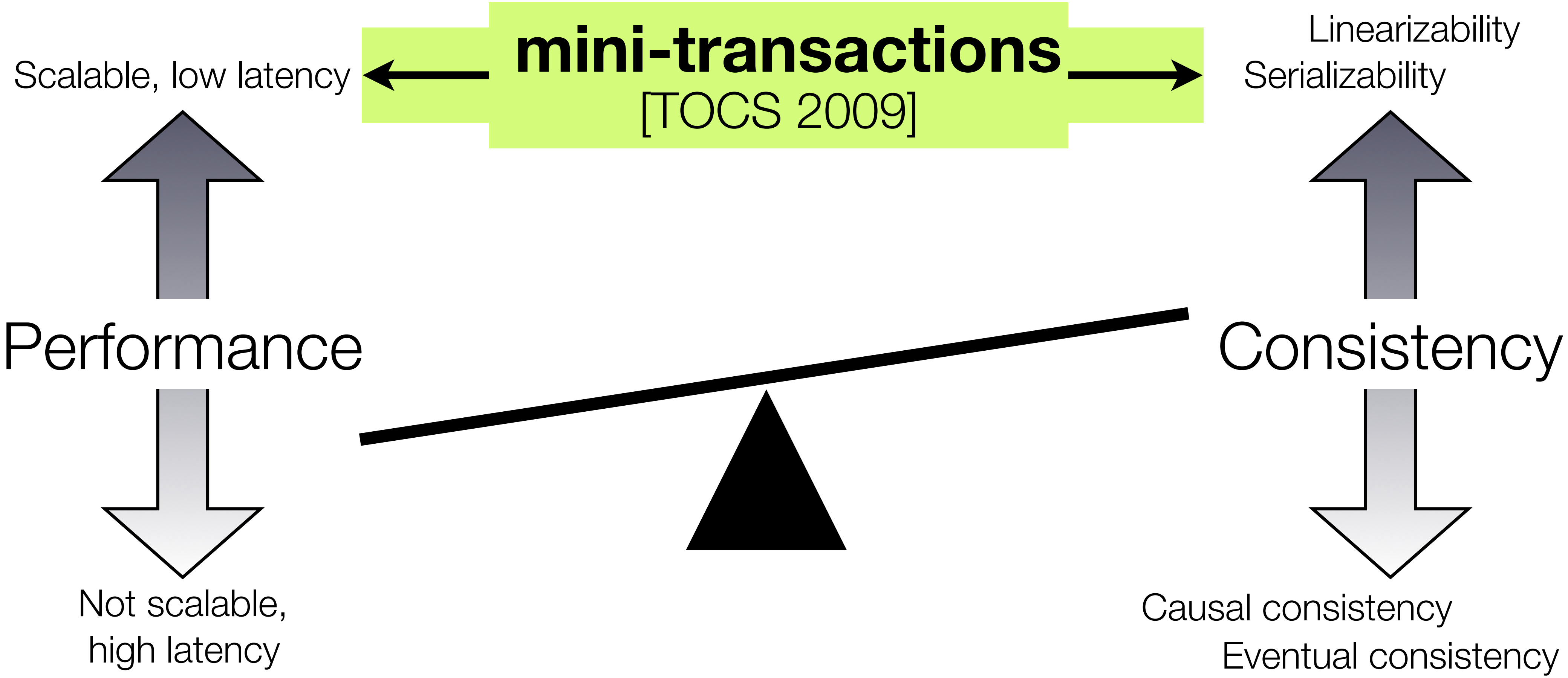
Callinicos: Robust Transactional Storage for Distributed Data Structures

Ricardo Padilha, Enrique Fynn, Robert Soulé and Fernando Pedone

University of Lugano (USI)

Switzerland

The storage tradeoff



mini-transactions
[TOCS 2009]

Scalable, low latency

Performance

Not scalable,
high latency

Linearizability
Serializability

Consistency

Causal consistency
Eventual consistency

Mini-transactions are elegant

- Simple transaction language
  - Three operations: cmp, read, write
  - Any number in transaction
- Simple execution model
  - If all compares successful, execute reads and writes
- Optimized implementation
  - Divide transaction in sub-transactions, one per partition
  - Two-phase commit among sub-transactions

Mini-TX Problem 1: Too Restrictive

- Example: swapping two entries

Transaction to swap entries A and B:

```
swap_tx (A, B) {
        x = read(A);
        y = read(B);
        write(A, y);
        write(B, x);
}
```

Swap with minitransactions:

```
swap_tx1(A, B) {
        x = read(A);
        y = read(B);
} // return x, y

swap_tx2(A, B, x, y) {
        cmp(A, x);
        cmp(B, y);
        write(A, y);
        write(B, x)
}
```

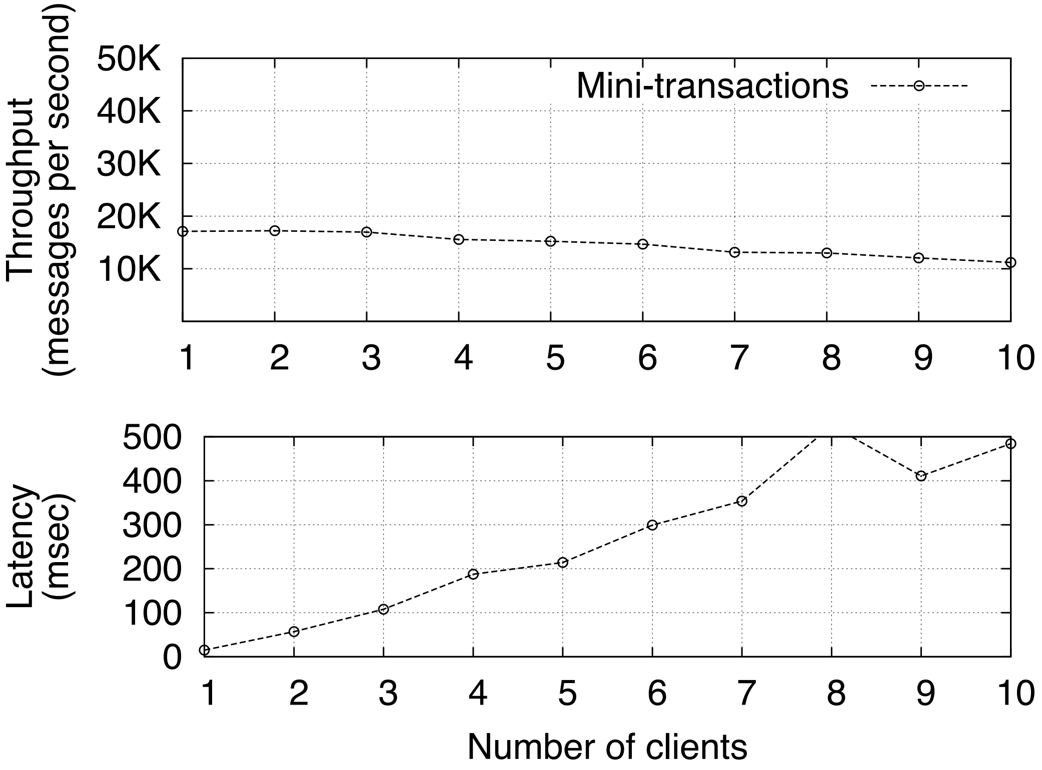
if abort then retry

Problem 1: No notion of variables

Problem 2: What if A and B are in different partitions?

Mini-TX Problem 2: High abort rate

- Optimistic concurrency control
  - ✦ Due to “short locks” during transaction execution
  - ✦ Due to restrictive data flow



↓ **Throughput**

↑ **Latency**

Mini-TX Problem 3: Failure Model

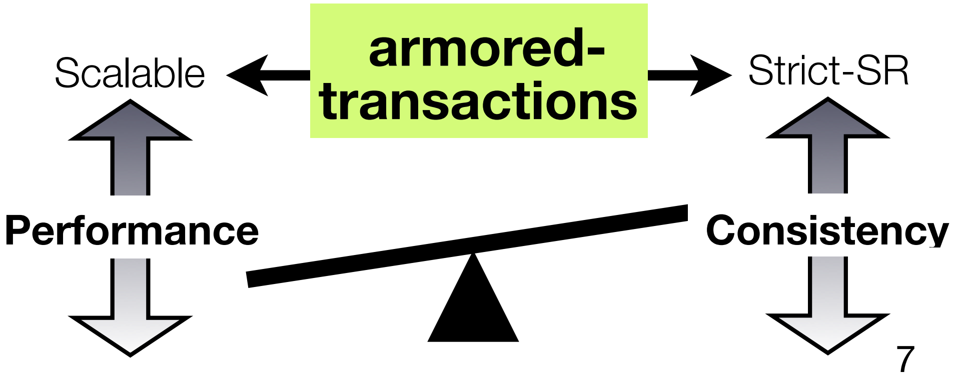
Mini-TX

Crash failures

Byzantine failures

Callinicos: armored transactions

- Address three problems of mini-transactions
  - ✦ Unrestricted operations and data flow across partitions
  - ✦ Contention management that avoids aborts
  - ✦ Byzantine fault tolerance
- Without giving up
  - ✦ Strong consistency (strict serializability)
  - ✦ Scalable performance



armored-transactions

Scalable

Strict-SR

Performance

Consistency

System design

Solving Mini-TX Problem 1:
Richer language for unrestricted operations

- Example: Swapping two entries in Callinicos

Transaction to swap entries:

```
swap_tx (A, B) {
    x = read(A);
    y = read(B);
    write(A, y);
    write(B, x);
}
```

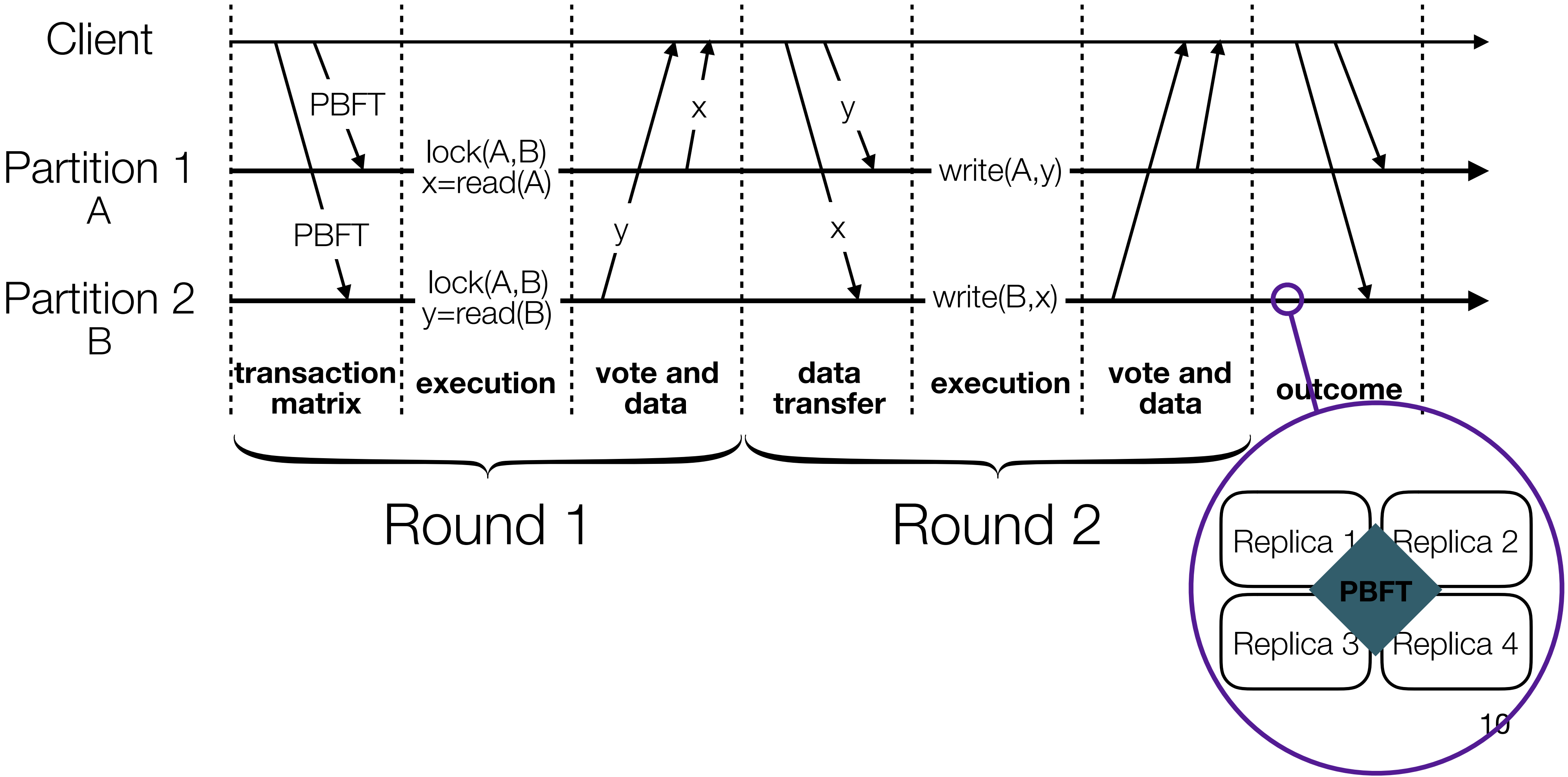
Transaction matrix for swap:

	Partition 1	Partition 2
Round 1	x = read(A) export(x)	y = read(B) export(y)
Round 2	import(y) write(A,y)	import(x) write(B,x)

A is in Partition 1
B is in Partition 2

Solving Mini-TX Problem 1:
Multi-round transactions for complex data flow

- Swap execution with data exchange

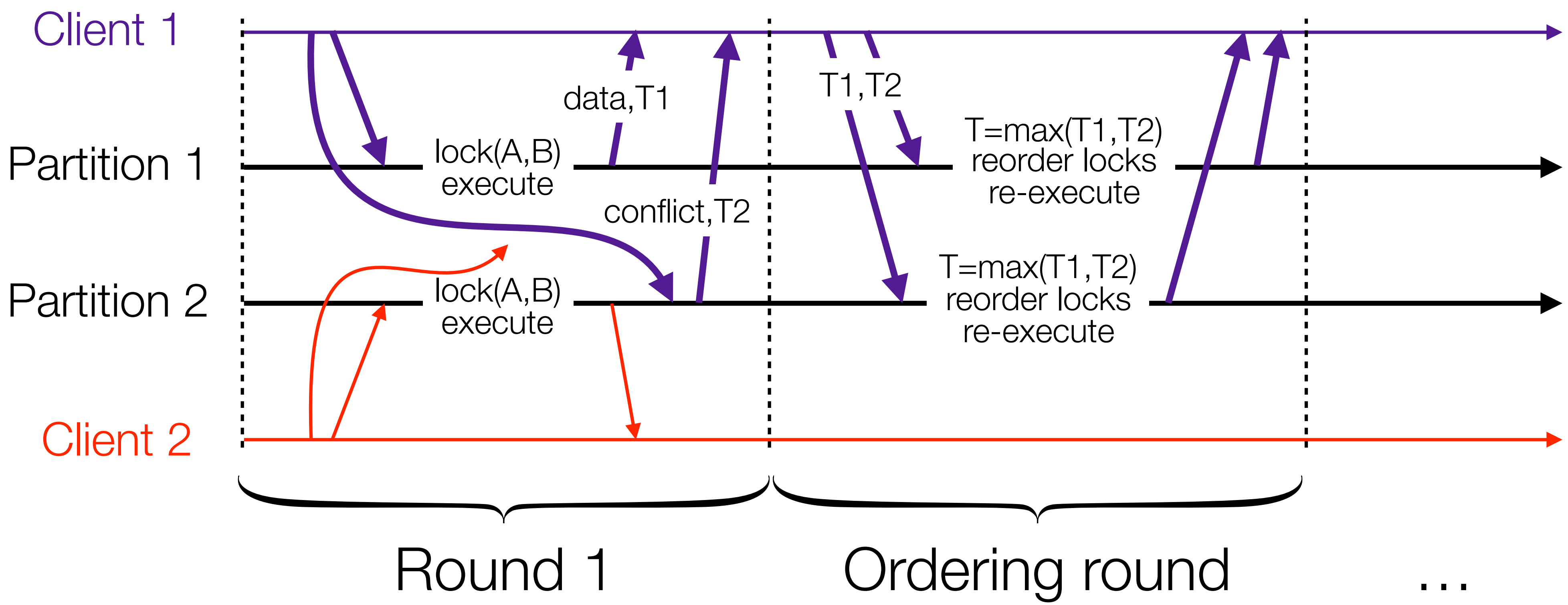


Solving Mini-TX Problem 1:
Multi-round transactions for complex data flow

- Round 1
  - Client multicasts transaction matrix to each partition (PBFT)
  - Replicas acquire locks, execute, return signed vote and data
  - Client collects signed votes and data
- Round i > 1
  - Client sends signed data collected in round i-1 to partitions
  - Replicas receive data, execute, return signed vote and data
  - Client collects signed vote and data

Solving Mini-TX Problem 2:
Order transactions instead of aborting them

- Two concurrent swap transactions



Solving Mini-TX Problem 2:
Order transactions instead of aborting them

- Round 1(extension)
  - Server assigns timestamp to transaction; locks are ordered
  - Server returns timestamp; upon conflict, notifies client
  - Client collects signed timestamps, votes and data
- Ordering round
  - Client sends signed timestamps and data to partitions
  - Replicas compute final timestamp, possibly [re-acquire locks and re-execute], and return signed vote and data
  - Client collects signed vote and data

Solving Mini-TX Problem 3: Coping with Byzantine clients and servers

- Byzantine servers [PBFT, TOCS 2002]
    - Within a partition: state machine replication
    - Each partition needs 3f+1 replicas, f : byzantine servers
- Byzantine clients [Augustus, Eurosys 2013]
    - Safety is not violated (strict serializability)
    - Liveness guarantees under attack
        - Unfinished transactions (fixed by correct clients), among others
        - Does not provide absolute liveness guarantees (possible?)

Implementation and evaluation

Implementation and evaluation

- Prototype implemented in Java 7
- PBFT and transaction processing engine
- Benchmarks
    - Kassia: distributed message queue
        - Natural point of contention (queue tail)
    - Buzzer: distributed graph store
        - Configurable contention, dependent on graph structure

Kassia: distributed message queue

- Producers
    - Add messages to the queue
    - Contend with producers and consumers
- Consumers
    - Scan the queue, without removing elements
    - Contend with producers

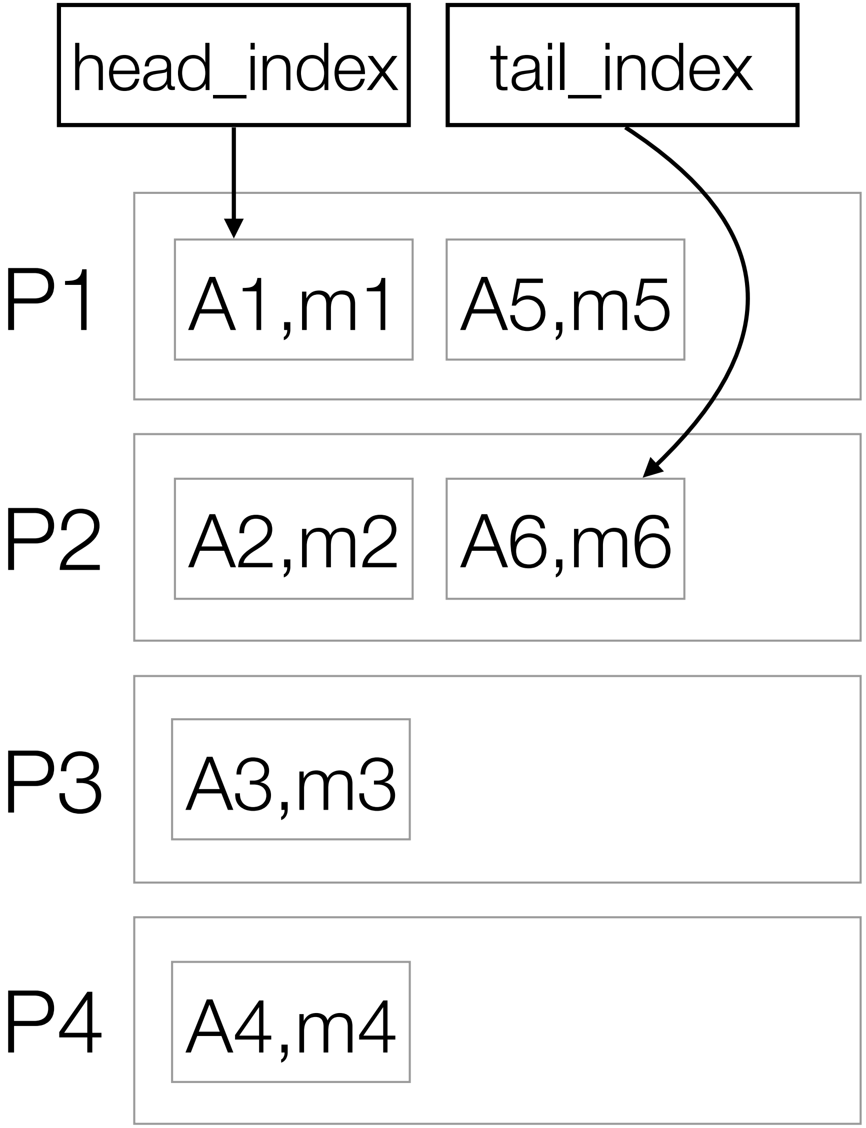
Kassia: distributed message queue

Producer transaction:

```
producer(new_msg) {
        x = read(tail_index);
        x = x + 1;
        write(x, new_msg);
        write(tail_index, x);
}
```

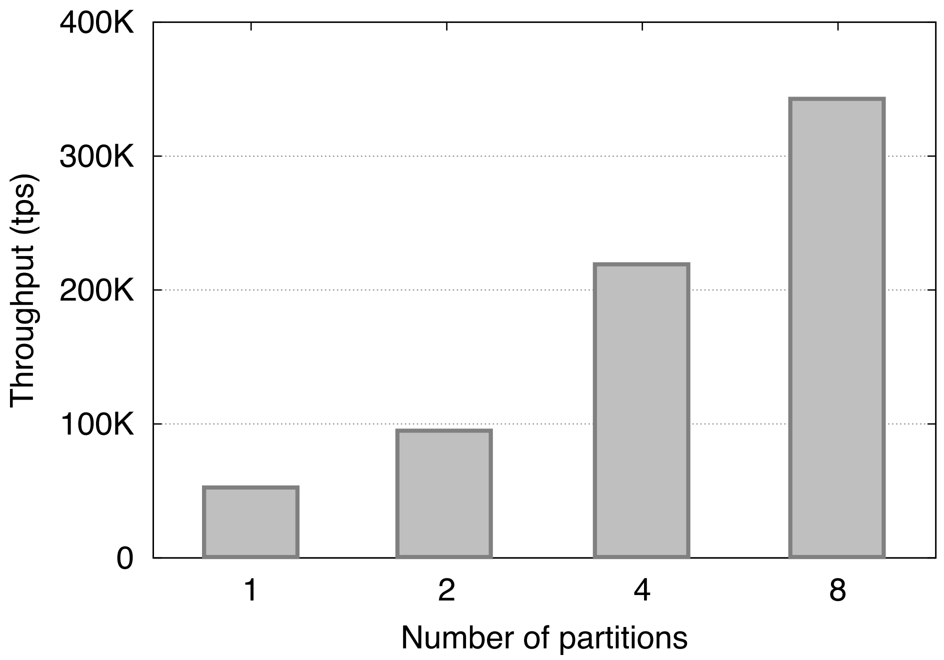
Consumer transaction:

```
consumer( ) {
        x = read(head_index);
        y = read(tail_index);
        z = range_query(x,y);
        return z;
}
```

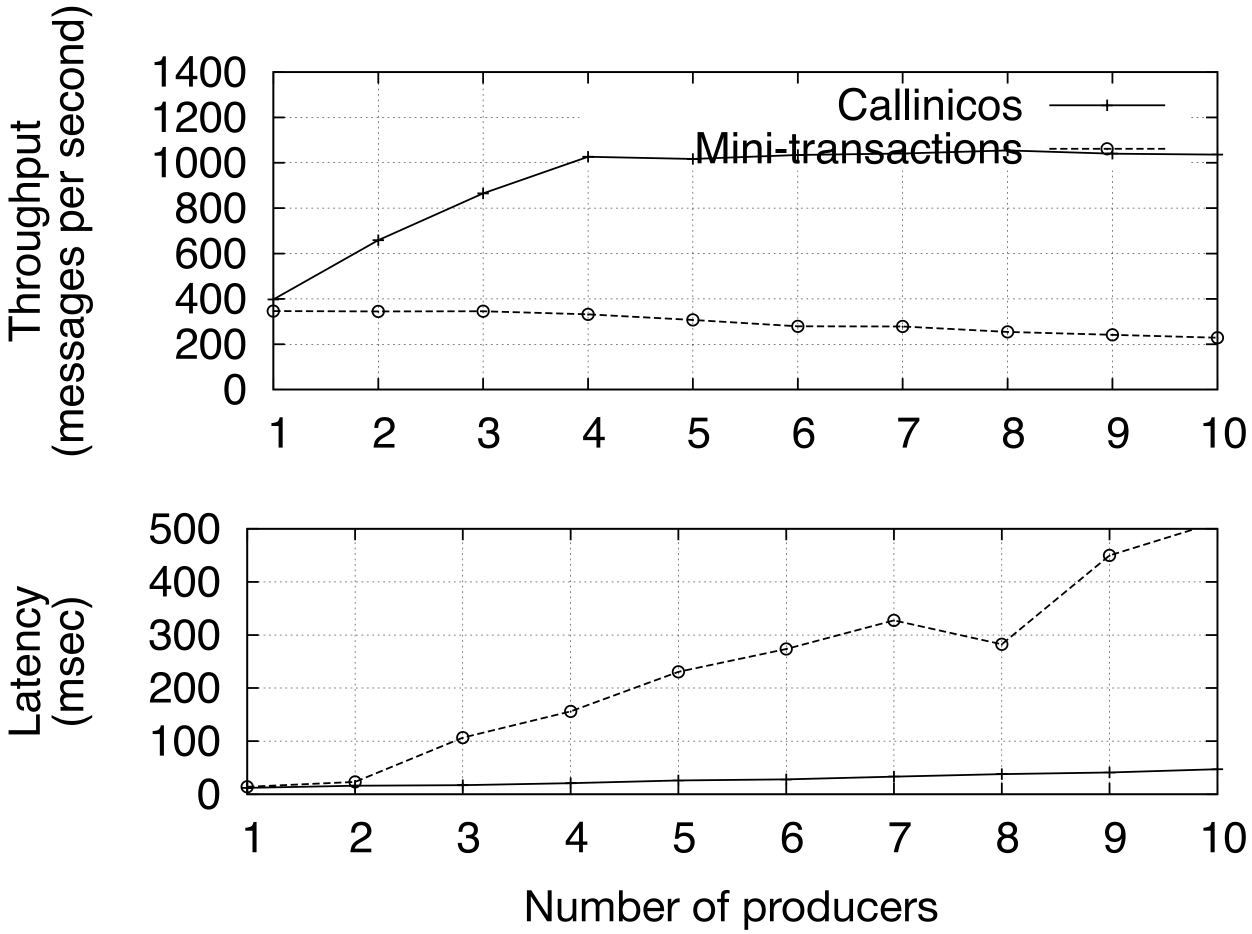


Performance highlight 1:
Callinicos scales with partitions

- One queue per partition, six producers per partition



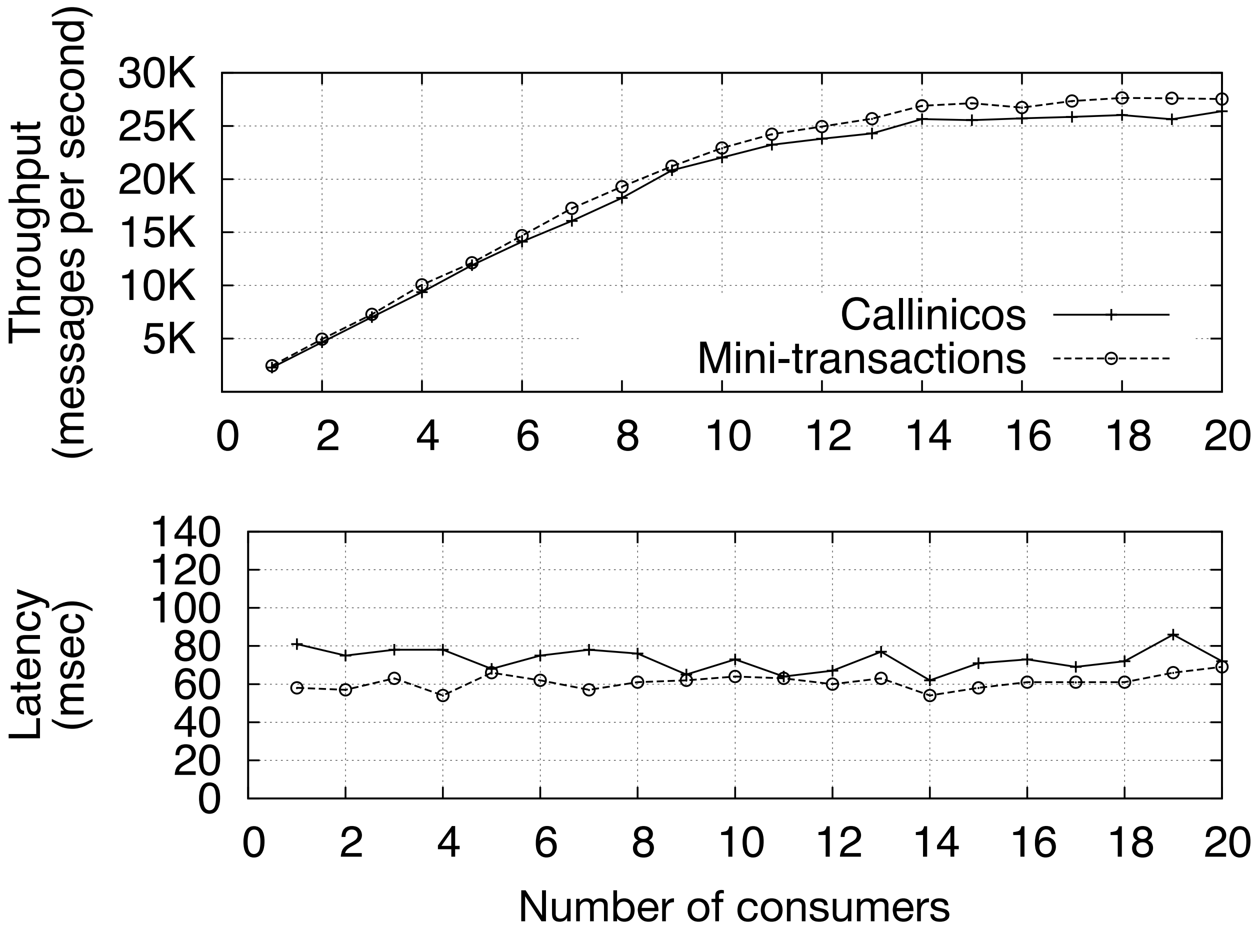
Performance highlight 2:
No performance loss with contention



10k-byte messages

Producers only: high contention

Performance highlight 3: Good performance without contention



10k-byte messages

Consumers only: no contention

Final remarks

- Callinicos extends mini-transactions
  - Unrestricted operations and data flow across partitions
  - Contention management
  - Byzantine fault tolerance
- Without giving up
  - Strong consistency (strict serializability)
  - Scalable performance

THANK YOU!

https://github.com/usi-systems/callinicos