Practical considerations of Distributed STM systems development

Maciej Kokociński, Tadeusz Kobus, Paweł T. Wojciechowski
Poznań University of Technology, Poland

Paxos STM
probably the best DSTM in the world

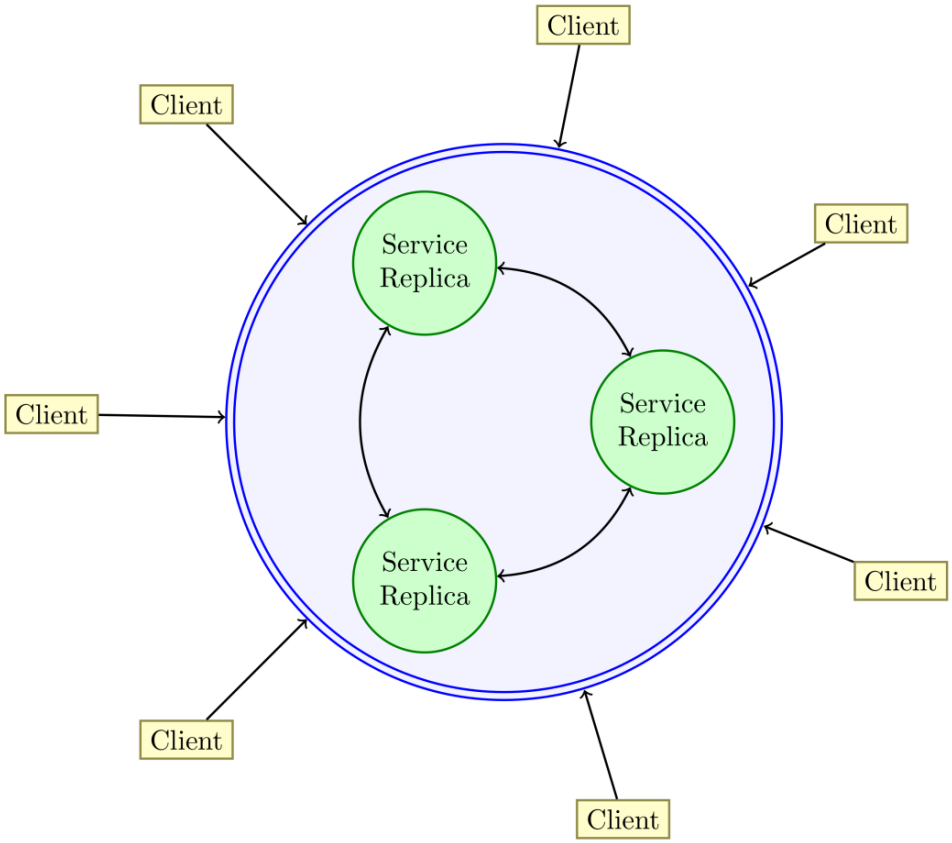
Maciej Kokociński, Tadeusz Kobus, Paweł T. Wojciechowski
Poznań University of Technology, Poland

Outline

- Motivations & model
- API
- Performance
- Evaluation

Motivations

- Service Replication
  - Fault-tolerance
  - Scalability
  - Ease of development
- Transactional Replication
- Paxos STM
  - Distributed STM
  - Platform for research
  - Preliminary work



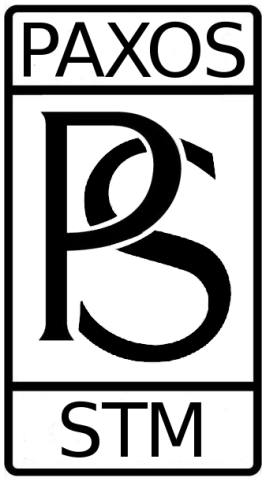
System model

- Distributed STM
  - TM on top of network
  - Support for multi-core
  - One-copy serializability
  - ACID guarantees
- Fault-tolerance
  - Crash-recovery failure model
  - $f = \left\lceil \frac{N}{2} \right\rceil - 1$

```
atomic {
   float amount = 100;
   if (accountA.balance() >= amount)
         accountA.withdraw(amount);
   else
         retry;
   accountB.deposit(amount);
}
```

Paxos STM – our system

- Distributed STM for Java
- Deferred Update
  - Objects replicated on each site
  - Optimistic concurrency control
  - Atomic Broadcast
- JPaxos as base layer
  - MultiPaxos algorithm
  - Crash-recovery failure model
  - Various recovery algorithms



API

- **No** language extensions
- **No** static code analysis

  (no pre-compilation required)
- **Automatic code instrumentation**

  (via Javassist)

API – example

```
@TransactionObject
class Account {
    private float funds;
    public float balance() { return funds; }
    public void deposit(float amount) {
        funds += amount;
    }
    public void withdraw(float amount) {
        if (amount > funds)
            throw new IllegalArgumentException();
        funds -= amount;
    }
}
```

API – behind the scene

```
@TransactionObject
class Account implements Observable, Externalizable
{
    private float funds;

    public TCID __tc_id;
    public long __tc_version;
    public volatile Account __tc_next;
    public int __tc_attr;

    public Account();
    public float balance();
    public void deposit(float f);
    public void withdraw(float f);

    public float __tc_Account_balance_internal();
    public void __tc_Account_deposit_internal(float f);
    public void __tc_Account_withdraw_internal(float f);

    public float __tc_Account_balance_noCheck();
    public void __tc_Account_deposit_noCheck(float amount);
    public void __tc_Account_withdraw_noCheck(float amount);

    public float __tc_Account_balance_wCheck();
    public void __tc_Account_deposit_wCheck(float amount);
    public void __tc_Account_withdraw_wCheck(float amount);

    public float __tc_Account_balance_sCheck();
    public void __tc_Account_deposit_sCheck(float amount);
    public void __tc_Account_withdraw_sCheck(float amount);

    public final boolean __tc_isVersion();
    public final boolean __tc_isShadowCopy();
    public final boolean __tc_isHandle();
    public final TCID __tc_getId();
    public final long __tc_getVersion();
    public final Observable __tc_getNext();
    public final void __tc_setNext(Object obj);
    public final boolean __tc_isWritten();
    public final void __tc_setWritten(boolean flag);

    public final Observable __tc_retrieve(long l);
    public final void __tc_release(long l);
    protected Account(SpecialConstructorMarker specialconstructormarker);
    public final void __tc_initHandle(TCID tcid);
    public final void __tc_initShadowCopy(TCID tcid);
    protected void __tc_copy(Account account);
    public void __tc_vcopy(Object obj);
    protected Account __tc_makeCopy();
    public Observable __tc_vmakeCopy();

    public void __tc_serializeHelper(DataOutputStream dataoutputstream) throws IOException;
    public byte[] __tc_serialize() throws IOException;
    public void __tc_deserializeHelper(DataInputStream datainputstream) throws IOException,
      ClassNotFoundException;
    public void __tc_deserialize(byte data[]) throws IOException, ClassNotFoundException;
    public void readExternal(ObjectInput objectinput) throws IOException, ClassNotFoundException;
    public void writeExternal(ObjectOutput objectoutput) throws IOException;
    public Observable __tc_addNewVersion(long l, byte data[]) throws IOException,
      ClassNotFoundException;
}
```

API – example

```
new Transaction() {
    public void atomic() {
        float amount = 100;
        if (accountA.balance() >= amount)
            accountA.withdraw(amount);
        else
            retry();
        accountB.deposit(amount);
    }
};
```

API – example

```
new Transaction() {
    public void atomic() {
        float amount = 100;
        if (accountA.balance() >= amount)
            accountA.withdraw(amount);
        else
            retry();
        accountB.deposit(amount);
    }
};
```

No need to manually specify RS and WS

Goodies

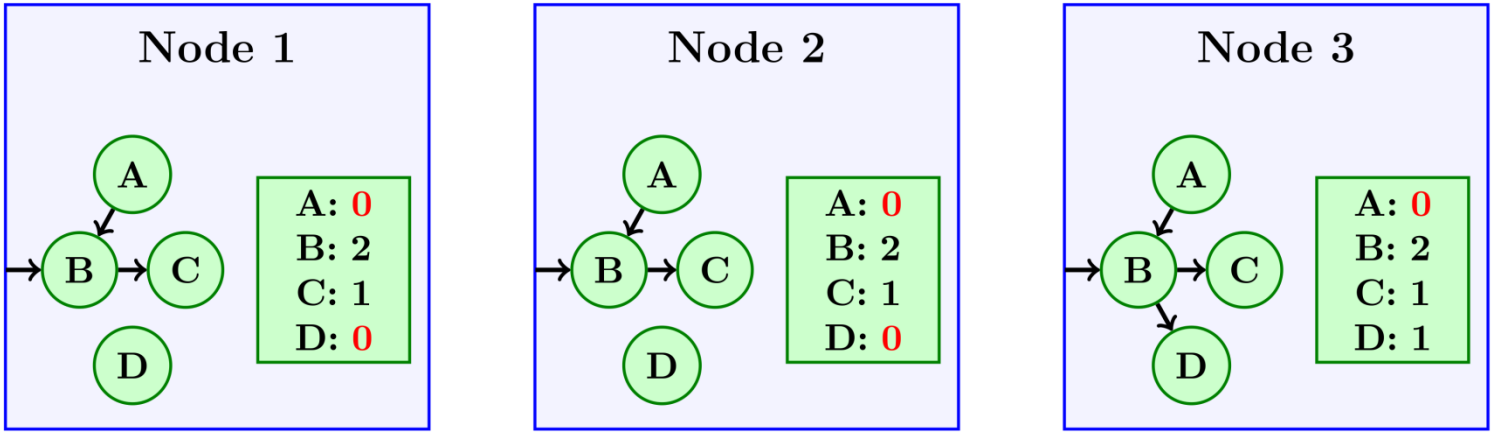
- Shared Object Registry

```
// node 1
Car car = new Car(„Rolls Royce”, „Phantom”);
PaxosSTM.addToSharedObjectRegistry(„Phantom”, car);
...
// node 2
Car car = (Car) PaxosSTM.getFromSharedObjectRegistry(„Phantom”);
```

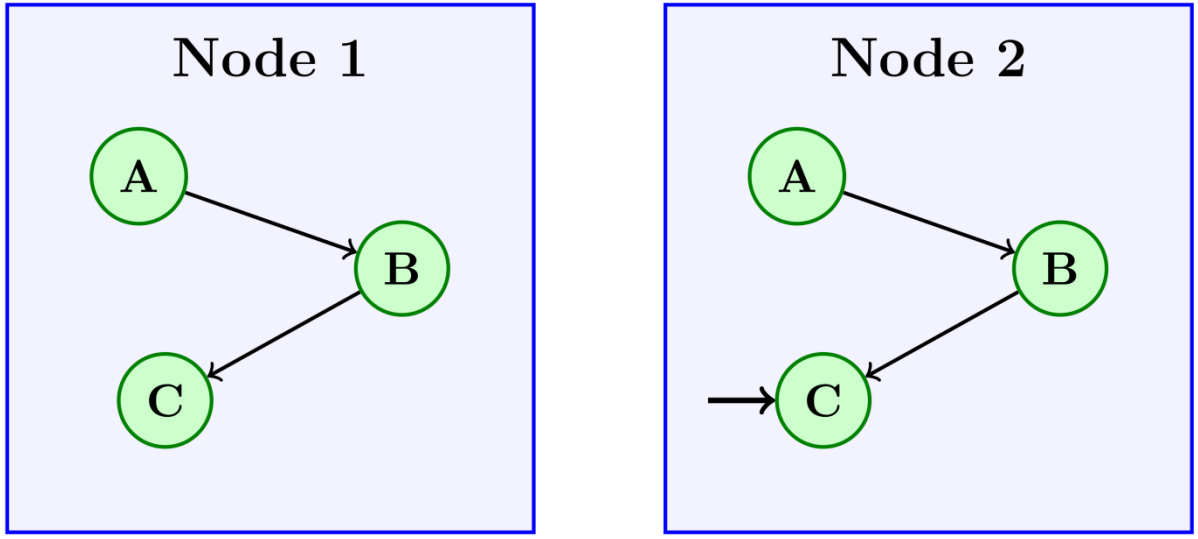
- Distributed Garbage Collector (next slide)

Distributed Garbage Collection

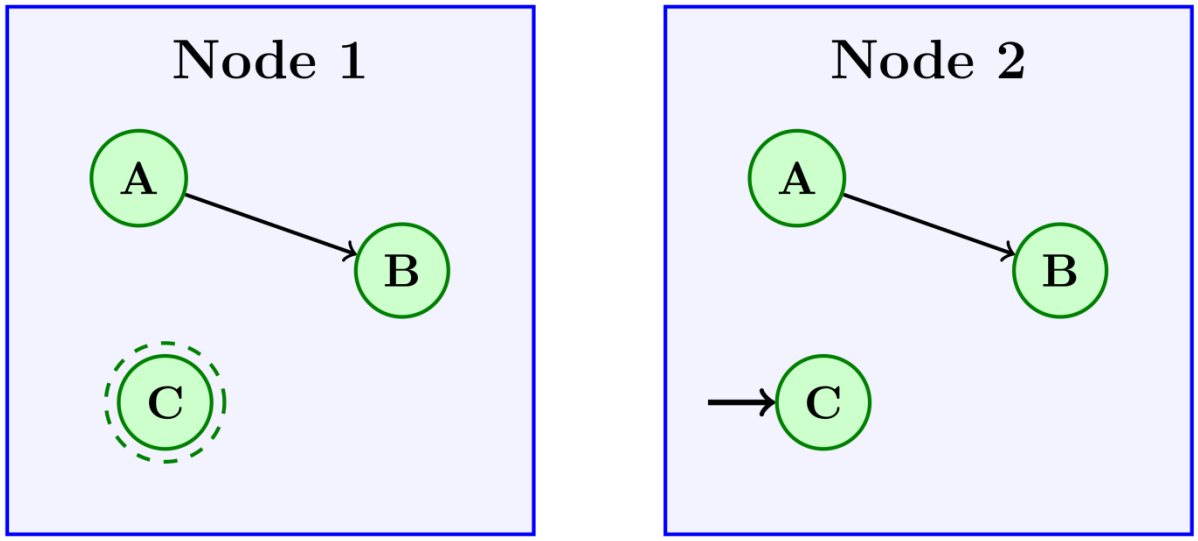
- Natural solution in Java
- Non-trivial solutions in distributed system
  - Counting references on different nodes
  - System-wide agreement on object removal



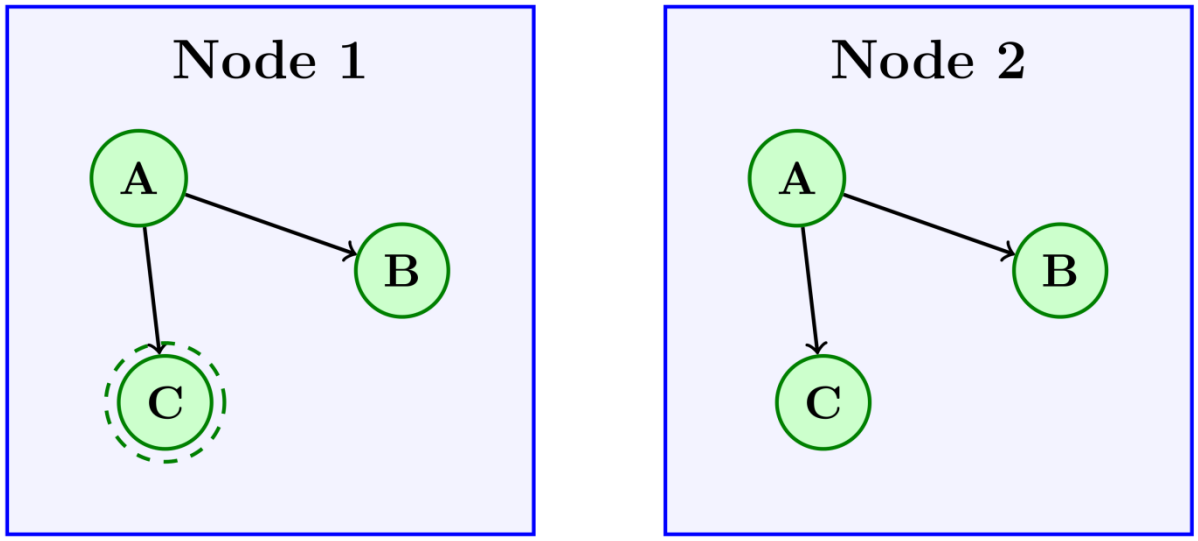
Distributed Garbage Collection – why Java GC is not enough?



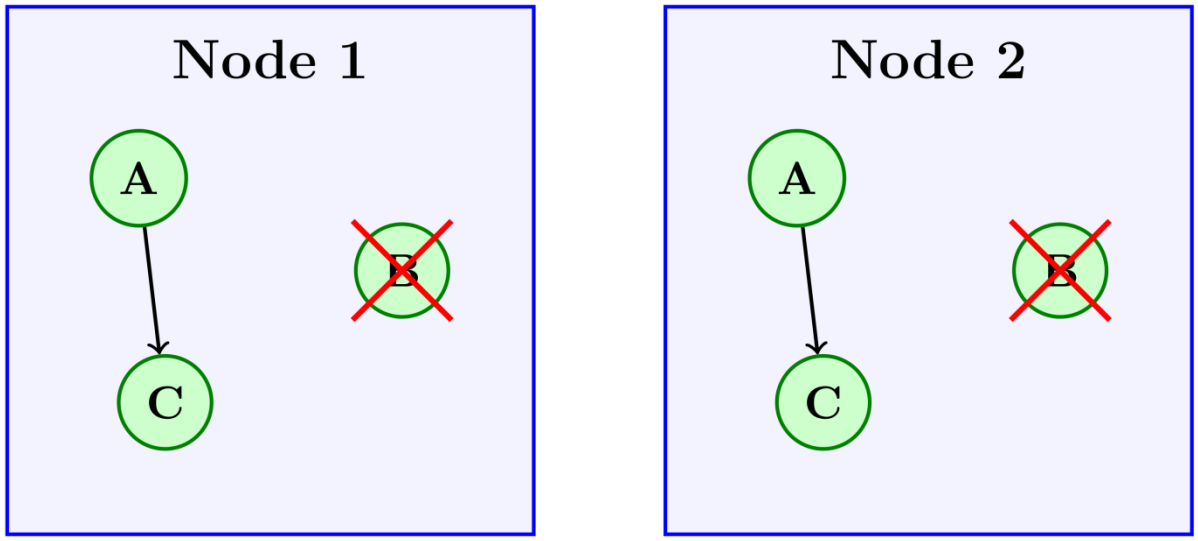
Distributed Garbage Collection – why Java GC is not enough?



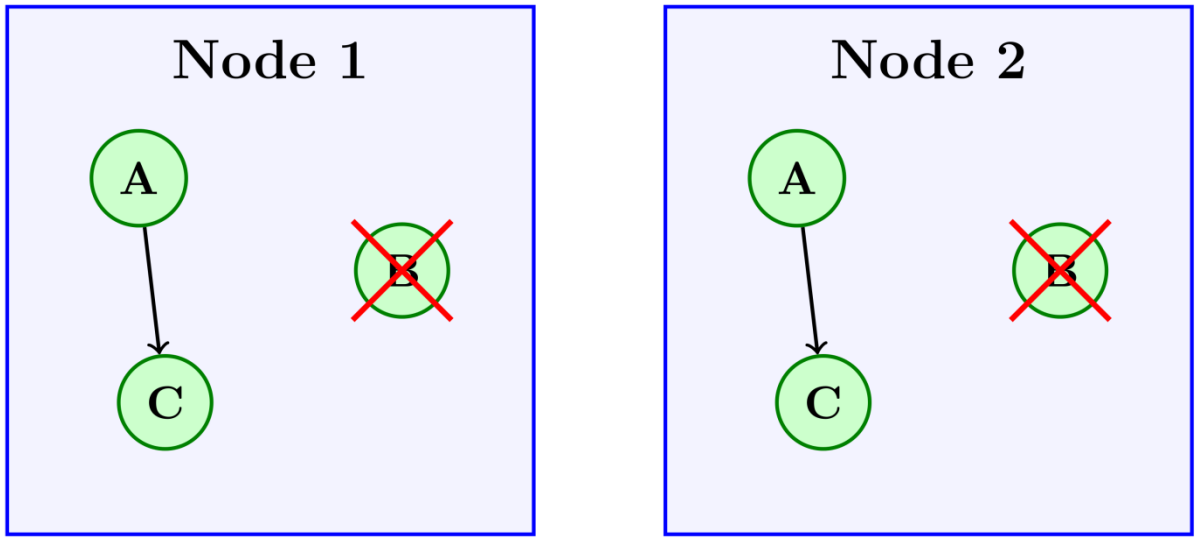
Distributed Garbage Collection – why Java GC is not enough?



Distributed Garbage Collection – why Java GC is not enough?



Distributed Garbage Collection – why Java GC is not enough?



Finally no memory leaks!

Performance consideration

Implementation matters

09/2010

- 1,500 transactions/s [hash] @ 8 nodes
- One version per object
- **Internal locks** for inter-thread synchronization

02/2012

- 150,000 transactions/s [hash] @ 8 nodes
- Added multi-versioning
- **Memory barriers** inter-thread synchronization

No changes to Deferred Update

Concurrency within Paxos STM

- No explicit locks
- Synchronization on memory barriers
- Single writer principle → Cache coherency-friendliness
- **Reader threads – no synchronization overhead**

```
Class Car { ... };
volatile Car root = new Car(„Honda”);

// Thread 1
Car newCar = new Car(„Toyota”);
newCar.next = root;
root = newCar; //memory barrier
```

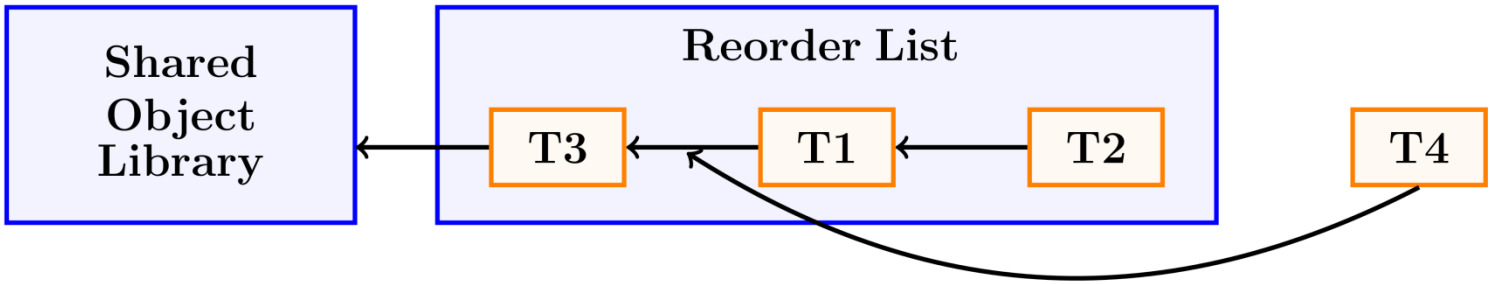
```
// Threads 2..n
Car newestCar = root;
while (newestCar != null) {
  System.out.println(newestCar);
  newestCar = newestCar.next;
}
// Toyota
// Honda
```

Other optimizations

- Abort-free execution of read-only transactions
- JPaxos – Paxos STM binding
- Custom object serialization
- Package size trim
- Heavy code profiling
  - Interface replaced by abstract class
  - Transaction exceptions' handling

Other (mis-)optimizations

- Reorder list



- Nested transactions object lookup index

```
atomic {
    x = 5;
    atomic {
        ...
        atomic {
            y = x;
        }
    }
}
```

Key observation

Performance & scalability

features of the underlying protocol

technical aspects of the implementation

Key observation

Performance & scalability

features of the underlying protocol

technical aspects of the implementation

Is the bandwidth the bottleneck?

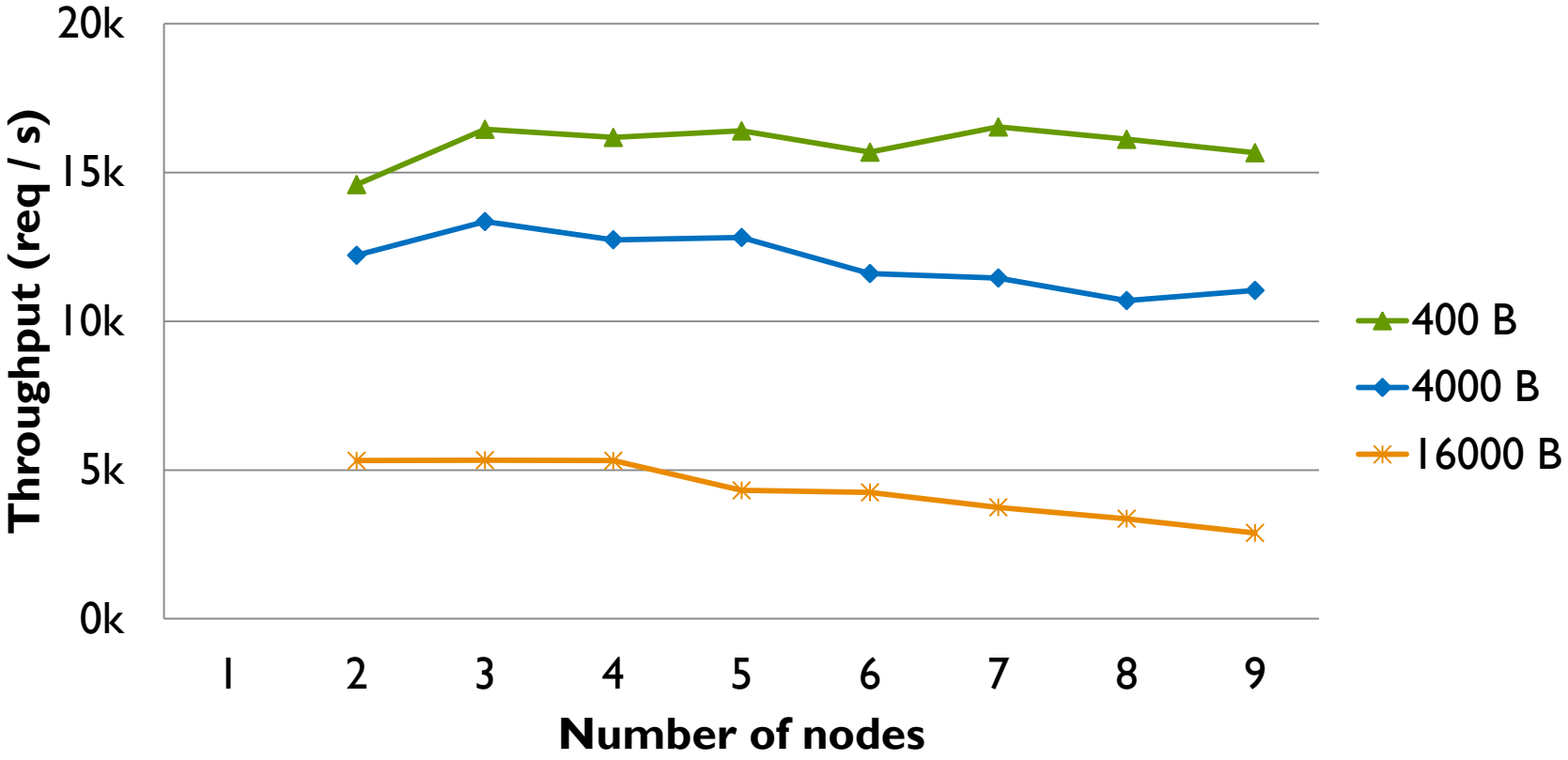
Evaluation

- Environment
  - 9 nodes x (4 x AMD Opteron 6174)
  - 4 x 12 cores @ 2.2GHz, 128KB / 512KB / 12MB cache
  - 192GB RAM
  - Infiniband QDR 32 Gbps
  - Gigabit ethernet
  - CentOS 5.6, OpenJDK 1.6

Evaluation

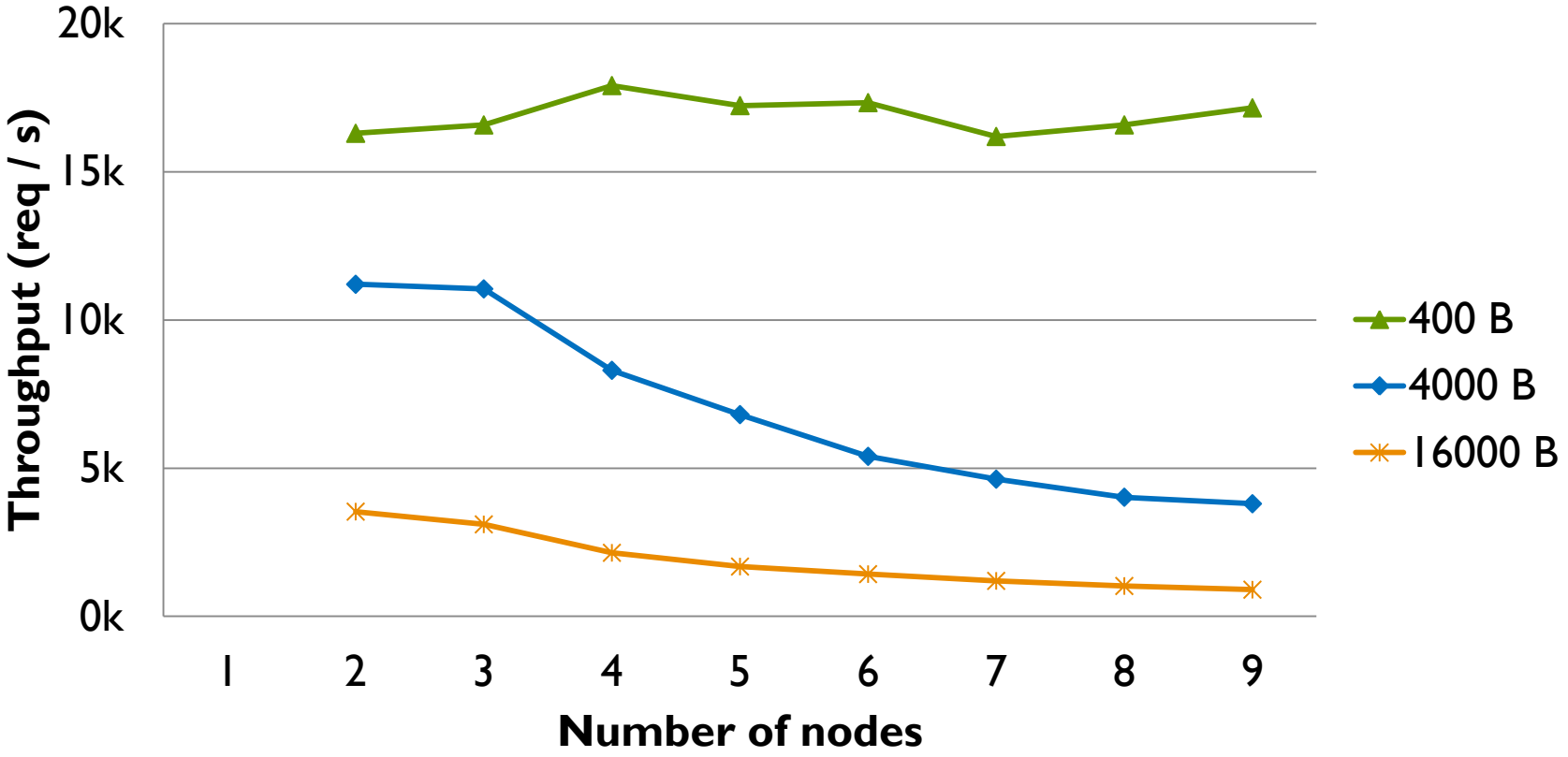
- Number of nodes: 2-9
- Number of threads/node: 48
- Scenarios RO/RW:
  - 90 / 10
  - 50 / 50
  - 10 / 90
- Microbenchmark – replicated hash table
  - Size: 10,000 elements
  - RO transaction – 100 get requests
  - RW transaction – 8 get requests + 2 put/remove request
- Bank benchmark
  - Size: 10,000 accounts
  - RO transaction – scan all the accounts
  - RW transaction – wire transfer between 2 accounts

Base layer – JPaxos (Infiniband)



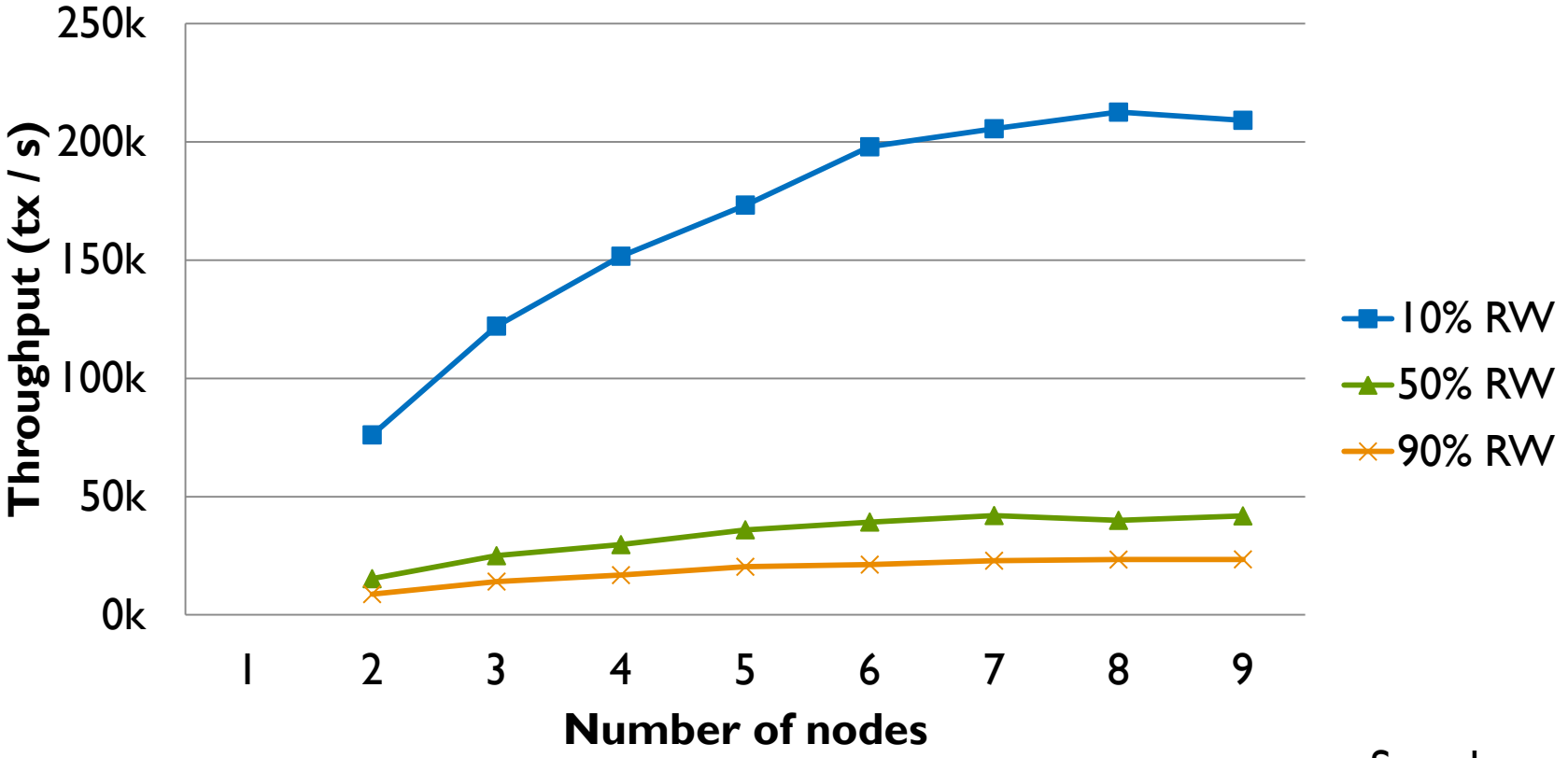
- Bigger the requests lower the throughput
- More nodes → slight performance decrease

Base layer – JPaxos (Gigabit Ethernet)



- Network gets saturated
- More nodes → dramatic performance decrease

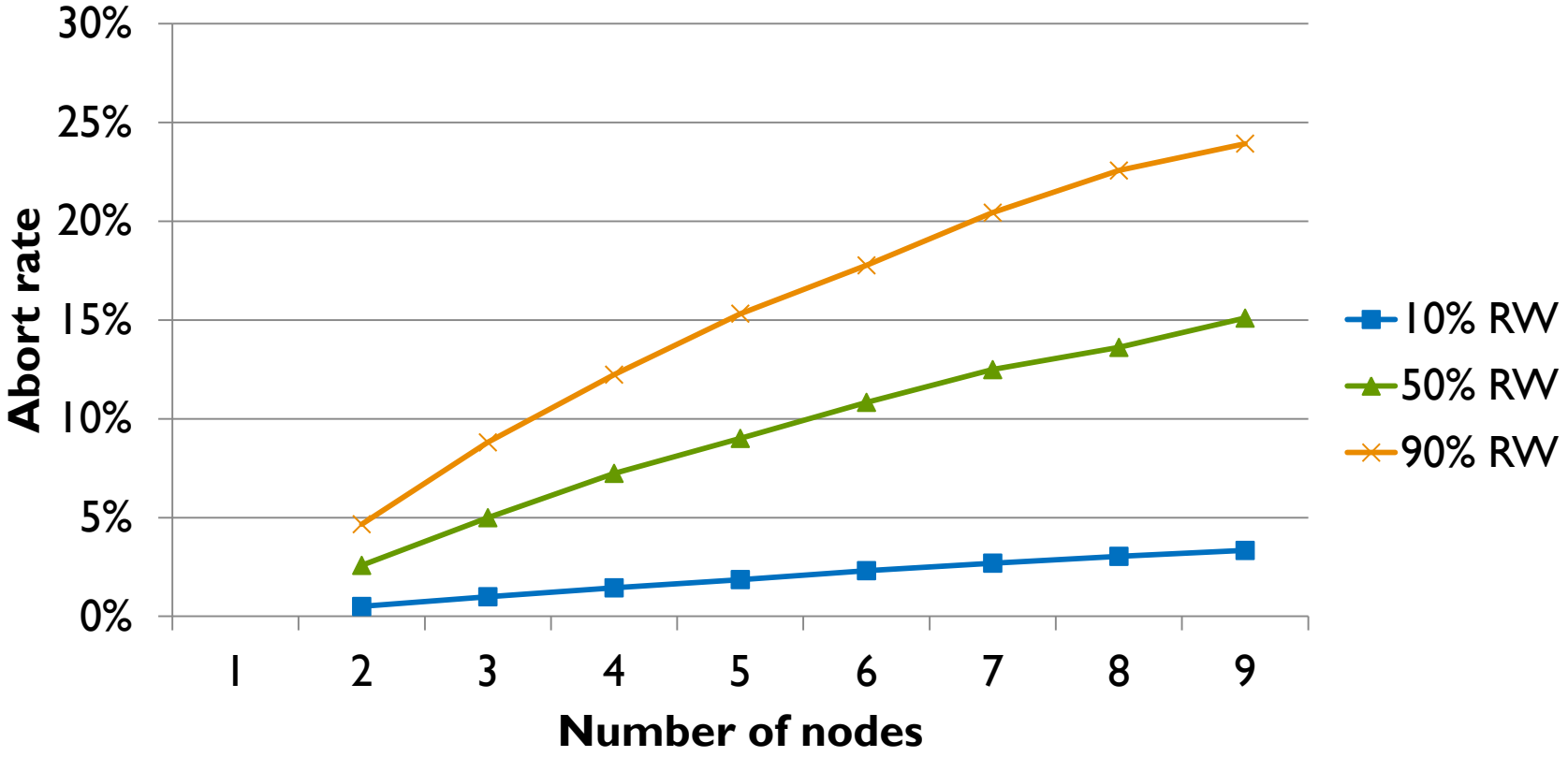
Hash table (Infiniband)



Speedup:
10% - 2.79
50% - 2.74
90% - 2.67

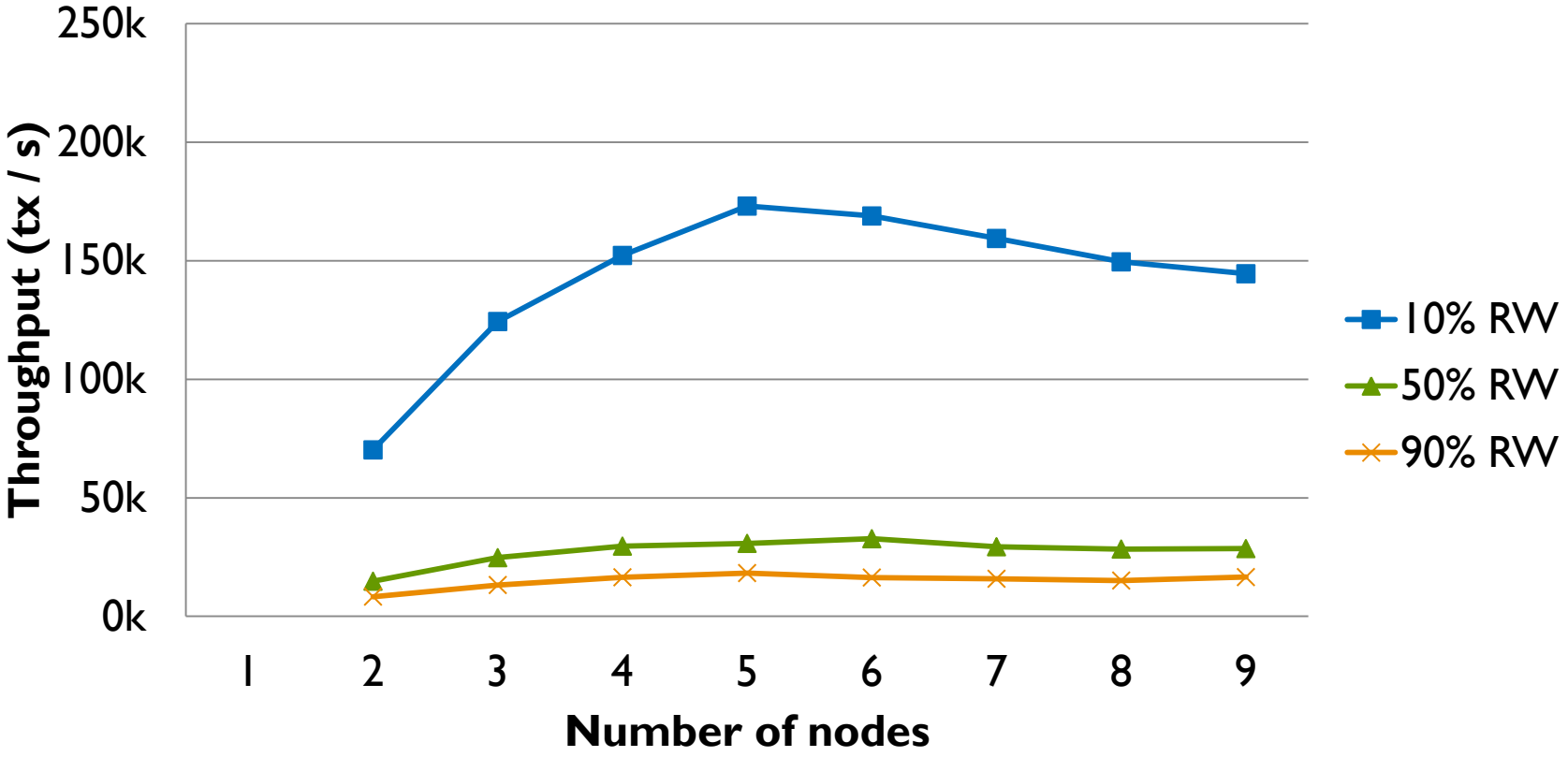
- Top result of 212k tx/s
- All the scenarios scale pretty well

Hash table (Infiniband)



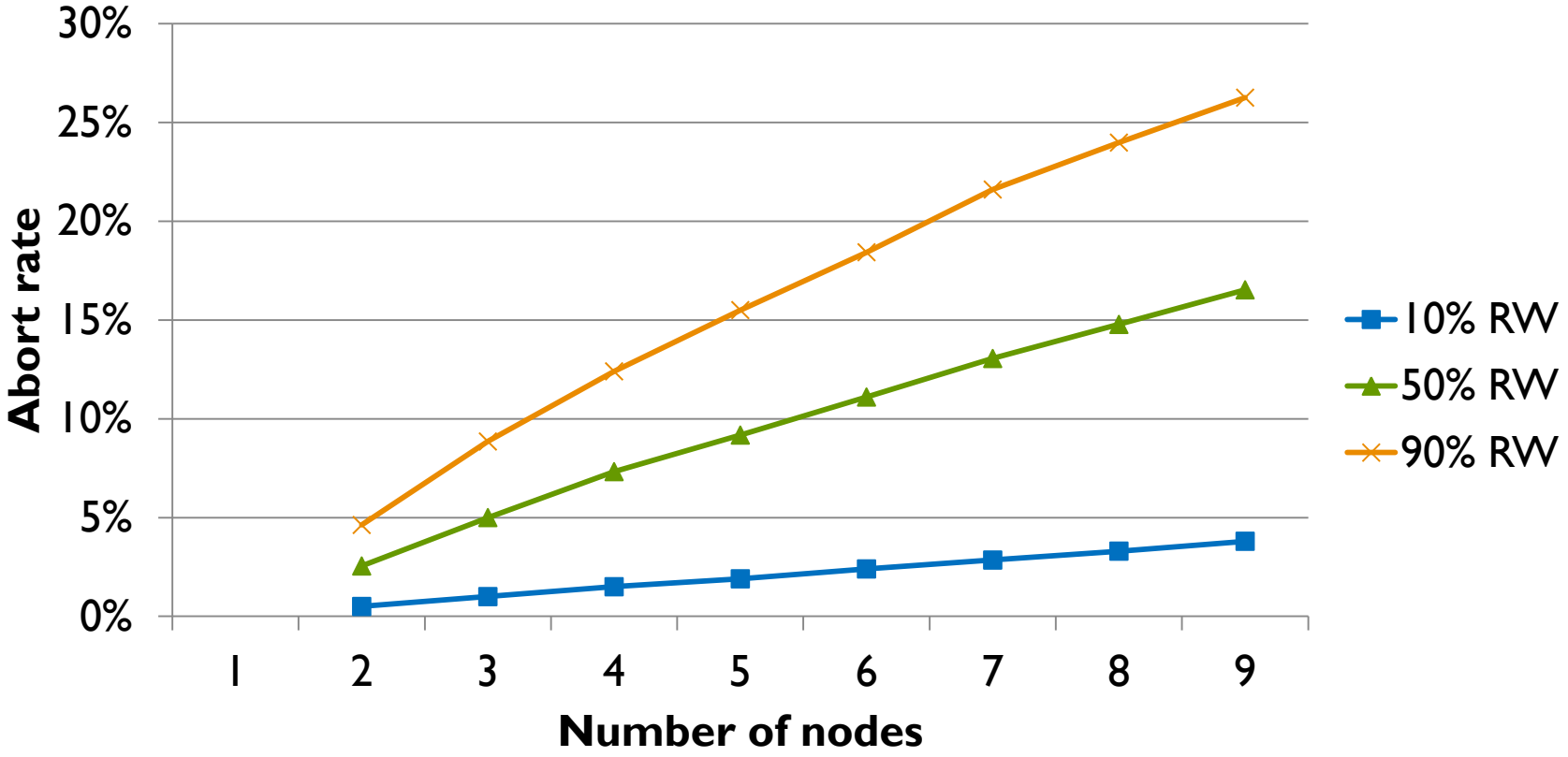
- Abort rate steadily increases with the number of nodes
- High contention (25%) limits scalability

Hash table (Gigabit Ethernet)



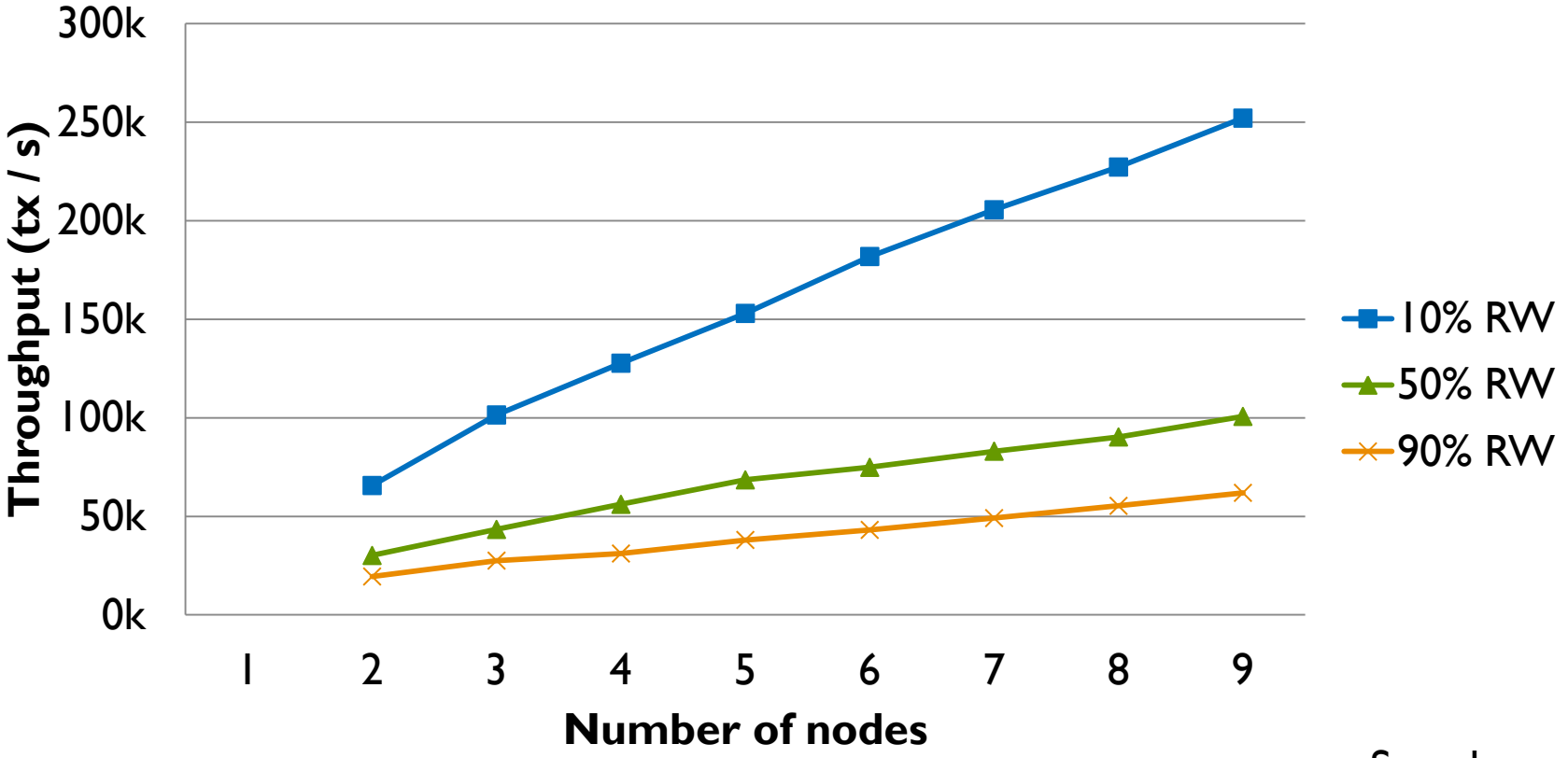
- Nearly identical throughput with up to 5 nodes
- Sudden decrease → limitation of the network interface

Hash table (Gigabit Ethernet)



- The same contention level as with Inifiniband

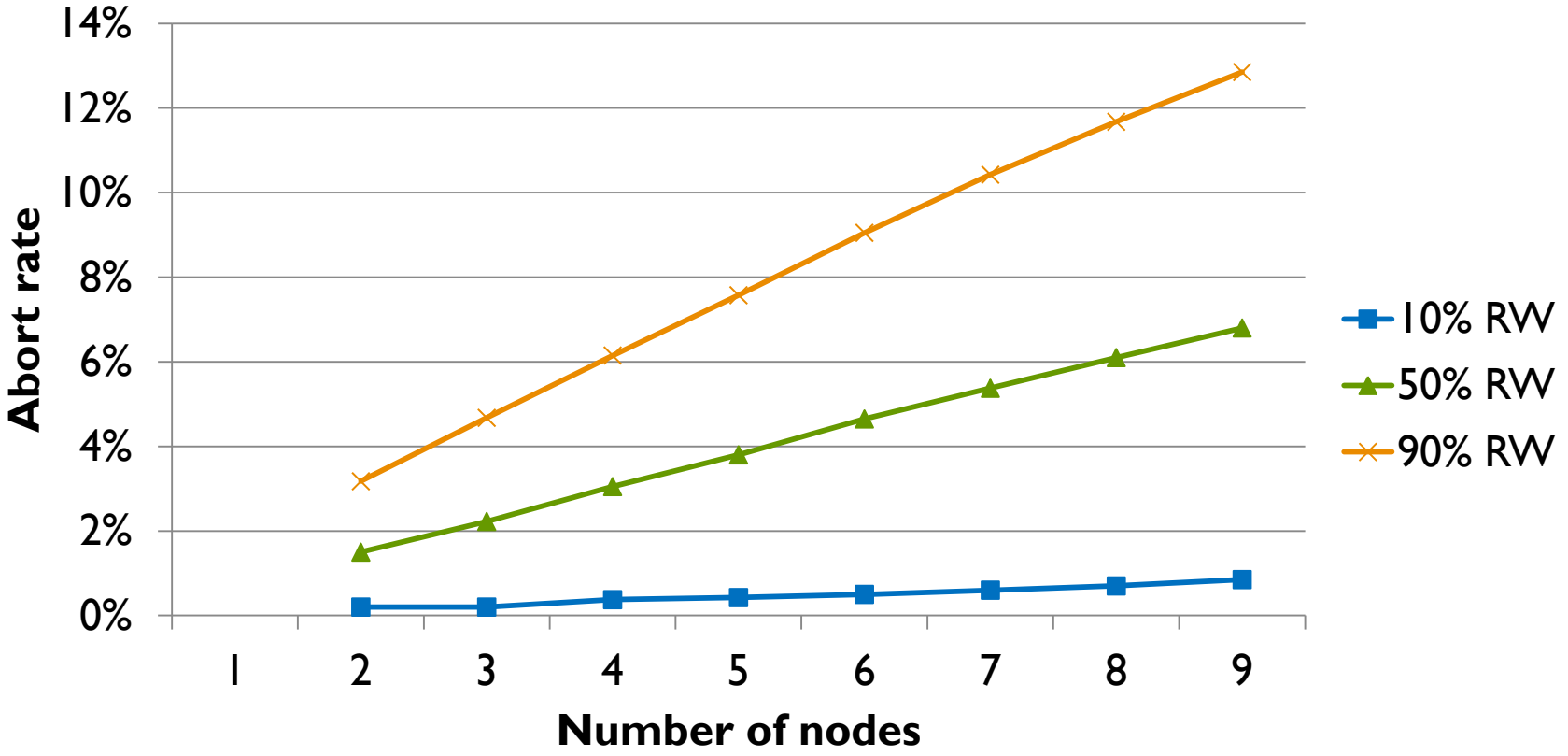
Bank (Infiniband)



- Top result of 252k tx/s
- Perfect scalability

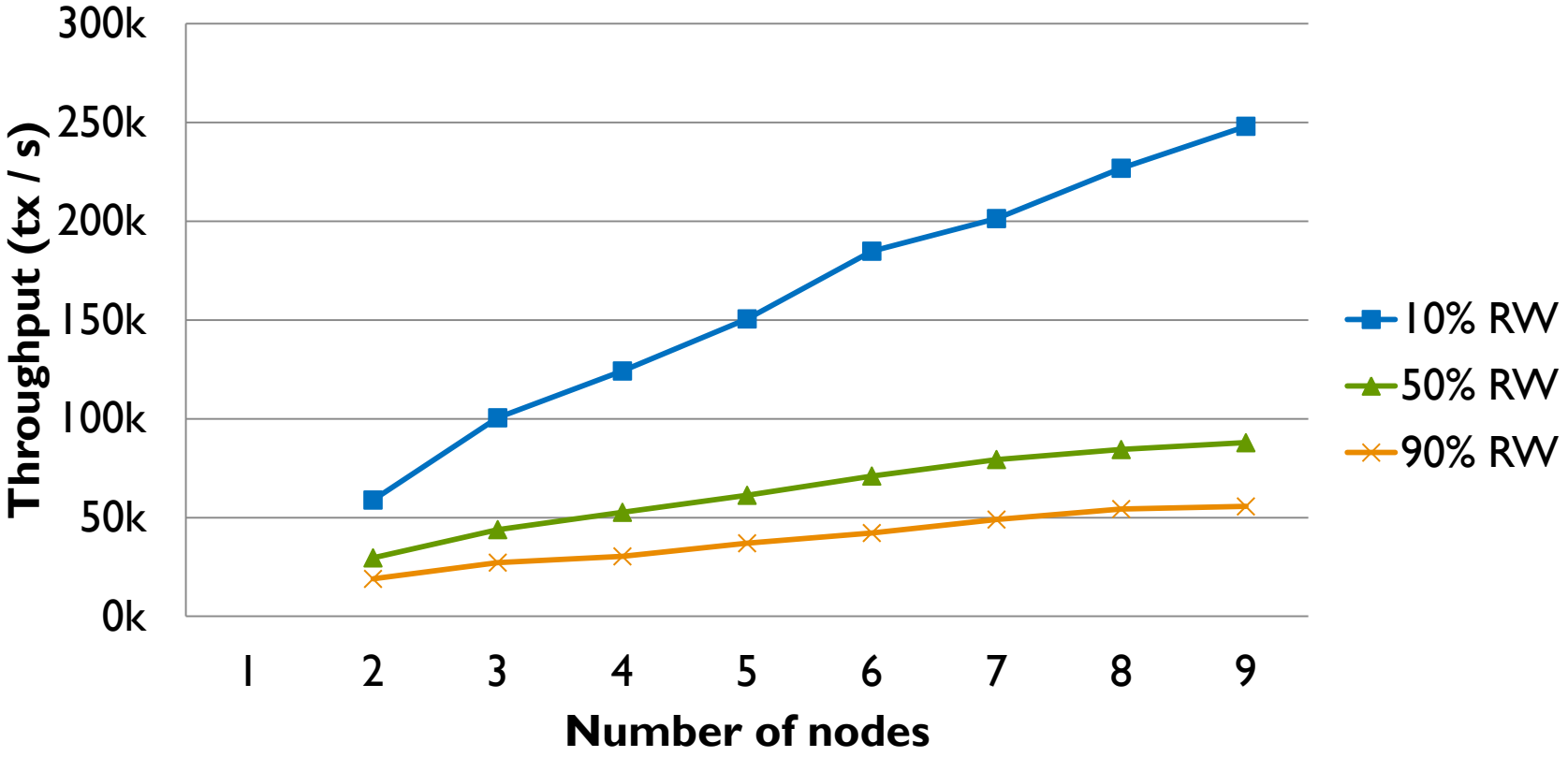
Speedup:
10% - 3.84
50% - 3.34
90% - 3.18

Bank (Infiniband)



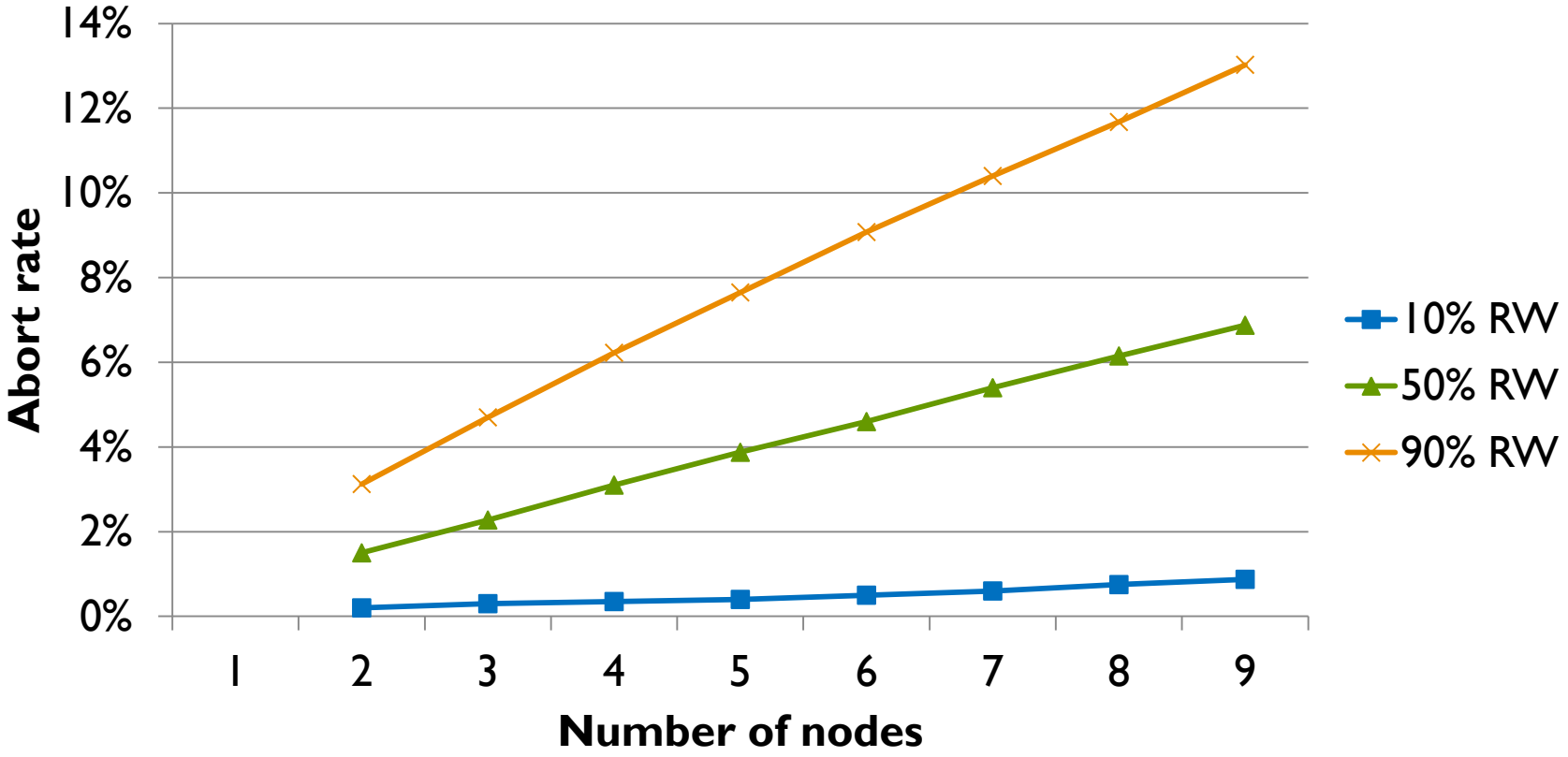
- Lower contention (14%) can be attributed to shorter tx
- No limit to scalability up to 9 nodes (what happens later?)

Bank (Gigabit Ethernet)



- No sudden decrease this time!
- Shorter tx → less data to be exchanged

Bank (Gigabit Ethernet)



- The same contention level as with Infiniband

Conclusions

- Implementation matters
  - Avoid locks!
  - Don't forget about cache
  - Profile your code
- Some techniques work only on paper
- Scalable software for scalable protocols assesment
- Network – the crucial factor

- Contact:
  - www.it-soa.pl/paxosstm
  - paxosstm@cs.put.poznan.pl