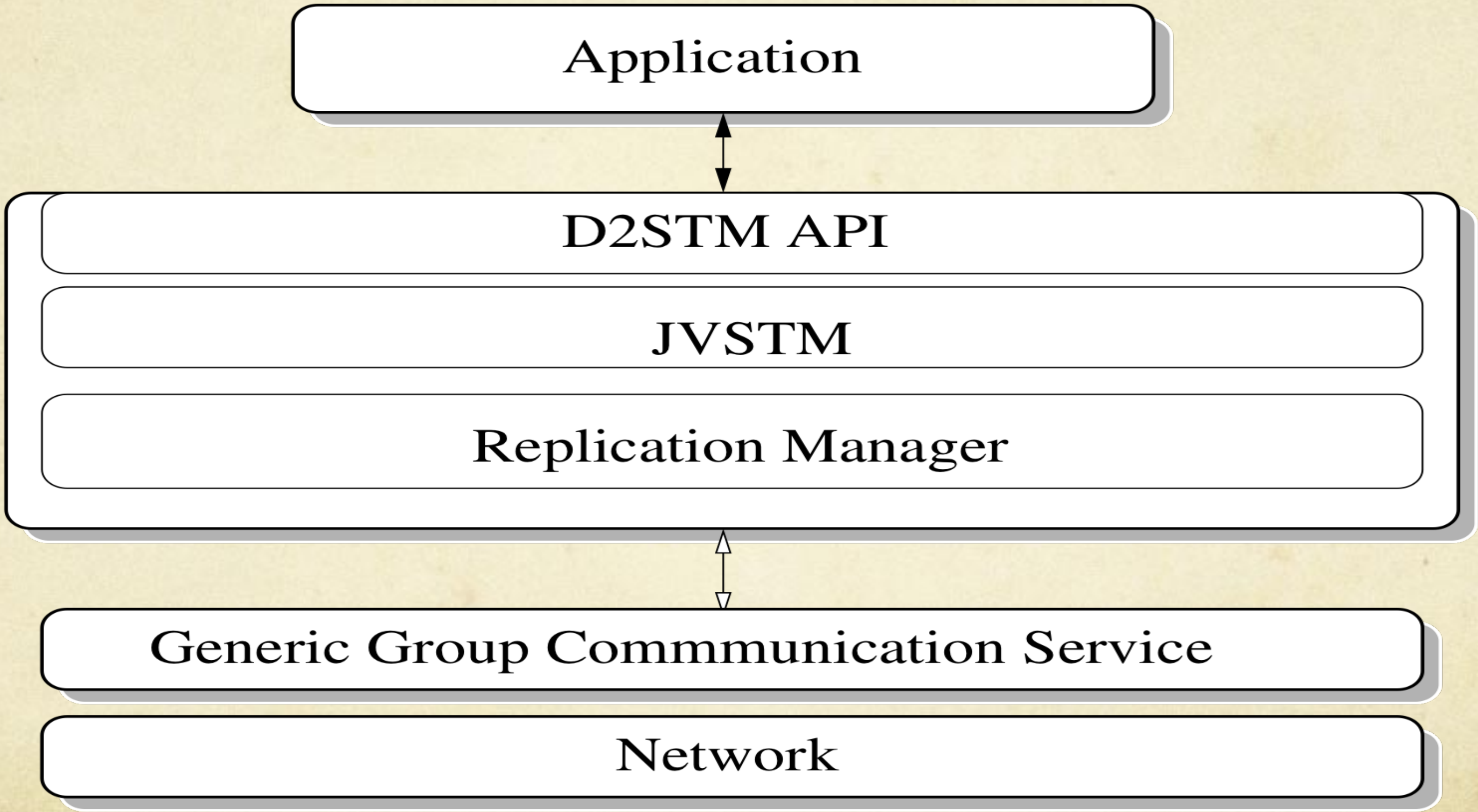
Problem:
(very) big message size

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

D^{2}STM

- In STMs, transaction's execution time is often 10-100 times short than in DBs:
  - the cost of AB is correspondingly amplified
- Bloom Filter Certification:
  - space-efficient encoding (via Bloom Filter) to reduce message size

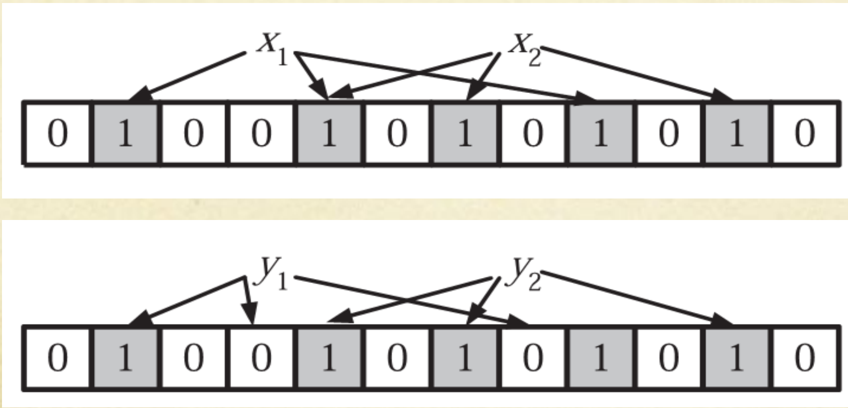
D²STM



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

D^2STM

- Bloom filters
  - A set of n items is encoded through a vector of m bits
  - Each item is associated with k bits through k hash functions having as image {1..m}:
    - insert: set k bits to 1
    - query: check if all k bits set to 1



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

D^{2}STM

- False Positives:
  - An item is wrongly identified as belonging to a given set
  - Depend on the number of bits used per item (m/n) and the number of hash functions (k)
- D2STM computes the size of the Bloom filter based on:
  - User-defined false positive rate
  - Number of items in the read set (known)
  - Number of BF queries, estimated via moving average over recently committed transactions

D^{2}STM

- Read-only transactions:
  - local execution and commit

D^{2}STM

- Write transaction T:
  - Local validation (read set)
  - If the transaction is not locally aborted, the read set is encoded in a Bloom filter
  - Atomic broadcast of a message containing:
    - the Bloom filter enconding of tx readset
    - the tx write set
    - the snapshotID of the tx
  - Upon message delivery: validate tx using Bloom filter's information

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

D^{2}STM

for each committed T' s.t. T'.snapshotID > T.snapshotID
for each data item *d* in the writeset of T'
if *d* is in Bloom filter associated with T's readset
abort T

// otherwise...

commit T

D²STM

- STMBench7: Results

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Part III: Case-Studies

- Partitioned Non-Replicated
  - STM for clusters (Cluster-STM)
- Partitioned (Replicated)
  - Static Transactions (Sinfonia)
- Replicated Non-Partitioned
  - Certification-based with Bloom Filters (D^2STM)
  - **Certification with Leases (ALC)**
  - Active Replication with Speculation (AGGRO)

ALC

- Asynchronous Lease Certification Replication of Software Transactional Memory
- Carvalho, Romano, Rodrigues, 2010
- Exploit data access locality by letting replicas dynamically establish ownership of memory regions:
  - replace AB with faster coordination primitives:
  - no need to establish serialization order among non-conflicting transactions
  - shelter transactions from remote conflicts

ALC

- Data ownership established by acquiring an Asynchronous Lease
  - mutual exclusion abstraction, as in classic leases…
  - …but detached from the notion of time:
  - implementable in a partially synchronous system
  - Lease requests disseminated via AB to avoid distributed deadlocks.

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

ALC

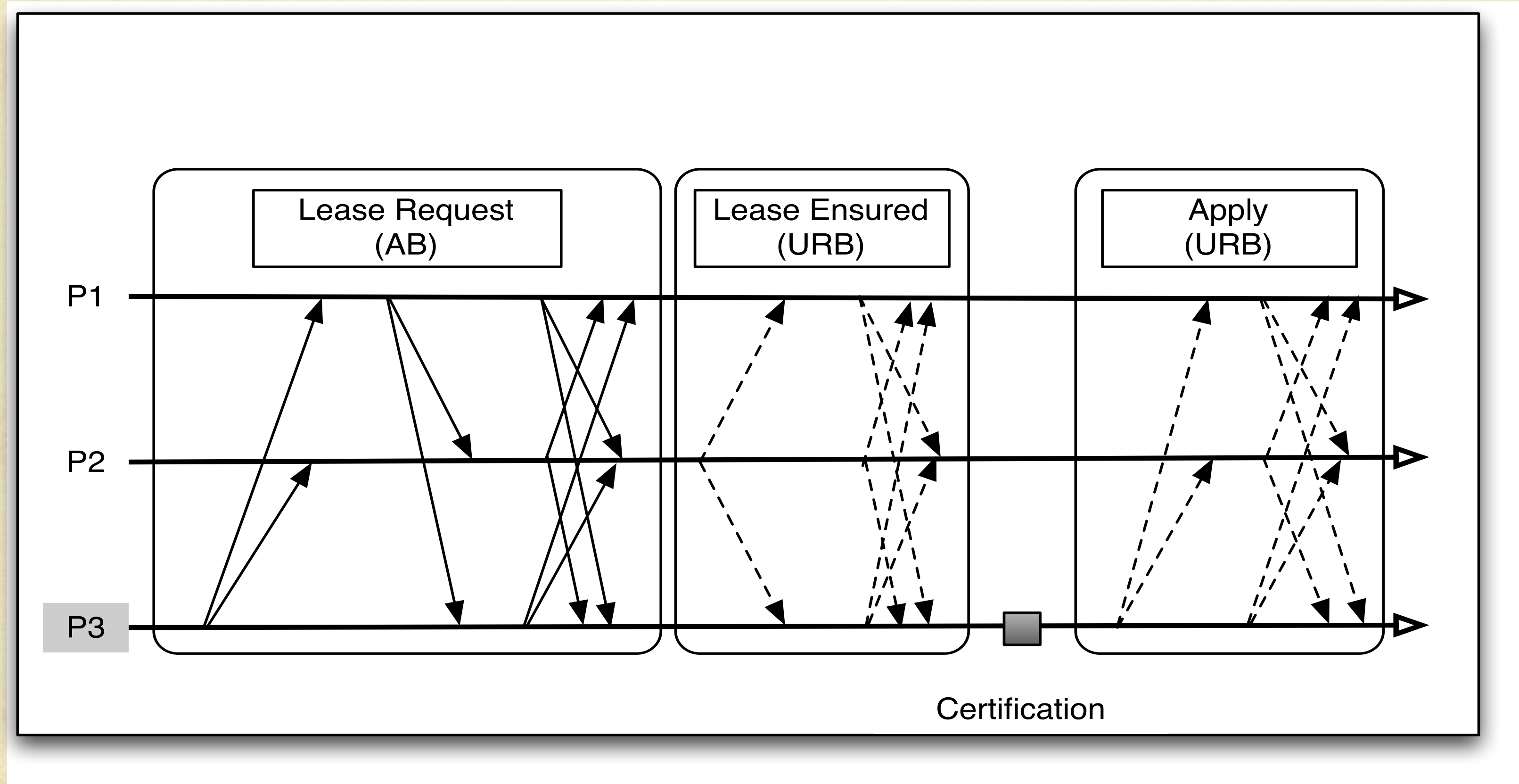
- Transactions are locally processed
- At commit time check for leases:
  - An Asynchronous Lease may need to be established
- Proceed with local validation

ALC

- If local validation succeeds, its writeset is propagated using Uniform Reliable Broadcast (URB):
  - No ordering guarantee, 30-60% faster than AB
- If validation fails, upon re-execution the node holds the lease:
  - Transaction cannot be aborted due to a remote conflict!

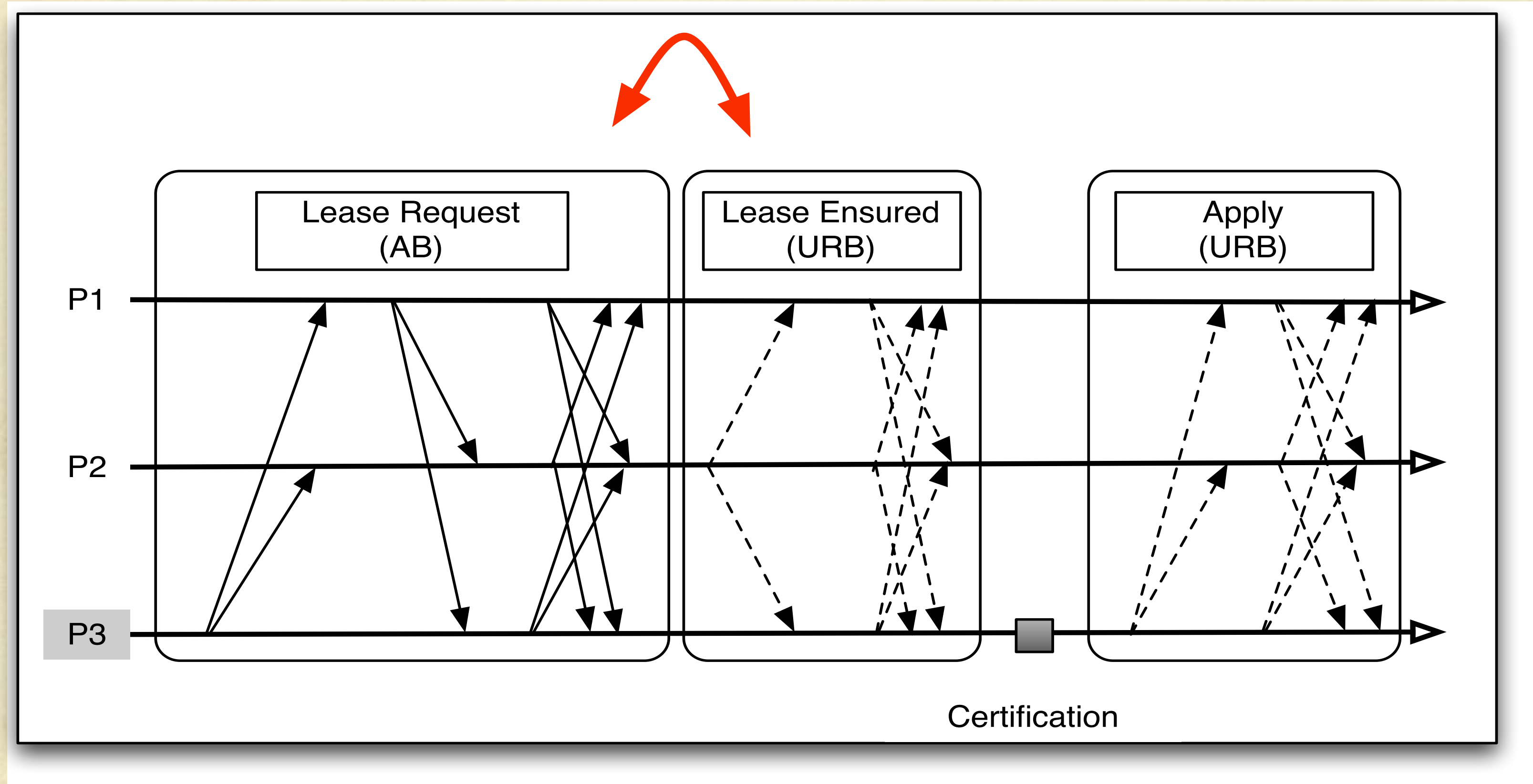
M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

ALC



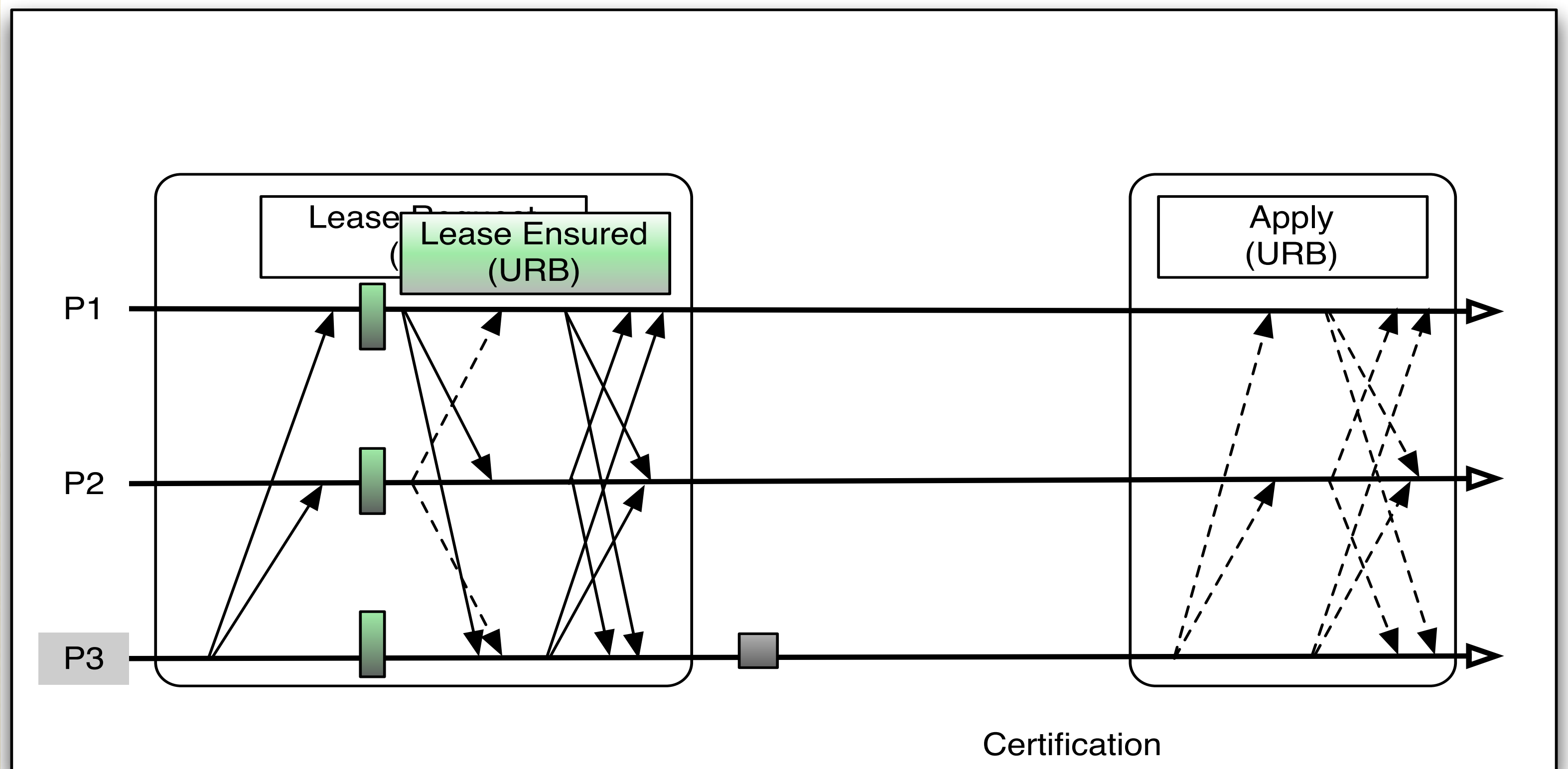
M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

ALC



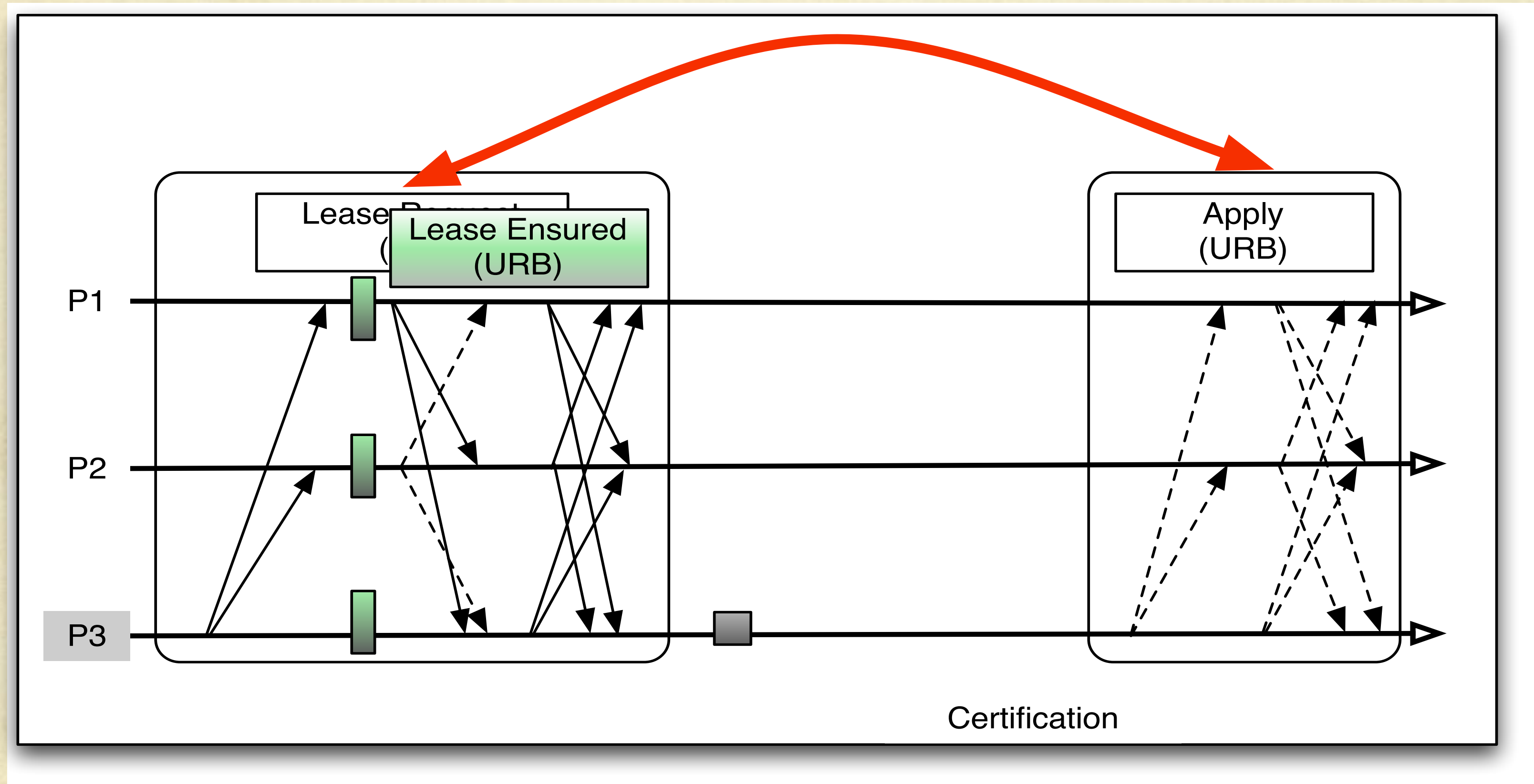
M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

ALC



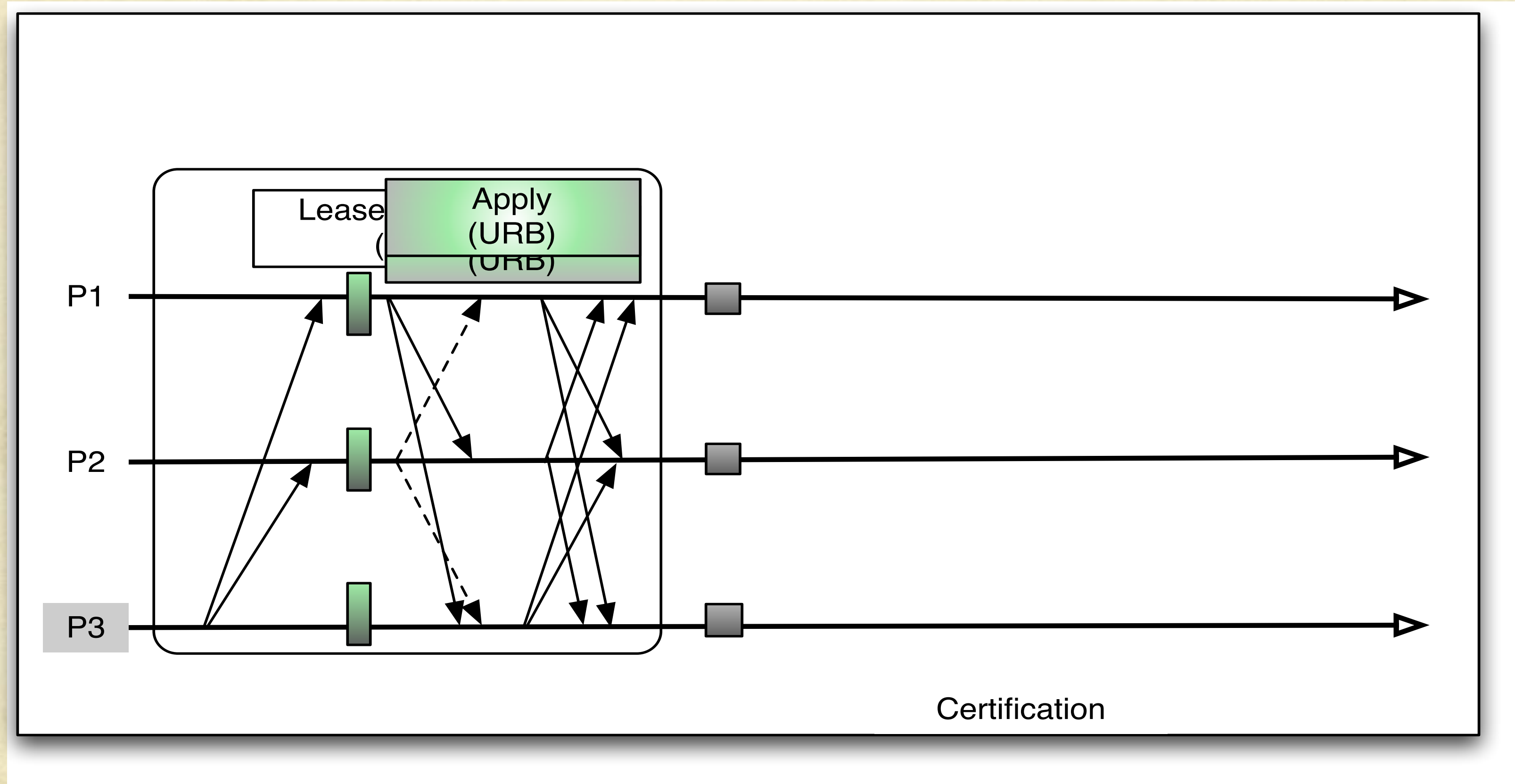
M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

ALC



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

ALC



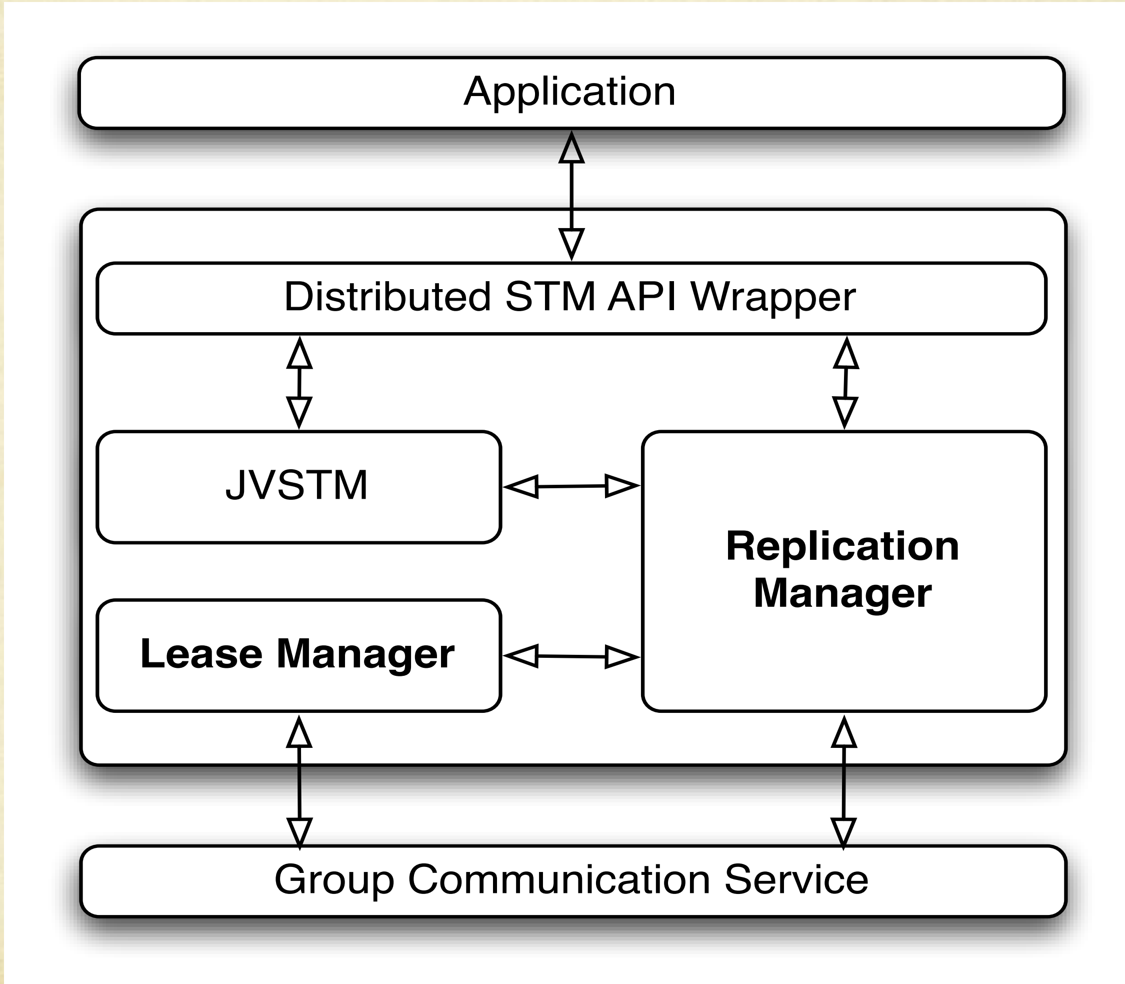
M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

ALC

- If applications exhibit some access locality:
  - avoid, or reduce frequency of AB
  - locality improved via conflict-aware load balancing
- Ensure transactions are aborted at most once due to remote conflicts:
  - essential to ensure liveness of long running transactions
  - benefic at high contention rate even with small running transactions

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

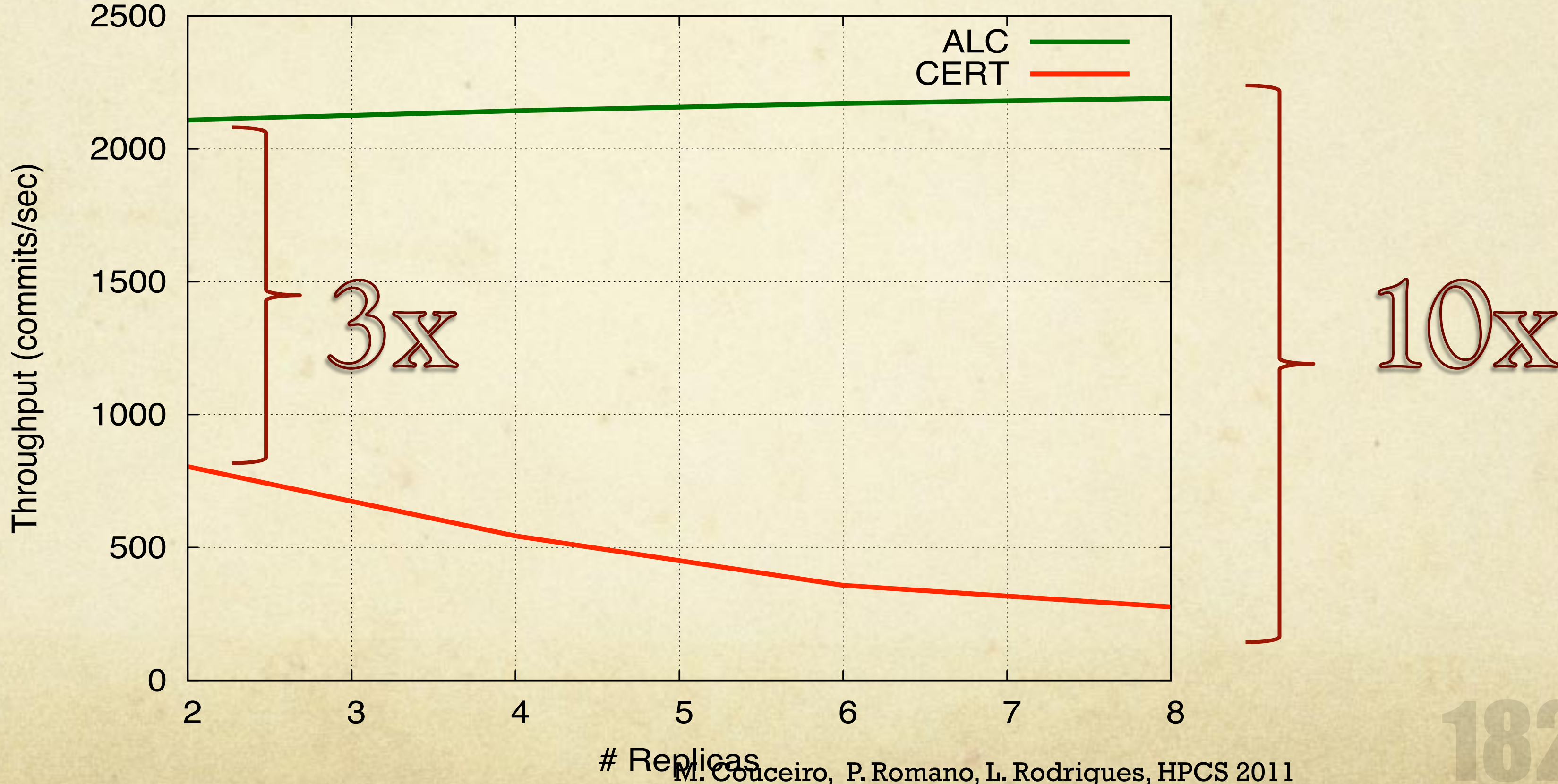
ALC



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

ALC

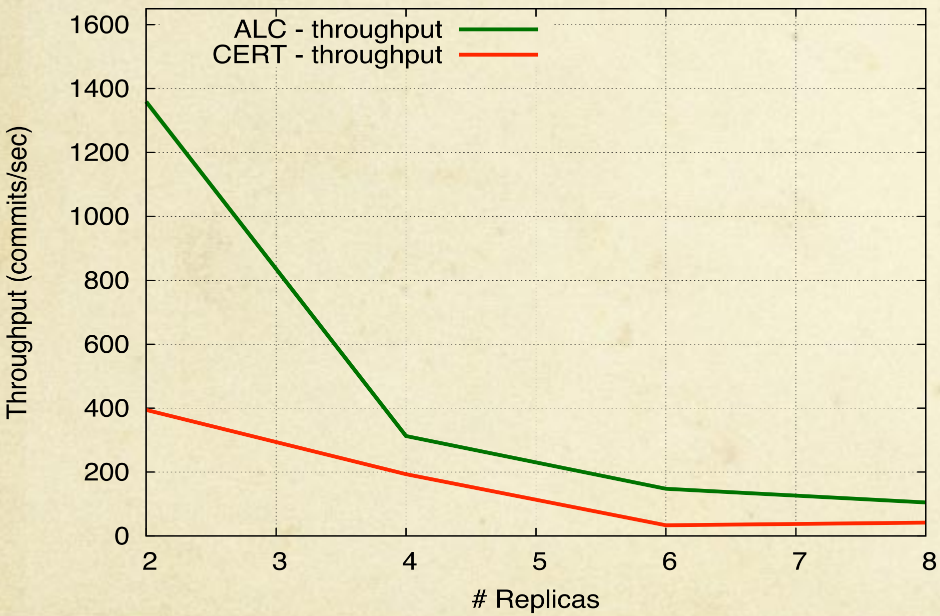
- Synthetic “Best case” scenario
  - Replicas accessing distinct memory regions

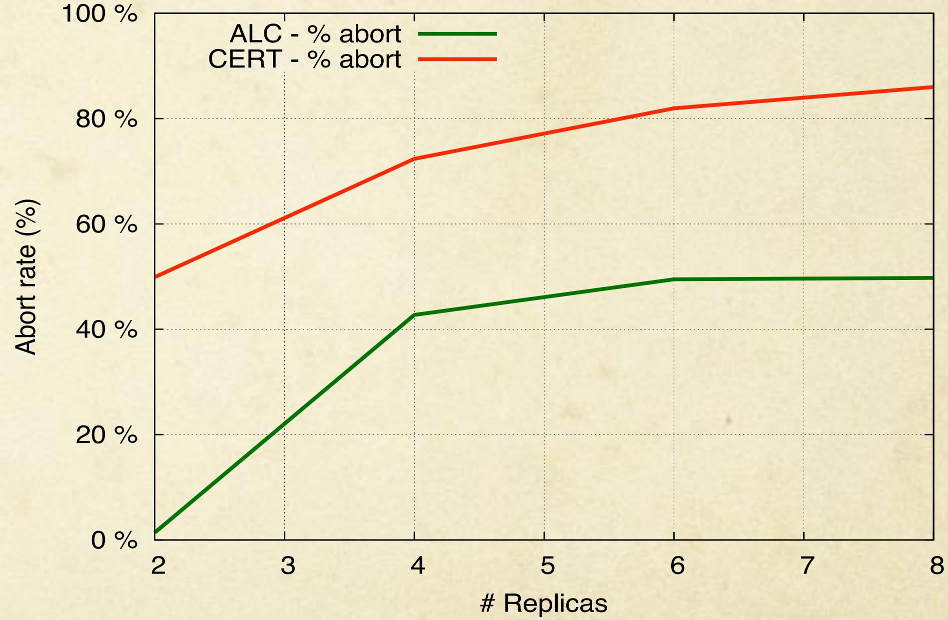


M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

ALC

- Synthetic "Worst case" scenario
  - All replicas accessing the same memory region

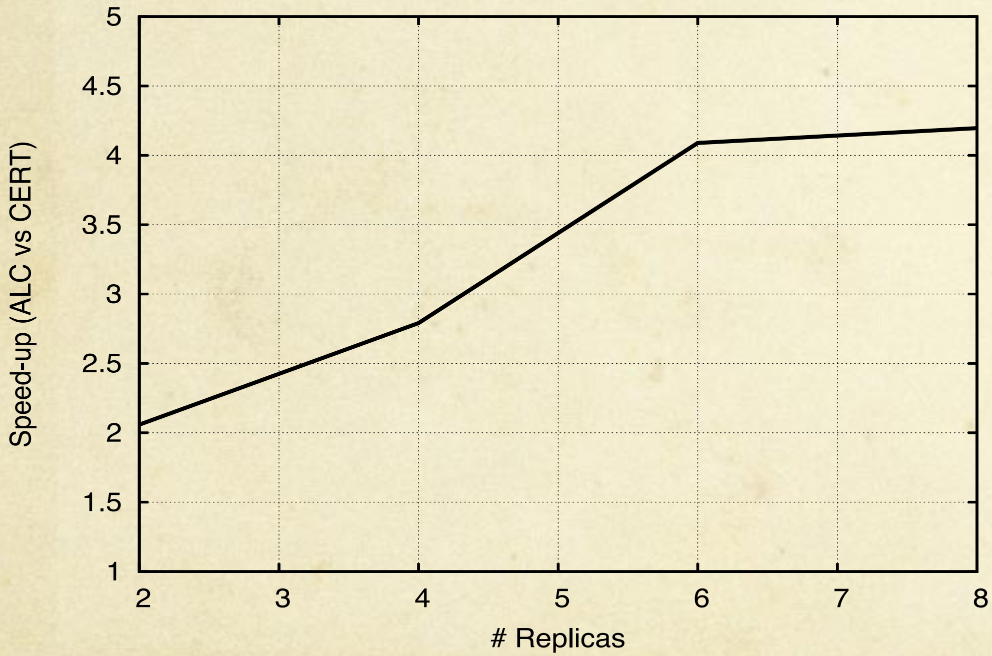


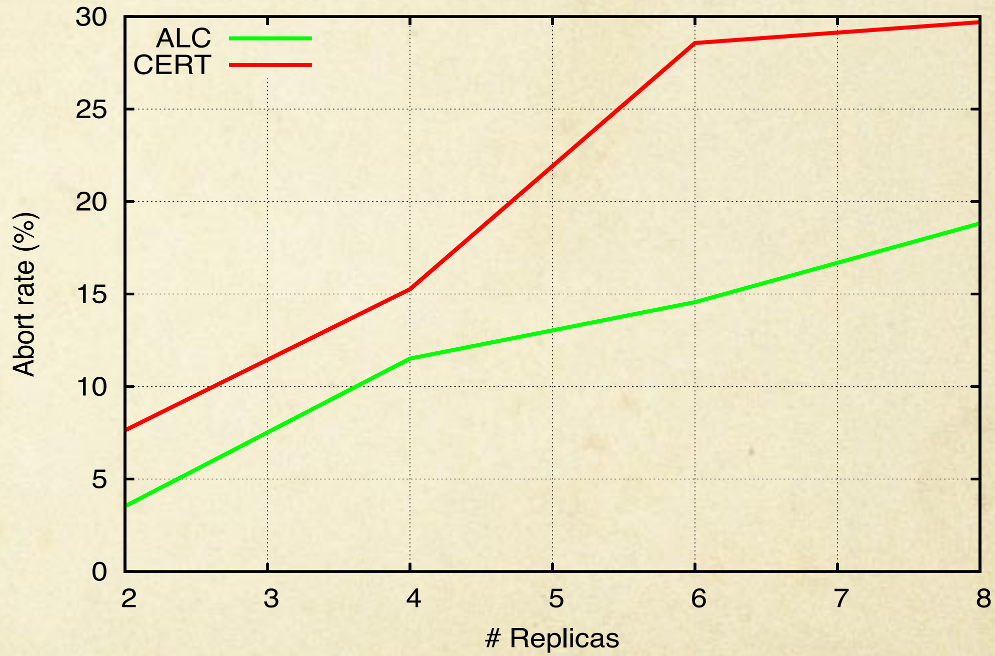


M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

ALC

- Lee Benchmark





M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

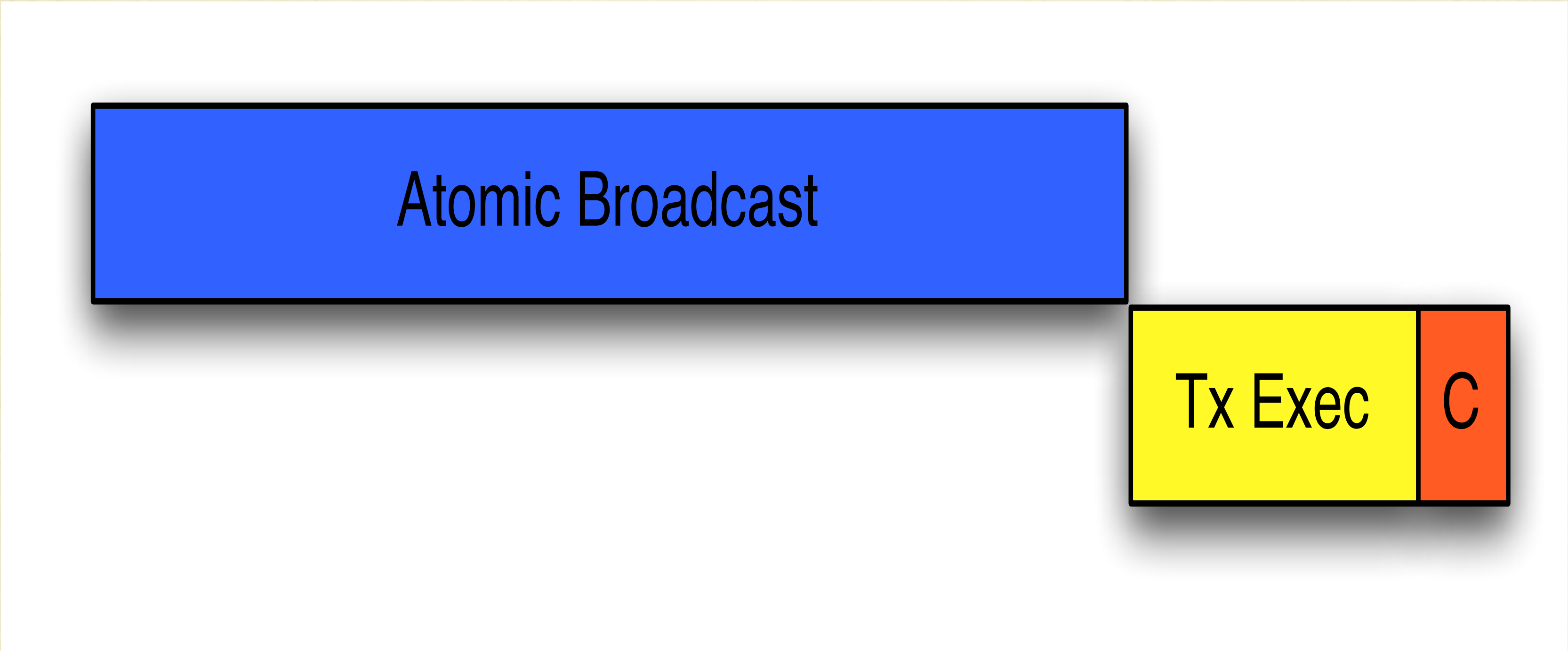
Part III: Case-Studies

- Partitioned Non-Replicated
  - STM for clusters (Cluster-STM)
- Partitioned (Replicated)
  - Static Transactions (Sinfonia)
- Replicated Non-Partitioned
  - Certification-based with Bloom Filters (D^2STM)
  - Certification with Leases (ALC)
  - **Active Replication with Speculation (AGGRO)**

AGGRO

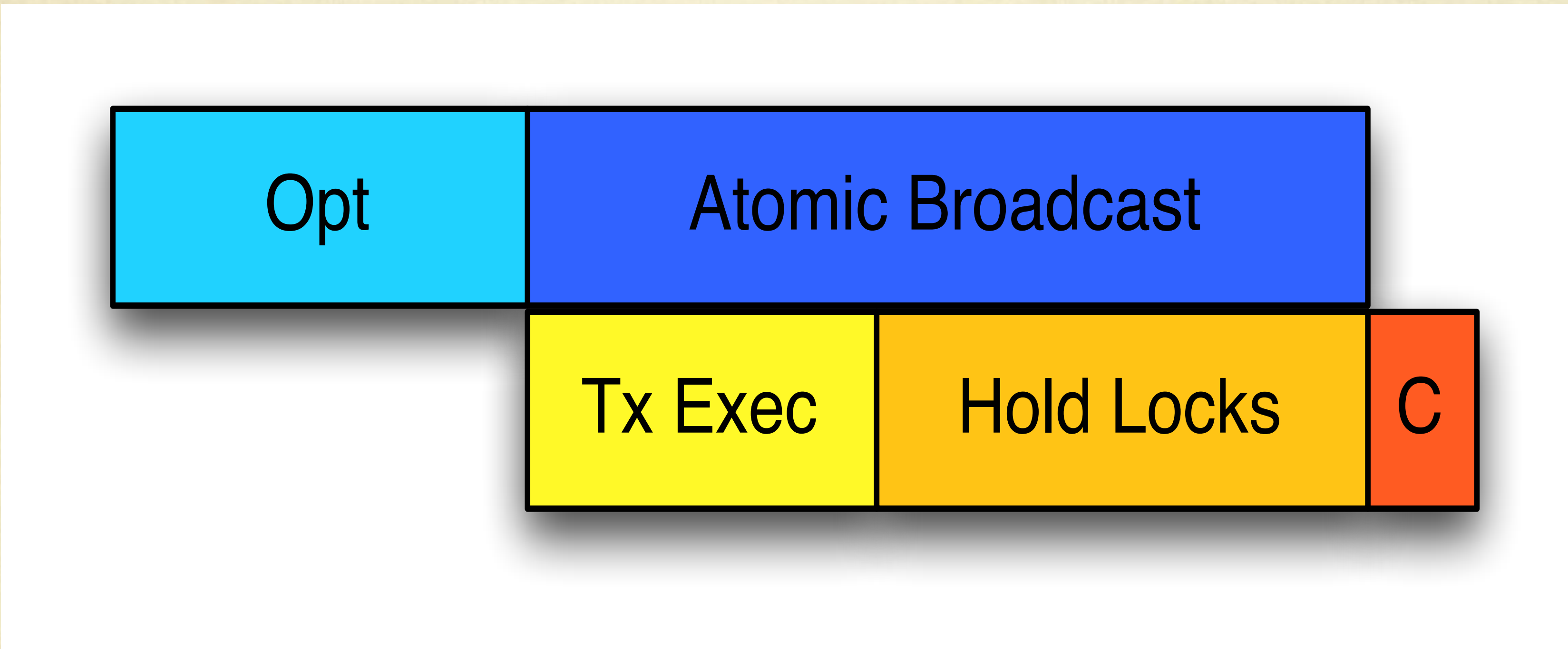
- AGGRO: Boosting STM Replication via Aggressively Optimistic Transaction Processing
- R. Palmieri, Paolo Romano and F. Quaglia, 2010
- Active Replication for STMs
  - Multiple replicas
  - All replicas execute update transactions
  - Read-only transactions can execute in any replica
  - Data survives failures of replicas

Basic Active Replication



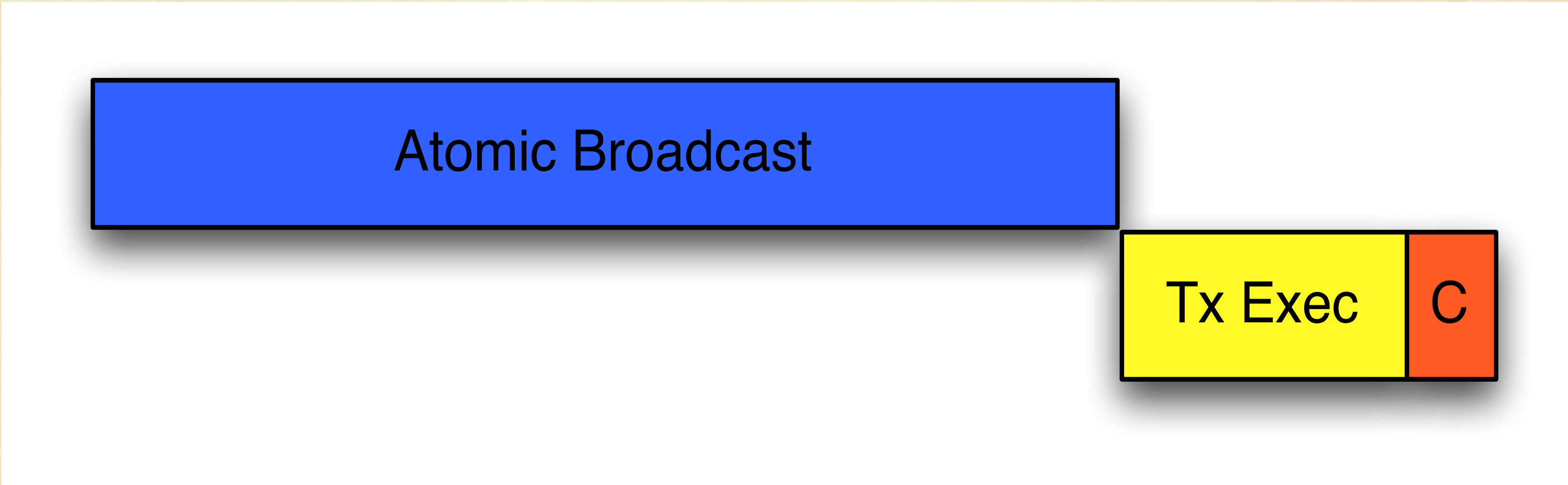
M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

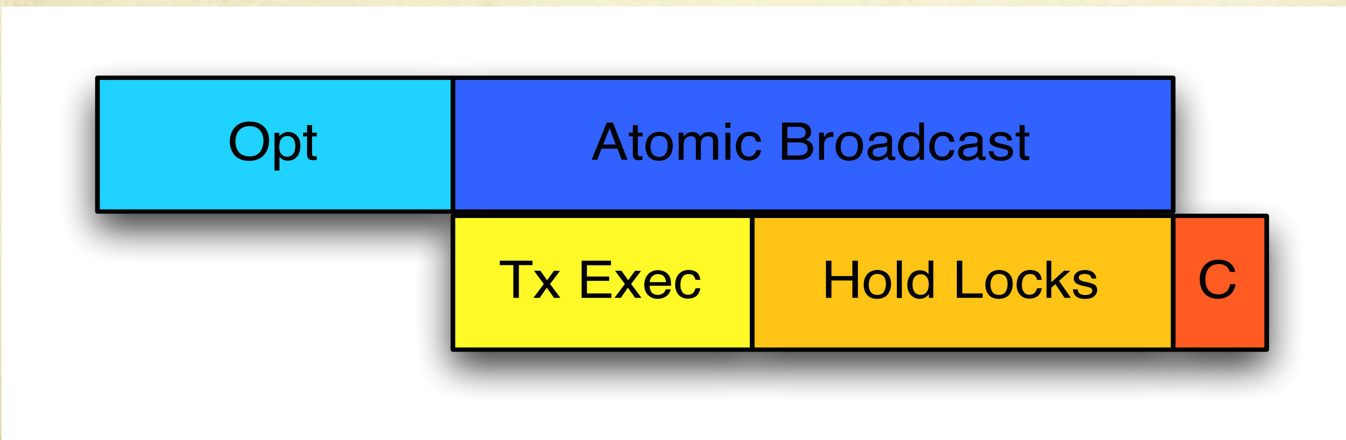
With Optimistic Delivery



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

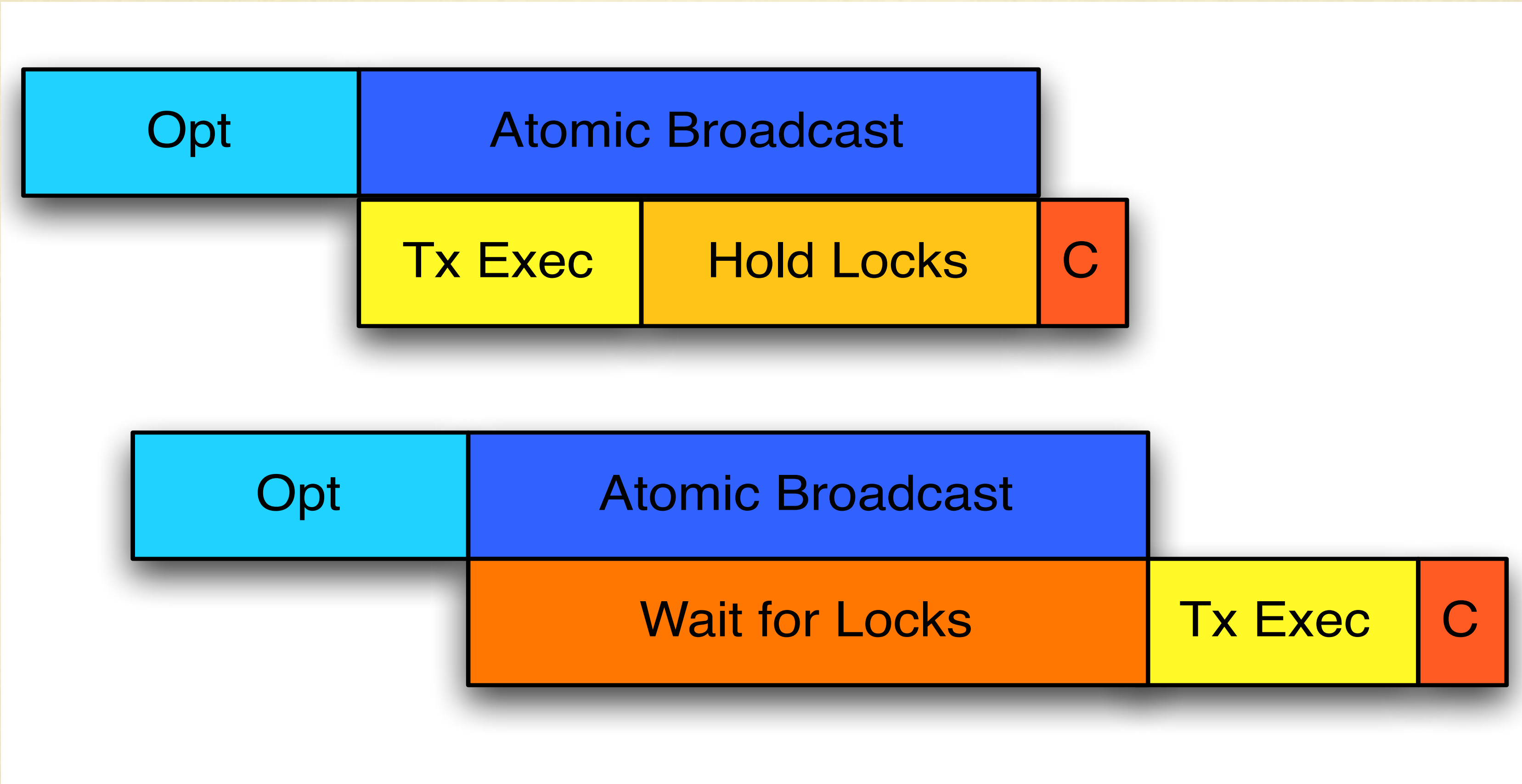
Improvement





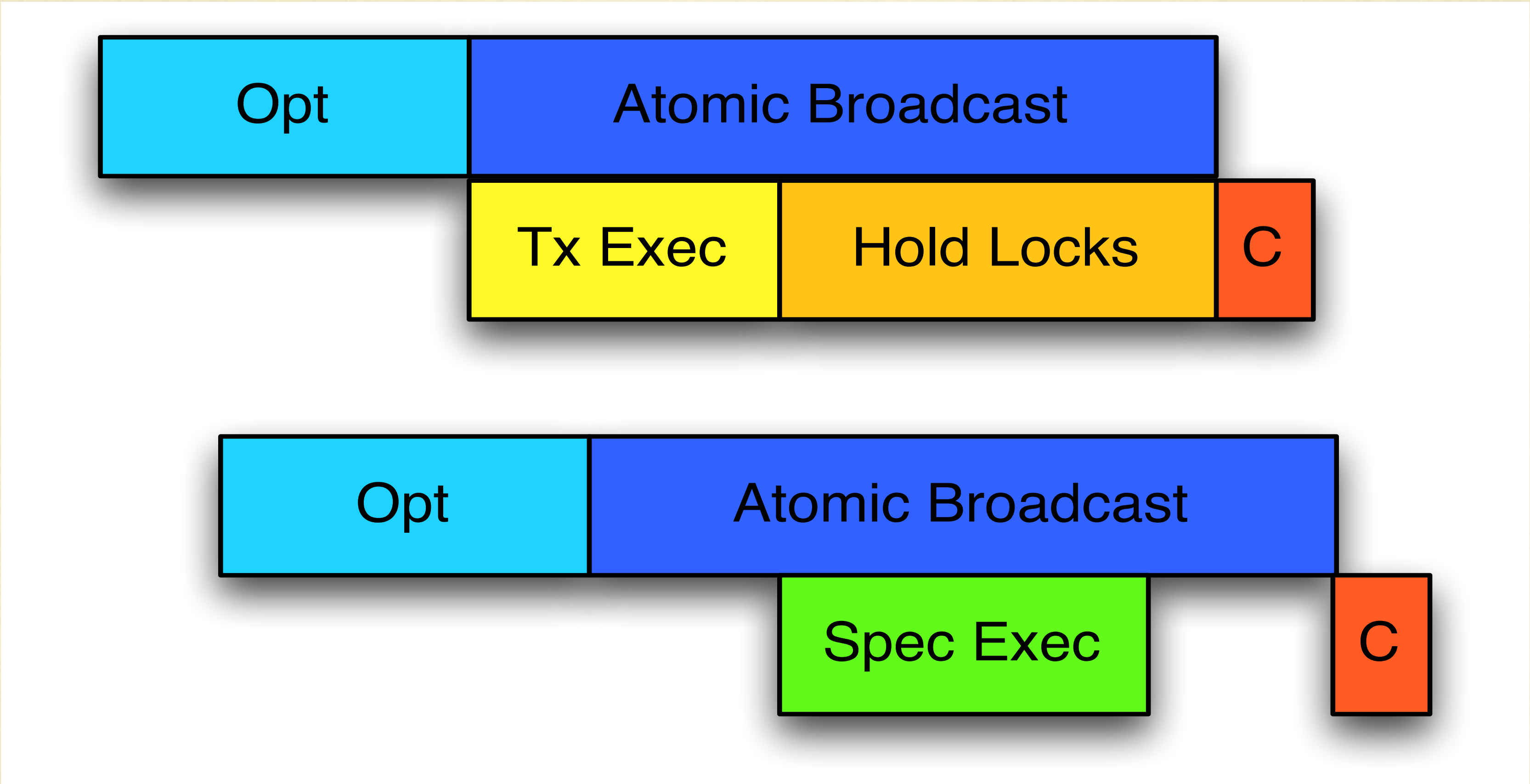
M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

But...



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Speculative



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

AGGRO

- Transactions are started in speculative order immediately after the optimistic delivery
- Writes kill all transactions that have read stale data
- Items touched by speculative transactions Tspec are marked as "work-in-progress (WIP)" while Tspec executes
  - When Tspec terminates (but not yet committed) items are unmarked as WIP.
  - Transaction only read values from terminated transactions.

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

AGGRO Algorithm

upon **opt-Deliver**(Ti)

start transaction Ti in a speculative fashion

AGGRO Algorithm

upon **write**(Ti, X, v)

if (X not already in Ti.WS)

add X to Ti.WS

mark X as WIP // C&S

for each Tj that follows Ti in OAB order:

if (Tj read X from **Tk** preceding Ti) abort Tj

else

update X in Ti.WS

AGGRO Algorithm

upon **read**(Ti, X)

if (X in Ti.WS) return X.value from Ti.WS

if (X in Ti.RS) return X.value from Ti.RS

wait while (X is marked WIP)

let Tj be tx preceding Ti in OAB order that wrote X

Ti.readFrom.add(Tj)

AGGRO Algorithm

```
upon completed (Ti)
    atomically {
        for each X in Ti.WS: unmark X as WIP by Ti
    }
upon commit(Ti)
    atomically {
        for each X in Ti.WS: mark X as committed
    }
```

AGGRO Algorithm

upon **abort**(Ti)

abort any transaction that read from Ti

restart Ti

upon **TO-Deliver**(Ti)

append Ti to TO-order

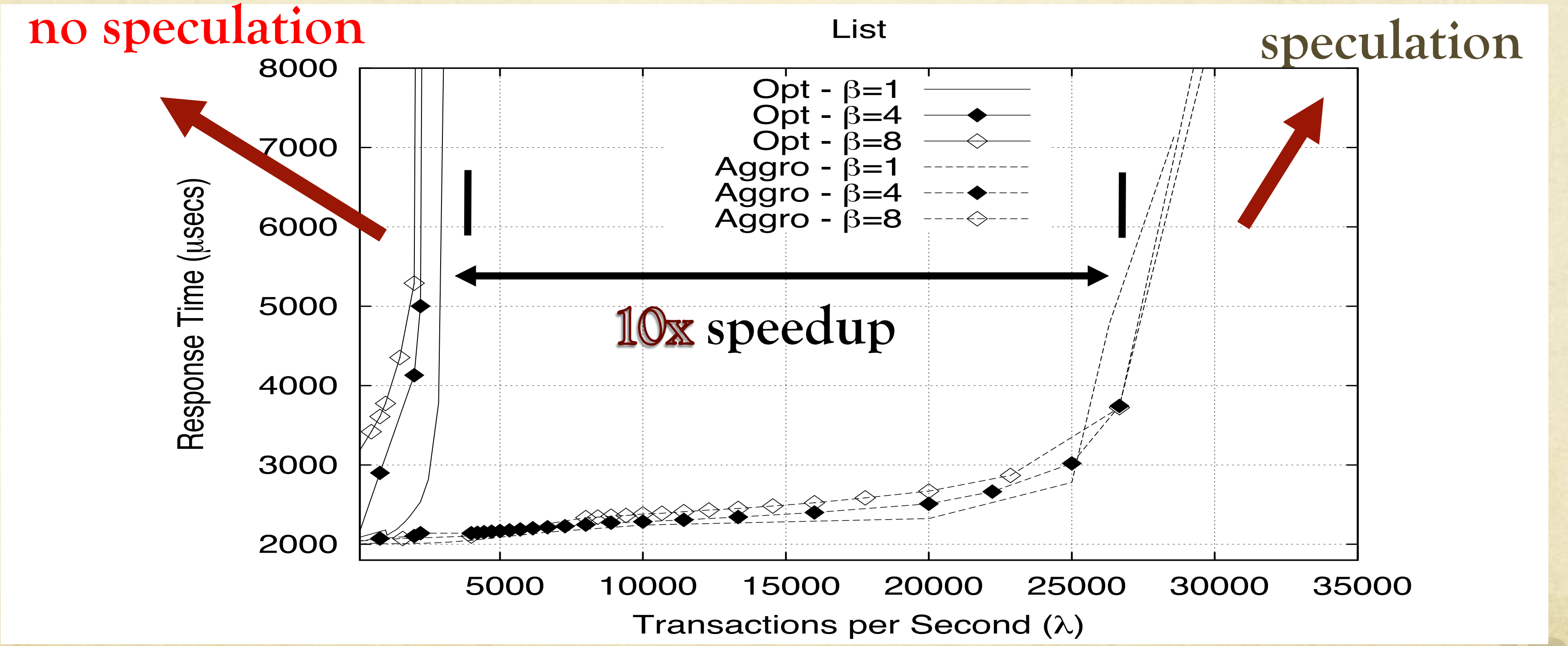
wait until all xacts preceding Ti in TO-order committed

if (validation of Ti's readset fails) abort (Ti)

else commit(Ti)

AGGRO Performance

- Performance speed-up
  (20% reordering, only one SO explored)



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Contents

Conclusions

- Replication helps in read-dominated workloads or when writes have low conflicts
- Replication provides fault-tolerance
- Some techniques have promising results

Conclusions

- No technique outperforms the others for all workloads, networks, number of machines, etc
- Autonomic management of the distributed consistency and replication protocols
  - Change the protocols in runtime, in face of changing workloads

A bit of publicity

- Time for the commercials

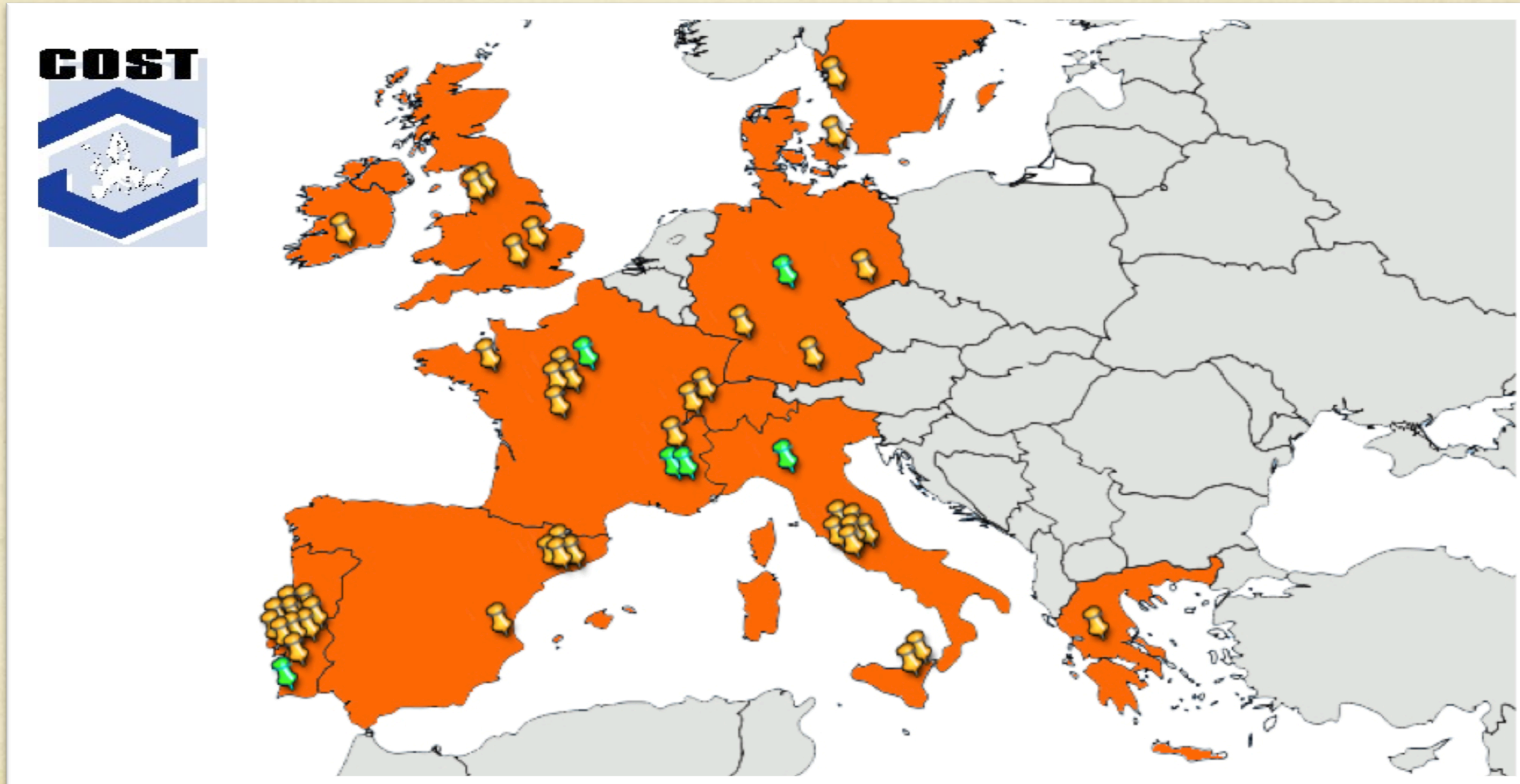
CLOUD-TM

- DTMs: a programming paradigm for the Cloud ?
- Stay tuned on **www.cloudtm.eu**

Cloud·TM
A novel programming paradigm for the Cloud

SEVENTH FRAMEWORK PROGRAMME

Euro-TM



M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011

Euro-TM Cost Action

- Research network bringing together leading European experts in the area of TMs
- Contact us if you are interested in joining it:
  - romano@inesc-id.pt
  - ler@inesc-id.pt
- **www.eurotm.org**

Bibliography

- [AGHK06] Hagit Attiya, Leah Epstein, Hadas Shachnai, and Tami Tamir, `Transactional contention management as a non-clairvoyant scheduling problem" PODC 2006.
- [AKWKLJ08] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson, Lee-TM: A Non-trivial Benchmark for Transactional Memory, In Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2008), Aiya Napa, Cyprus, June 2008.
- [AMSVK09] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. 2009. Sinfonia: A new paradigm for building scalable distributed systems. ACM Trans. Comput. Syst. 27, 3, Article 5 (November 2009)

Bibliography

- [BAC08] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. 2008. Software transactional memory for large scale clusters. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP '08). ACM, New York, NY, USA, 247-258

- [CRRC09] M. Couceiro, Paolo Romano, L. Rodrigues and N. Carvalho, D2STM: Dependable Distributed Software Transactional Memory, Proc. IEEE 15th Pacific Rim International Symposium on Dependable Computing (PRDC'09)

- [CS06] João Cachopo and Antonio Rito-Silva. 2006. Versioned boxes as the basis for memory transactions. Sci. Comput. Program. 63, 2 (December 2006), 172-185.

Bibliography

- [DSS06] D. Dice, O.Shalev, N. Shavit, Transactional Locking II, In In Proc. of the 20th Intl. Symp. on Distributed Computing (2006)

- [FC10] Sérgio Fernandes and João Cachopo, A scalable and efficient commit algorithm for the JVSTM, 5th ACM SIGPLAN Workshop on Transactional Computing

- [GK08] Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP '08). ACM, New York, NY, USA, 175-184.

Bibliography

- [GHP05] Rachid Guerraoui, Maurice Herlihy, Bastian Pochon: Toward a theory of transactional contention managers. PODC 2005: 258-264

- [GU09] Rachid Guerraoui, Tutorial on transactional memory, CAV 2009

- [HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. 2003. Software transactional memory for dynamic-sized data structures. In Proceedings of the twenty-second annual symposium on Principles of distributed computing (PODC '03). ACM, New York, NY, USA, 92-101.

Bibliography

- [JKV07] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. 2007. STMBench7: a benchmark for software transactional memory. SIGOPS Oper. Syst. Rev. 41, 3 (March 2007), 315-324.

- [MCKO08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, Kunle Olukotun, STAMP: Stanford Transactional Applications for Multi-Processing, In IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization, Sept. 2008.

- [MMA06] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. 2006. Exploiting distributed version concurrency in a transactional memory cluster. In Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '06). ACM, New York, NY, USA, 198-208.

Bibliography

- [KAJLKW08] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham and Ian Watson, DiSTM: A Software Transactional Memory Framework for Clusters, In the 37th International Conference on Parallel Processing (ICPP'08), September 2008

- [PQR10] R. Palmieri, Paolo Romano and F. Quaglia, AGGRO: Boosting STM Replication via Aggressively Optimistic Transaction Processing, Proc. 9th IEEE International Symposium on Network Computing and Applications (NCA), Cambridge, Massachussets, USA, IEEE Computer Society Press, July 2010

- [RPQCR10] Paolo Romano, R. Palmieri, F. Quaglia, N. Carvalho and L. Rodrigues, An Optimal Speculative Transactional Replication Protocol, Proc. 8th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA), Taiwan, Taipei, IEEE Computer Society Press, September 2010

Bibliography

- [RRCC10] Paolo Romano, L. Rodrigues, N. Carvalho and J. Cachopo, Cloud-TM: Harnessing the Cloud with Distributed Transactional Memories , ACM SIGOPS Operating Systems Review, Volume 44 , Issue 2, April 2010
- [SR11] Saad and Ravindran, Distributed Transactional Locking II, Technical Report, Virgina Tech, 2011.

The End

Thank you.