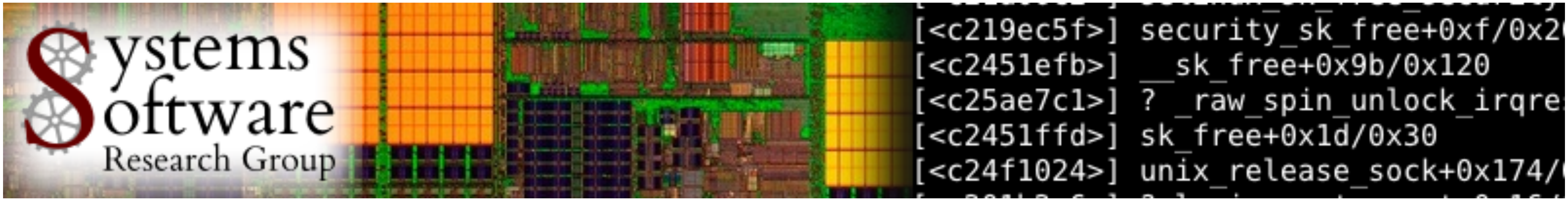


On the Fault-tolerance and High Performance of Replicated Transactional Systems

Dr. Sachin Hirve
Virginia Tech

ece Bradley Department of Electrical & Computer Engineering

10th September 2015

VirginiaTech Invent the Future

Distributed Operations

- In today's world distributed operations are ubiquitous
- Example -

Image sources: rikbasra.com, database.bio, iconsplace.com, iconfinder.com, guillaumekurkdjian.com, mytay.mobi, prepareyourgroundzero.com, iconshut.com

What are Distributed Operations?

- A logical unit of work that accesses shared data involving two or more servers on the network
- Servers coordinate to service client requests while ensuring consistency of data
- Properties : Atomicity, Consistency, Isolation, Durability
- Example -

```
tx_start:
            x = x -10;
            y = 20;
tx_end
```

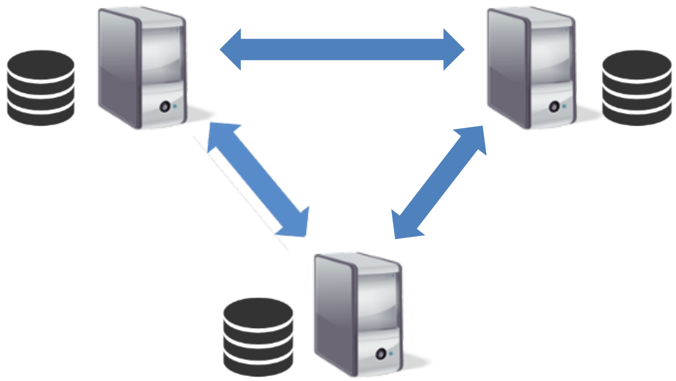


Image sources: rikbasra.com, database.bio, iconsplace.com

Distributed Operations

- Desired properties
  - Fault-tolerance
  - High resiliency
  - Failure masking
- State Machine Replication (SMR) [Schneider, 93] is a general approach to achieve these dependability properties.

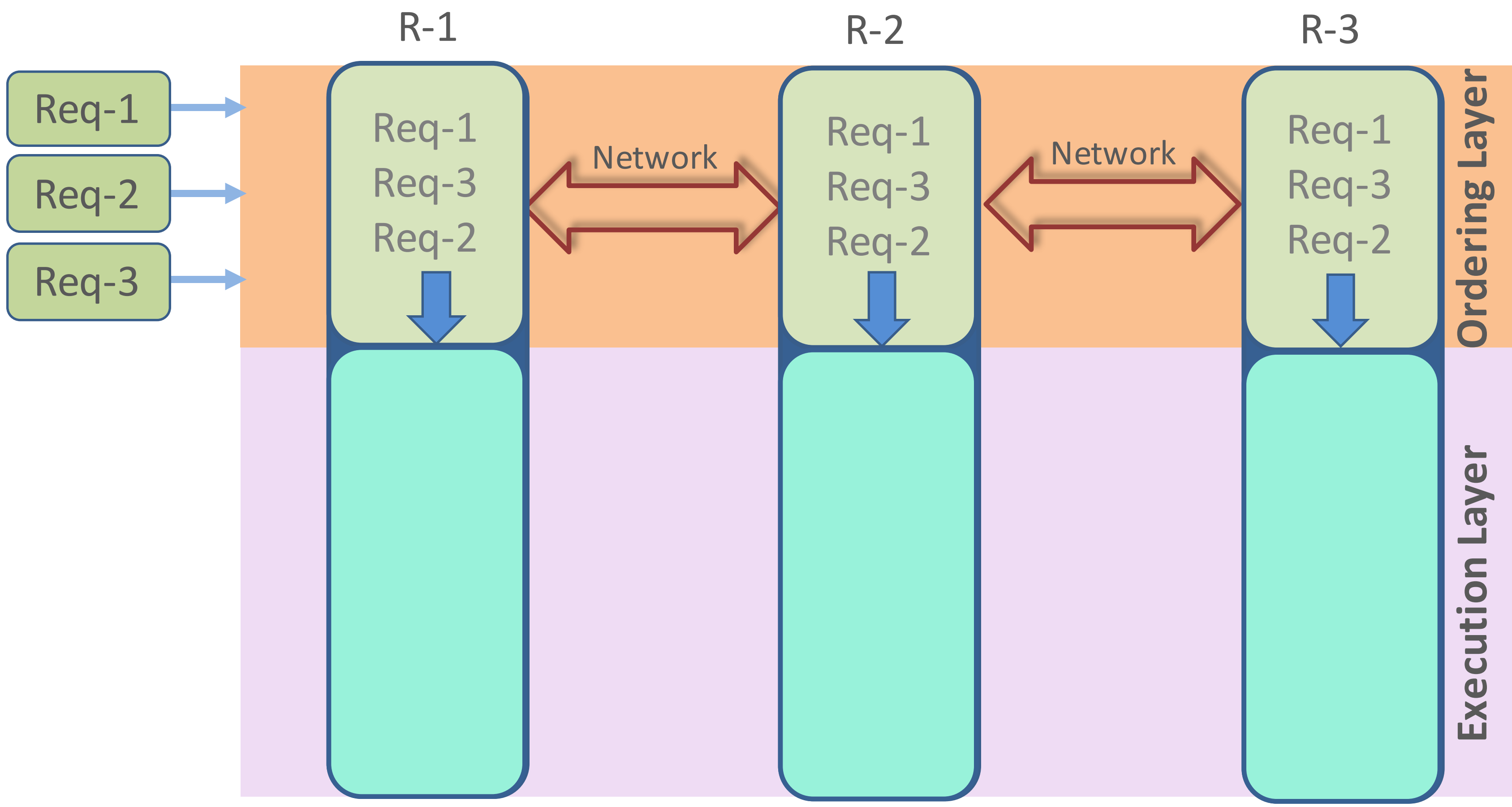
System Model

- A distributed system consists of **N** nodes $\{P_1, P_2, \ldots, P_n\}$, also called servers/replicas
- For ***f*** number of faults, system size **N = 2*f* +1** [Lamport, 98]
- Data is replicated on all nodes
- Only replica crash (non-byzantine) faults are considered
- Clients may or may not be co-located with replicas
- Commands are client requests, that includes operations on shared data

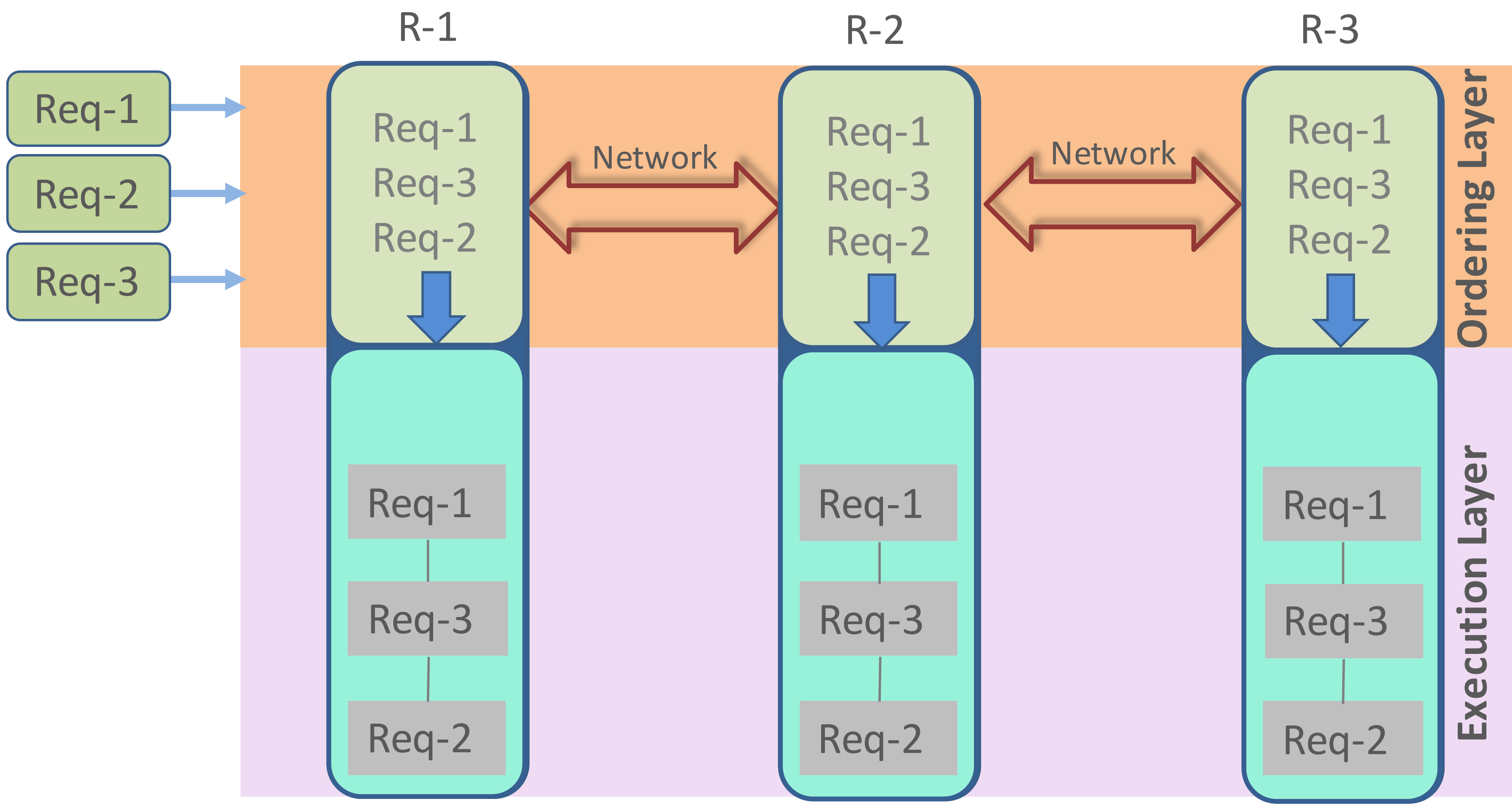
State Machine Replication (SMR)

- SMR implements fault-tolerant services by replicating servers and coordinating client interactions with servers
- State machine consists of
  - *State variables* that encode the *state* of the system
  - Commands that transform this *state*
- Building blocks
  - Ordering layer
  - Execution layer

State Machine Replication (SMR)



State Machine Replication (SMR)



How SMR meets dependability properties?

- Properties of SMR
  - Consistent state
  - High availability
  - Failure masking

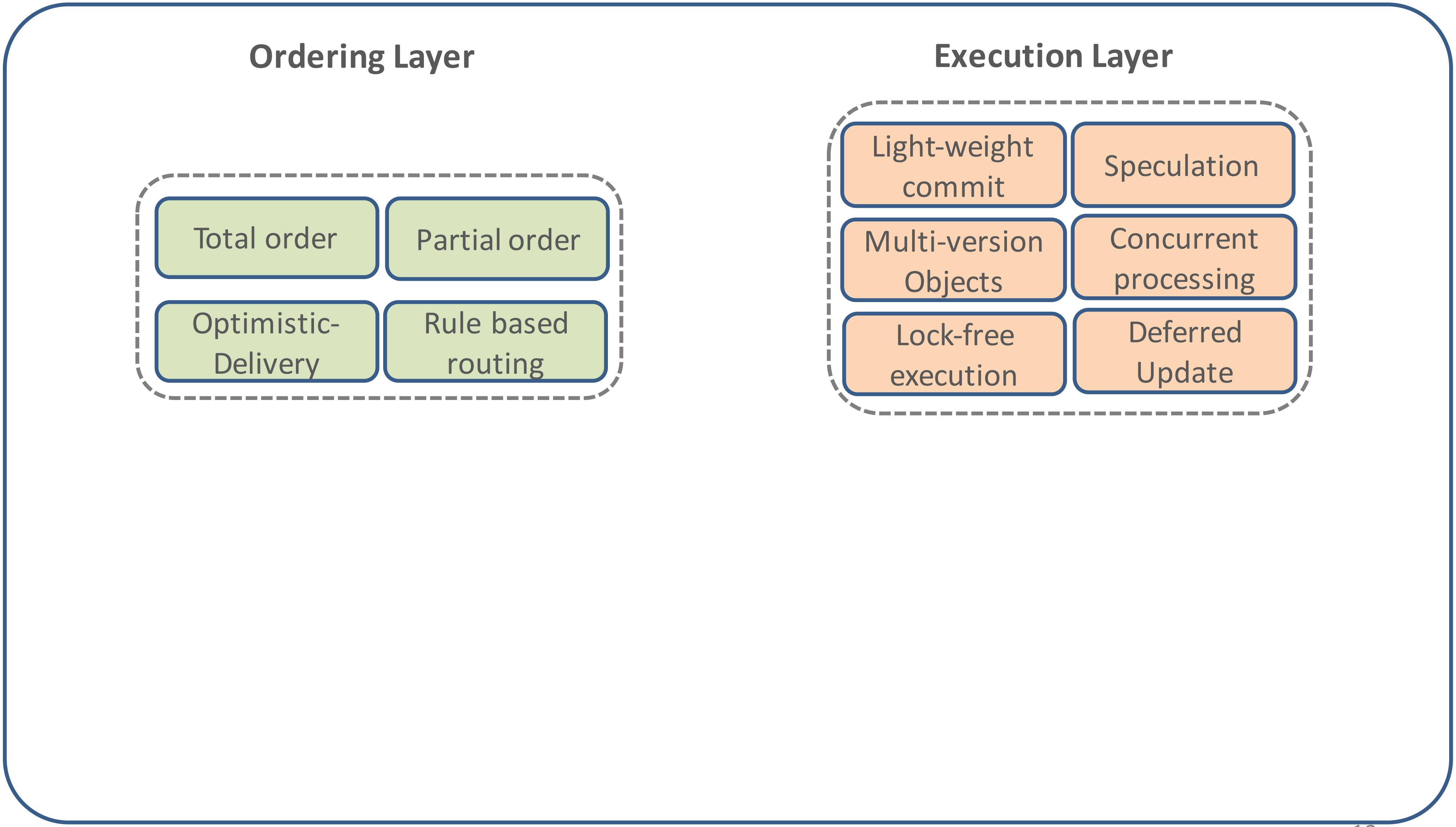
SMR – Ordering layer

- Total order
  - Replicas define order of requests "blindly", without looking at conflicts
  - Generally request are serially executed
  - Examples – Paxos [Lamport, 98], Mencius (baseline) [Mao, 08]
- Partial Order
  - Order is defined among conflicting requests
  - Better possible concurrency for request execution
  - Examples – Generalized Paxos [Lamport, 05], Epaxos [Moraru, 13]

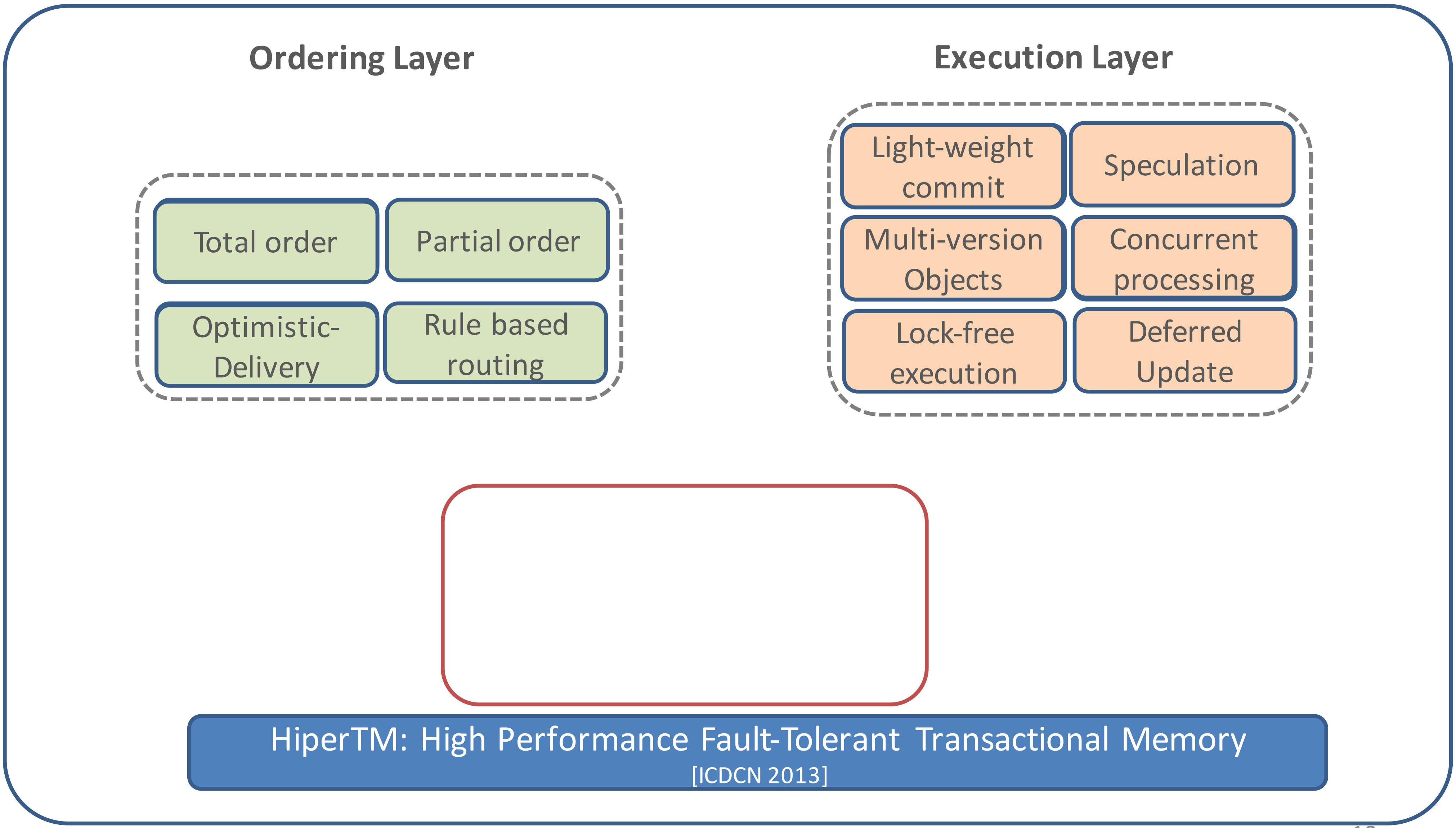
SMR – Execution layer

- Deferred Update Replication (DUR)
  - Requests are executed optimistically prior to order finalization and at final order, they are validated and committed
  - High concurrency and performance for rare conflicts among requests
  - Fails to exploit concurrency in high conflict scenarios
- Deferred Execution Replication (DER)
  - Requests are executed after the order is finalized
  - Requests are executed post final-order, therefore conflicts do not lead to aborts
  - Fails to benefit from concurrency

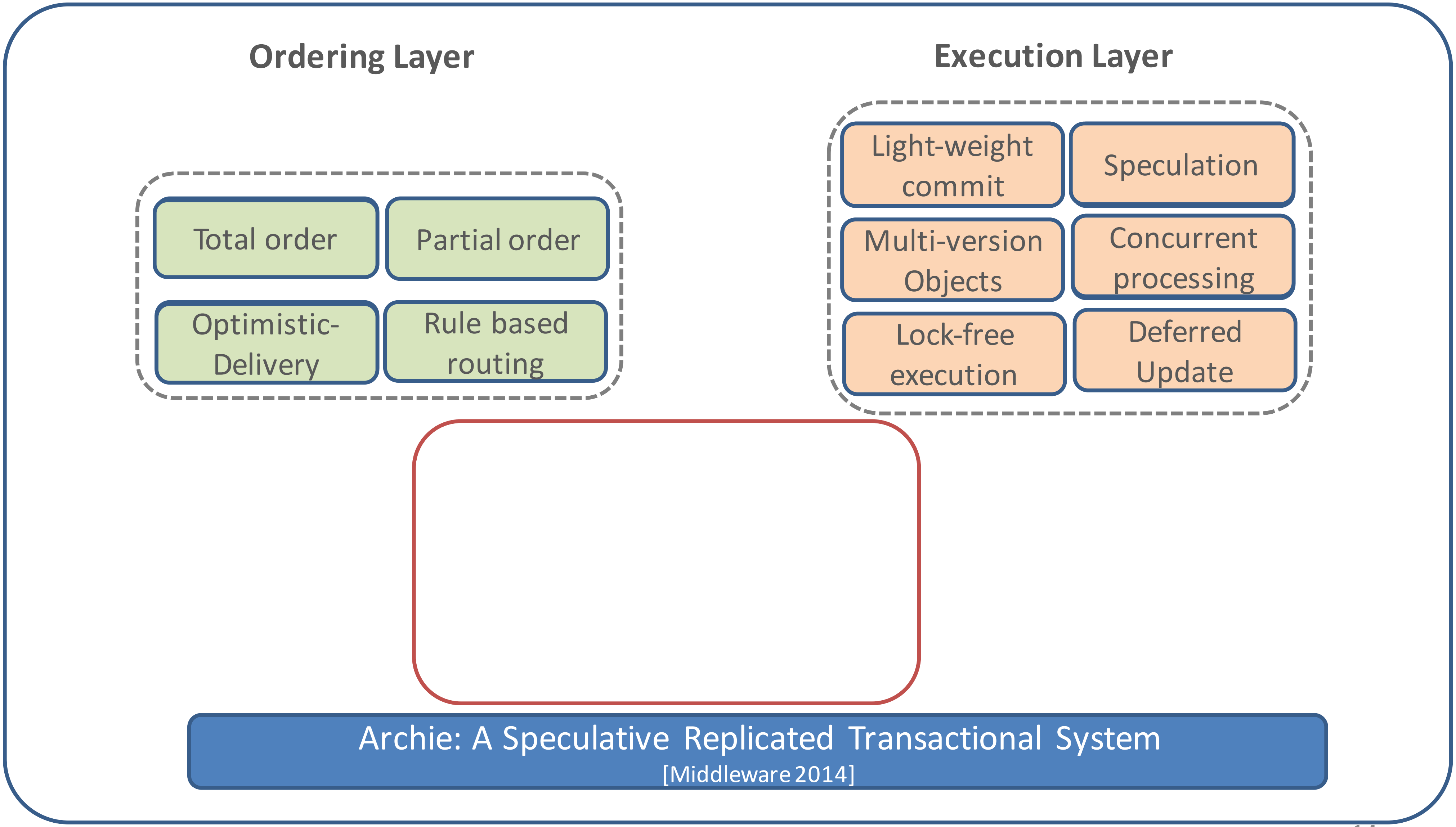
Research Contributions



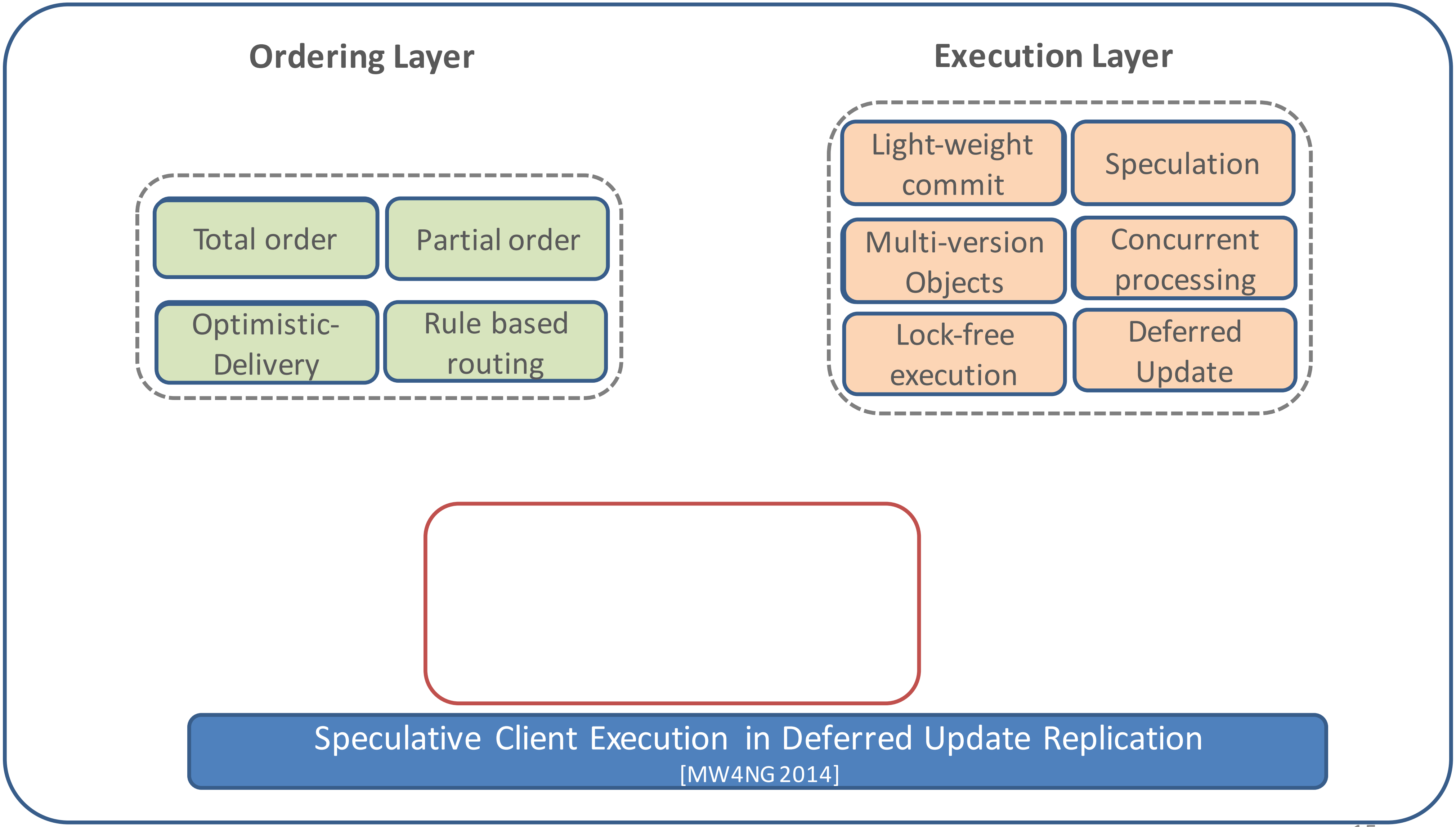
Research Contributions



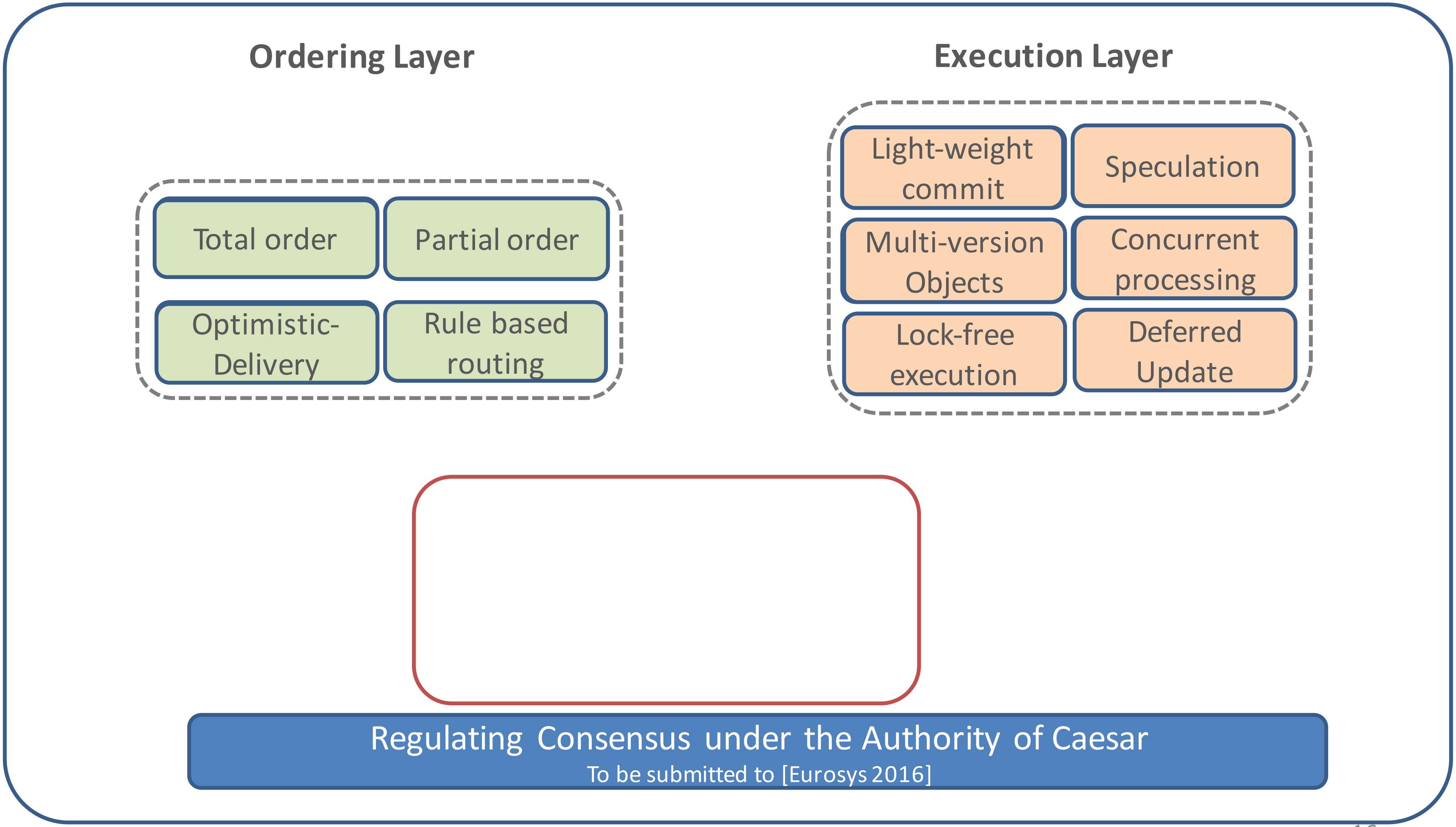
Research Contributions



Research Contributions



Research Contributions



Research Contributions

Ordering Layer

- Total order
- Partial order
- Optimistic-Delivery
- Rule based routing

Execution Layer

- Light-weight commit
- Speculation
- Multi-version Objects
- Concurrent processing
- Lock-free execution
- Deferred Update

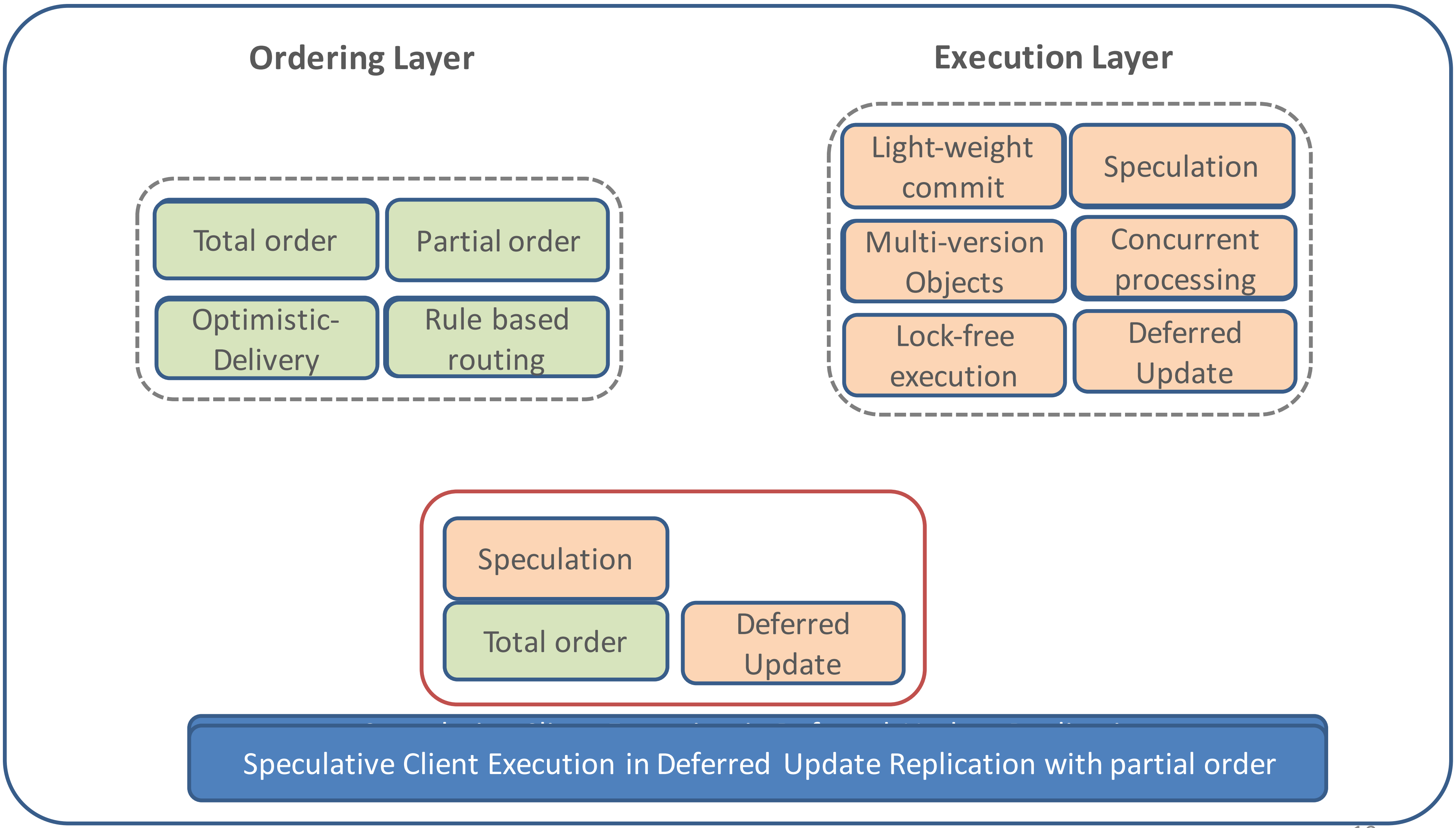
Scaling up Active Replication using Staleness

Submitted to [TPDS]

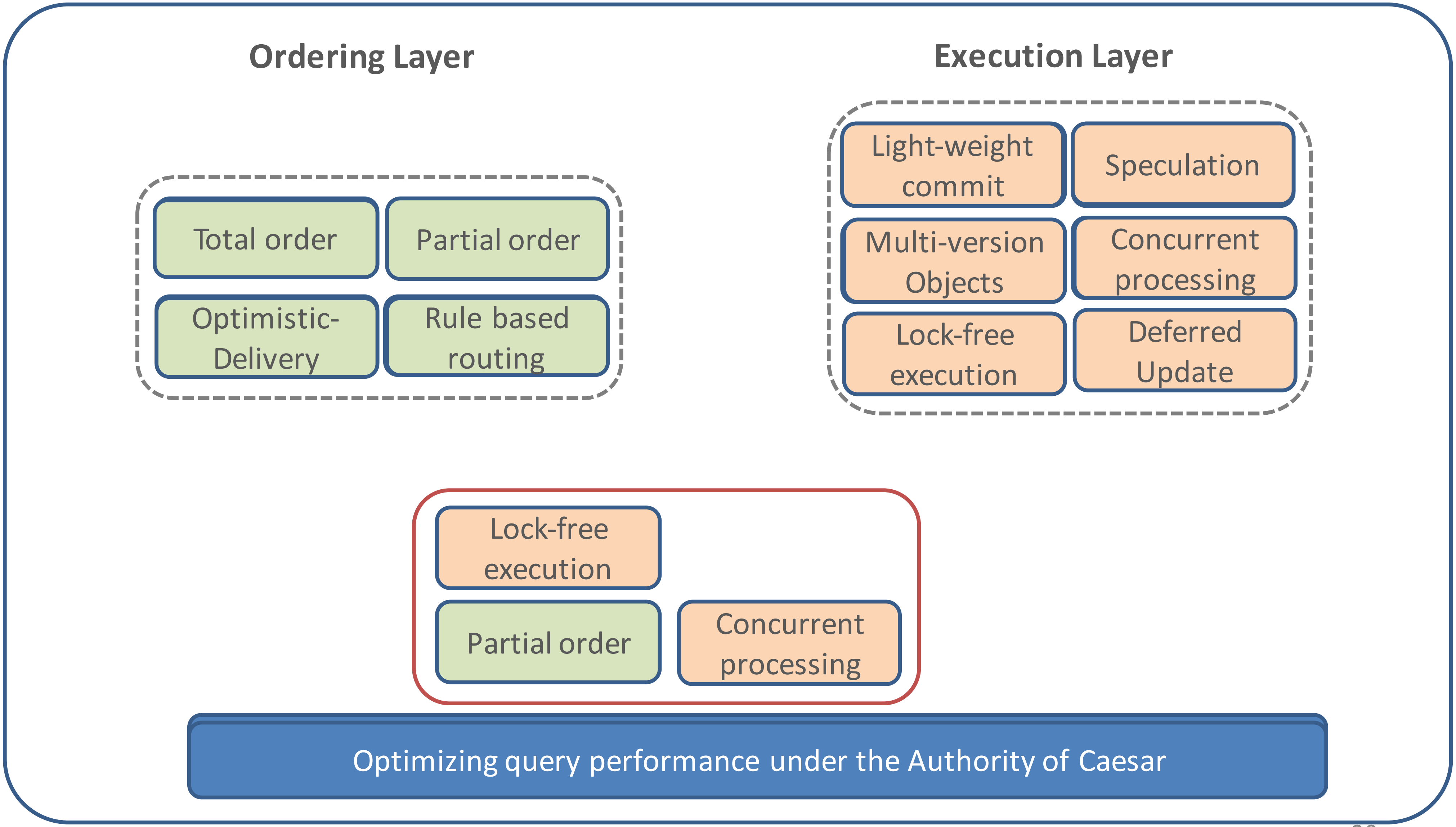
Research Contributions

- What is so special about this set of contributions?
  - These systems are composed of plugins
  - Plugins are not specific to a single system or problem
  - Can be mix-matched to create another system solving different problem

Portability of Contributions – Example1



Portability of Contributions – Example2



Post-Prelim Contributions

- Speculative Client Execution in Deferred Update Replication
  - ACM/IFIP/USENIX 15th Middleware Workshop for Next Generation Computing (MW4NG 14)
- Regulating Consensus under the Authority of Caesar
  - To be submitted to EuroSys 16

Post-Prelim Contributions

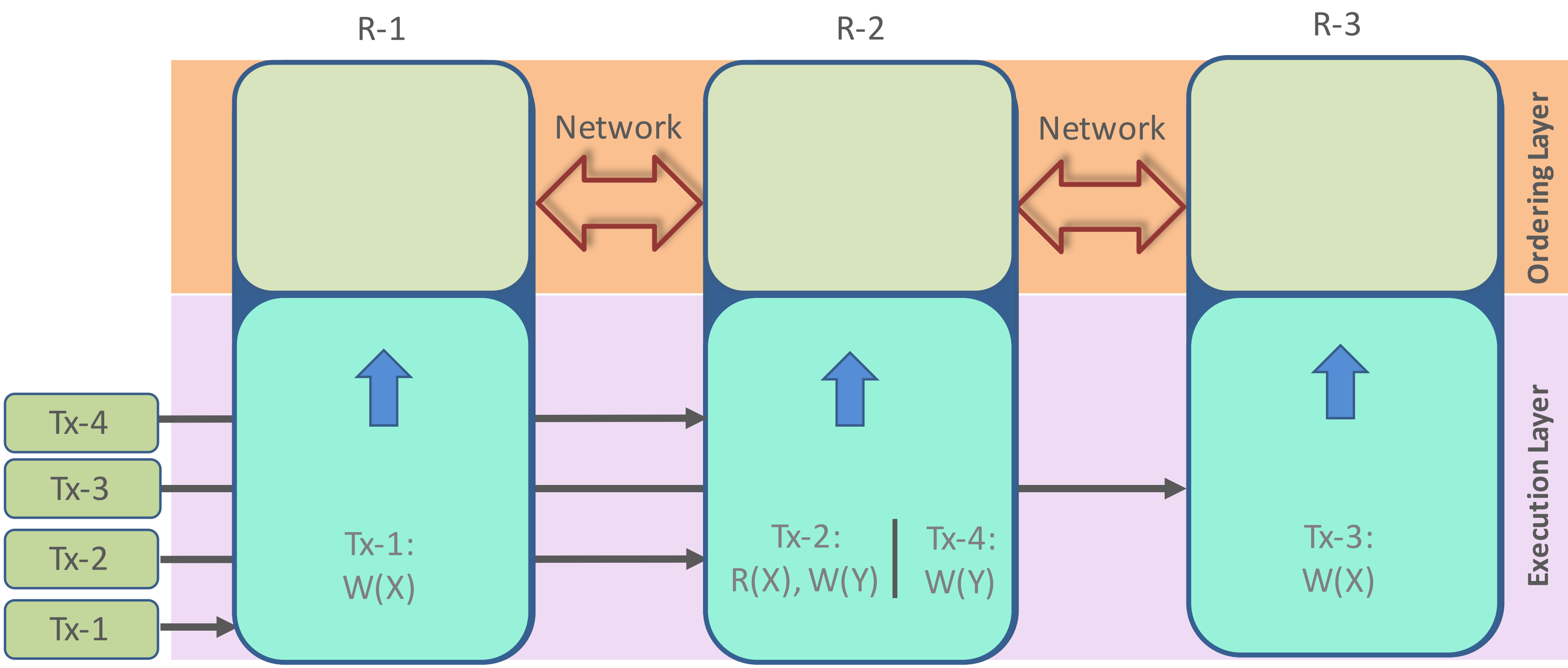
- Speculative Client Execution in Deferred Update Replication
  - ACM/IFIP/USENIX 15th Middleware Workshop for Next Generation Computing (MW4NG 14)
- Regulating Consensus under the Authority of Caesar
  - To be submitted to EuroSys 16

Deferred Update Replication - Definitions

- Optimistic execution
  - A transaction execute assuming all objects accessed by it are up-to-date and no other concurrent transaction accesses those objects
- Readset
  - Collection of objects and versions that are read by transaction
- Writeset
  - Collection of objects that are updated by transaction
- Validation
  - Verifying the validity of objects at commit time that were read earlier during optimistic execution
- Commit
  - Updating the main memory with object updates by the current transaction

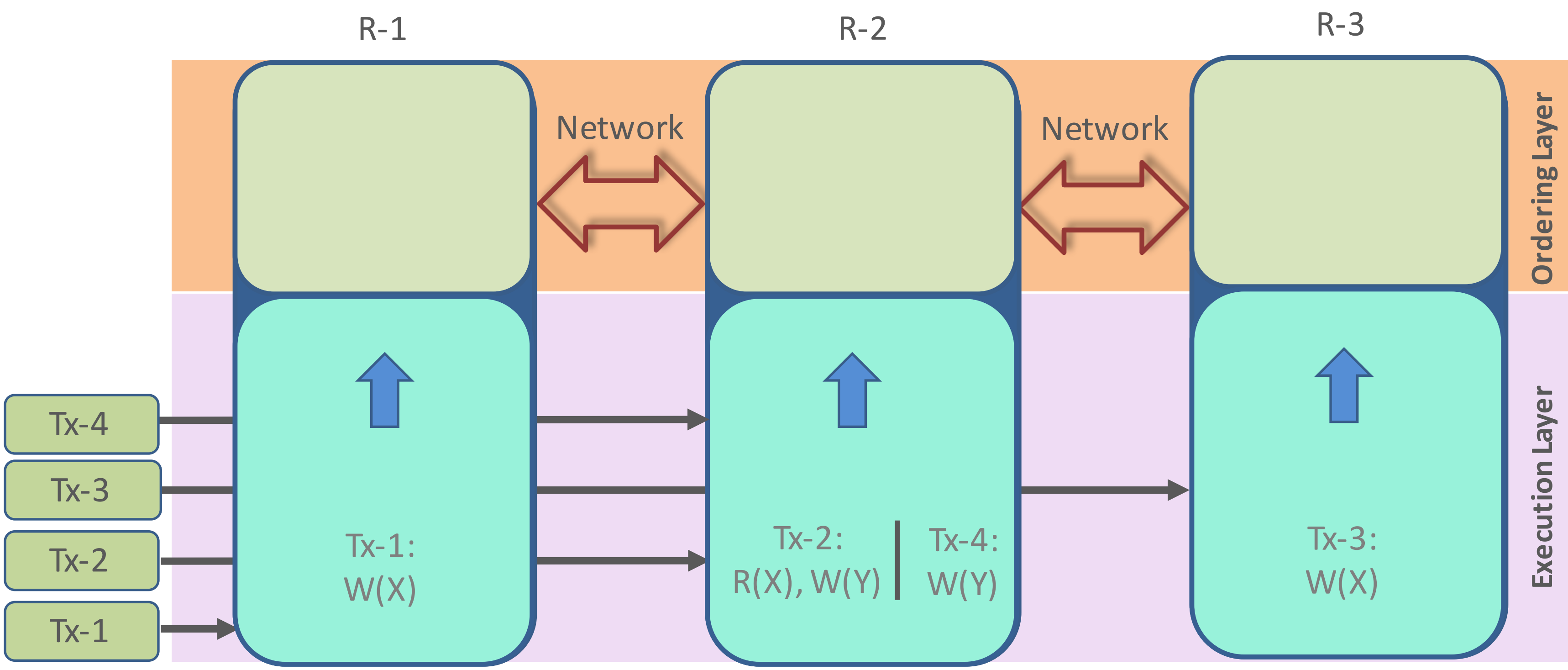
Deferred Update Replication

- Execution model
  - Requests are executed optimistically
  - Transaction updates go through certification phase before they can be committed



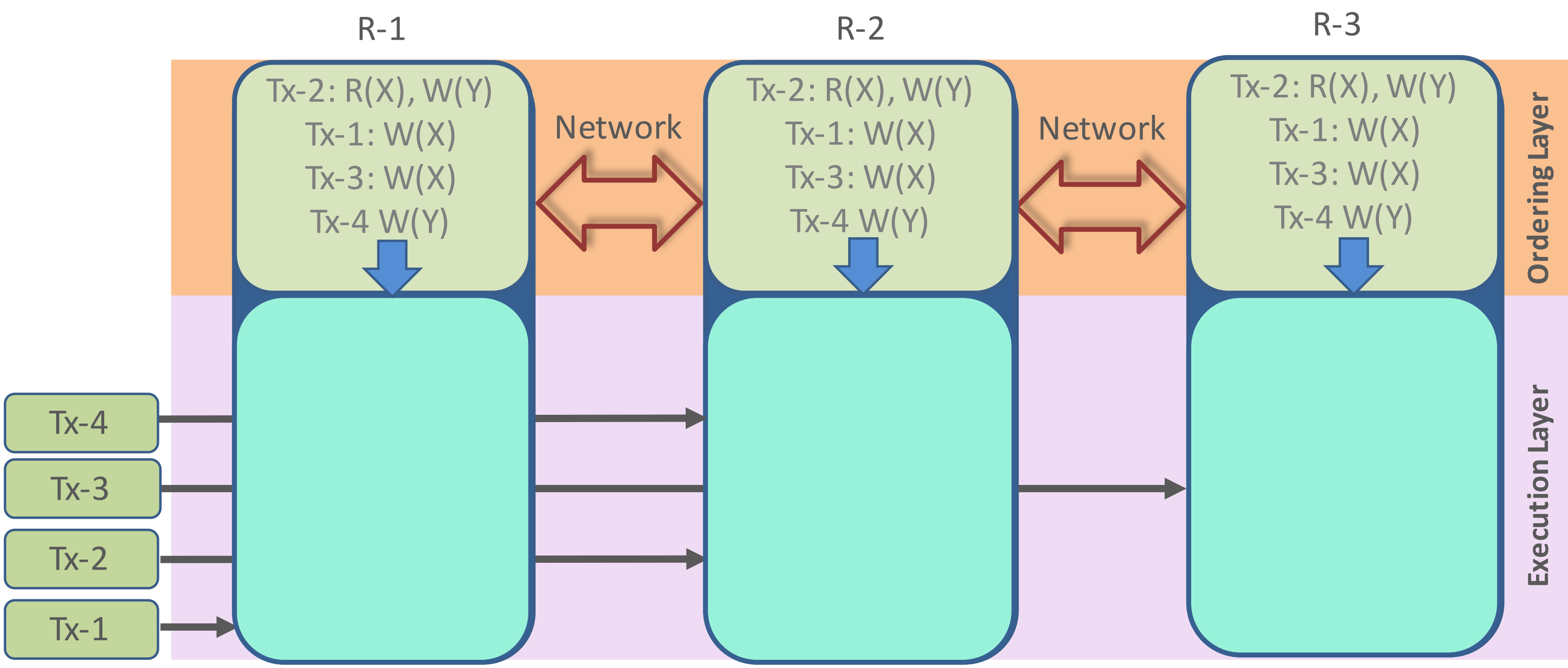
Deferred Update Replication

- A transaction execution model
  - Requests are executed optimistically
  - Transaction updates go through certification phase before they can be committed



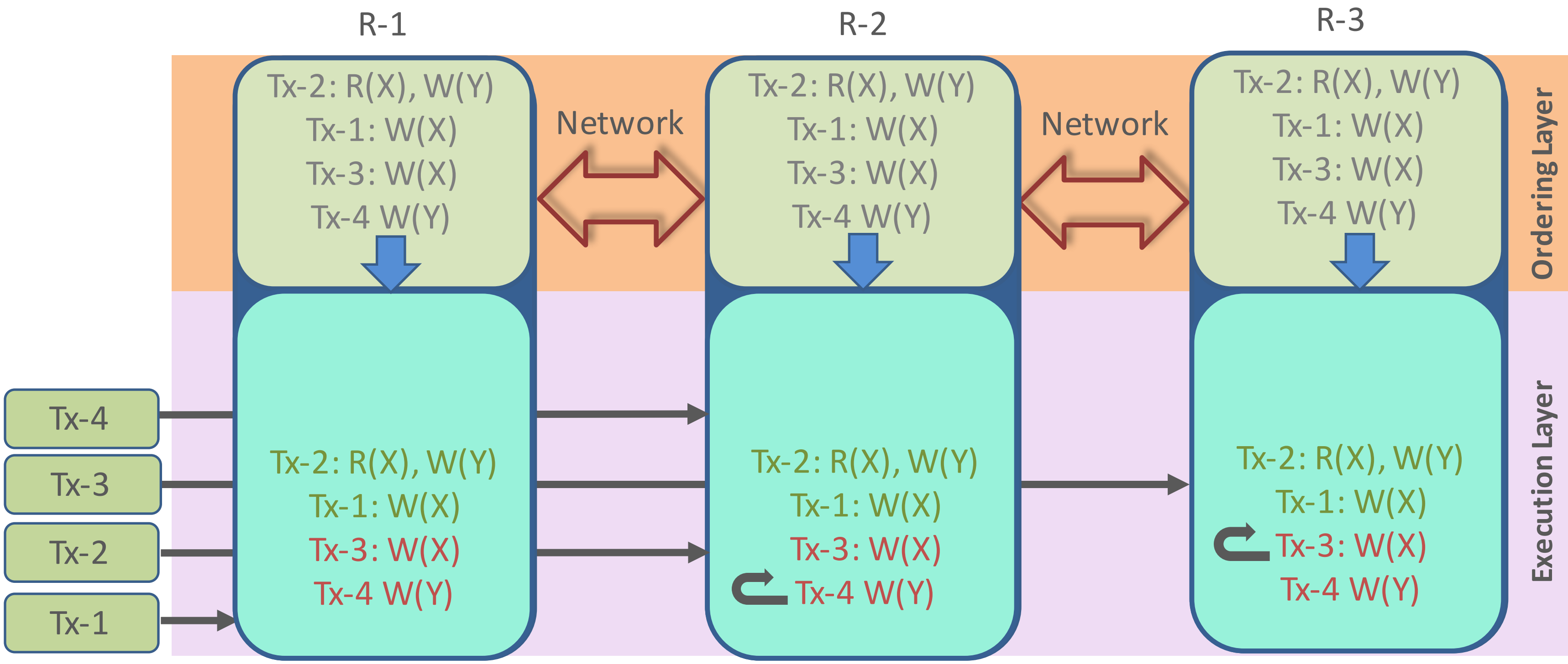
Deferred Update Replication

- Certification phase
  - Defines an order for transaction updates



Deferred Update Replication

- Certification phase
  - Validates transaction updates w.r.t. the defined order
  - On successful validation commits transaction by updating objects
  - On failing validation, aborts the transaction and re-executes



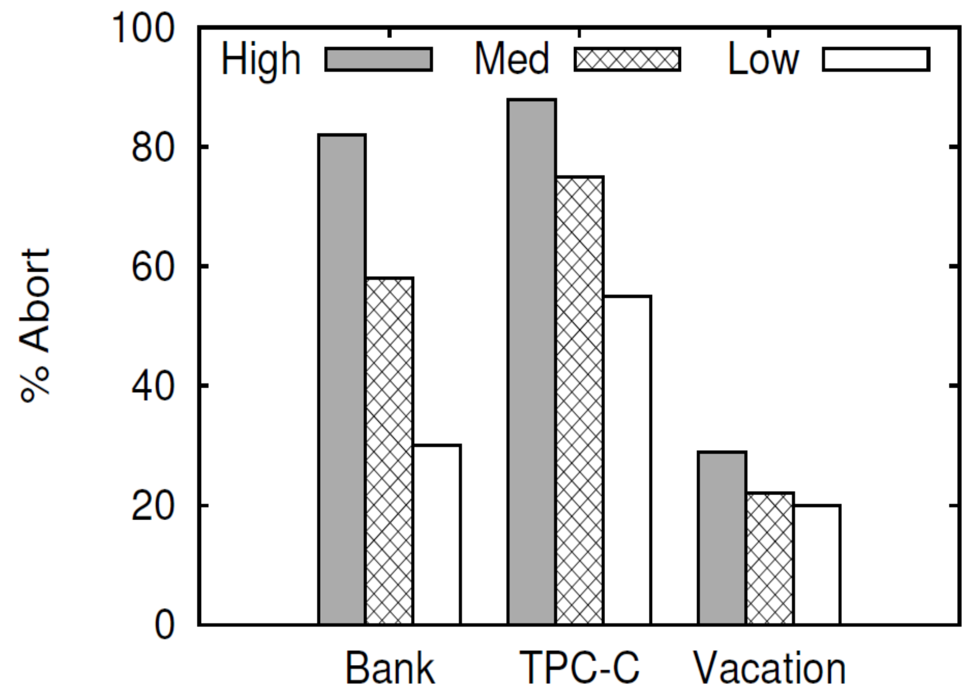
Deferred Update Replication

- Salient points
  - Inherent parallelism of transaction processing
  - In case of rare conflicts among transactions, DUR gives the best performance
  - In high conflict situations, DUR performs poorly due to high number of aborts
  - Even in partitioned access, DUR suffers from aborts among local transactions
- DUR presents an interesting problem to address
  - Applicable to certain applications e.g., TPC-C, an OLTP benchmark
  - Can we avoid aborts among local transactions, even in presence of higher number of conflicts?

Deferred Update Replication

- Impact of local aborts with varying the degree of conflicts
  - Performance of DUR various benchmarks and different contention levels

Contention Level	Accounts	WH	Relations
High	500	23	250
Medium	2000	115	500
Low	5000	230	1000



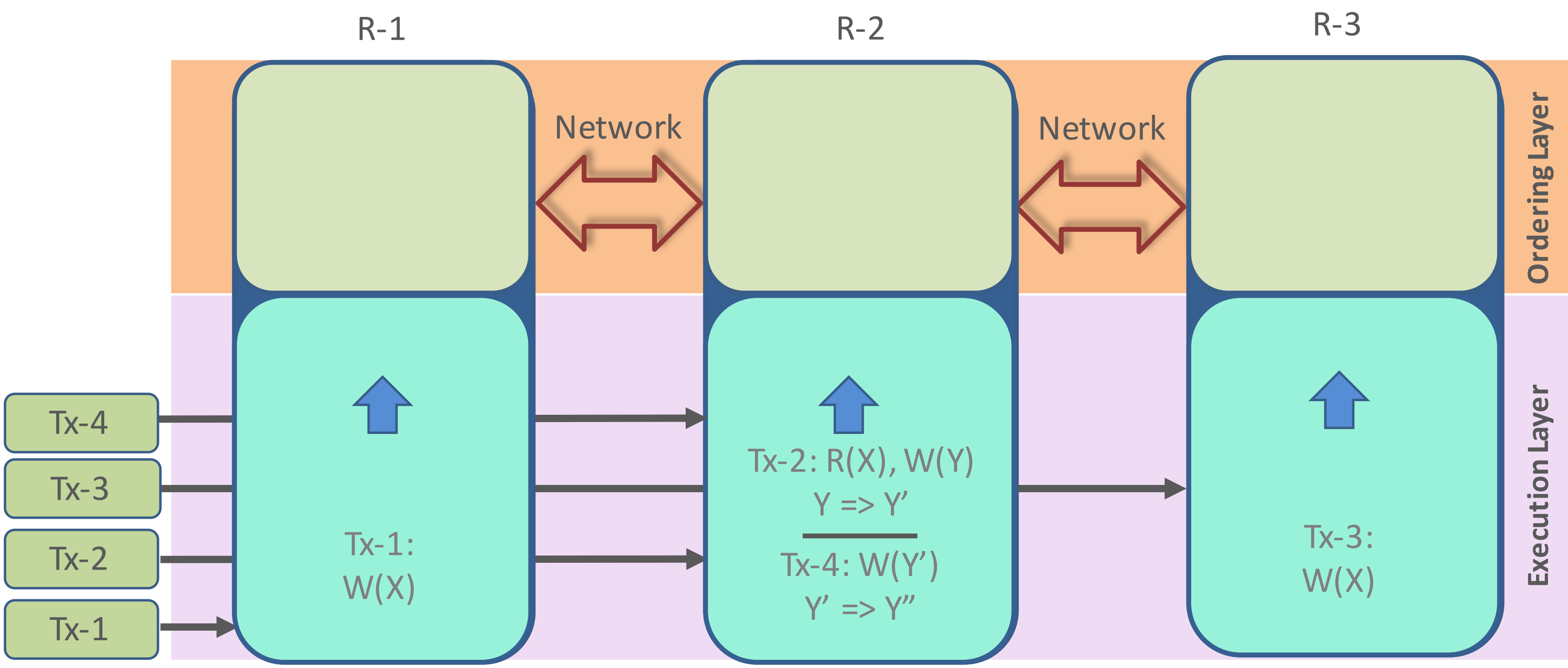
% of aborted transactions on 11 nodes using PaxosSTM

X-DUR – Design goals

- Eliminating conflicts among local concurrent transactions
  - Local transaction ordering
  - Speculation in optimistic execution
- Eliminating aborts from possible reorder in certification phase
  - Enforcing local transaction order to certification phase

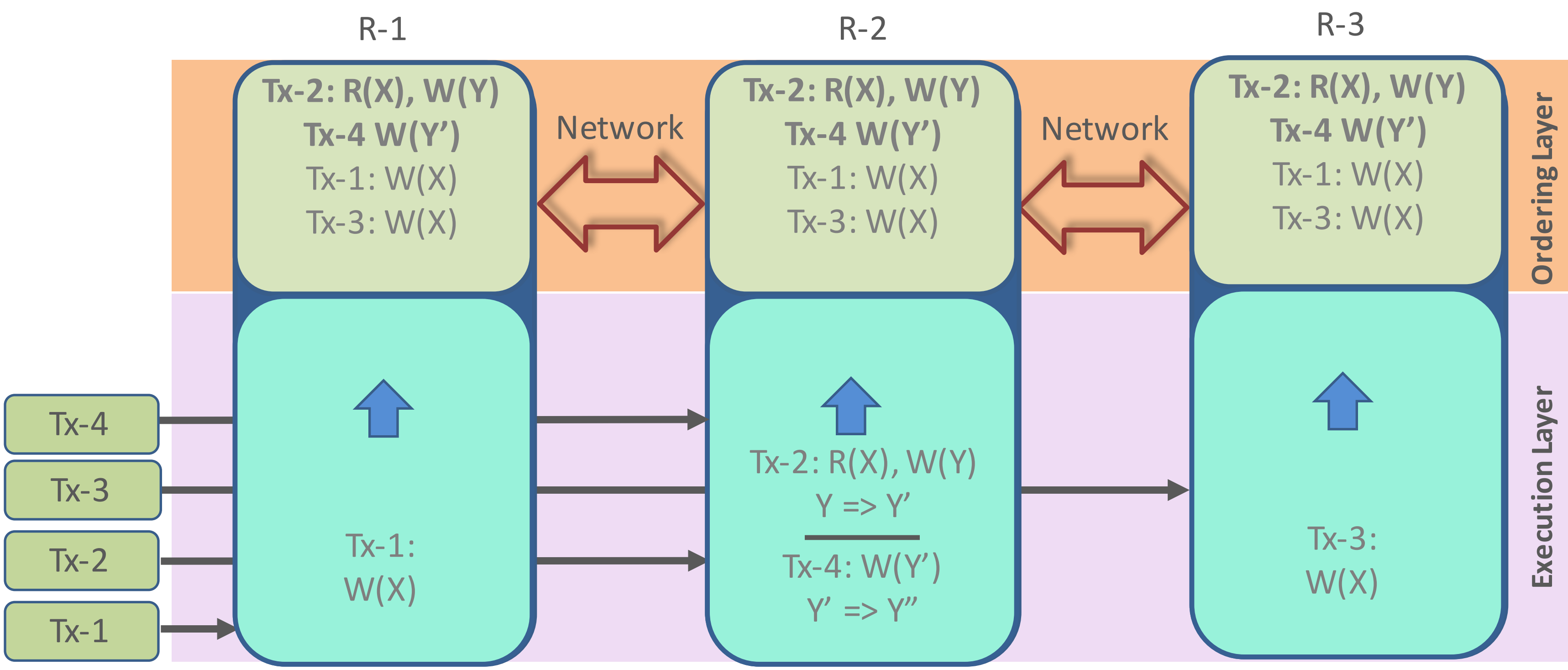
X-DUR

- Execution model
  - A local order is defined among requests
  - Speculation helps to pass on the object updates among locally ordered transactions



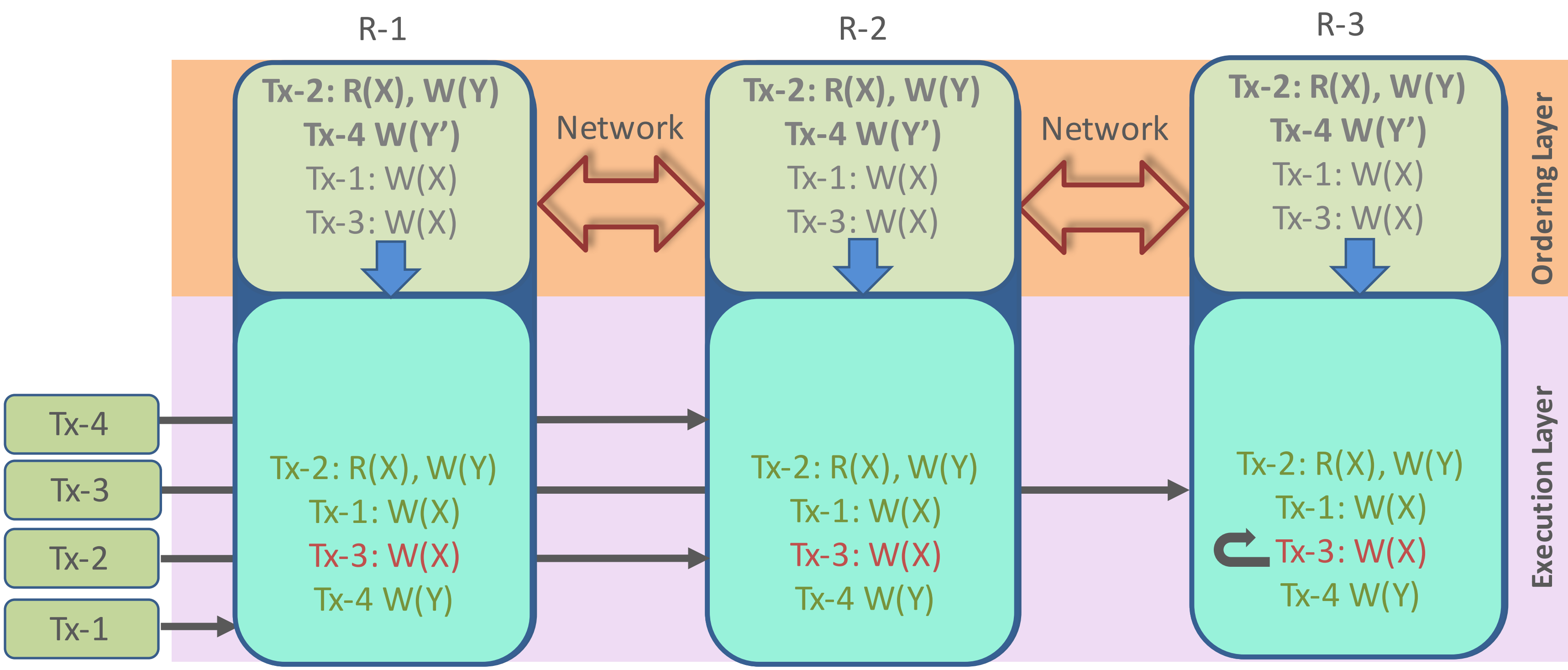
X-DUR

- A transaction execution model
  - Requests are executed optimistically
  - Transaction updates go through certification phase before they can be committed



X-DUR

- Certification phase
  - Validates transaction updates w.r.t. the defined order
  - On successful validation commits transaction by updating objects
  - On failing validation, aborts the transaction and re-executes

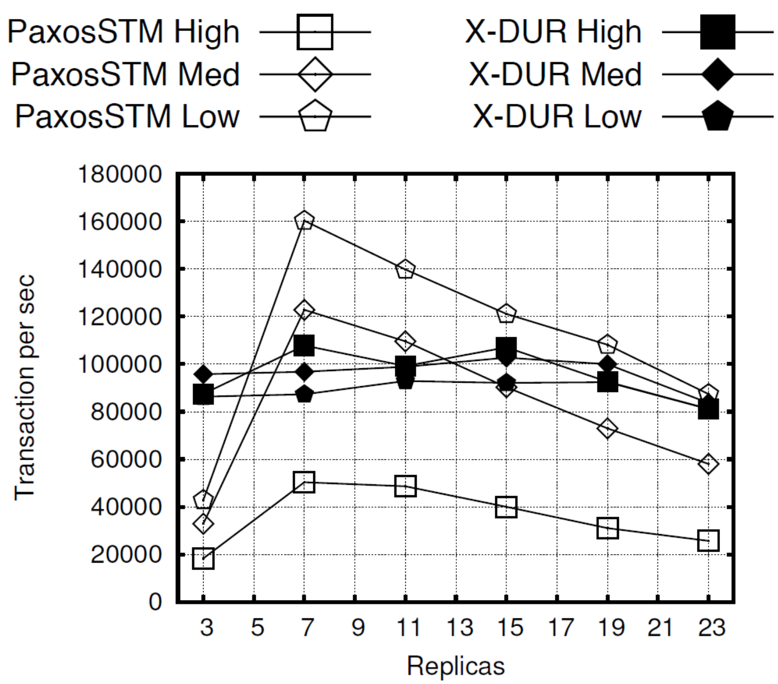


X-DUR : Evaluation

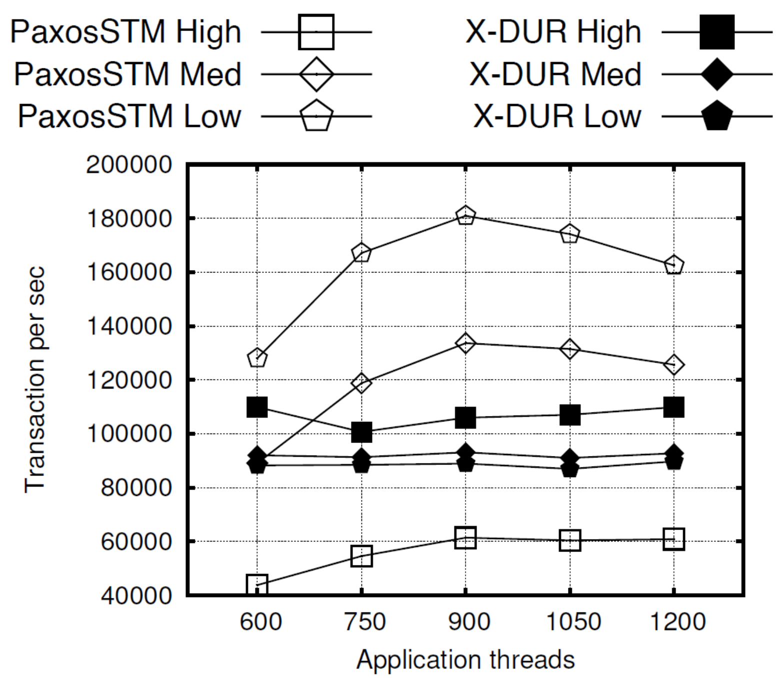
- Testbed – PRObE cluster (23 nodes)
  - AMD Opteron 6272, 64-core, 2.1 GHz CPU
  - 128 GB RAM and 40 Gbps ethernet
- Benchmarks
  - Bank: A micro-benchmark that mimics bank operations
  - TPC-C: A popular OLTP benchmark
  - Vacation: Distributed version of vacation application in STAMP [Minh, 08]
    - Mimics the operations of reserving flight, car etc. for vacation
- Competitor
  - PaxosSTM: a DUR-based system; it suffers from local aborts

Evaluation: Bank

- Contention: 500 objects(high), 2000 objects (medium) and 5000 objects (low)
- For low conflicts, PaxosSTM performs great due to high amount of parallelism
- X-DUR outperforms PaxosSTM in medium-high conflict scenarios



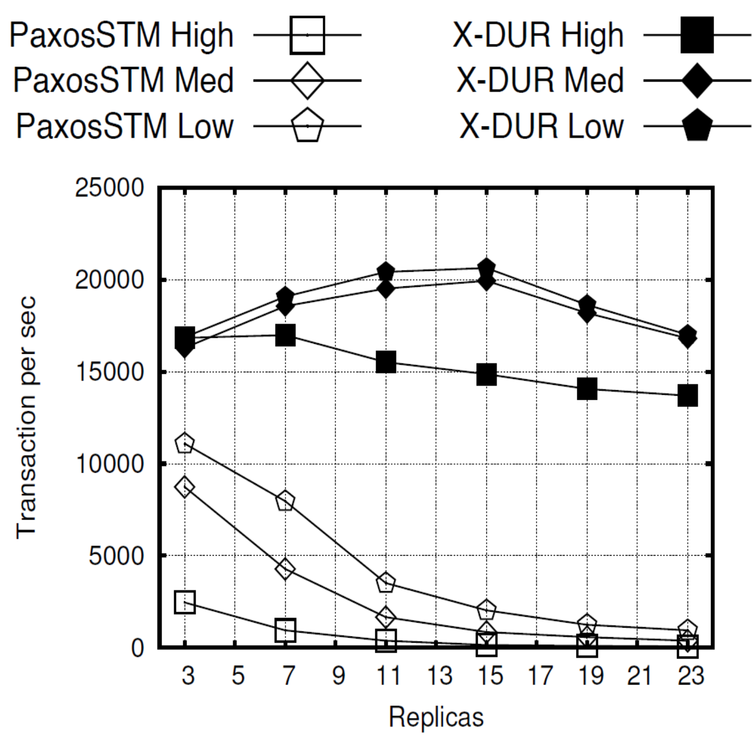
Throughput for varying the number of nodes



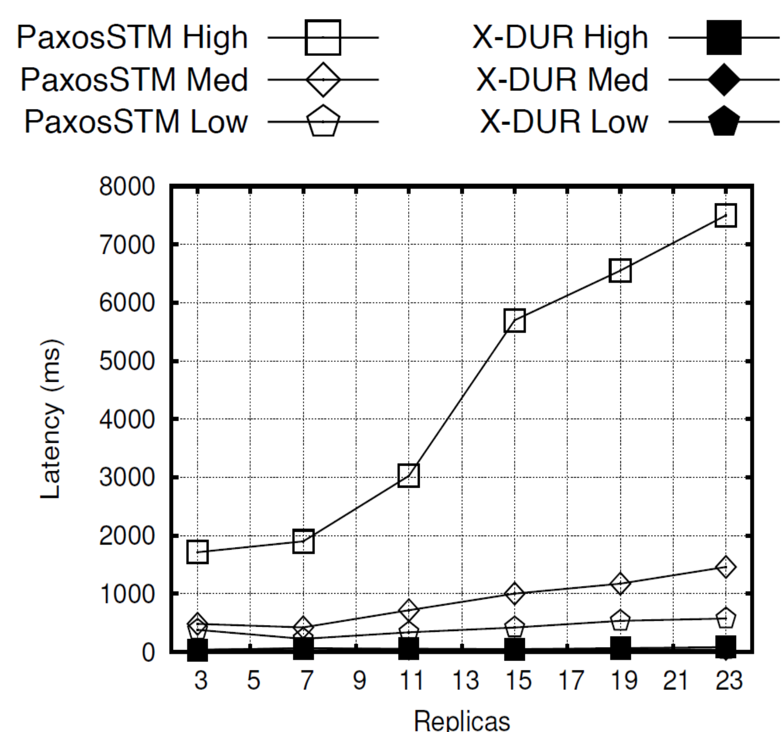
Throughput for 7 nodes with varying number of clients

Evaluation: TPC-C

- Contention: High, medium and low
- X-DUR outperforms PaxosSTM in all scenarios
  - Transaction length is moderately long
  - Even low conflict leads to high number of aborts for PaxosSTM



Throughput for varying the number of nodes



Latency for varying the number of nodes

Post-Prelim Contributions

- Speculative Client Execution in Deferred Update Replication
  - ACM/IFIP/USENIX 15th Middleware Workshop for Next Generation Computing (MW4NG 14)
- Regulating Consensus under the Authority of Caesar
  - To be submitted to EuroSys 16

Can ordering layer be improved further?

- All our previous works used total-order based ordering layer
- Research contributions majorly focused on transaction execution
  - Speculation
  - Concurrent processing
  - Lightweight commit
- It seems total-order is restricting further improvement
  - In DER, requests have to execute in order, irrespective of conflicts
  - In DUR, transactions commit in order, irrespective of conflicts
  - Are we loosing performance due to total-order?

Ordering layer definitions

- Leader
  - A replica that is elected by all replicas
  - Gets the right to propose the order of requests
  - Tries to convince other replicas about the proposed order
- Single-leader approaches
  - Only one elected replica gets to propose the order of requests
- Multi-leader approaches
  - Each replica in the system gets to propose the order of requests
- Communication steps
  - Number of times a leader has to send messages to finalize the order for a proposed request

Existing distributed ordering layer implementations

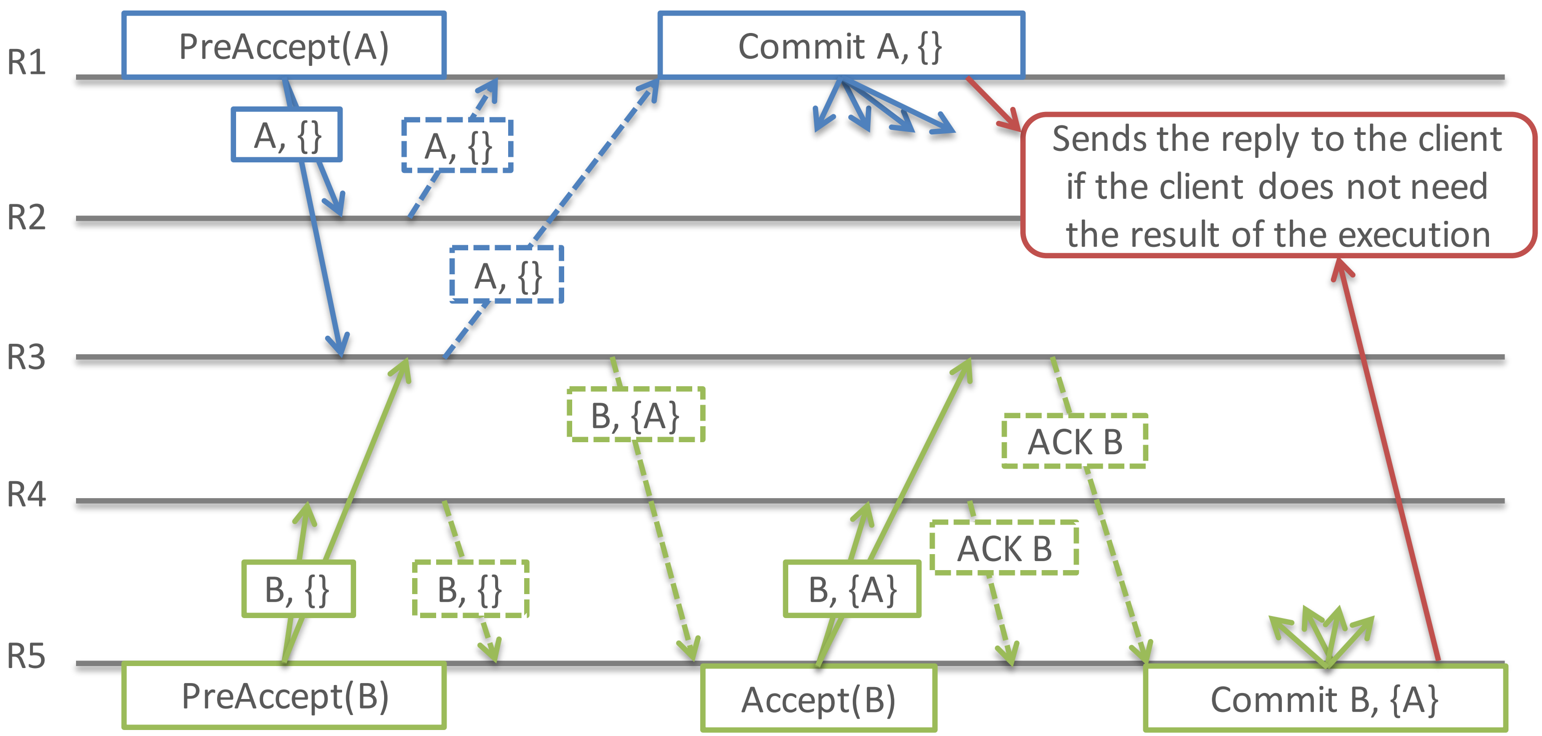
- Total-order
  - Multi-Paxos
    - An optimization over Paxos [Lamport, 98]
    - Single leader based ordering protocol
  - Mencius (baseline) [Mao, 08]
    - Multi-leader based ordering protocol
    - Response from all nodes required to make progress
    - Performance is defined by the slowest replica in the system
- Partial-order
  - Generalized Paxos [Lamport, 05]
    - Multi-participant partial-order protocol with single conflict resolver
  - EPaxos [Moraru, 13]
    - Multi-leader based partial-order protocol
    - Local conflict resolution using graph analysis

State-of-the-art solution: EPaxos

- Multi-leader approach: Each replica is leader for its proposals
- Distributes load evenly among all replicas
- Exploits fast replicas
- Decouples request dependency finalization and deterministic order
  - Network layer finalizes dependencies for each request
    - The set of committed requests and their dependencies form a directed dependency graph
  - Local execution layer defines order among conflicting requests
    - Deterministic order using directed graph analysis at the time of execution of a command

EPaxos: Protocol Details

- Request finalization process:



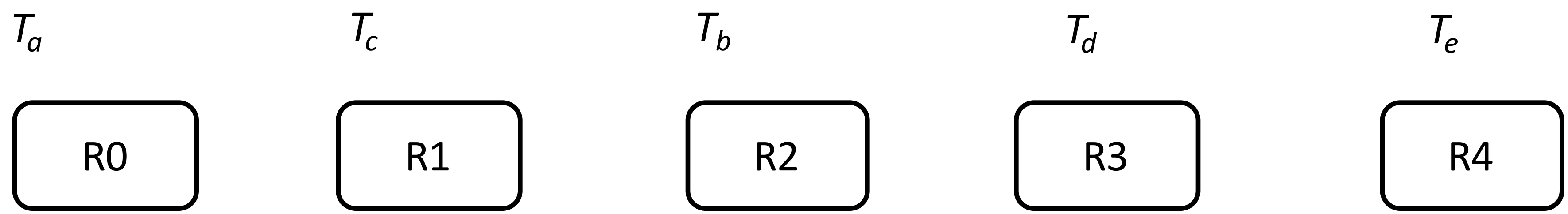
State-of-the-art solution: EPaxos

- What could go wrong?
  - If a client waits for the result of an execution then the expensive cost of the graph analysis appears in the client-perceived latency

Can we do better?

- Wish list
  - Multi-leader approach
    - All replicas help each other to improve ordering layer performance
  - Use of quorum to decide the order
    - Exploit fastest replicas
  - Finalize the request order in minimum possible communication delays
    - Effort to reduce the expensive network communication steps
  - Partial-order
    - Order is defined only among conflicting requests
  - Highly concurrent execution of transactions
    - Exploit the partial order to achieve higher concurrency for request execution
  - Use loosely synchronized clocks to timestamp requests
    - Exploit natural advancement of physical clocks
    - Ensure monotonically increasing clock

Caesar



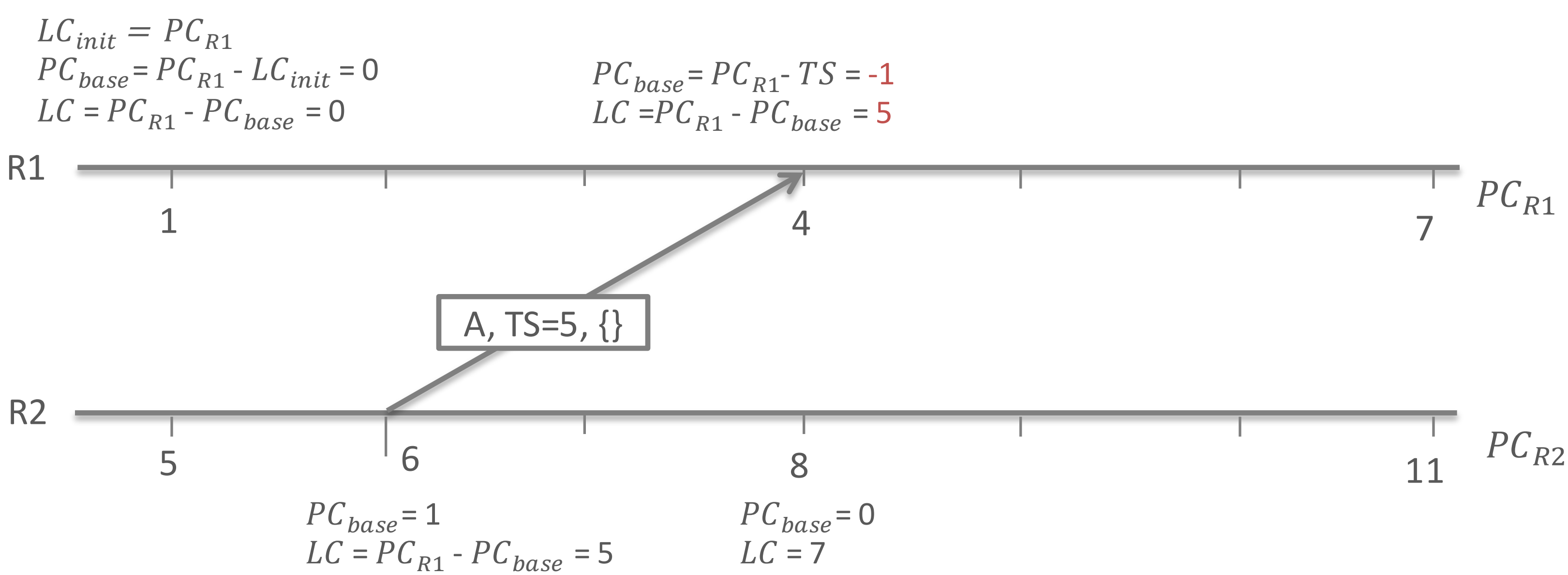
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	X		X		X	X	X												

Burnt slot: txs that conflict with T_b cannot be delivered in 1

- T_b does not depend on T_c
- T_d depends on T_e

Caesar

- No predefined slots for requests originating from a replica
  - Caesar uses naturally advancing physical clocks to timestamp requests
- No external clock synchronization required
  - Caesar forwards local clock in case timestamp received from other replica is in future



Handling Pre-Accept messages

Receive **Pre-Accept/ T_b, 2** from R2

0	1	2	3	4
	T_a	T_b		

Reply **T_b,2 →{T_a}|Ack**

T_a and T_b conflict

Handling Accept/Stable messages

Receive **Accept / Commit** $T_b, 2 \rightarrow \{T_a\}$ from R2

0	1	2	3	4
X	T_a	T_b ACCEPTED $\{T_a\}$		

Reply **ACK**

Don’t miss dependencies: Wait Condition 1

Receive **Pre-Accept / T_c, 0** from R0

0	1	2	3	4
T_c	T_a	T_b ACCEPTED $\{T_a, T_c\}$		

T_b and T_c conflict. T_b may burn slot 0.
Wait for T_b acceptance/stabilization

Reply **T_c,0 → {} | Ack**

Aborting a message delivery: : Wait Condition 1

Receive **Pre-Accept / $T_{c,}$ 0** from R0

0	1	2	3	4
X T_c	T_a	T_b ACCEPTED $\{T_a\}$		

T_b and T_c conflict. T_b may burn slot 0.
Wait for T_b acceptance/stabilization

Reply **T_c ,3 → $\{T_b\}$ | NACK**

Suggest a retry at slot 3 for T_c

Bound the delivery aborts: Wait Condition 2

Receive **Pre-Accept / T_{d}, 7** from R2

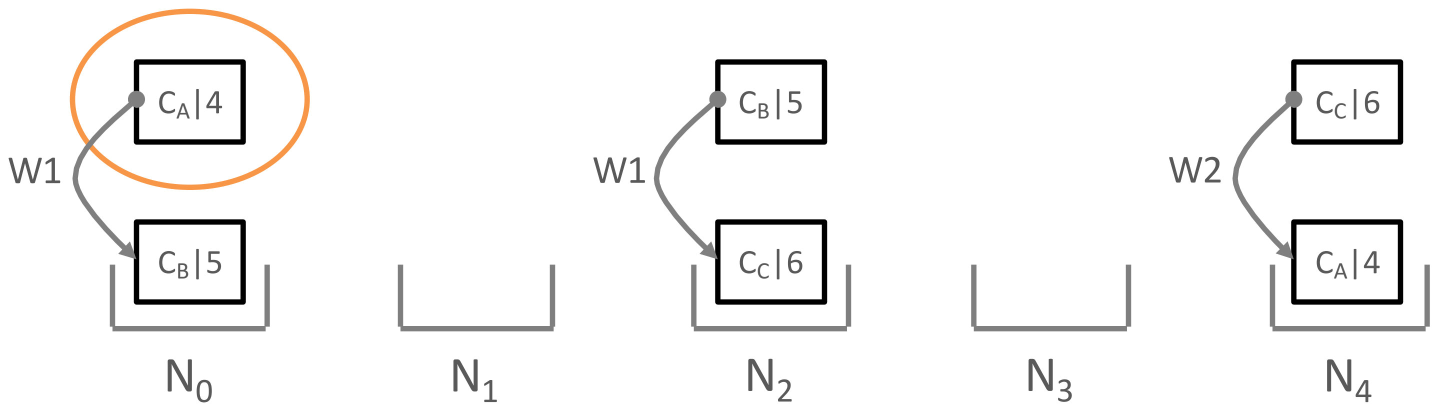
There is a burnt, conflicting and non-empty slot. T_d waits for T_c annihilation

Receive **Accept / T_{c}, 5** from R0

0	1	2	3	4	5	6	7
X T_c	T_a	T_b COMMIT $\{T_a\}$			T_c		T_d

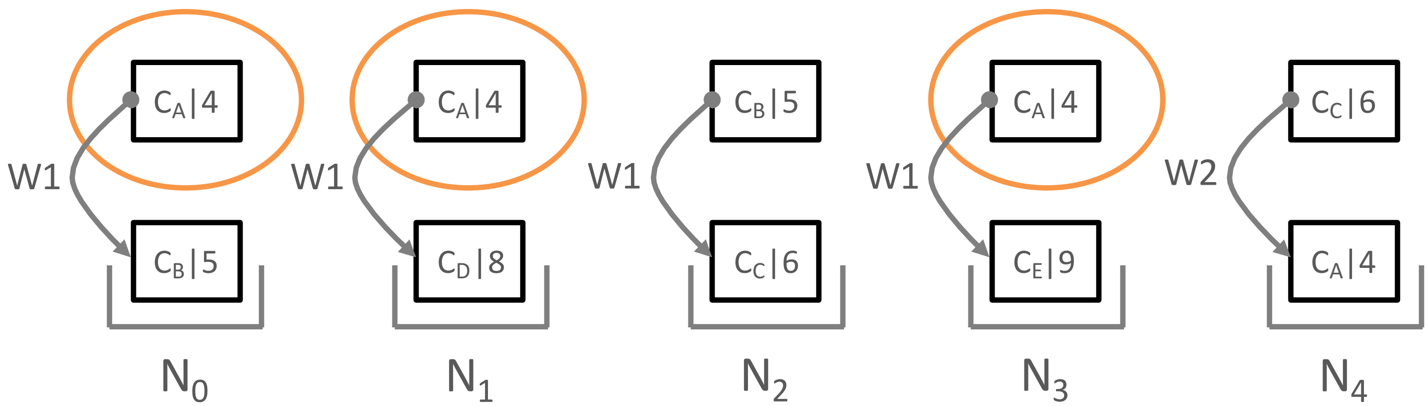
But did we get it right?

- There is a potential deadlock situation



But did we get it right?

- There is a potential deadlock situation



How can we remove deadlocks?

- Reason of deadlocks
  - Both waiting conditions W1 and W2 conflict
  - Waiting condition W1 ensures performance
  - Waiting condition W2 ensures correctness
- Can we get rid of W2?
  - Exchange dependencies in response to Accept message

Avoiding wait condition W2: 1

Receive **Pre-Accept / T_{d}, 7** from R2

There is a burnt, conflicting and non-empty slot. T_d waits for T_c annihilation

Receive **Accept / T_{c}, 5** from R0

0	1	2	3	4	5	6	7
X T_c	T_a	T_b COMMIT $\{T_a\}$			T_c		T_d

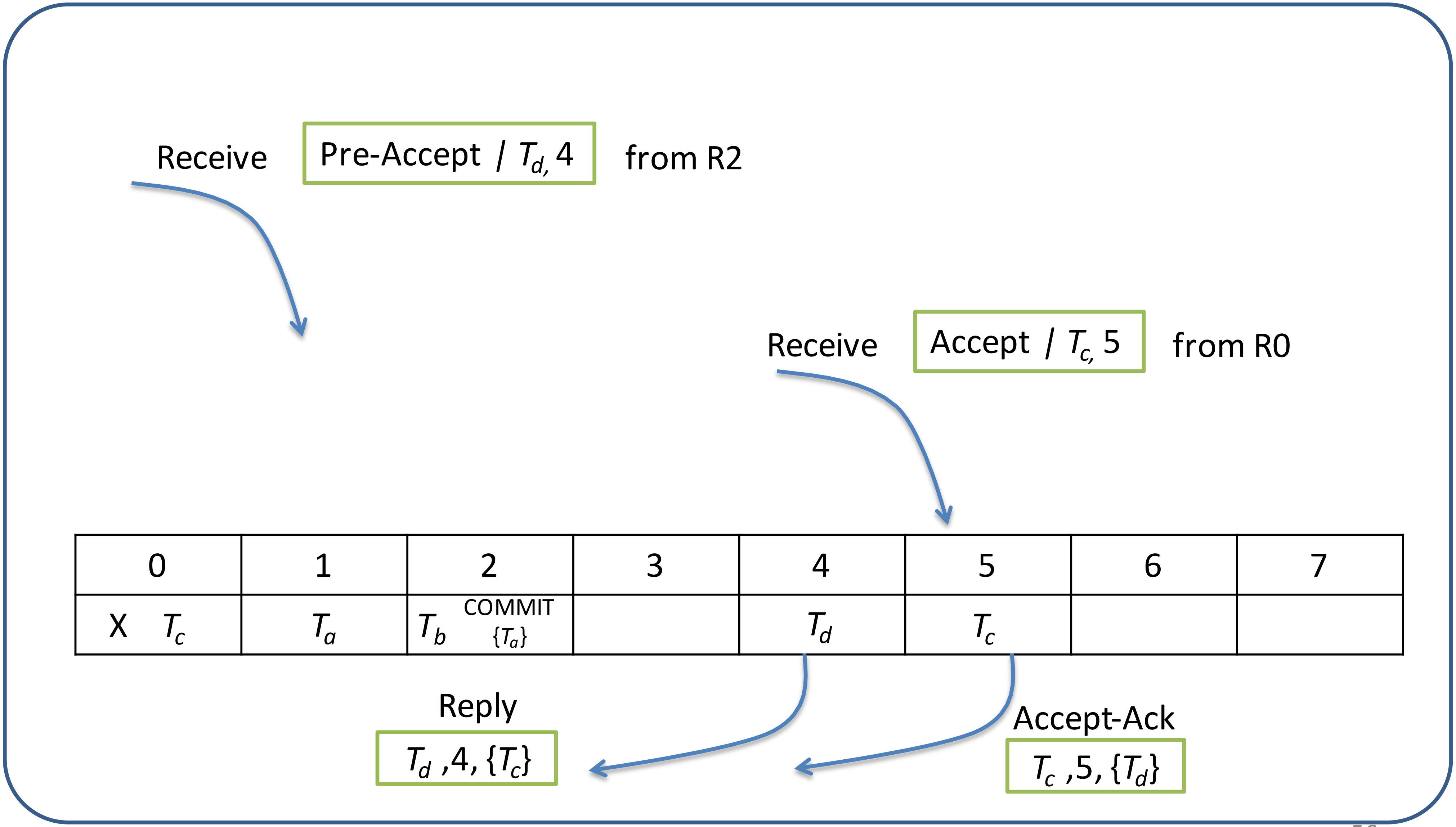
Accept-Ack

T_c ,5, {}

Reply

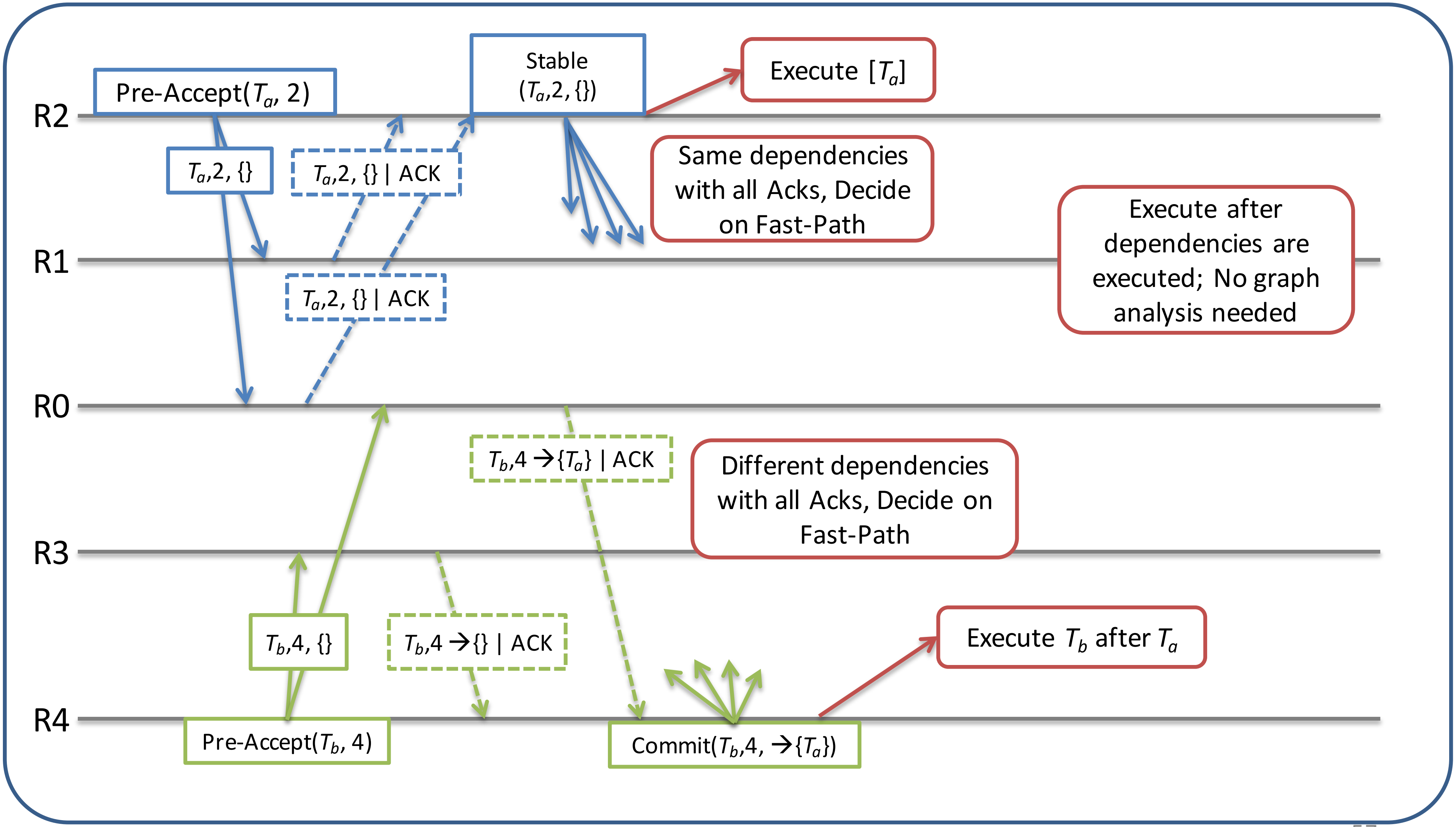
T_d ,7, $\{T_c\}$

Avoiding wait condition W2: 2



0	1	2	3	4	5	6	7
X T_c	T_a	T_b COMMIT {T_a}		T_d	T_c		

Caesar at work

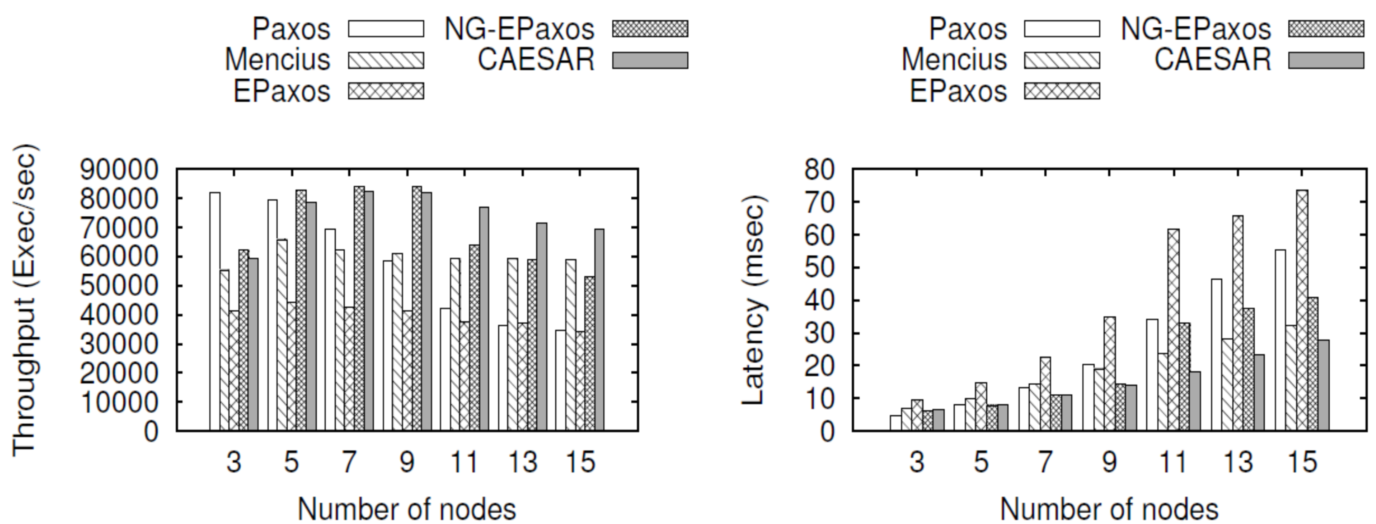


Caesar: Evaluation

- Testbed – PRObE cluster (15 nodes)
  - AMD Opteron 6272, 64-core, 2.1 GHz CPU
  - 128 GB RAM and 40 Gbps ethernet
- Benchmarks
  - Key-Value: A micro-benchmark that does single object read/write operations
  - TPC-C: A popular OLTP benchmark
  - Vacation: Distributed version of vacation application in STAMP [Minh, 08]
    - Mimics the operations of reserving flight, car etc. for vacation
- Competitors
  - Multi-Paxos : Total order, post final delivery serial execution
  - Mencius: Multi-leader total order, post final delivery serial execution
  - EPaxos: Multi-leader partial order, post final delivery parallel processing

Evaluation: Key-Value

- Partitioned access: 0-conflicts
- EPaxos suffers from high cost of graph processing
  - Performace of NG-Epaxos i.e., EPaxos without graph processing, confirms high cost of graph processing
- Mencius suffers from serial execution and need to hear from all replicas
- Paxos shows single-leader bottelneck

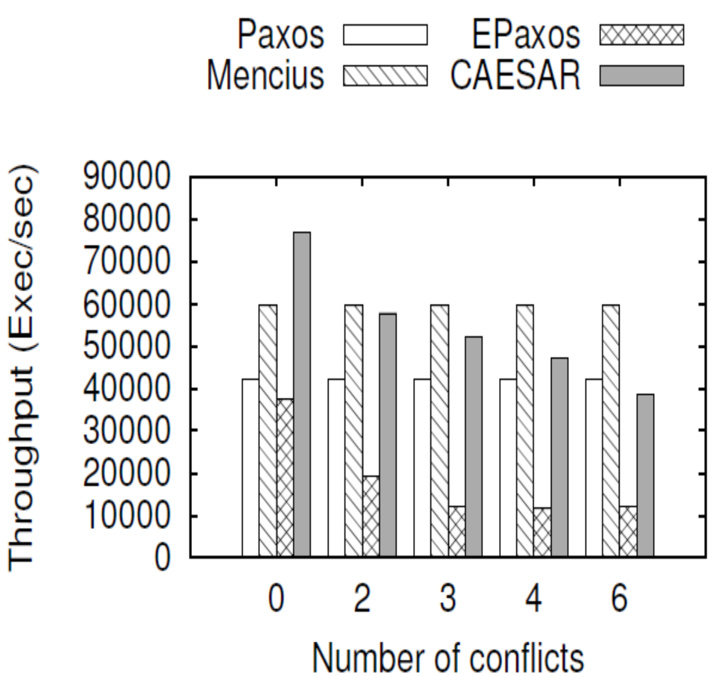


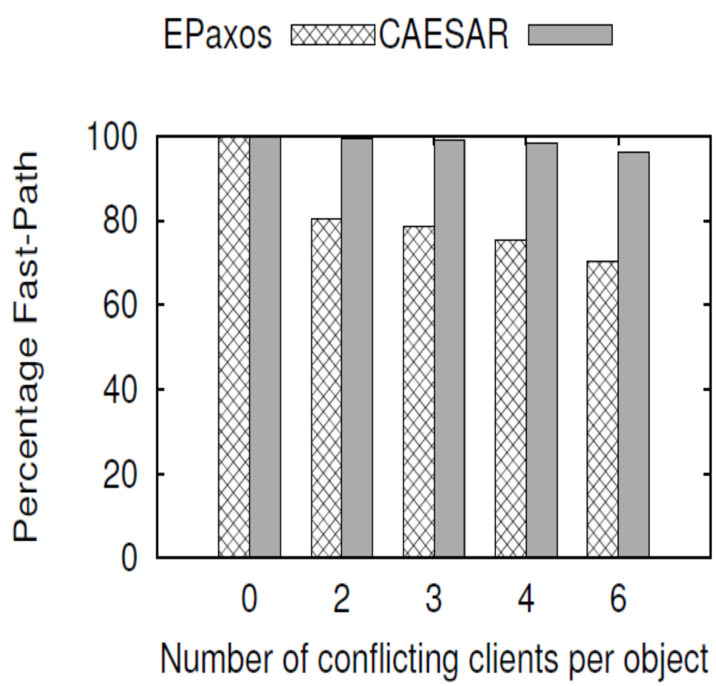
(a) Throughput. (b) Latency.

Ordering layer performance with varying the number of nodes

Evaluation: Key-Value

- Performance under varying conflicts
- EPaxos suffers from high cost of graph processing with increasing conflicts
- Increasing conflicts also impact EPaxos's probability of fast-paths

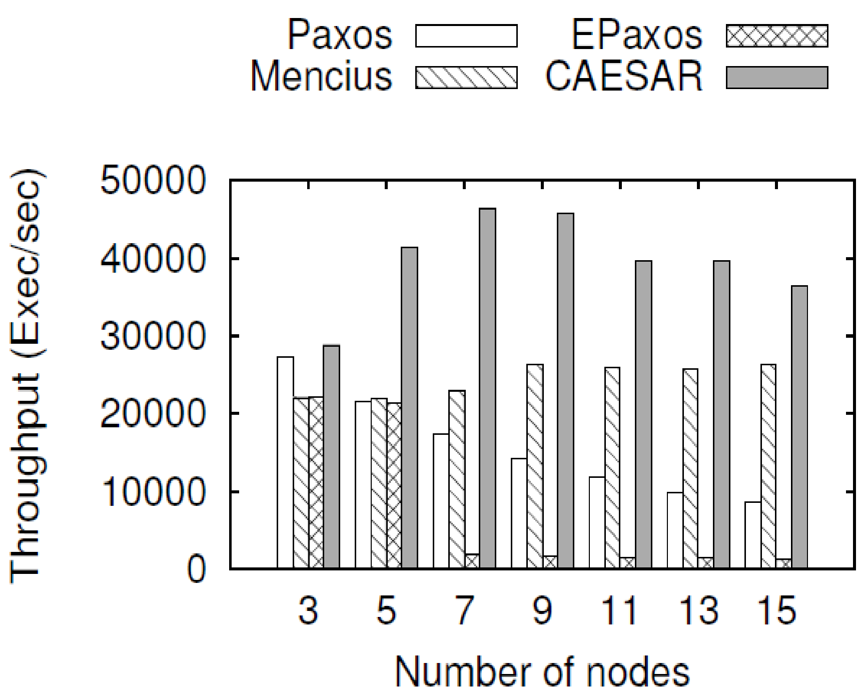




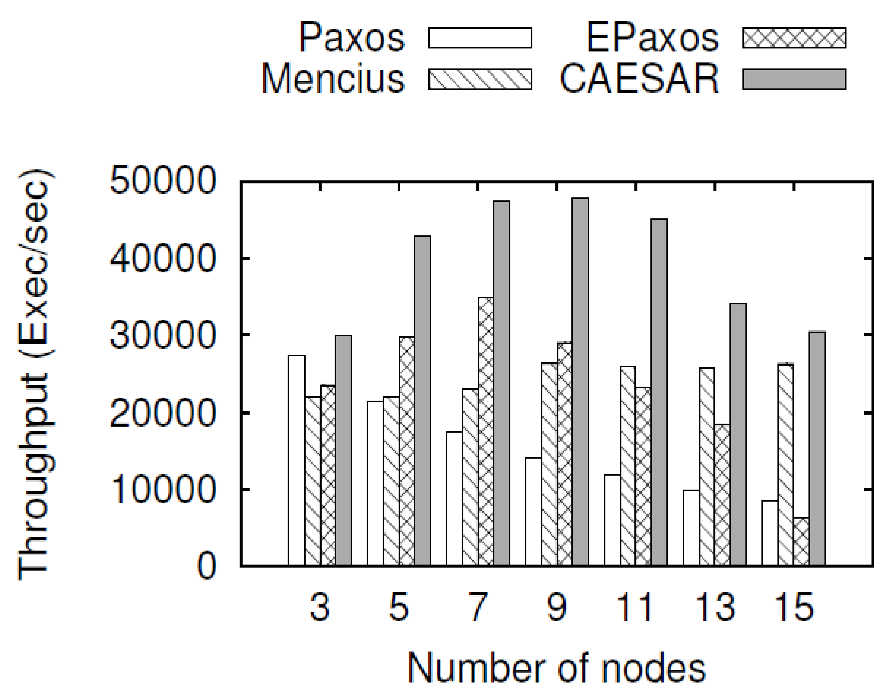
Ordering layer performance for 11 nodes and varying number of conflicting clients per object

Evaluation: TPC-C

- Contention: high (200 warehouses) and low (1000 warehouses)
- Cost of transaction processing impacts serial execution in Paxos and Mencius
- Epaxos exploits concurrency in low conflict scenarios
- Caesar outperforms all of the competitors



TPC-C transaction throughput for varying number of nodes under high conflicts(200 WH)



TPC-C transaction throughput for varying number of nodes under low conflicts(1000 WH)

Conclusion

- Contributions are modular in design
  - Different contributions can be mix-matched to solve another set of problems in distributed transaction processing
- Speculation pays off
  - DER and DUR both can benefit
- Ordering layer optimizations help execution layer too
  - Optimistic order helps speculation; partial order helps concurrent processing

Thank You! Questions?

List of Contributions

- HiperTM: High Performance, Fault-Tolerant Transactional Memory
  - **ICDCN 14**
- Extended version of HiperTM: High Performance, Fault-Tolerant Transactional Memory
  - Submitted to **TCS**
- SMASH: speculative state machine replication in transactional systems
  - **Middleware 13**
- Archie: A Speculative Replicated Transactional System
  - **Middleware 14**
- Speculative Client Execution in Deferred Update Replication
  - **MW4NG 14**
- Regulating Consensus under the Authority of Caesar
  - To be submitted to **EuroSys 16**
- Scaling Up Active Replication using Staleness
  - Submitted to **TPDS**

- Automated Data Partitioning for Highly Scalable and Strongly Consistent Transactions
  - **TPDS 15**
- On Transactional Memory Concurrency Control in Distributed Real-time Programs
  - **Cluster 13**

Thank You!!

Image sources: rikbasra.com