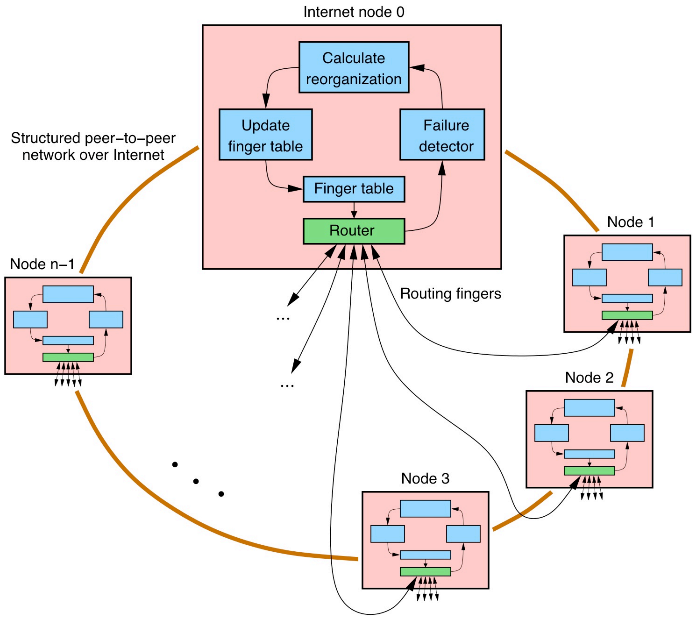
THE ADVENTURES OF SELFMAN

Self Management for Large-Scale Distributed Systems



Peter Van Roy
and SELFMAN partners

May 8, 2008

Grid@Mons 2008

Université catholique de Louvain
Louvain-la-Neuve, Belgium

Vision

- Software is fragile!
  - A single bit error can cause a catastrophe
- Software complexity is ramping up quickly due to:
  - The sufficient bandwidth and reliability of the Internet
  - The increasing number of networked devices
  - The increasing computing power of these devices
- Many new applications are appearing
  - File-sharing (Napster, Morpheus, Freenet, BitTorrent,…), information sharing (Youtube, Flickr, …), social networks (LinkedIn, FaceBook, …), collaboration (Wikis, Skype, Messengers, …), MMORPGs, on-line vendors (Amazon, eBay, PriceMinister, …), etc.
  - These applications are currently a mix of client/server and peer-to-peer, but they are getting more complicated
- How can we build such applications so they are not fragile?
  - They should be self managing

What is self management?

- **The system should be able to reconfigure itself to handle changes in its environment or its requirements without human intervention but according to high-level management policies**
  - Human intervention is lifted to the level of the policies
- Typical self-management operations include: add/remove nodes, tune performance, auto-configure, failure detection & recovery, intrusion detection & recovery, software rejuvenation
- Self management is needed **at all levels**
  - Such as: single node level (failures), network level, services (transactional storage, broadcast), application level
- For large-scale systems, environmental changes that require recovery by the system become normal and even frequent events
  - **"Abnormal" events are normal occurrences** (failure is a normal occurrence)

How to build large-scale self-managing applications?

- We start with systems that already solve the problem !
  - Structured overlay networks (derived from peer-to-peer)
- These systems already handle the lower layers
  - Self-managed communication and storage
- We add the higher layers needed by applications
  - First we complete the overlays by handling network partitioning and improving lookup consistency
  - Then we add replicated storage and a transaction service
- The needs are guided by three application scenarios
  - Machine-to-machine messaging (France Telecom)
  - Distributed Wiki (ZIB)
  - On-demand video streaming (Stakk)

Three application scenarios

- Our self-management architecture is designed so that these three scenarios can work well:
- Machine-to-machine messaging (France Telecom): decentralized messaging application, must recover on node failure, must gracefully degrade and self optimize, have transactional behavior
- Distributed Wiki (ZIB): Wiki distributed over SON using transactions with versioning and replication, both editing and search support
- P2P video streaming (Stakk): distributed live media streams with quality of service to large numbers of customers, need dynamic reconfiguration to handle fluctuating structure

Application requirements

Use Case	Self-* Properties	Components	Overlay Networks	Transactions
Machine To Machine	++	++	+	+
Distributed Wiki	++	+	++	++
P2P Video Streaming	++	+	++	
J2EE Application Server	++	++		+

Successive steps to build a self-management architecture

- First, we fix the overlay networks
  - Improving lookup consistency: relaxed ring
  - Handling network partitioning: merge algorithm
- Second, we add services
  - Replicated storage
  - Distributed transaction service
- We explain each of these steps
  - This is work being done in the SELFMAN project

THE ADVENTURES OF SELFMAN

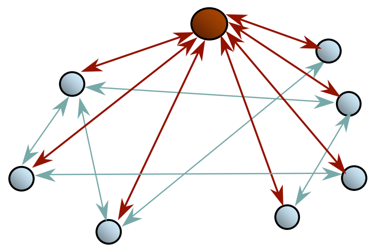
SELFMAN project

- STREP in IST Software and Services, 3 years starting June 2006
- Partners:
  - Université catholique de Louvain (Belgium) (coordinator)
  - Kungliga Tekniska Högskolan (Sweden)
  - Institut National de Recherche en Informatique et Automatique (France)
  - France Télécom Recherche et Développement (France)
  - Konrad-Zuse-Zentrum für Informationstechnik Berlin (Germany)
  - National University of Singapore (Singapore)
  - Peerialism AB (Sweden)

Structured overlay networks

Three generations of peer-to-peer networks

- Hybrid (client/server)
  - Napster

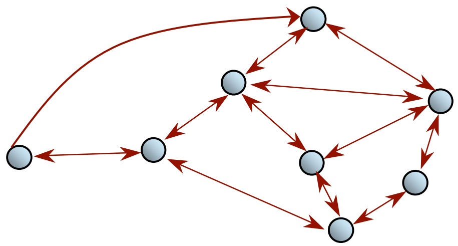


R = N-1 (hub)

R = 1 (others)

H = 1

- Unstructured overlay
  - Gnutella, Kazaa, Morpheus, Freenet, …
  - Uses flooding

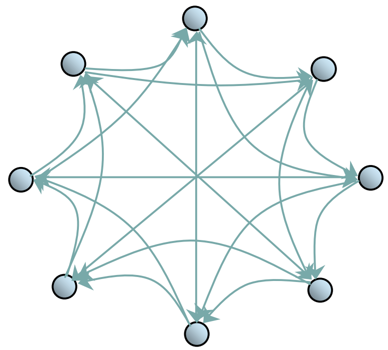


R = ? (variable)

H = 1…7

(but no guarantee)

- Structured overlay
  - Exponential network
  - DHT (Distributed Hash Table), e.g., Chord, DKS, P2PS



R = log N

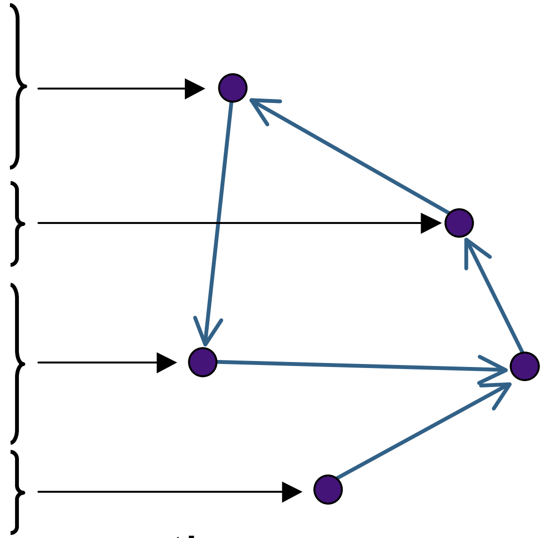
H = log N

(with guarantee)

What is a Structured Overlay Network (also known as Distributed Hash Table)?

- An ordinary hash table that is distributed

Key	Value
Bengt	193.20.10.2
Ali	202.49.2.44
Björn	241.13.11.19
Peter	169.14.33.1
Olle	10.0.0.1
Nisse	211.113.9.12



- Every node provides a lookup operation
  - Provide the value associated with a any key
- Nodes keep routing pointers
  - If item not found, route to another node

Properties of SONs/DHTs

- Scalability
  - Number of nodes
  - Number of items

Maximum **log(*n*)** re-routes
log(*n*) routing table size

1/***n*** portion of items per node

- Self-manage in presence *joins*/*leaves*/*failures*
  - Routing information
  - Data items

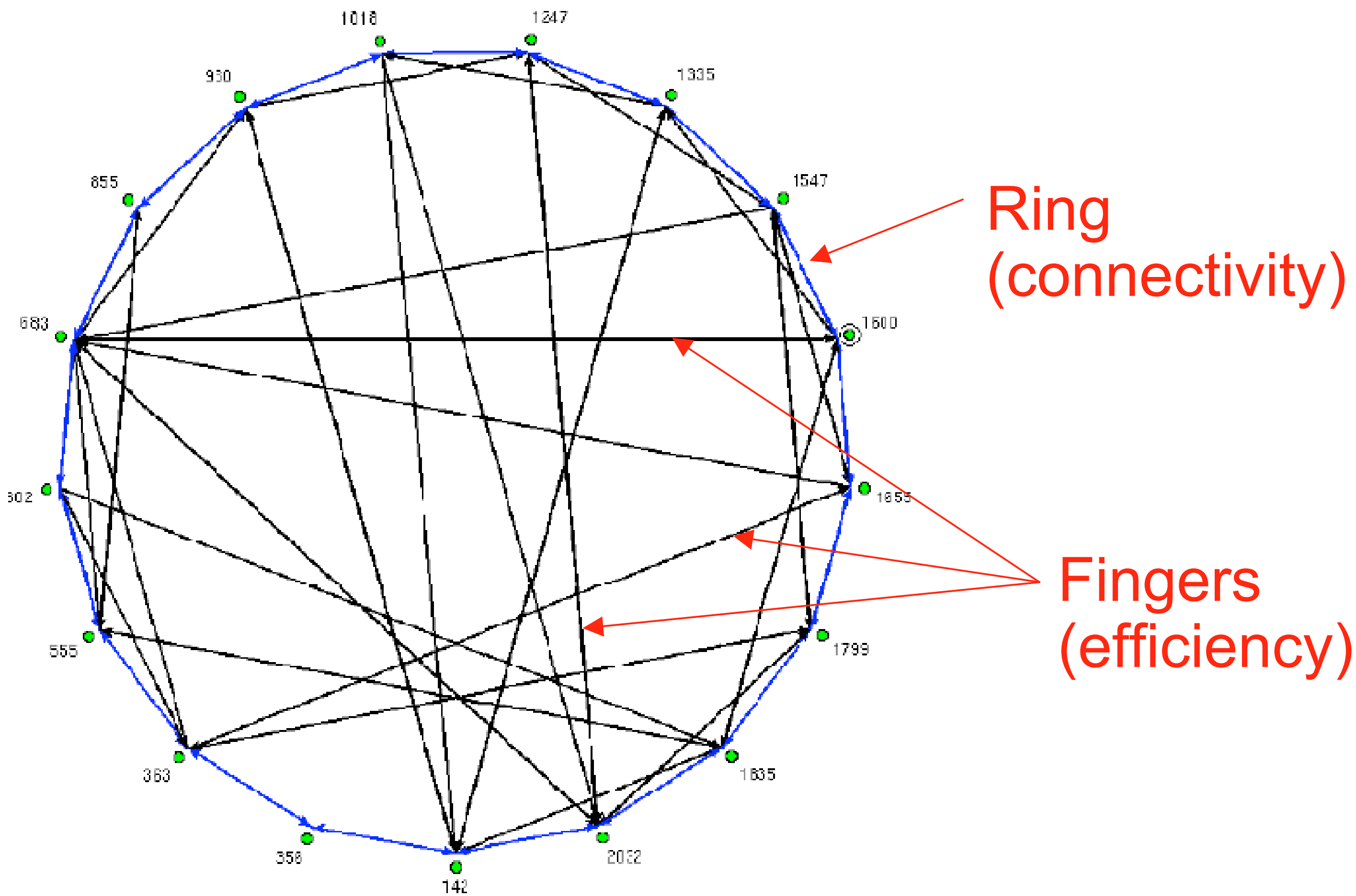
Update routing tables continuously

Replicate data for reliability

- Guarantees: fast routing, finding the item

THE ADVENTURES OF SELFMAN

Based on a ring topology



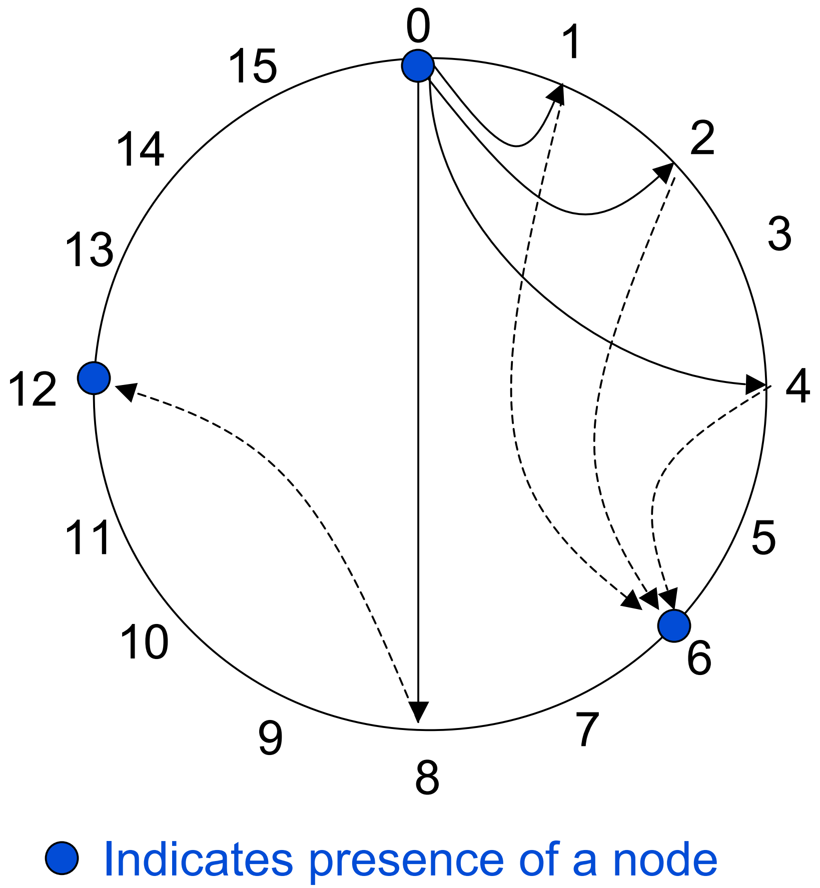
Lookup illustrated in Chord

We illustrate lookup in Chord, a simple SON. Nodes sparsely populate a circular identifier space.

Given a key, find the value associated to the key (here, the value is the IP address of the node that stores the key)

Assume node 0 searches for the value associated to key K with identifier 7

Interval	node to be contacted
[0,1)	0
[1,2)	6
[2,4)	6
[4,8)	6
[8,0)	12



Where are SONs used?

- Internet Architecture
  - Routing On Flat Labels (ROFL) [sigcomm'06]
- Mobility
  - Session Initiation Protocol, Host Identity Protocol (HIP), I3, …
- File sharing and Streaming
  - e-Mule, Azureus, PPLive [sigcomm'07], …
- Application Servers
  - amazon.com DYNAMO [sosp'07]
- Other uses
  - databases (PIER), DFS (WheelFS [sosp'07], …), caches (squirrel, …)

Relaxed ring algorithm

Ring maintenance

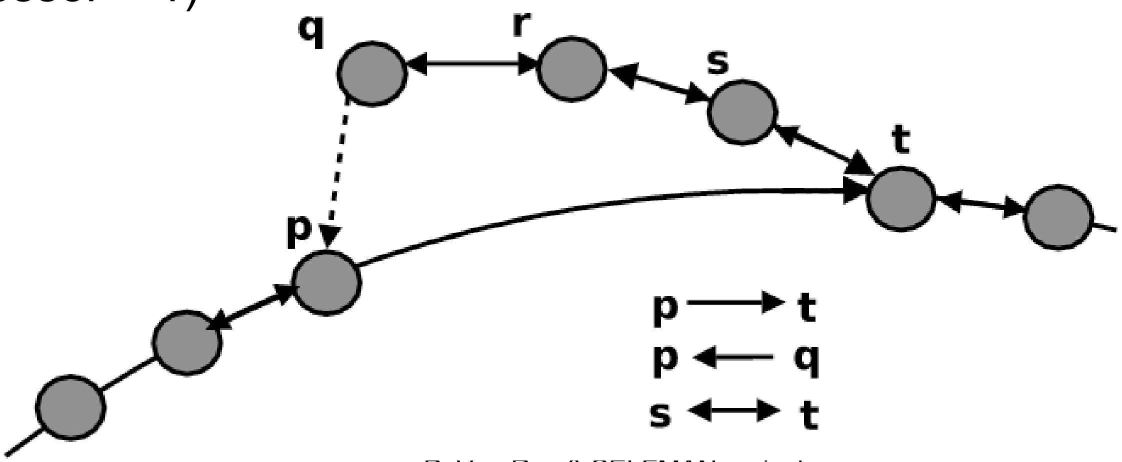
- In a SON based on a ring topology, self organization is done at **two levels**:
  - The ring ensures connectivity (correctness): it must always exist despite joins, leaves, and failures
  - The fingers reduce number of routing hops (efficiency): they can be temporarily in an inconsistent state
- The relaxed ring algorithm improves the connectivity maintenance
  - It has improved behavior for failures
  - It greatly reduces the probability of inconsistent lookups

Connectivity maintenance

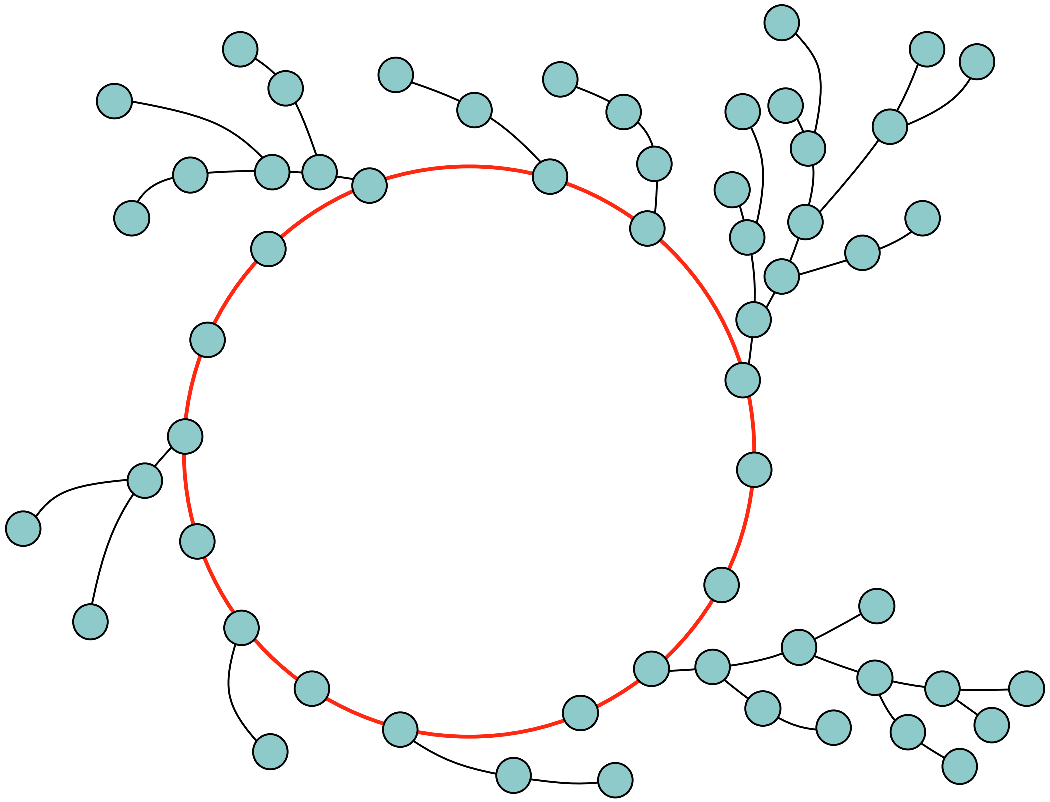
- Connectivity maintenance is not trivial
  - Peers can join and leave at any time
  - Peers that crash are like peers that leave but without notification
  - Temporarily broken links create false suspicions of failure
- Crucial properties to be guaranteed
  - Lookup consistency
  - Ring connectivity

The relaxed-ring architecture

- The ring is constructed using an invariant:
  Every peer is in the same ring as its successor
- Connectivity maintenance is completely asynchronous
- Nodes communicate through message passing
  - For a join, instead of one step involving 3 peers (as in DKS, also developed in SELFMAN), we have two steps each with 2 peers → we do not need locking
- A peer can never indicate another peer as the responsible node (a peer knows only its own responsibility, which starts with the key of the predecessor + 1)



Example of a relaxed ring



- It looks like a ring with “bushes” sticking out
- The bushes appear only if there are failure suspicions
  - Usually the ring is not as bushy as in this example!
- There always exists a perfect ring (in red) as a subset of the relaxed ring.
- The relaxed ring is always converging toward the perfect ring
  - The number of bushes existing at any time depends on the churn (rate of change of the ring, failures/joins per time)

Lookup consistency

- **Definition:** Lookup consistency means that at any time there is exactly one responsible node for a particular key ***k***
  - Lookup consistency is difficult to achieve
    - Strong (atomic) data consistency, availability, and partition tolerance are impossible to achieve simultaneously (Brewer's conjecture)
  - What can we do in the case of the Internet's failure model?
    - Crash failures of nodes and networks and false failure suspicions
    - Eventually perfect failure detection
- **Theorem:** The relaxed-ring join algorithm guarantees lookup consistency at any time in presence of multiple joining peers
  - This is not true for many other SONs, e.g., Chord
- **Theorem**: Multiple failing peers never introduce inconsistent lookup unless the network is partitioned
  - In practice, the probability of inconsistency is vastly reduced

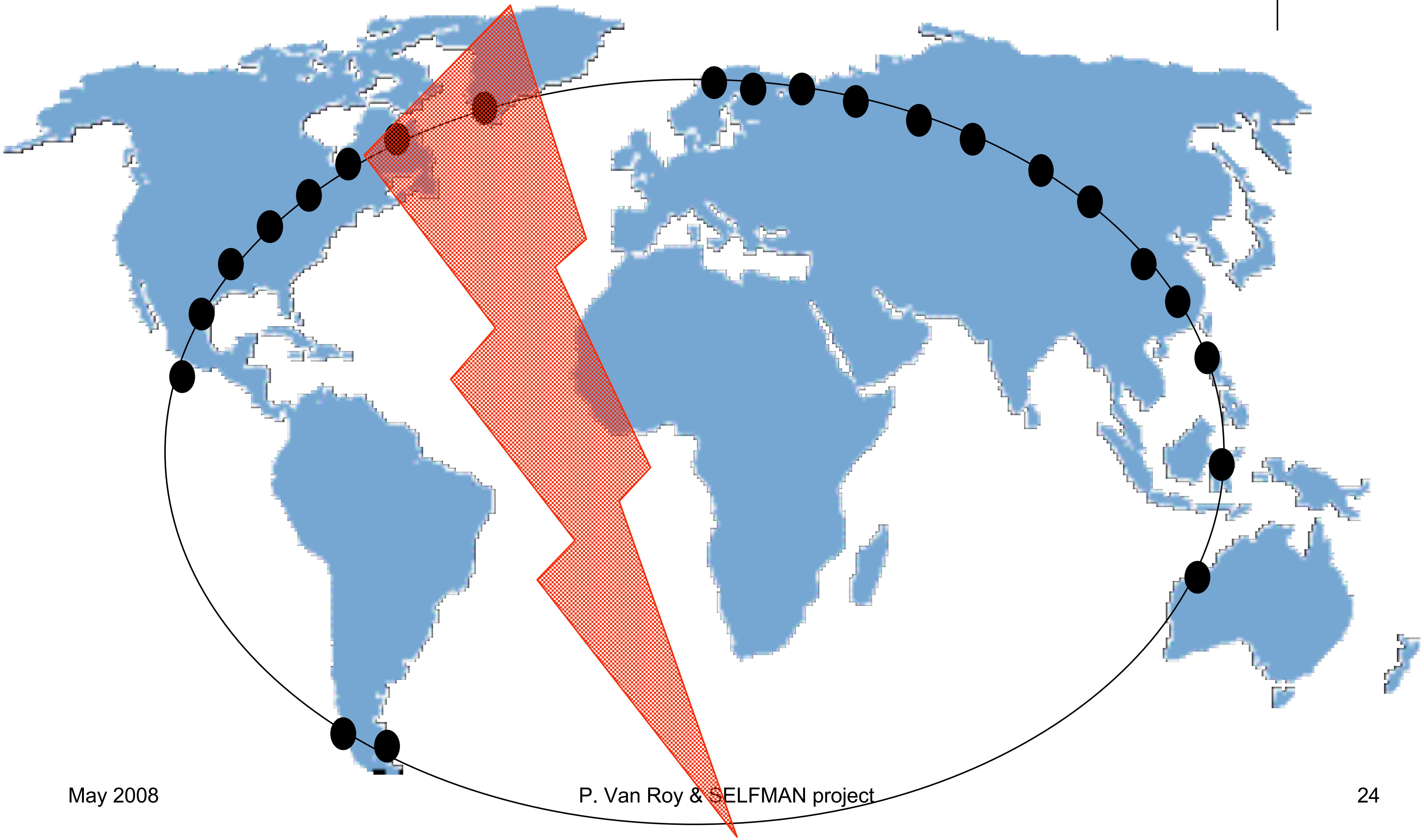
THE ADVENTURES OF SELFMAN

Ring merge algorithm

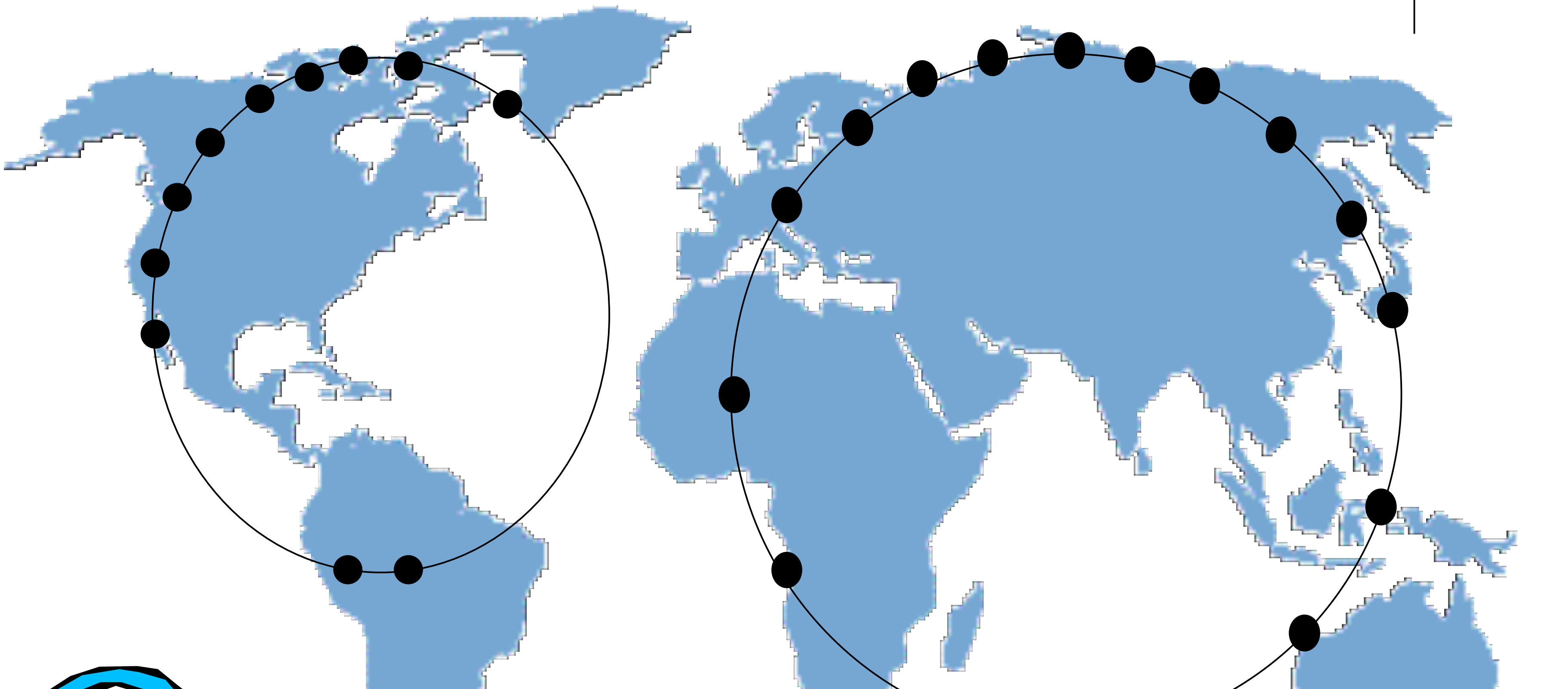
Problem statement

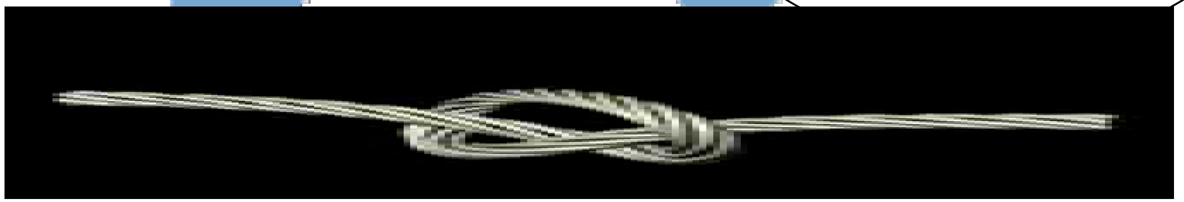
- Network partitions occur frequently
  - Often small, occasionally large
- Any long-lived DHT will experience partitions
  - Problem barely studied at all
- This is an important problem
  - Studied in other contexts: databases (80s), file systems (90s)

Real world example



Real world example





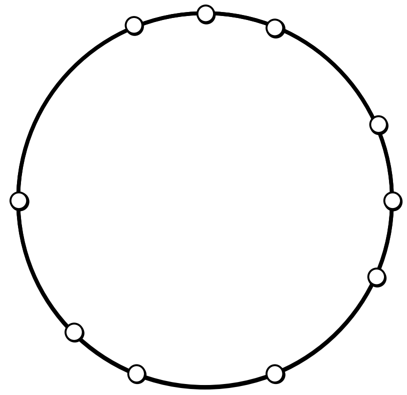
Current beliefs about partitions & SONs are wrong!

- Ring-based DHTs ”cannot function at all until the whole merge process is complete”, Datta *et al.* [iwsos’06, best paper award]

- Ring-based SONs are inherently ill-suited for dealing with network partitions, Ken Birman [gossip-leiden’06]

Existing systems

- Most existing DHTs survive network partitions



- How to efficiently merge several rings?
  - Automatic merge when partition detected
  - Manual merge decided by external management

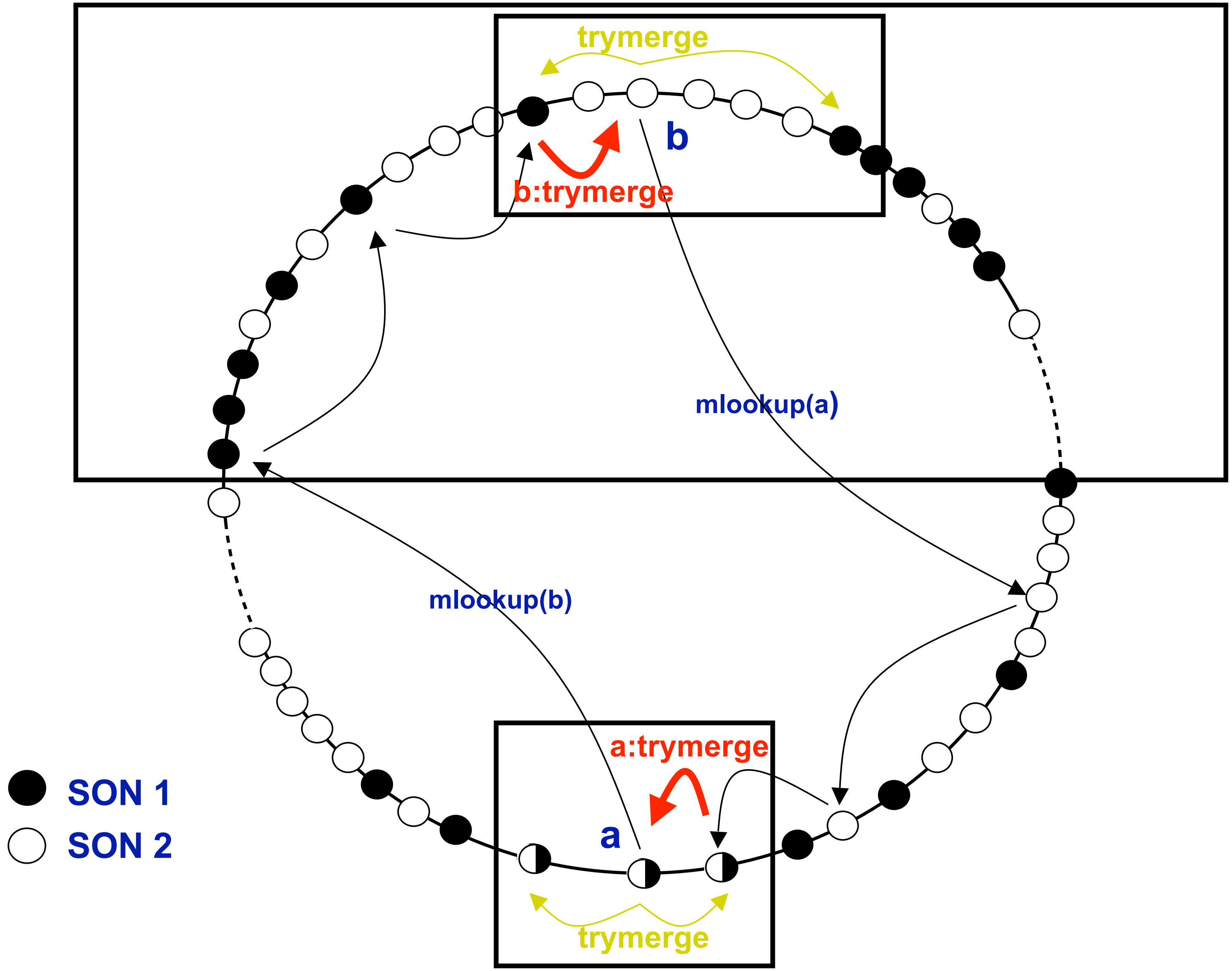
Automatically detecting need for a merge

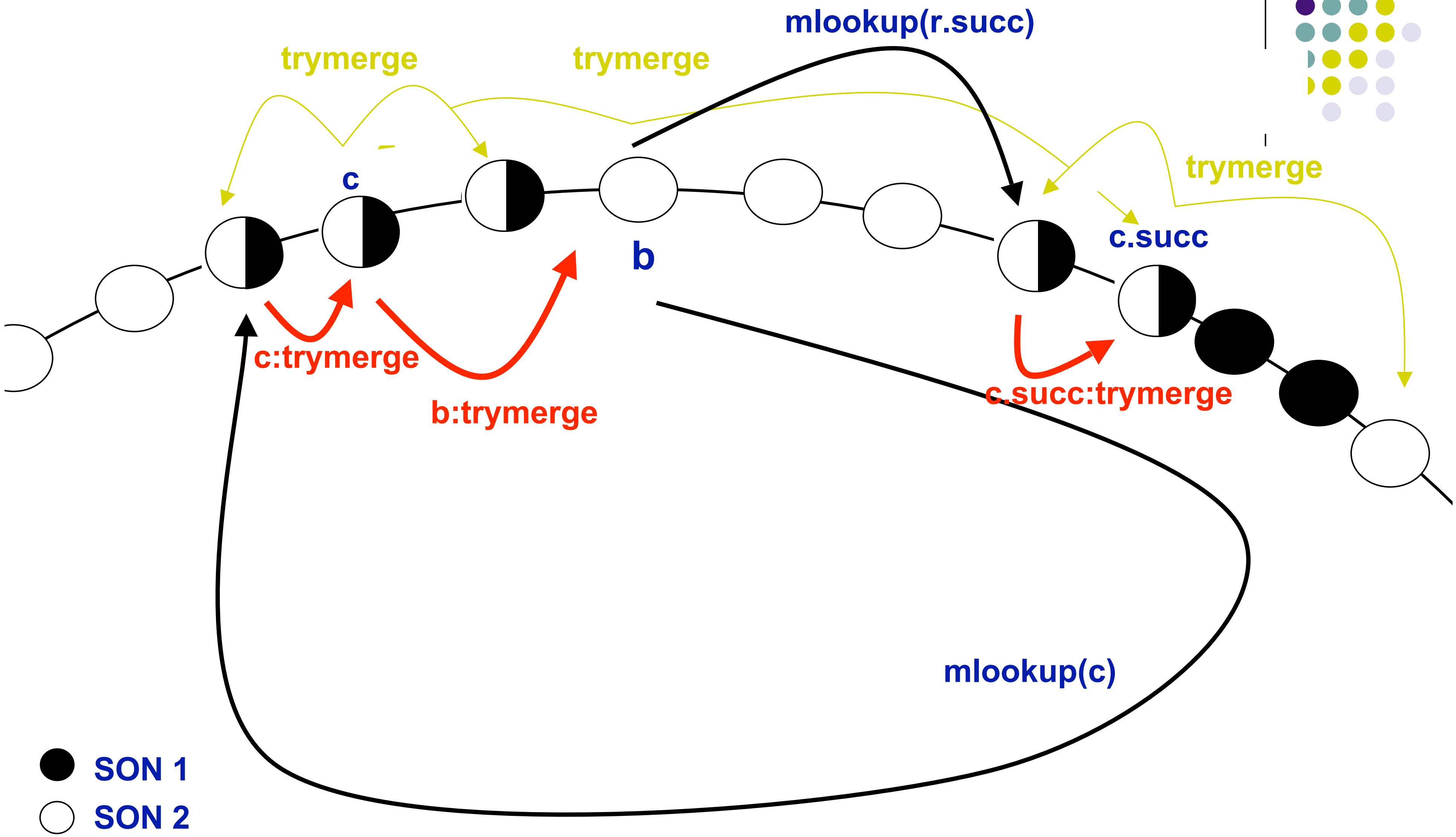
- Each node maintains a *passive list*
  - Stores every locally stored crashed node
- Ping passive lists periodically
  - Alive node indicative of merger
- If passive list contains no live node
  - “Kick start” the merge by using external mechanism to add one node to passive list

THE ADVENTURES OF SELFMAN

Simple ring unification algorithm

- Assume **a** detects **b** on a different ring
- a calls **mlookup(b)**
  - **mlookup** traverses ring to get close to **b**
  - It then calls **trymerge(cpred, csucc)**
- **trymerge(cpred, csucc)**
  - Try merging by updating pointers to candidates (cpred, csucc)
  - Recursively call **mlookup** to continue the merger


trymerge
b
b:trymerge
mlookup(a)
mlookup(b)
a:trymerge
a
trymerge
SON 1
SON 2

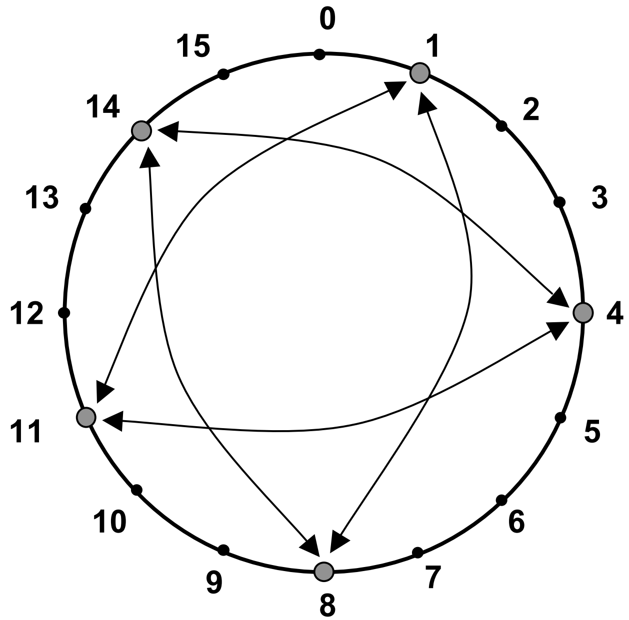

mlookup(r.succ)
trymerge
trymerge
trymerge
c
b
c.succ
c:trymerge
b:trymerge
c.succ:trymerge
mlookup(c)
SON 1
SON 2

Improved algorithm

- The algorithm needs linear time to merge rings
  - This can be acceptable:
    - Partitions are rare
    - Let algorithm run in background
    - Low cost, low performance
- With gossiping we improve the algorithm to merge rings in logarithmic average time
  - Let detecting node share info with M random nodes
  - Caveat: node does not know M random nodes
    - Spread this process during the merger

Fixing "loopy rings"

- Sometimes DHTs end up in a *loopy ring*



- The gossip algorithm can recover from loopy rings

Distributed transactions

Transactions on a SON

- Transactions on a SON are challenging because of high churn:
  - Frequent node leaves, crashes, and joins
  - Results in changing data responsibilities of nodes
- We use a crash stop failure model
- We assume an eventually perfect failure detector
  - Failure detection on Internet is notoriously difficult
  - We use a majority algorithm based on a modified Paxos
  - Inconsistent lookups are hidden by the majority algorithm
- We build the transactions on top of a reliable storage service that uses symmetric replication

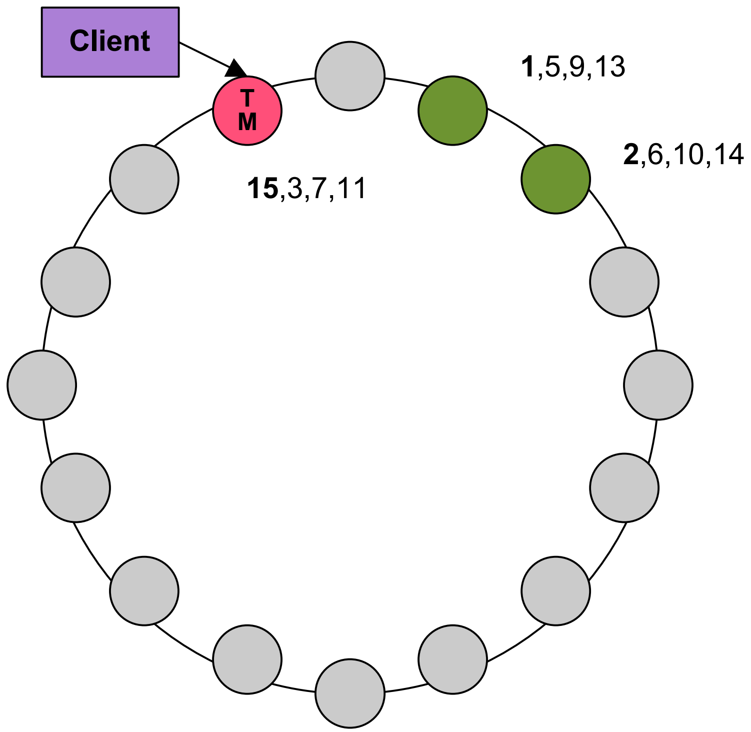
Concurrency control

- **Pessimistic CC** is used …
  - in scenarios with high contention
  - in DBs (with in *crash recovery* model)

- **Optimistic CC** is used …
  - in scenarios with low contention
  - when long network latencies cause much blocking

Atomic commit on a SON

Start of validation phase

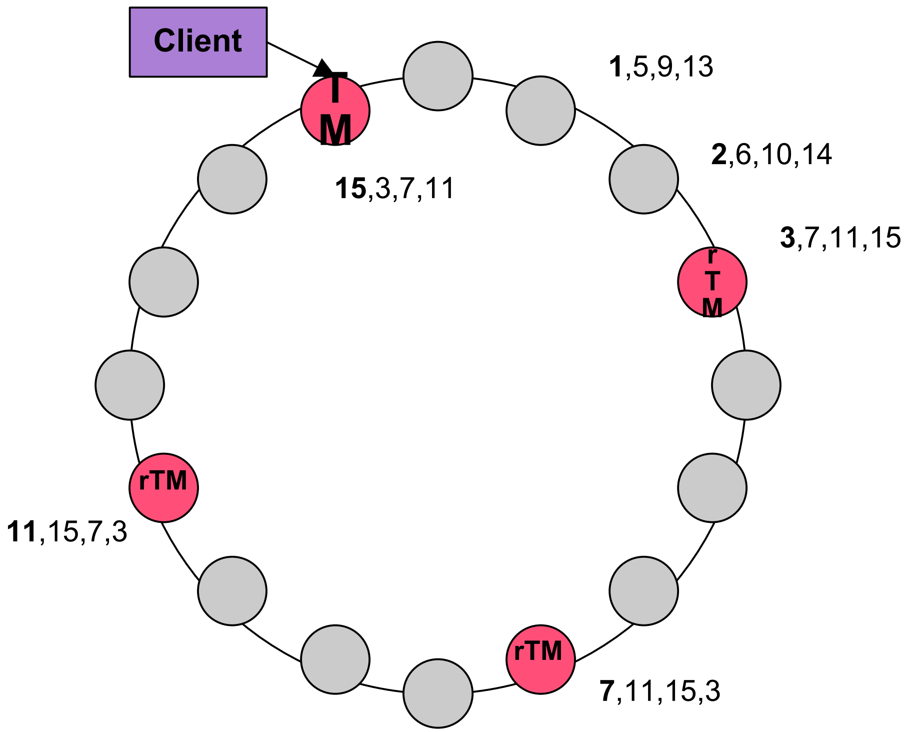


Client:

BOT
Write item(1)
Write item(2)
EOT

- Client **asks nearest node**, e.g. node 15
- Node 15 becomes the **Transaction Manager (TM)**
- TM creates a **transaction item** with a key for which it is responsible for (e.g. key = 15)

Atomic commit on a SON

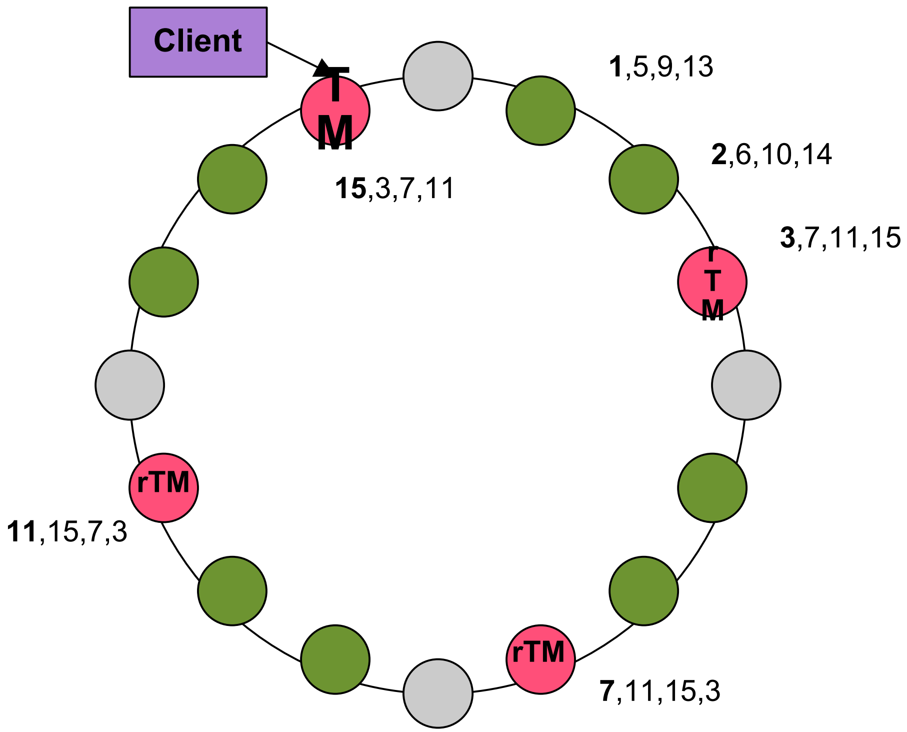


Client:

BOT
Write item(1)
Write item(2)
EOT

- Assuming **symmetric replication**, let the replication degree **f = 4**
- Nodes 3, 7, 11 become **replicated Transaction Managers (rTM)**, according to the replication of the transaction item

Atomic commit on a SON



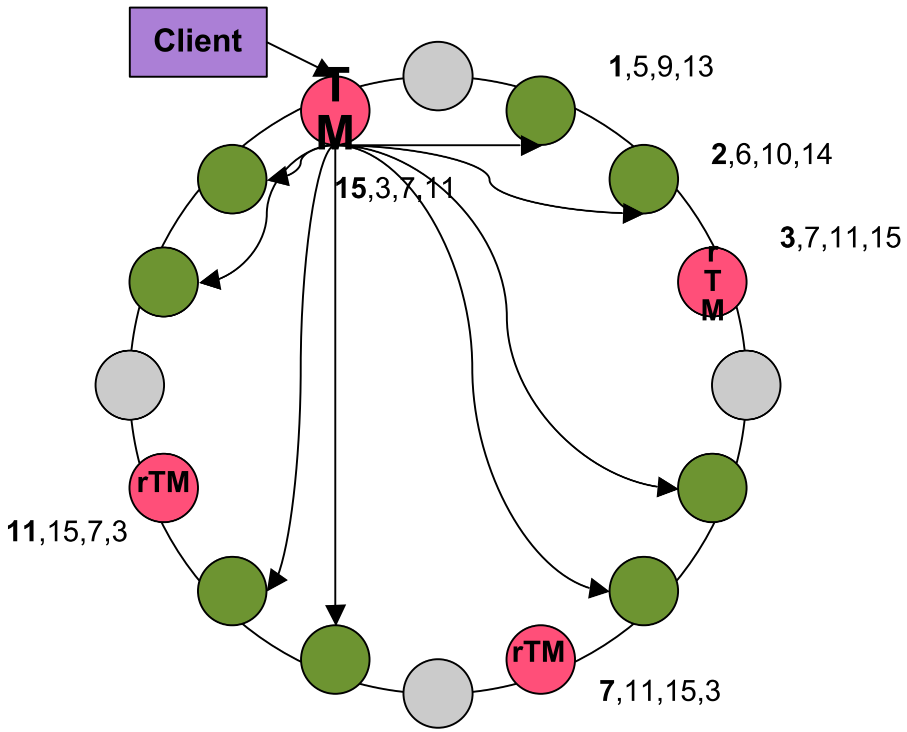
Client:

BOT
Write item(1)
Write item(2)
EOT

- Nodes 1, 5, 9, 13 and 2, 6, 10, 14 become **Transaction Participants (TP)**

1. and 2. Step

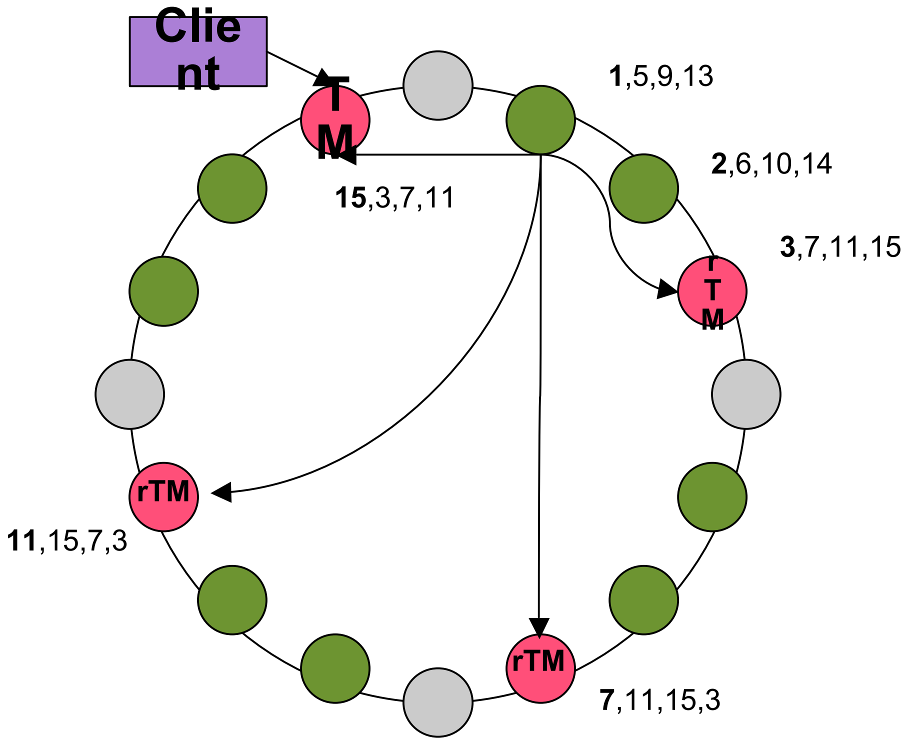
Atomic commit on a SON



- When the transaction is complete, we start the atomic commit algorithm
- TM **sends "Prepare"** together with the information needed for validation to all TPs

3. Step

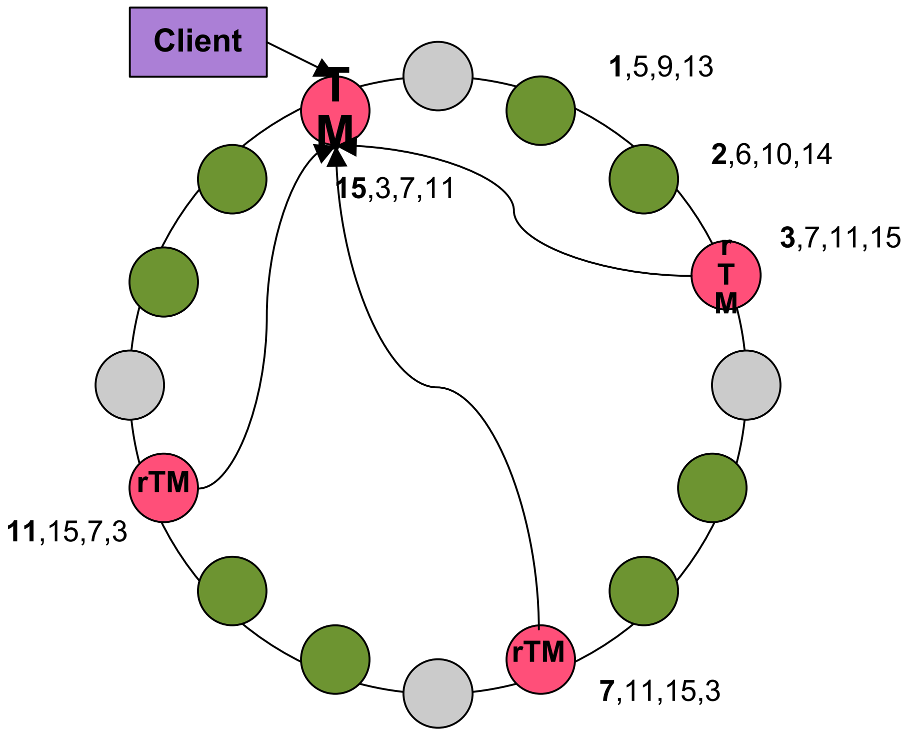
Atomic commit on a SON



- After having received "Prepare" from the TM, **each TP sends a "Prepared"** or **"Abort"** message **to all rTMs**

4. Step

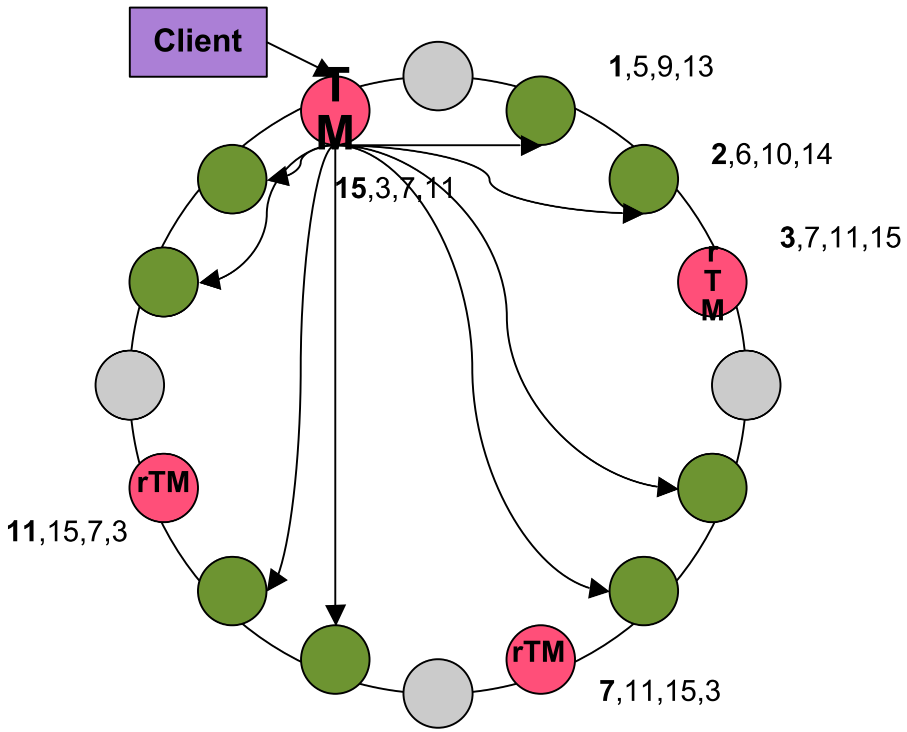
Atomic commit on a SON



- The **rTMs collect votes** from a majority of TPs per item and **locally decide** on abort or commit
- **Each rTM** sends the outcome to the **TM**

5. Step

Atomic commit on a SON



- The **TM collects the outcome** from at least a majority of rTMs
- After having collected a **majority**, the TM sends the **decision** to all TPs
- If the TM fails, this is detected and a new leader is chosen

6. Step

Current status

- Performance
  - 6 communication rounds
- Succeeds if more than f/2 nodes alive
  - Time outs are not used
- Simulations in progress
  - For validating assumptions and performance
- Implementations
  - Transaction algorithm and Distributed Wiki application implemented in Erlang at ZIB
    - *This implementation won first prize in the First IEEE International Scalable Computing Challenge (SCALE 2008) (May 2008)*
  - Implementations in progress on PlanetLab/EverLab and using network simulator

Conclusions

THE ADVENTURES OF SELFMAN

Conclusions

- Structured overlay networks are a good starting point for building large-scale self-managing systems
- Current SON research is *almost* mature enough for building self-management architectures
  - We have fixed the main problems: network merge and lookup consistency
- We are currently implementing and evaluating a replicated transactional storage algorithm
  - Majority algorithm (modified Paxos for atomic commit) together with network merge seems to be adequate to deal with Internet failure model
  - We implemented a distributed Wiki using this algorithm which won first prize in the First IEEE International Scalable Computing Challenge (SCALE 2008).
- This work is being done as part of the SELFMAN project
  - See **www.ist-selfman.org**