Scalaris – Methods for a Globally Distributed Key-Value Store with Strong Consistency

Thorsten Schütt
Zuse Institute Berlin

XtreemOS Summer School 2010

Outline

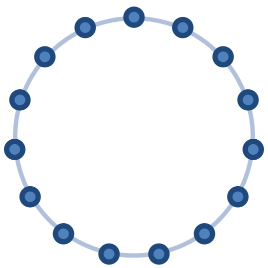
Clients

Key/Value Store (simple DBMS)

ACID

Transactions

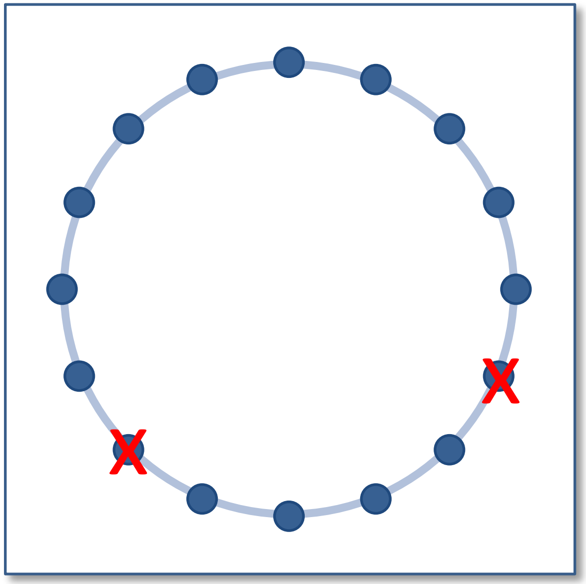
Replication



P2P Overlay

scalability and self-management

P2P LAYER



P2P Layer

- implements a primitive key/value store
  - synonyms: “key/value store” = “dictionary” = “map”, = …

- just 3 ops
  - `insert(key,value)`
  - `delete(key)`
  - `lookup(key)`

Example:
A Key/value store with all Turing award winners

Key	Value
Backus	1977
Hoare	1980
Karp	1985
Knuth	1974
Wirth	1984
. . .	. . .

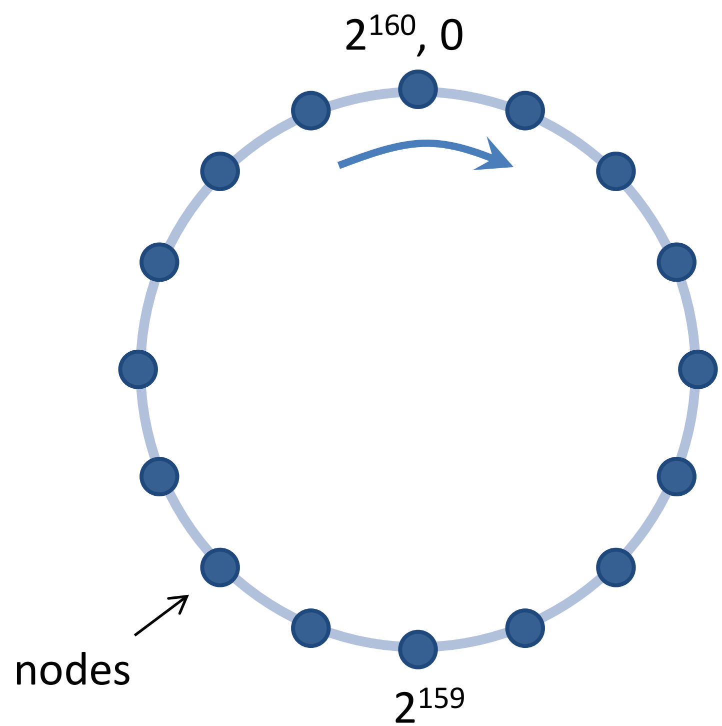
Nodes Maintain a Ring Structure

Keys

- define positions in the ring, e.g. $0 - 2^{160}$ or strings

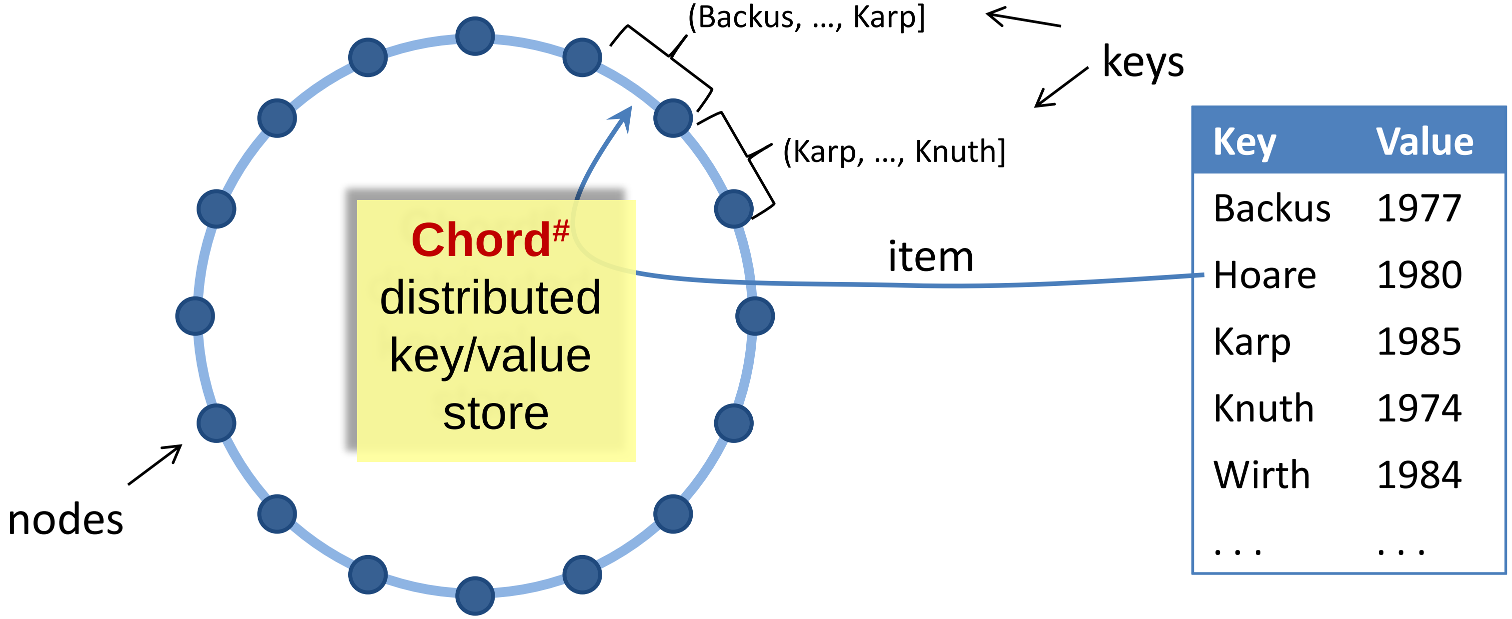
Nodes

- may join, leave or fail (churn)
- know several successors
- know their predecessor
- have random position in the ring



P2P Layer with Chord#

- Chord# uses keys directly as addresses in the ring
  - no hashing, thereby order-preserving → enables range queries
  - just need a total order on items
- The next node in the ring (clockwise) is responsible for a key

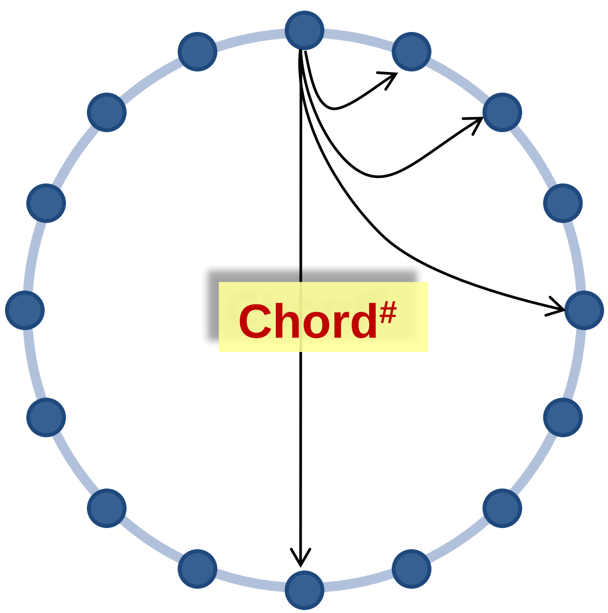


Routing Table and Data Lookup

Routing Table

- table contains $\log_2 N$ pointers
- pointers are exponentially spaced

$$pointer_i = \begin{cases} successor & : i = 0 \\ pointer_{i-1} \, . \, pointer_{i-1} & : i \neq 0 \end{cases}$$

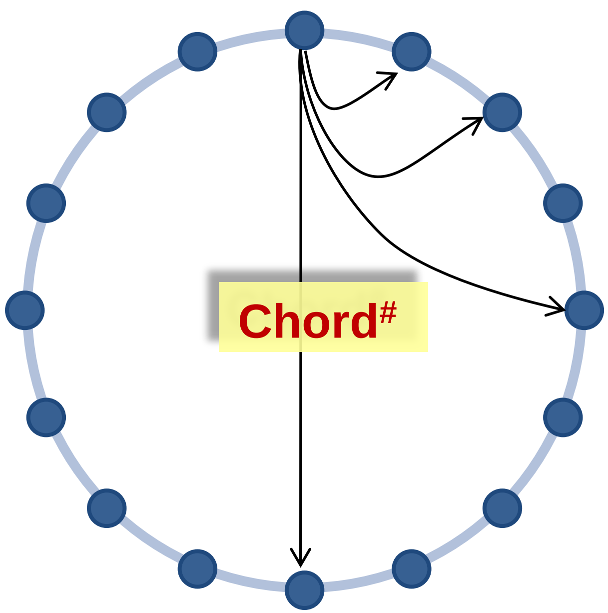


Routing Table and Data Lookup

Routing Table

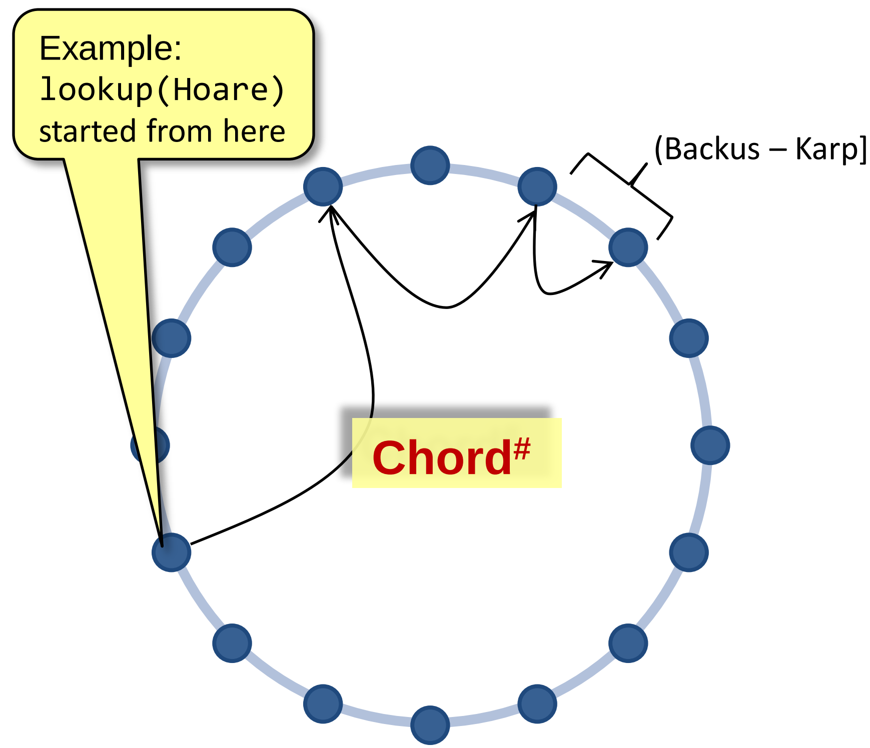
- table contains $\log_2 N$ pointers
- pointers are exponentially spaced

$$pointer_i = \begin{cases} successor & : i = 0 \\ pointer_{i-1} \,.\, pointer_{i-1} & : i \neq 0 \end{cases}$$



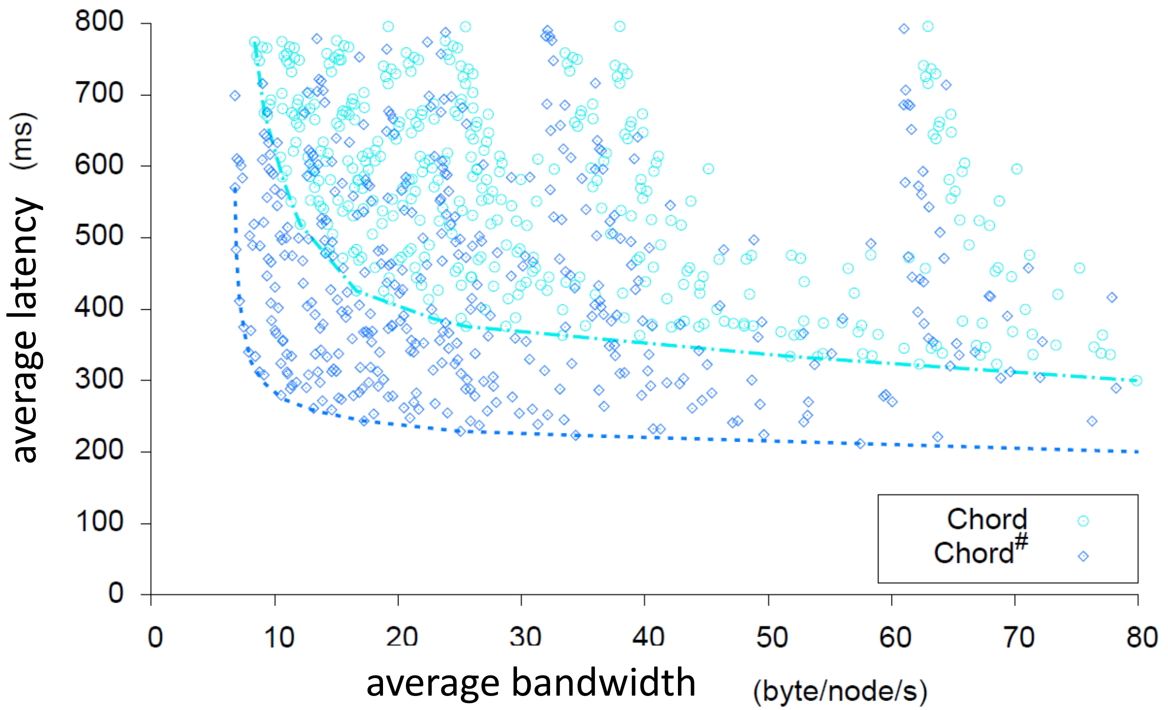
Retrieving Items

- $\leq \log_2 N$ hops



P2P Layer with Chord#

- Chord# features
  - fully decentralized, operations require only local knowledge
  - self-organizes as nodes join, leave, and fail
  - easy routing table maintenance
  - ≤ log (N) hops
  - range queries

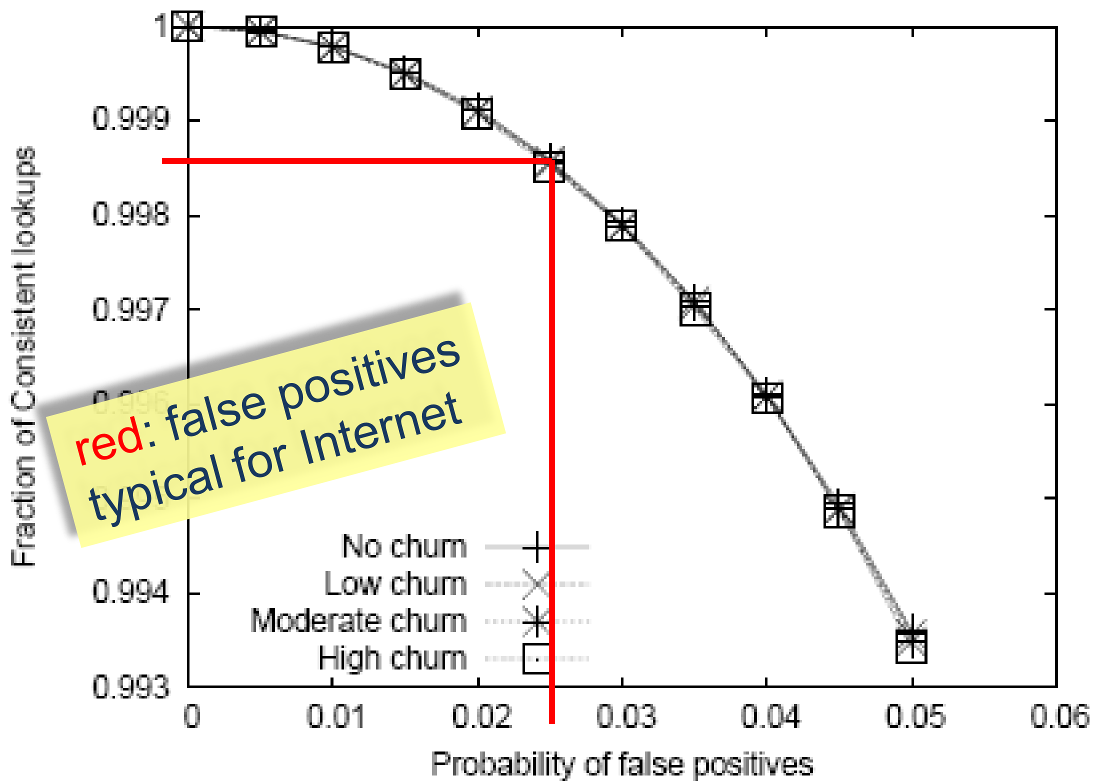


Failure detector

- Need a **failure detector** to check if nodes are alive.
- But failure detector may be wrong.
  - Node dead? Or just slow?
  - **Even without churn, inconsistencies may occur!**
- Two types of inconsistency
  - responsibility inconsistency
  - lookup inconsistency

How often does this occur?

- We simulated nodes with imperfect failure detectors
  (A node detects another living node as dead probabilistically)

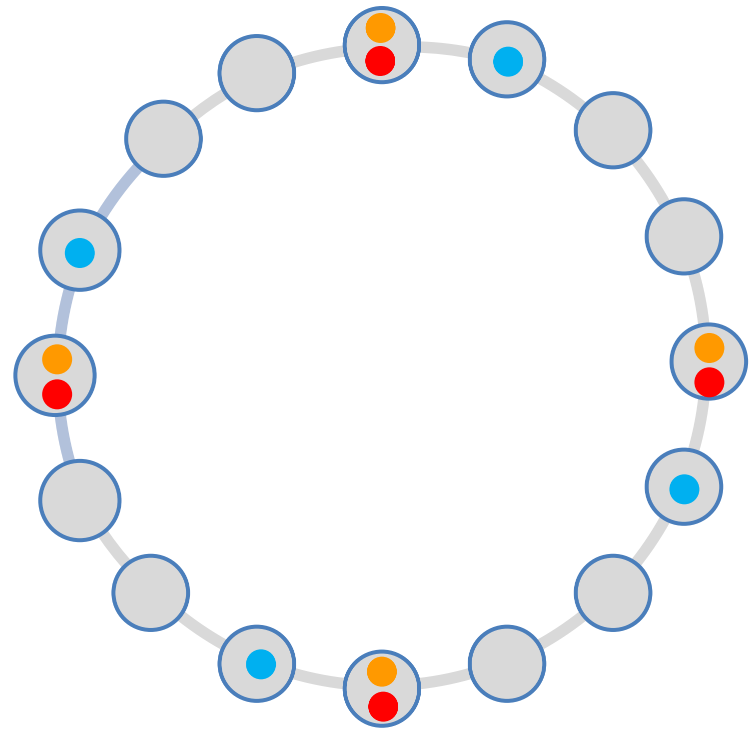


providing data availability

REPLICATION LAYER

Replication

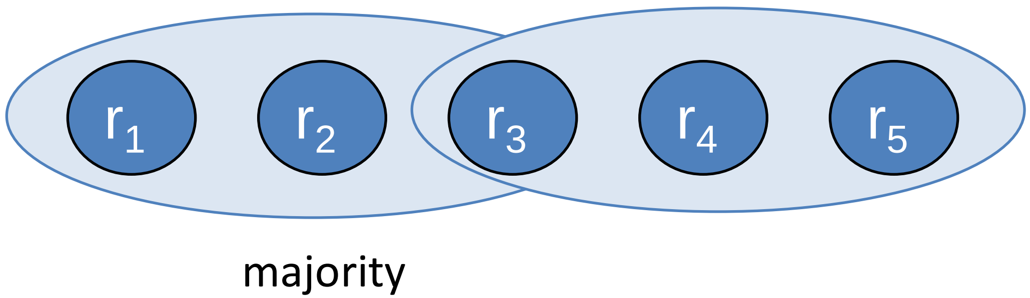
- We use symmetric replication
  - Use a globally known function to determine a set of keys under which the data is stored →



- Must ensure data consistency
  - need quorum-based methods

Replication and Quorum-based Algorithms

- Only read/write on majorities:

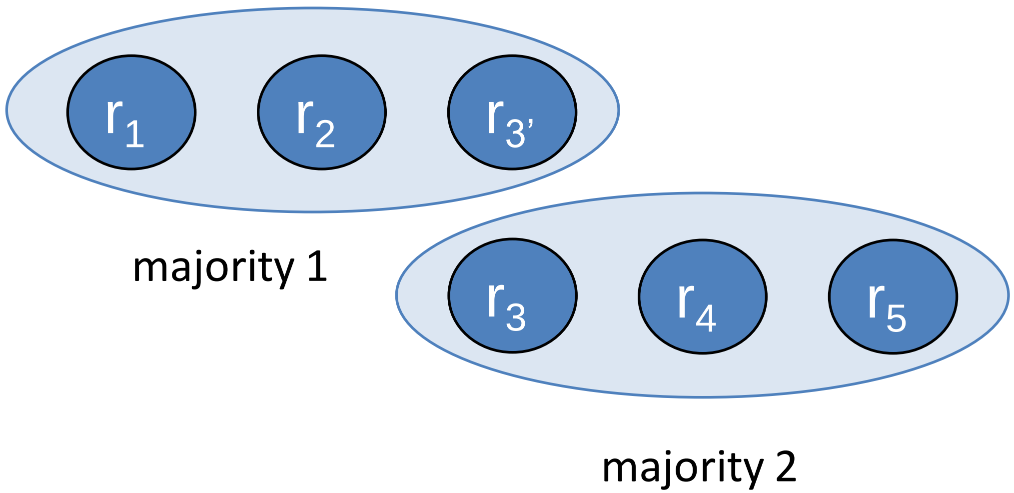


- Concurrent operations have overlapping majorities
  => conflict detection

- Comes at the cost of increased latency
  - but latency can be avoided by intelligent distribution over datacenters

Replication and Quorum-based Algorithms

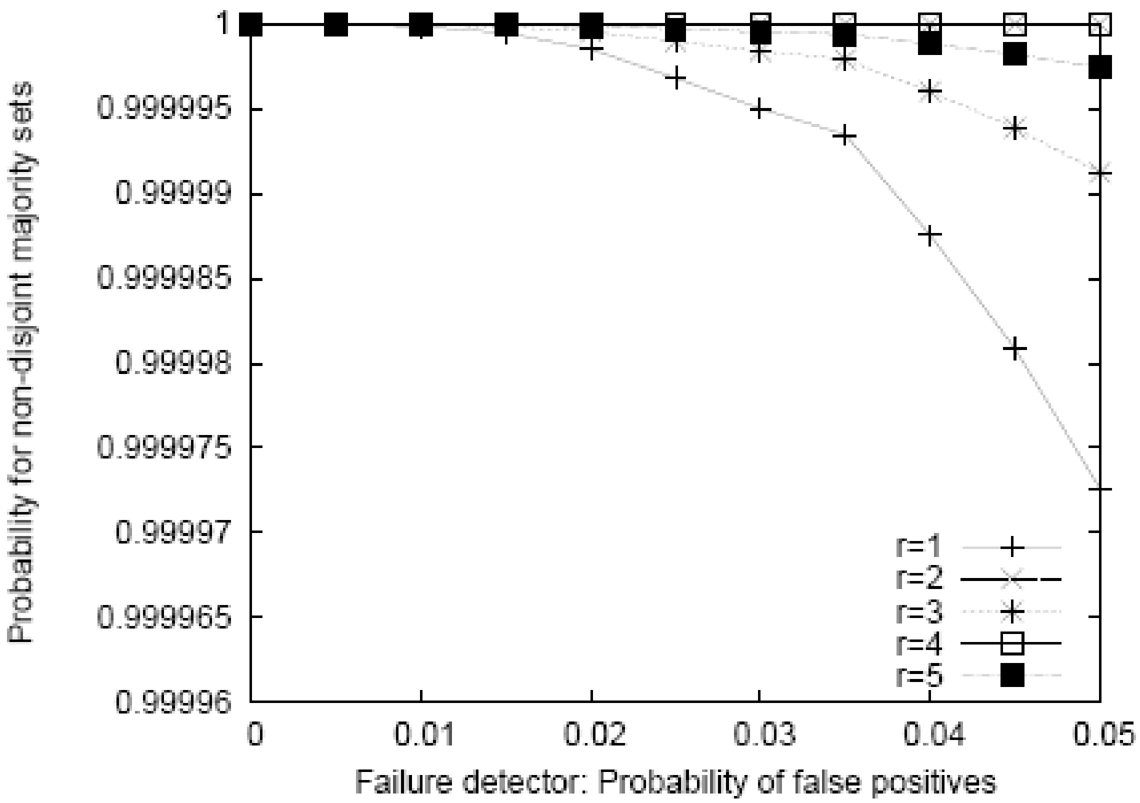
- Lookup inconsistency may result in more than f replicas
- Then, two (or more) non-overlapping majorities exist:



=> Must ensure that number of replicas is always ≤ f when using *simple* majority access

  - relaxed with stronger majorities

More consistent accesses with replication and quorum access



coping with concurrency

TRANSACTION LAYER

Challenges for Transactions in SONs

- churn
  - nodes may leave, join, or crash at any time
    → changing responsibilities
- "crash stop" fault model
- no perfect "failure detector"
  - never know whether a node crashed or just slow network

Goal: Strong Consistency

- What is it?
  - When a write is finished, all following reads will return the new value.

- How to implement?
  - Always read/write majority $\lfloor f/2 \rfloor + 1$ of f replicas.

    => **Latest version** is always in the read/write set

Goal: Atomicity

- What is it?
  - Make **all** or **no** changes!
  - Either ‘commit’ or ‘abort’.

- How to implement?
  - 2PC? Blocks if the transaction manager fails.
  - We use a variant of **Paxos Commit**
    - non blocking, because of *multiple acceptors*

Transactions + Replicas

BOT

debit (a, 100);

deposit (b, 100);

EOT

BOT

debit (a_1, 100);

debit (a_2, 100);

debit (a_3, 100);

deposit (b_1, 100);

deposit (b_2, 100);

deposit (b_3, 100);

EOT

Scalaris Transactions in Erlang

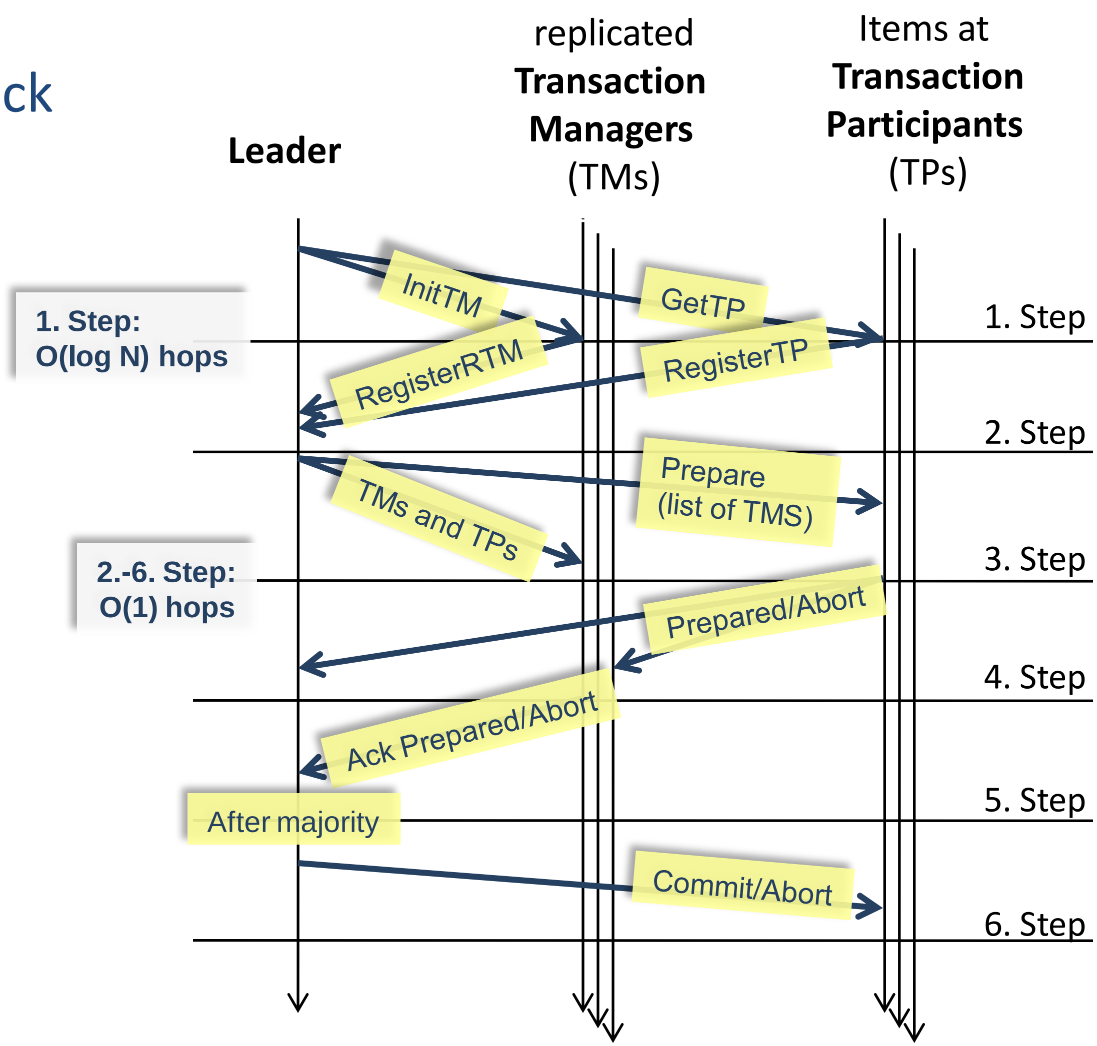
```
F = fun (TransLog) ->
     {X, TL1} = scalaris:read(TransLog, "Account A"),
     {Y, TL2} = scalaris:read(TL1, "Account B"),
      if
         X > 100 ->
              TL3 = scalaris:write(TL2, "Account A", X - 100),
              TL4 = scalaris:write(TL3, "Account B", Y + 100),
              {ok, TL4};
         true ->
              {ok, TL2};
      end
end,
MyTransLog = F(EmptyTransLog),

scalaris:commit(MyTransLog).
```

Build translog with quorum reads:
for a read: (value, version)
for a write: (value, quorum read version + 1)
+ infos on read/write locks

Validate and commit transaction
using Paxos commit.

Adapted Paxos Commit

- Optimistic CC with fallback
- Write
  - 3 rounds
  - non-blocking (fallback)
- Read even faster
  - reads majority of replicas
  - just 1 round
- succeeds when
  >f/2 nodes alive



scalaris in practice

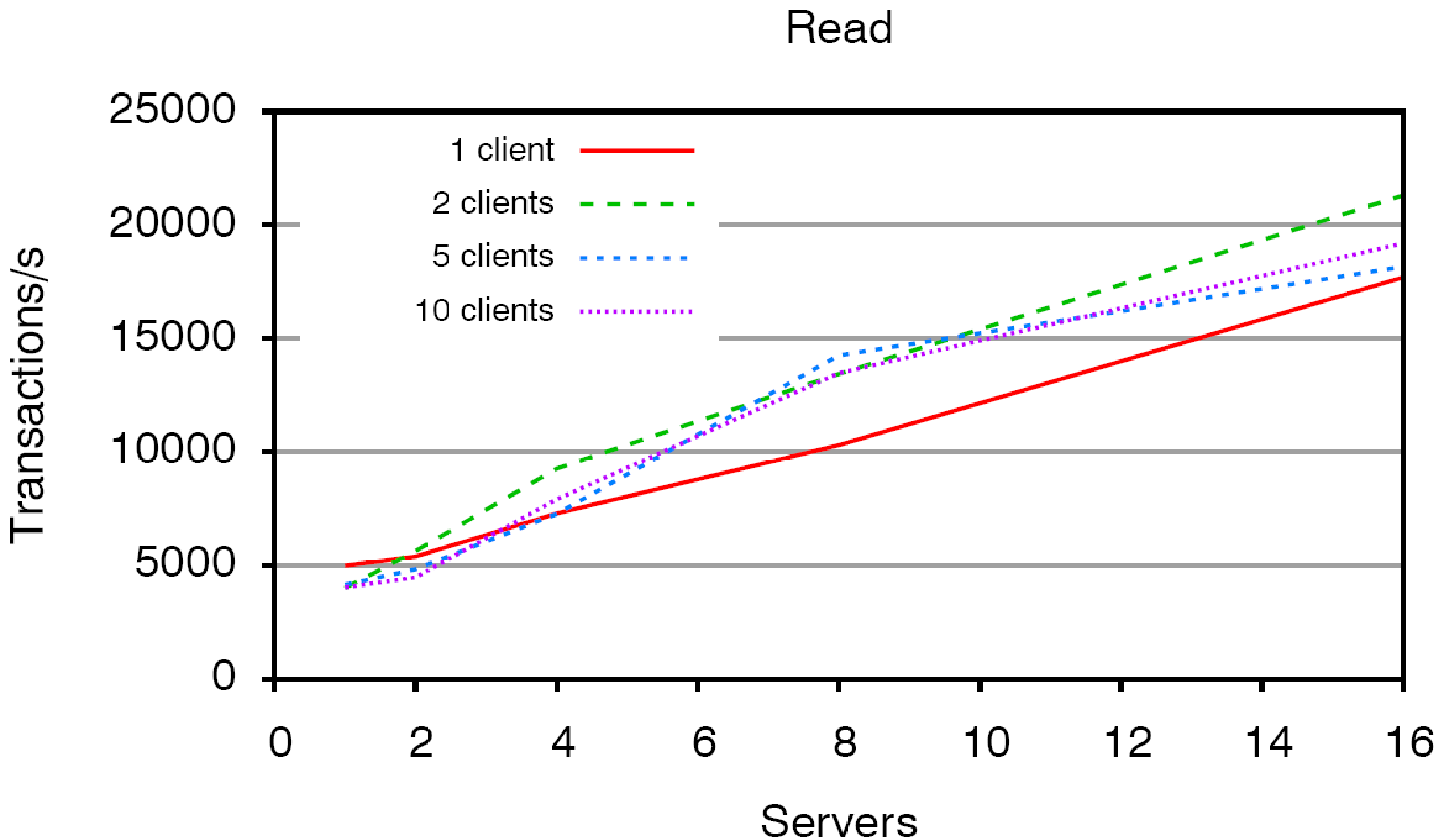
IMPLEMENTATION

Scalaris Implementation

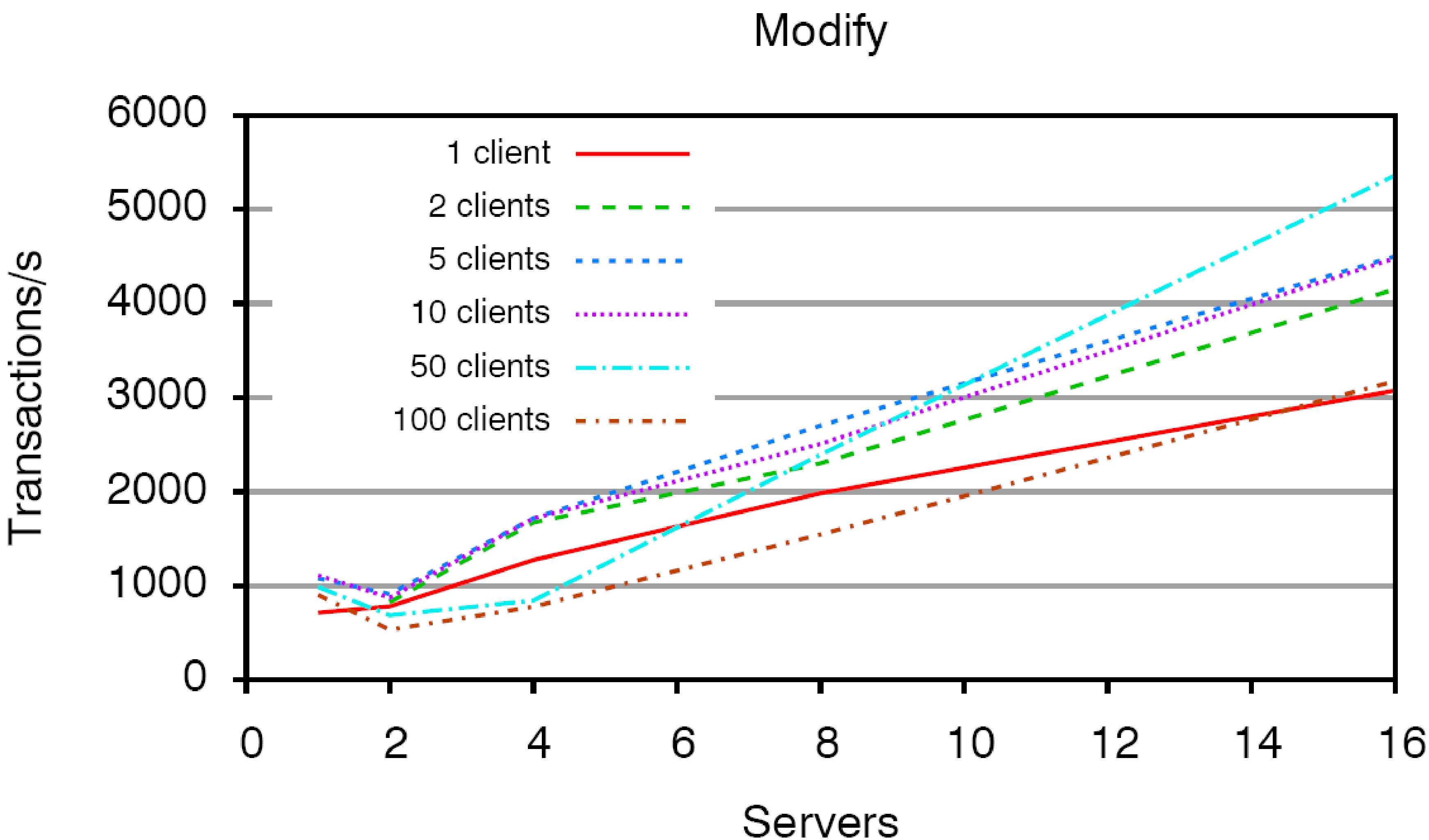
- Scalaris: 9,700 lines of Erlang code
  - 7,000 for Chord$^{\#}$ and infrastructure
  - 2,700 for transactions

- Application specific code
  - 1,300 for our Wikipedia code
  - Java for rendering and user interface

Read Performance



Modify Performance



Publish-Subscribe

- Application running on top of Scalaris
  - Subscription Database
- Operations:
  - Subscribe(Topic, URL)
    - updates database
  - Unsubscribe(Topic, URL)
    - update database
  - Publish(Topic, Message)
    - sends HTTP-JSON to all subscribed URLs

Key	Value
Server Failures	[URL1]
New Nodes	[URL1, URL2, URL3]
Weather Changes	[URL2]
DAX Changes	[URL13]
Soccer Goals	[URL1, URL2, URL3, URL4, URL5, …]
. . .	. . .

Summary

- http://scalaris.googlecode.com (BSD-License)
- scalable, transactional key-value store
- Java-API, JSON-HTTP, Ruby client, command line client, Erlang client

scalaris.googlecode.com