Building large-scale distributed applications on top of self-managing transactional stores

June 3, 2010

Peter Van Roy
with help from SELFMAN partners

Overview

- Large-scale distributed applications
  - Application structure: multi-tier with scalable DB backend
  - Distribution structure: peer-to-peer or cloud-based
- DHTs and transactions
  - Basics of DHTs
  - Data replication and transactions
  - Scalaris and Beernet
- Programming model and applications
  - CompOz library and Kompics component model
  - DeTransDraw and Distributed Wikipedia
- Future work
  - Mobile applications, cloud computing, data-intensive computing
  - Programming abstractions for large-scale distribution

European Commission
Information Society

THE ADVENTURES OF SELFMAN

Application structure

- What can be a general architecture for large-scale distributed applications?
- Start with a database backend (e.g., IBM's "multitier")
  - Make it distributed with distributed transactional interface
  - Keep strong consistency (ACID properties)
  - Allow large numbers of concurrent transactions
- Horizontal scalability is the key
  - Vertical scalability is a dead end
  - "NoSQL": Buzzword for horizontally scalable databases that typically don't have a complete SQL interface
    - *Key/value store* or column-oriented

↑ **our choice (simplicity)**

European Commission

Information Society

The NoSQL Controversy

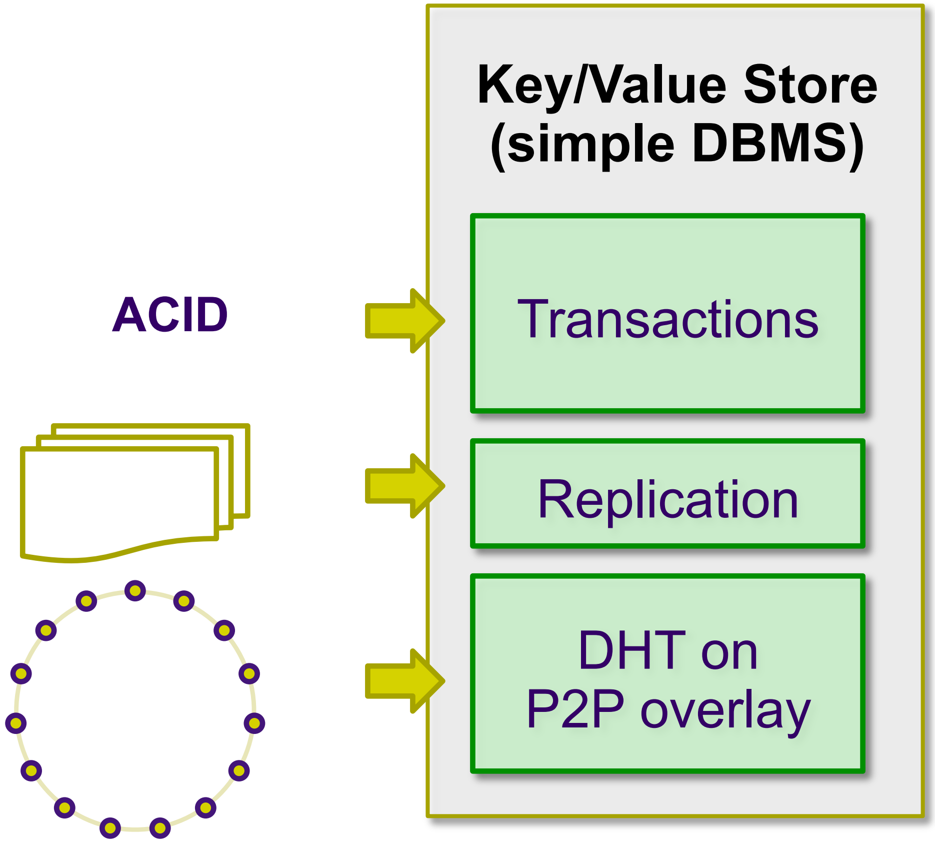
- NoSQL is a current trend in non-relational databases
  - May lack table schemas, may lack ACID properties, no join operations
  - Main advantages are excellent performance, with good horizontal scalability and elasticity (ideal fit to clouds)
    - SQL databases have good vertical scalability but are not elastic
- Often only weak consistency guarantees, such as eventual consistency (e.g., Google BigTable)
  - Some exceptions: Cassandra also provides strong consistency, Scalaris and Beernet provide a key-value store with transactions and strong consistency

THE ADVENTURES OF SELFMAN

Distribution structure

- Two main infrastructures for large-scale applications
- Peer-to-peer: use of client machines ← **our choice (loosely coupled)**
  - Very popular style, e.g., BitTorrent, Skype, Wuala, etc.
  - Different degrees of organization (unstructured to structured)
  - Supports horizontal scalability
- Cloud-based: use of datacenters (another good choice)
  - Becoming very popular too, e.g., Amazon EC2, Google AppEngine, Windows Azure, etc.
  - Supports horizontal scalability
  - Also supports elasticity
- Hybrids will appear
  - Combine elasticity & high availability of clouds with high aggregate bandwidth & low latency of peer-to-peer

Architecture



- This is the final architecture that we have built for large-scale distributed applications
- Distributed transactions provide consistency and fault tolerance
- The whole is built in modular fashion using concurrent components
- Each layer has self-managing properties
- We explain how it works and give some of the applications

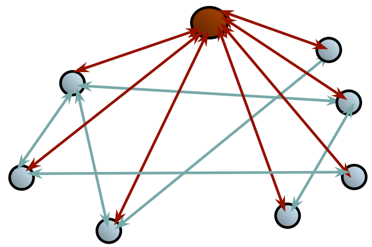
European Commission

Information Society

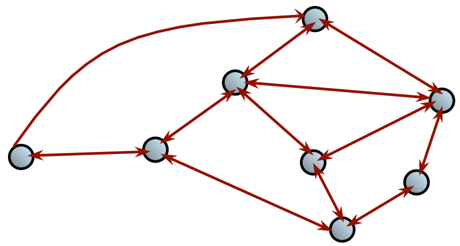
Distributed Hash Tables

DHTs: third generation of P2P

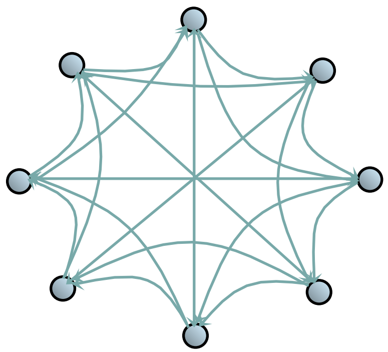
- Hybrid (client/server)
  - Napster
- Unstructured overlay
  - Gnutella, Kazaa, Morpheus, Freenet, …
  - Uses flooding
- Structured overlay
  - Exponential network with augmented ring structure
  - **DHT (Distributed Hash Table)**, e.g., Chord, DKS, Scalaris, Beernet
  - Self-organizes upon node join/leave/failure



R = N-1 (hub)
R = 1 (others)
H = 1



R = ? (variable)
H = 1…7
(but no guarantee)



R = log N
H = log N
(with guarantee)

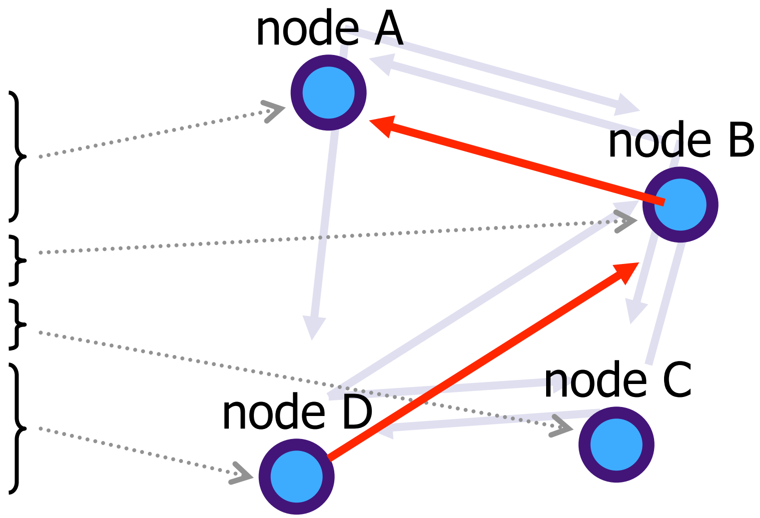
European Commission

Information Society

DHT functionality

- A dynamic distribution of a *hash table* onto a set of cooperating nodes

Key	Value
1	Algorithms
9	Routing
11	DS
12	Peer-to-Peer
21	Networks
22	Grids

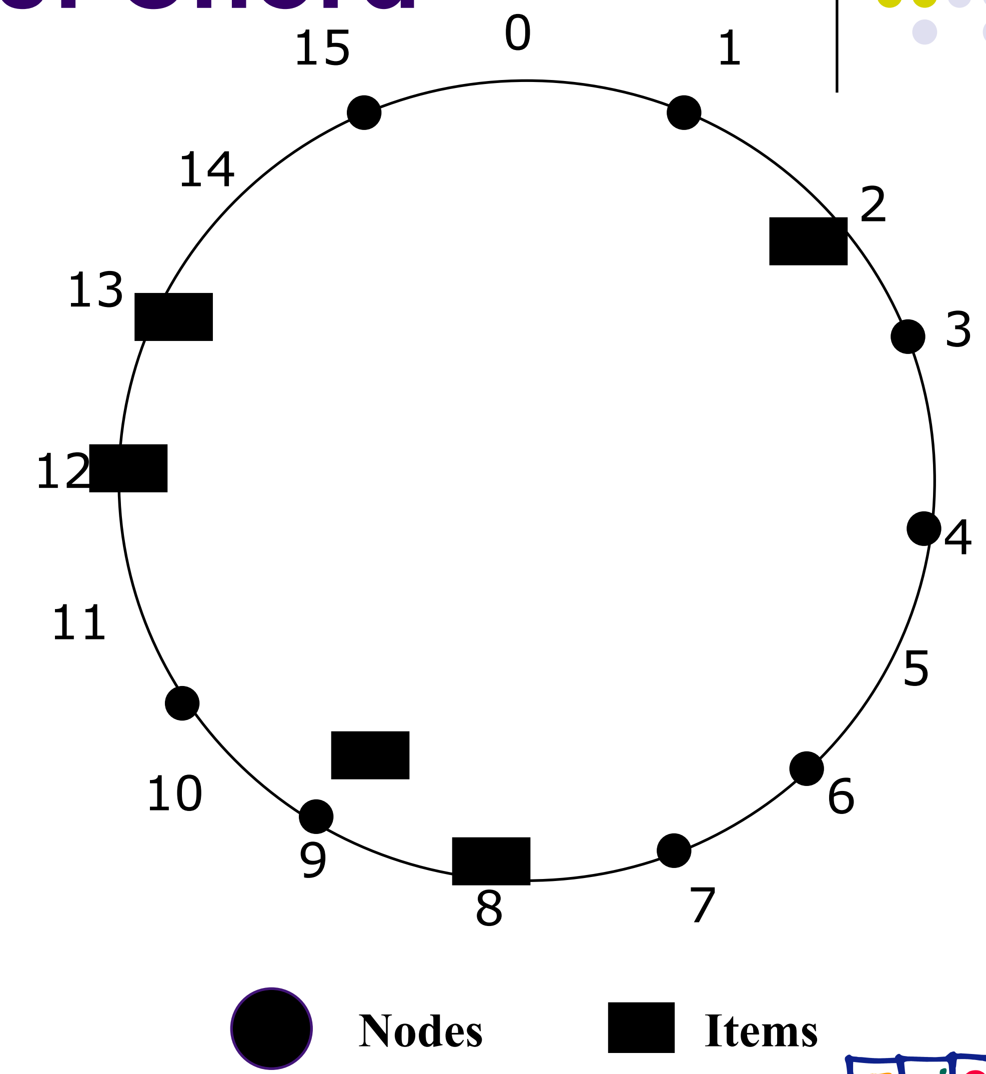


→ **Node D : get(9)**

- Hash table: get/set/delete operations
- Each node has a routing table
  - Pointers to some other nodes
  - Typically, a constant or a logarithmic number of pointers
- Fault tolerance: reorganizes upon node join/leave/failure

A DHT Example: Chord

- Ids of nodes and items are arranged in a circular space
- An item id is assigned to the first node id that follows it on the circle.
- The node at or following an id on the space (circle) is called the successor. This gives a connected ring.
- Not all possible ids are actually used (sparse set of ids, e.g., 2^{128})!
- Extra links, called fingers, are added to provide efficient routing

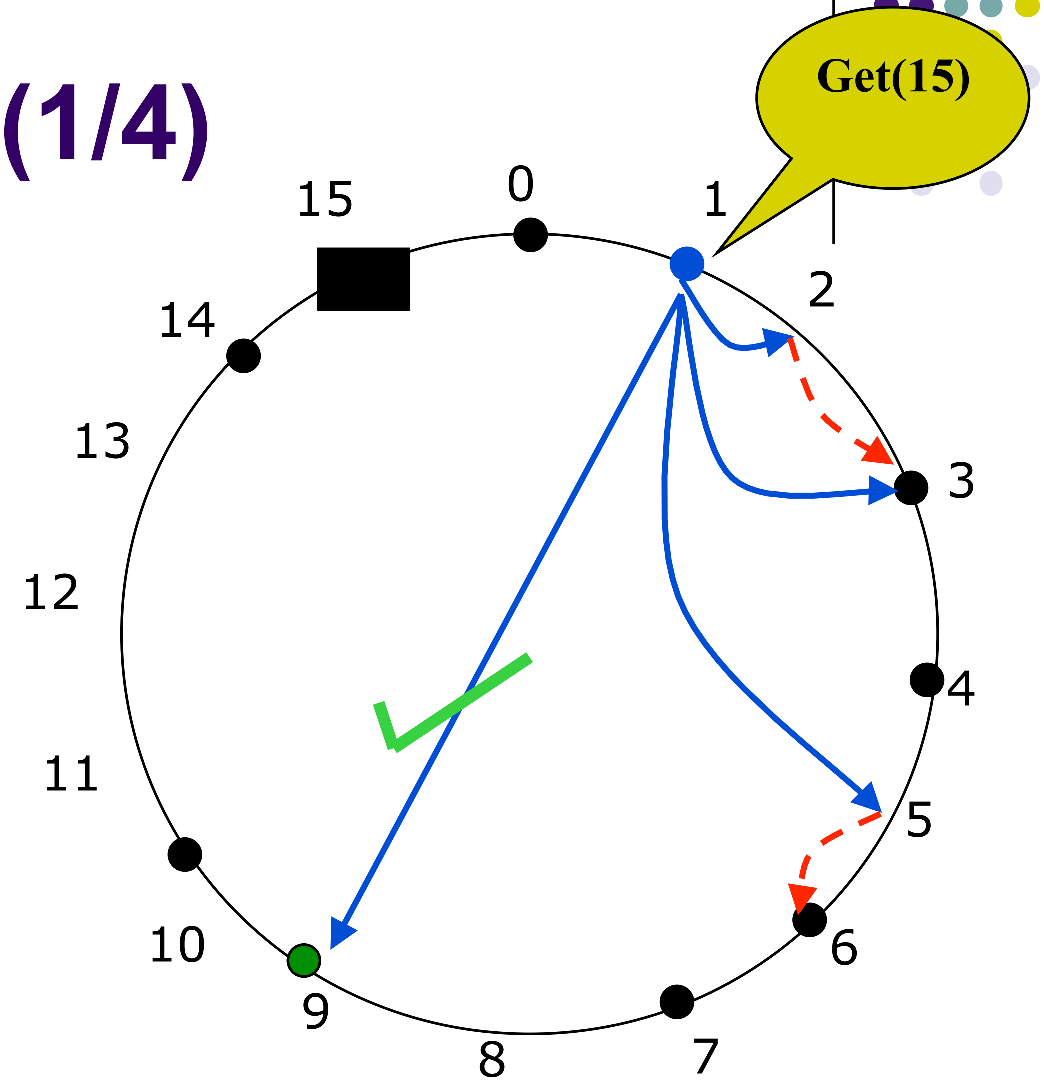


DHT self-maintenance

- In all ring-based DHTs inspired by Chord, self-organization is done at two levels:
  - The ring ensures connectivity: it must always exist despite joins, leaves, and failures
  - The fingers provide efficient routing: they may be temporarily in an imperfect state, but this affects only the efficiency of routing, not the correctness
- We now explain how routing works
  - We will explain connectivity maintenance later when we introduce the relaxed ring
  - The relaxed ring has much simpler connectivity maintenance than Chord

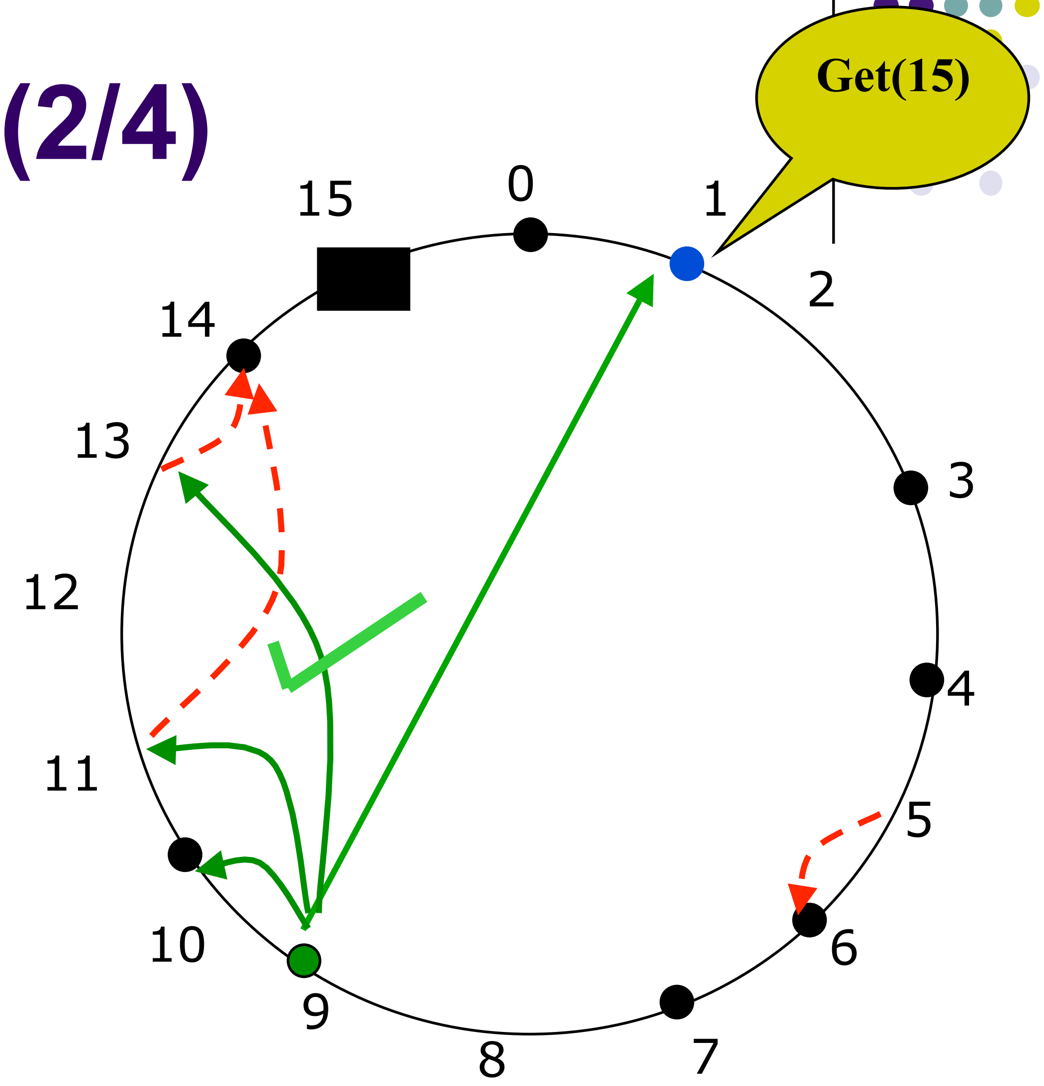
Chord routing (1/4)

- Routing table size: M, where $N = 2^M$
- Every node n knows successor $(n + 2^{i-1})$, for $i = 1..M$
- Routing entries = $log_2(N)$
- $log_2(N)$ hops from any node to any other node



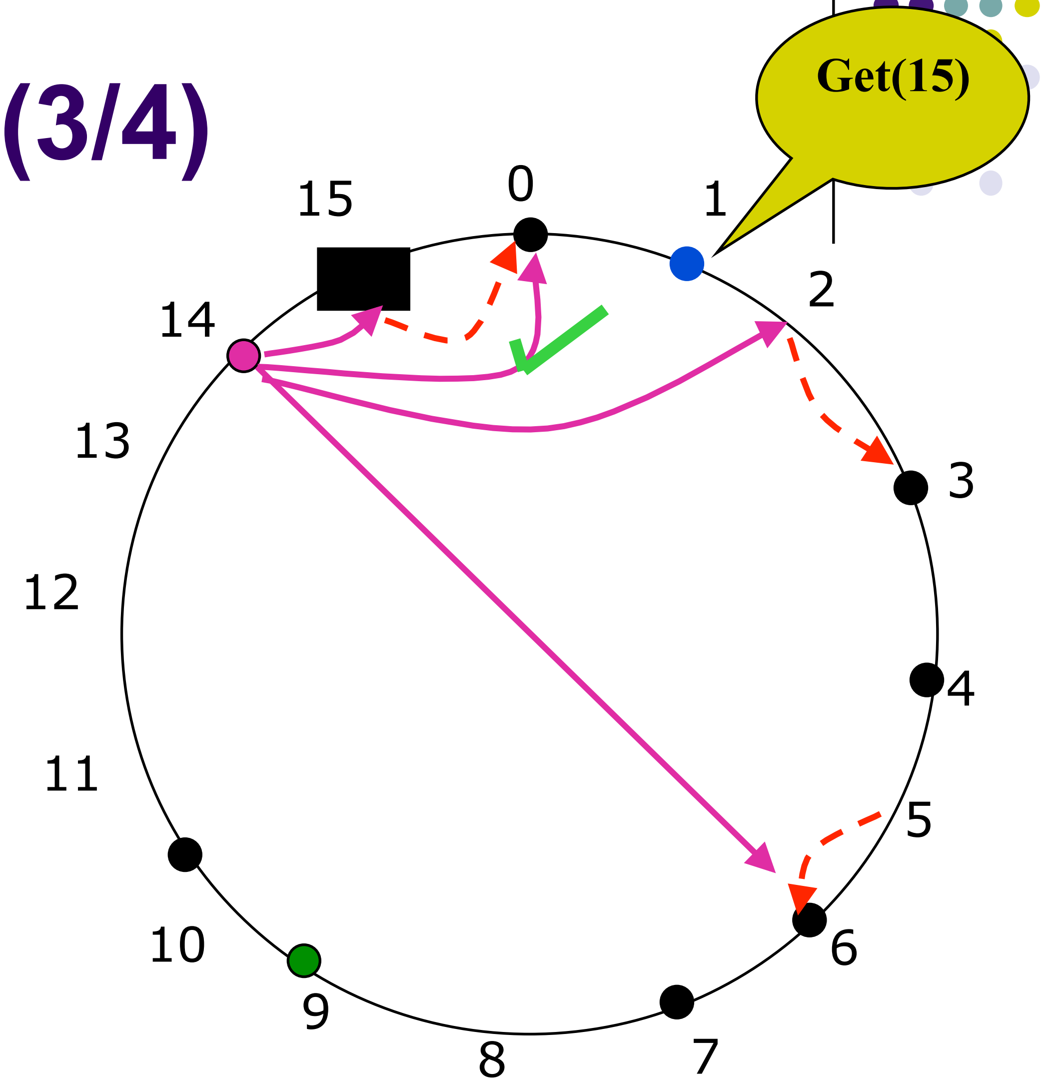
Chord routing (2/4)

- Routing table size: M, where $N = 2^M$
- Every node n knows successor ($n + 2^{i-1}$) , for $i = 1..M$
- Routing entries = $log_2(N)$
- $log_2(N)$ hops from any node to any other node



Chord routing (3/4)

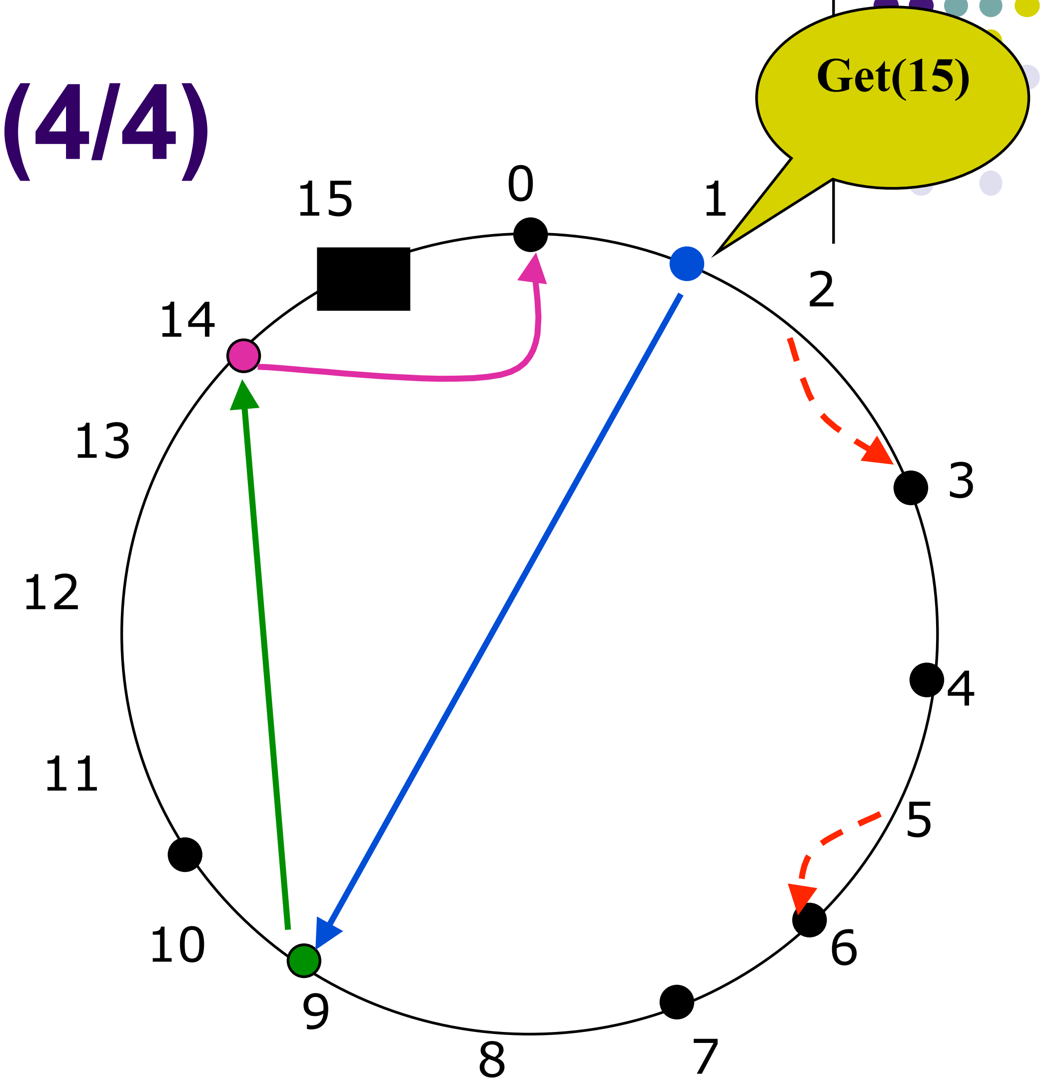
- Routing table size: M, where $N = 2^M$
- Every node n knows successor($n + 2^{i-1}$) , for $i = 1..M$
- Routing entries = $log_2(N)$
- $log_2(N)$ hops from any node to any other node



THE ADVENTURES OF SELFMAN

Chord routing (4/4)

- From node 1, it takes 3 hops to node 0 where item 15 is stored
- For 16 nodes, the maximum is $\log_2(16)$ = 4 hops between any two nodes



European Commission

Information Society

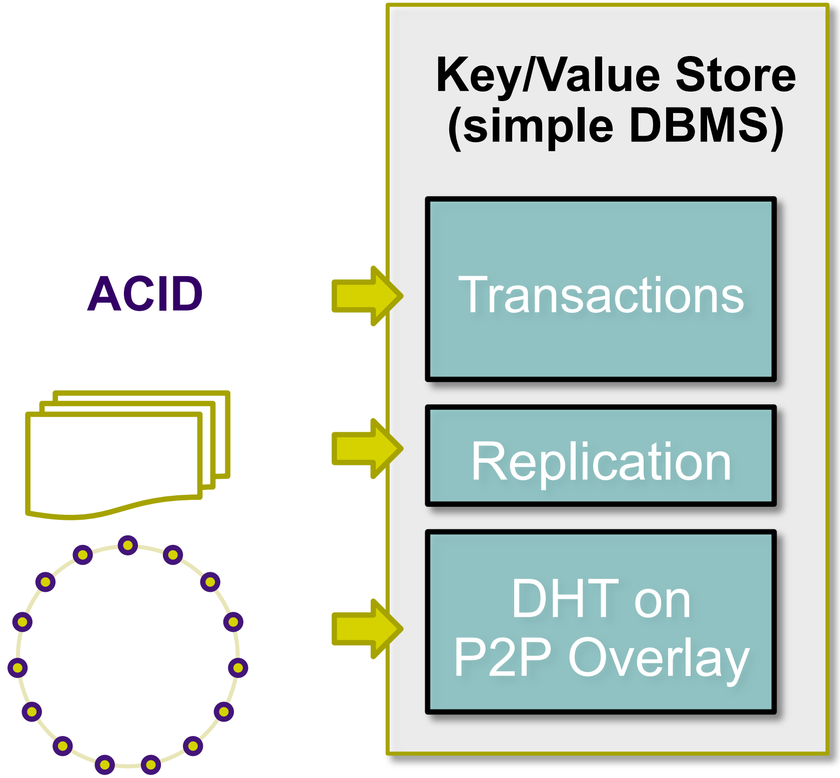
DHT-based Application Infrastructure

DHT-based application infrastructure

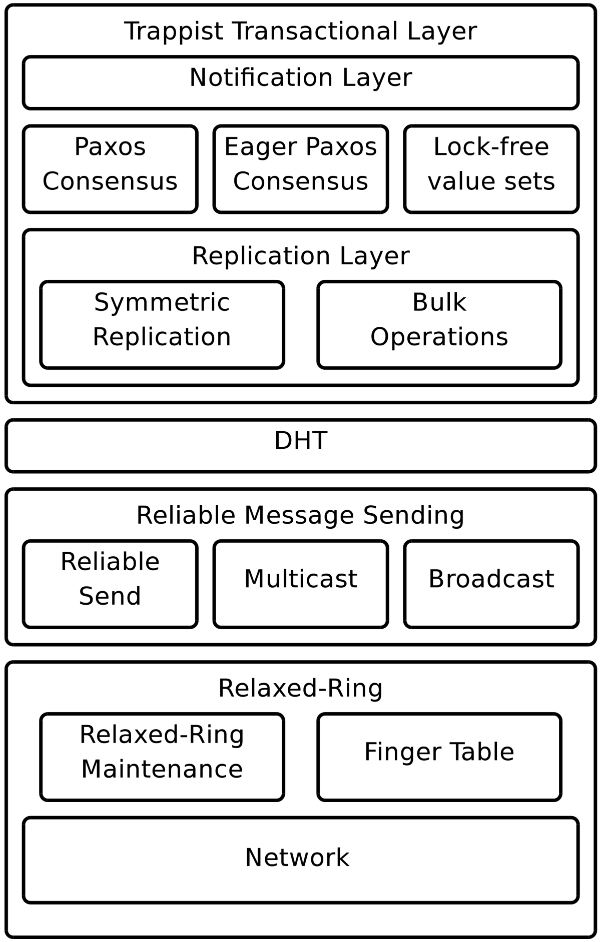
- We use the DHT as a foundation for building large-scale distributed applications
  - Using a concurrent component model with message passing
  - First layer: ring maintenance, efficient routing maintenance
  - Second layer: communication and storage
  - Third layer: replication and transactions
- A scalable decentralized application can be built on top of the transaction layer
- We built several applications using this architecture
  - Collaborative drawing (DeTransDraw), Distributed Wikipedia
  - As student project in a course: they complain it is too easy!

Scalaris and Beernet

- Scalaris and Beernet are key/value stores developed in the SELFMAN project (www.ist-selfman.org)
  - They provide transactions and strong consistency on top of loosely coupled peers using the Paxos uniform consensus algorithm for atomic commit
  - They are scalable to hundreds of nodes; with ten nodes they have similar performance as MySQL servers
  - Scalaris won first prize in the IEEE Scalable Computing Challenge 2008
- We focus on these two systems and the applications we have built on them



Detailed architecture



- Layered architecture
  - Relaxed ring and routing
  - Reliable message sending
  - DHT (basic storage)
  - Replication and transactions
- The relaxed ring maintains connectivity and efficient routing despite node failures, joins, and leaves
- The DHT provides basic storage without replication
- This figure shows the Beernet architecture; Scalaris is similar

Simplified ring maintenance

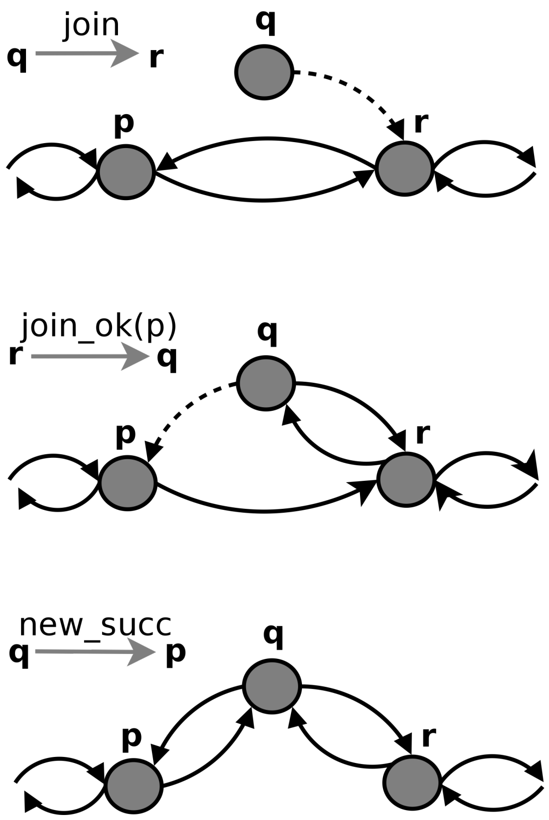
- We now continue our discussion of how DHTs work
- Ring maintenance is not a trivial issue
  - Peers can join and leave at any time
  - Peers that crash are like peers that leave without notification
  - Temporarily broken links create false failure suspicions
- Crucial properties to be guaranteed
  - Lookup consistency
  - Ring connectivity
- We define a relaxed ring which gives a very simple ring maintenance compared to Chord
  - E.g., no periodic stabilization needed like in Chord and many related structures

The relaxed-ring architecture

(0)

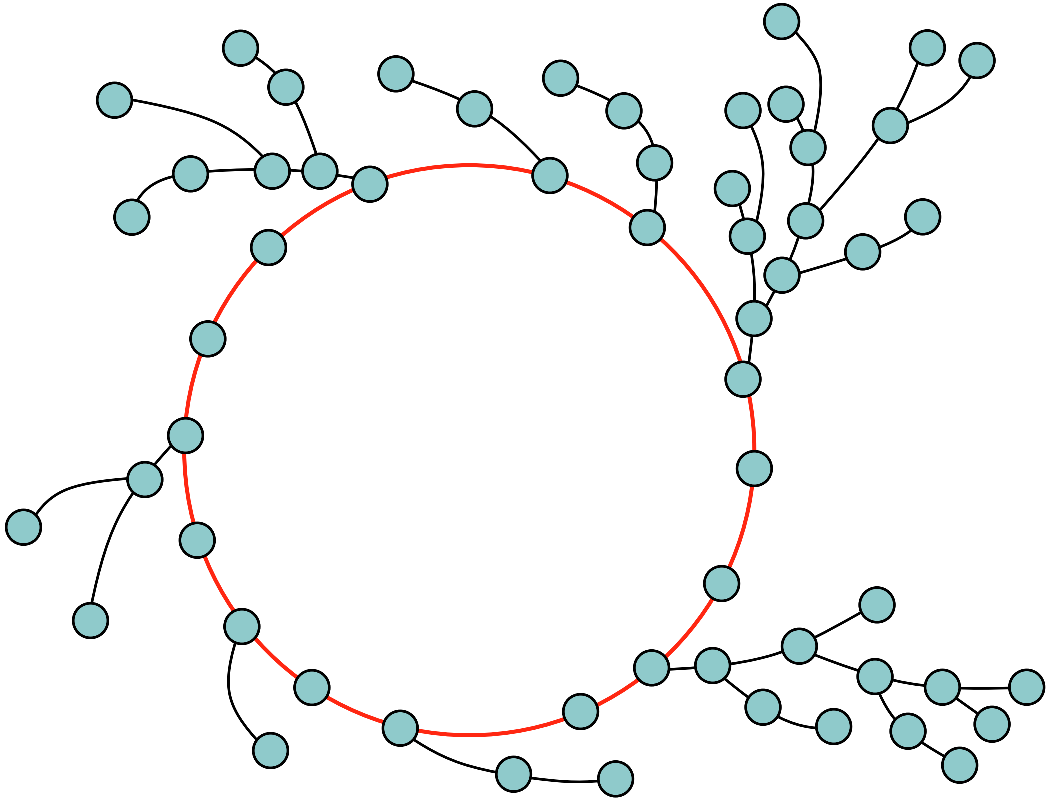
(1)

(2)



- The relaxed ring is the basis of the Beernet DHT
- The ring is based on a simple invariant:
  - Every peer is in the same ring as its successor
- Relaxed ring maintenance is completely asynchronous (no locking)
  - Joining is done in two steps, each involving two peers (instead of locking algorithm for insertion involving three peers as in Chord and DKS)
  - After first step, the node is in

Example of a relaxed ring

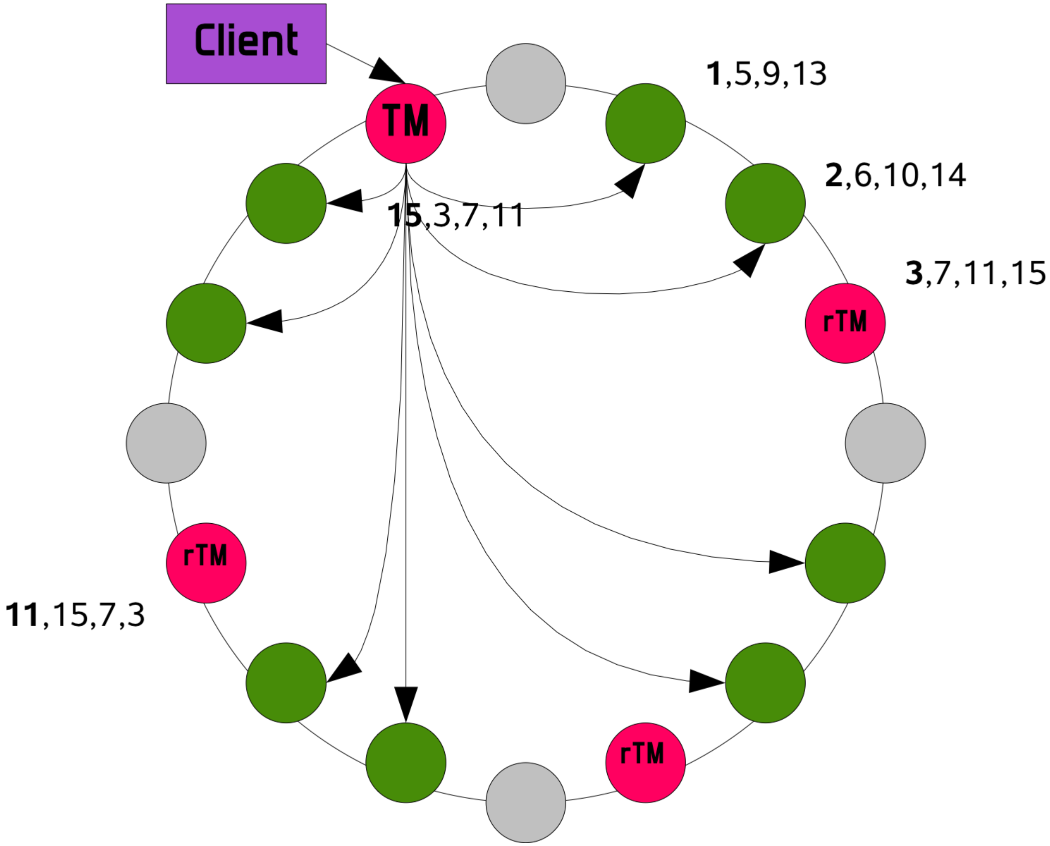


- It looks like a ring with “bushes” sticking out
- The bushes are long only for many failure suspicions
  - Average size of branch is less than one in typical executions
- There always exists a core ring (in red) as a subset of the relaxed ring. No branches means core ring = perfect ring.
- The relaxed ring is always converging toward a perfect ring
  - The size of bushes existing at any time depends on the churn (rate of change of the ring, failures/joins per time)

Lookup consistency

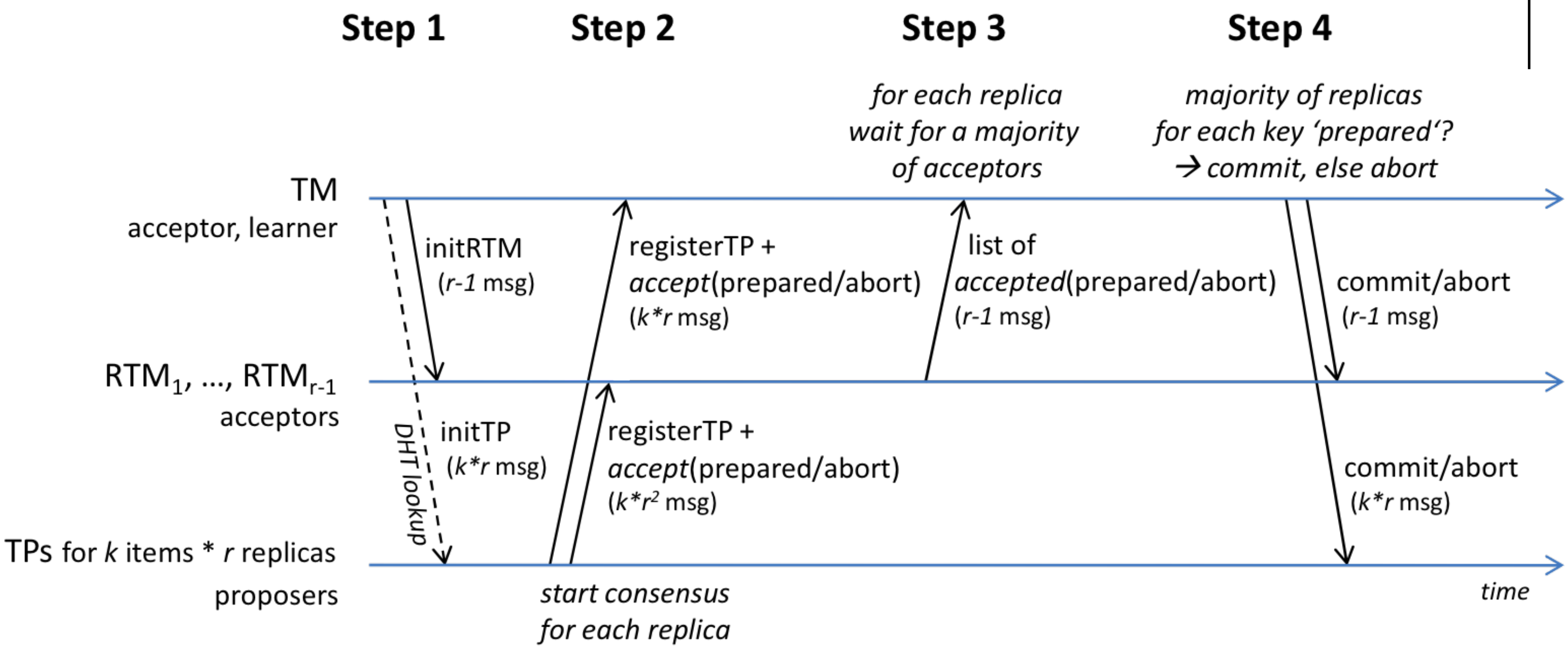
- **Definition:** Lookup consistency means that at any instant of time there is only one responsible node for a particular key ***k***
  - In the case of temporary failures (imperfect failure detection) lookup consistency cannot always be guaranteed: we may temporarily have more than one responsible node
  - Failure model: nodes may fail permanently and network links may fail temporarily, with eventually perfect failure detector (eventually accurate: false suspicion is possible, but only temporarily, strongly complete: failed nodes are always detected)
- **Theorem:** When there are no failures, the relaxed-ring join algorithm guarantees lookup consistency at any time for multiple joining peers
  - This is not true for Chord
- In realistic situations with false failure suspicions, the time interval for inconsistency is greatly reduced with respect to Chord
- Let us now explain the replication scheme, which practically eliminates inconsistency for data items

Symmetric replication



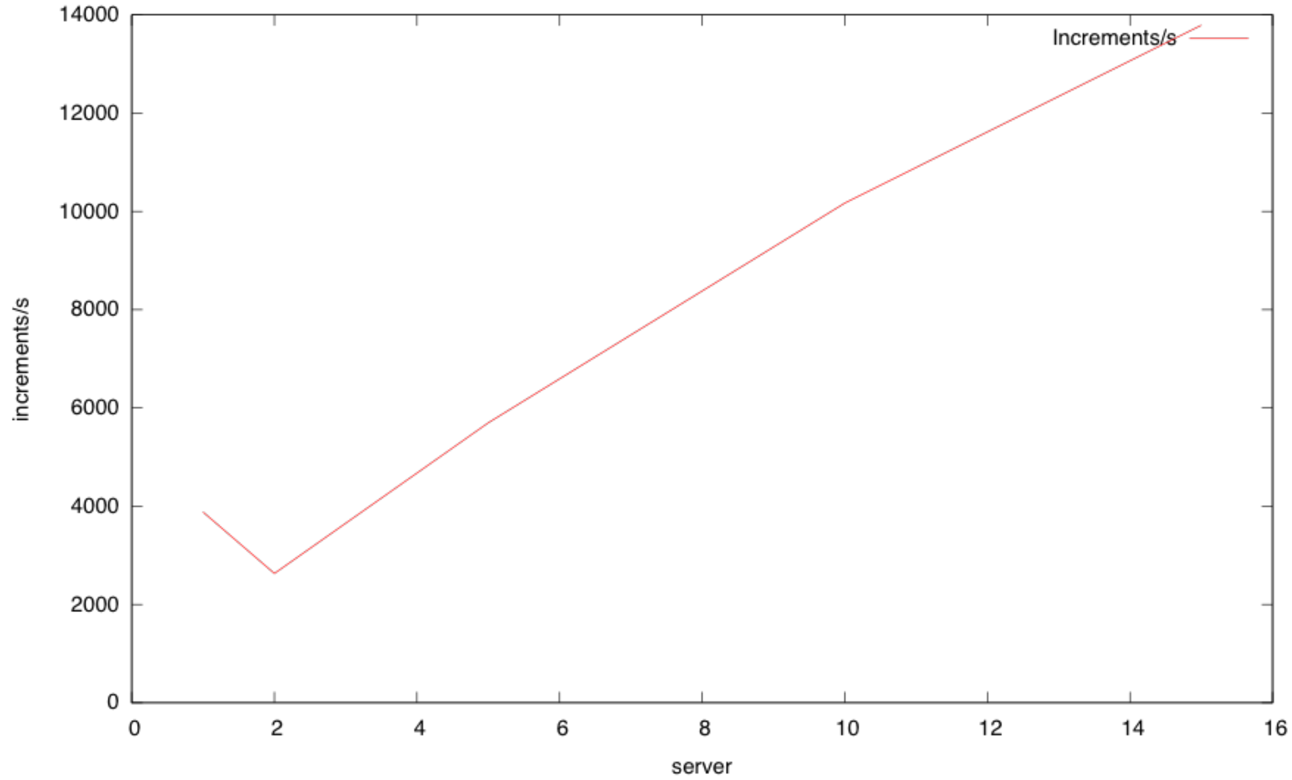
- Example network with 16 nodes and replication factor $r = 4$
- Load spread over ring; replica nodes can be accessed in decentralized fashion
- A client initiates a transaction by asking its nearest node, which becomes a transaction manager. Other nodes that store data are transaction participants.
- There are r transaction managers and r replicas for the other items

Transaction commit protocol



- Non-blocking commit protocol based on adapted Paxos that uses replicated transaction managers and replicated transaction participants
  - Paxos ≈ uniform consensus protocol for asynchronous systems assuming majority correct
- Assumes a majority of transaction managers {TM,RTM$_i$} and a majority of replicas {TP$_i$ with r replicas} for each item are correct

Scalaris performance



- Number of read-modify-write transactions per second
- Each server has two dual-core Intel Xeons at 2.66 GHz (4 cores in all) and 8 GB of main memory, with Gigabit Ethernet interconnection
- Total of 16 or 32 Scalaris nodes in the ring with replication factor of 4

Programming Model

Programming model

- One of the goals of SELFMAN was to explore the programming support for self-managing applications
- Both Scalaris and Beernet are implemented using concurrent component models with message passing and failure detection
  - Scalaris in Erlang and Beernet in Oz
- We also explored more sophisticated component models inspired by the Fractal framework
  - Components have management interface
  - CompOz library, Kompics component model
- This work is only the first step toward languages for large-scale distributed systems

CompOz

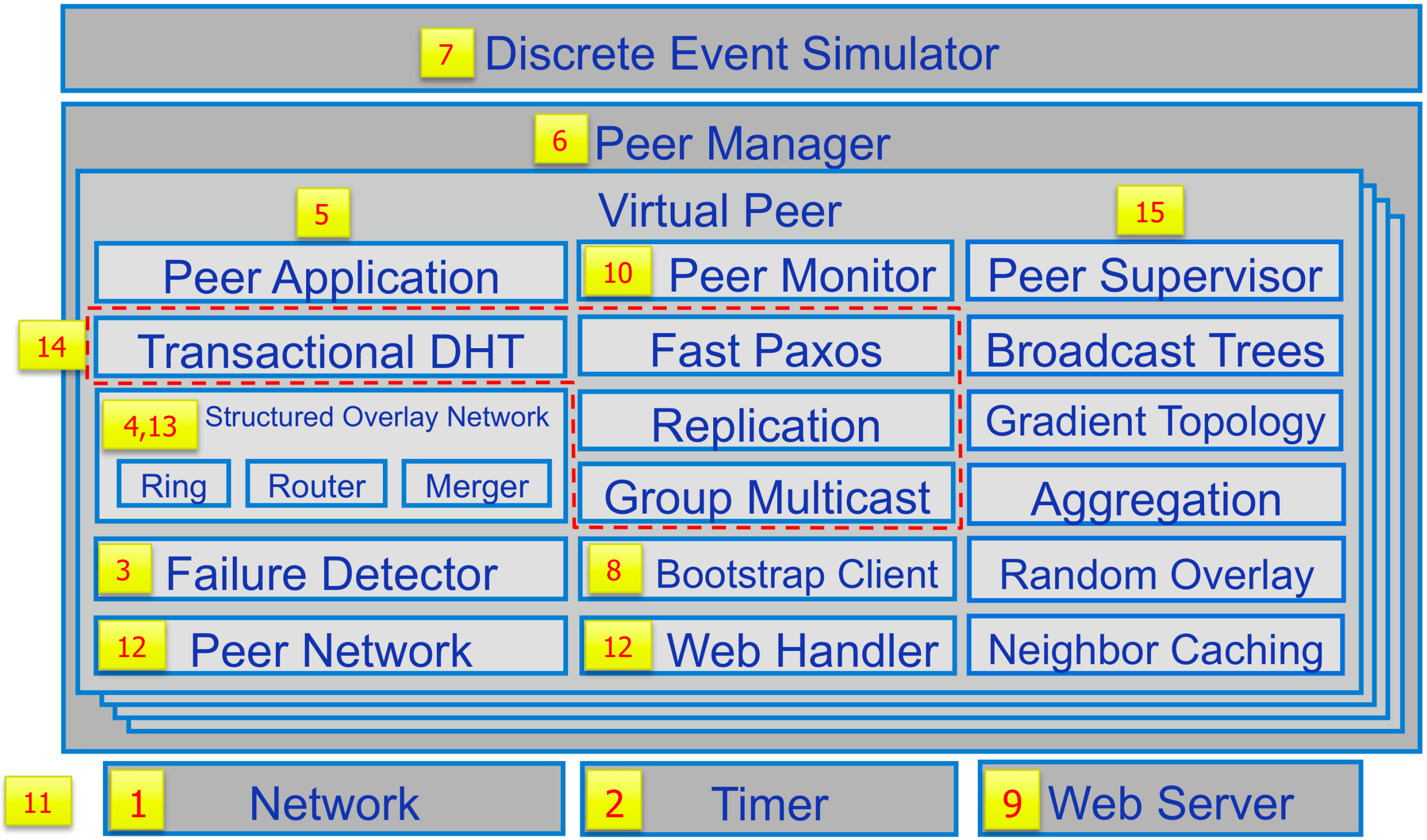
- Complete self-configuration library written in Oz
- Three complementary parts
  - Component construction and deployment (FructOz library)
    - Supports distribution, self configuration, lazy and dynamic deployment
    - Lifecycle control including termination and failures
  - Navigation and monitoring of dynamic architectures (LactOz library)
    - Distributed event bus, architecture as dynamic graph, filters
  - Distributed workflows (composing tasks) (WorkflOz library)
    - Libraries of workflow patterns as higher-order combinators
    - Can be monitored using LactOz

Kompics

- Concurrent event-driven component model implemented in Java (open-source software)
  - Supports multi-core execution and comes with full set of utility components (publish/subscribe, life-cycle management, failure handling)
- Supports dynamic reconfiguration
  - Protocol composition and hot software update
- Dual implementation for reproducible simulation / real execution of unmodified Kompics programs
  - Java-based DSL for experiment scenarios
  - Complete implementation of Chord P2P and Cyclon membership management

THE ADVENTURES OF SELFMAN

Self-management architecture implemented in Kompics



European Commission

Information Society

THE ADVENTURES OF SELFMAN

Explanation of the design

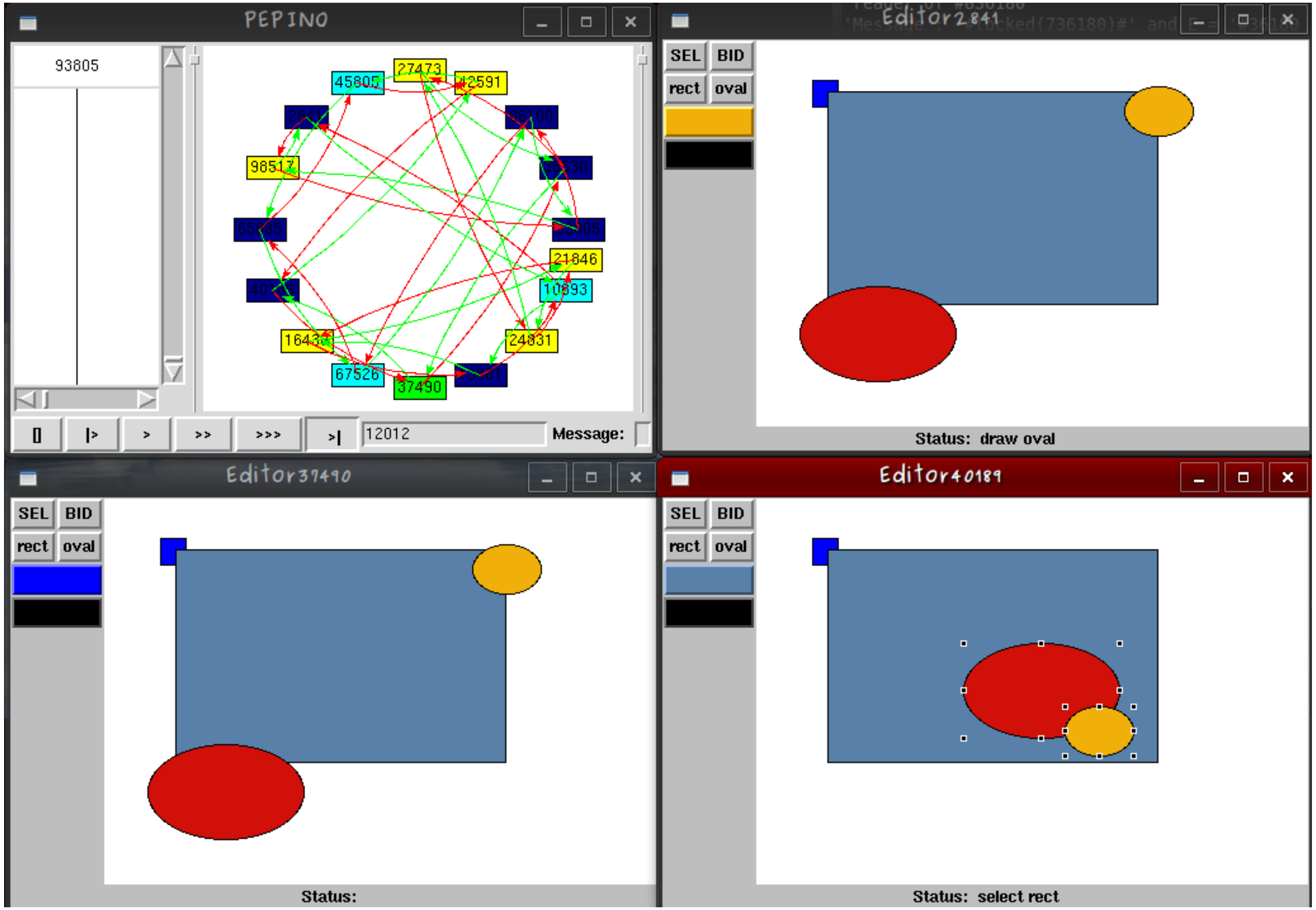
1. Encapsulate communication inside Network abstraction
2. Encapsulate timeout and alarm inside Timer abstraction
3. Encapsulate failure detection inside a Failure Detector
4. Decompose SON into Ring, Router, and Merger
5. Encapsulate all so far into a Virtual Peer component
6. Allow enclosing Peer Manager to add and remove Virtual Peers
7. Peer Manager can now be driven by a Discrete Event Simulator
8. Encapsulate bootstrapping into the Bootstrap Client
9. Enable Web-based visualization with Web Server component
10. Collect global state from new Peer Monitor component
11. Share Network, Timer, and Web Server among Virtual Peers
12. Inside Virtual Peer, add proxy Peer Network and Web Handler
13. The three SON components can be replaced
14. Add protocol components: Transactional DHT, Fast Paxos, Replication, and Group Multicast
15. Add new pillar inside Virtual Peer, to provide other useful services: Peer Supervisor, Broadcast Trees, etc.

Applications

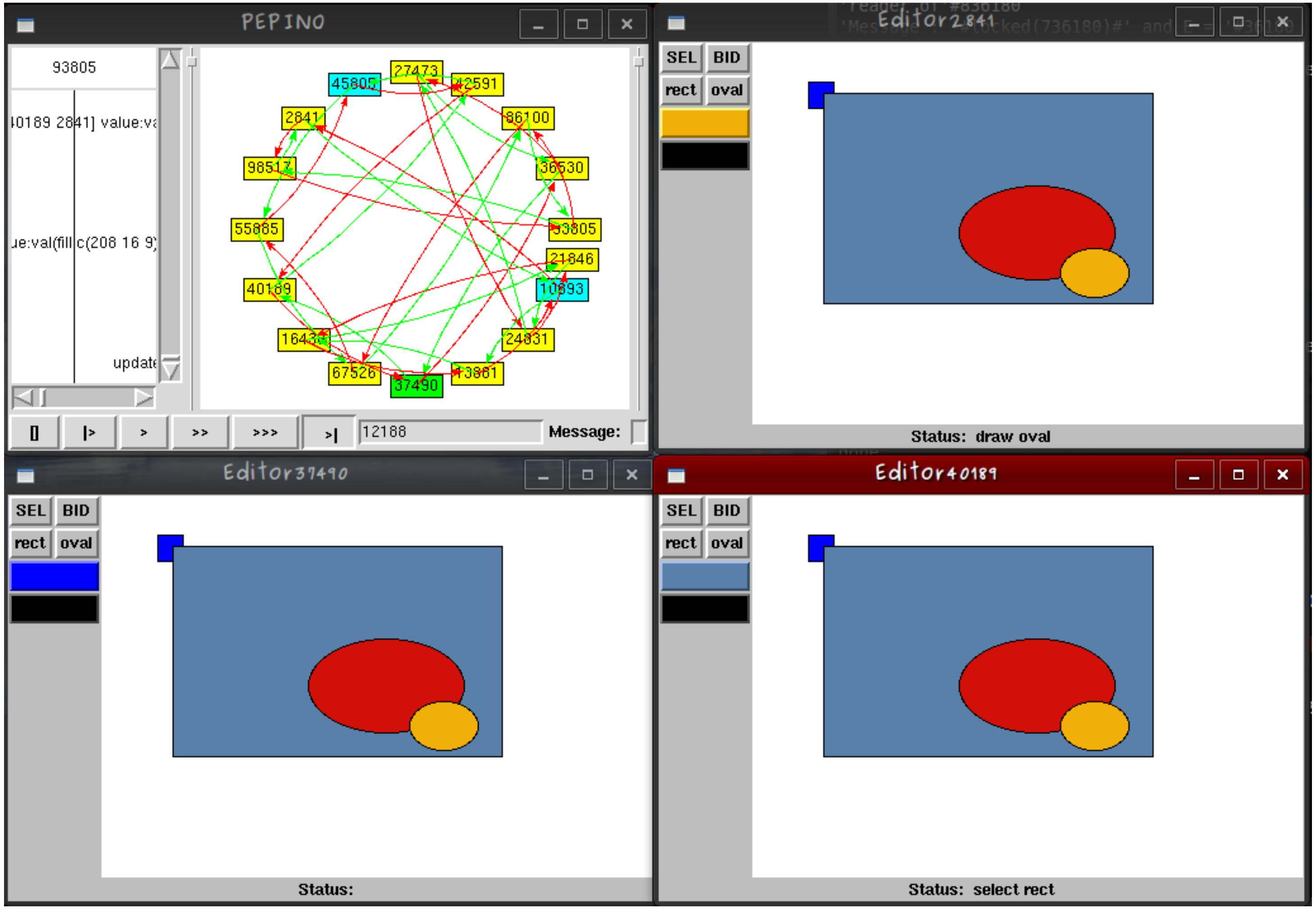
DeTransDraw Application

- DeTransDraw is a collaborative drawing application
  - Each user sees exactly the same drawing space
  - Users update the drawing space using transactions
  - For quick response time, the transaction is initiated concurrently with the display update
- Prototype application implemented on top of Beernet
  - Beernet written in Oz using Mozart, ported to gPhone with Android operating system (binary compatibility)

DeTransDraw – Getting Locks



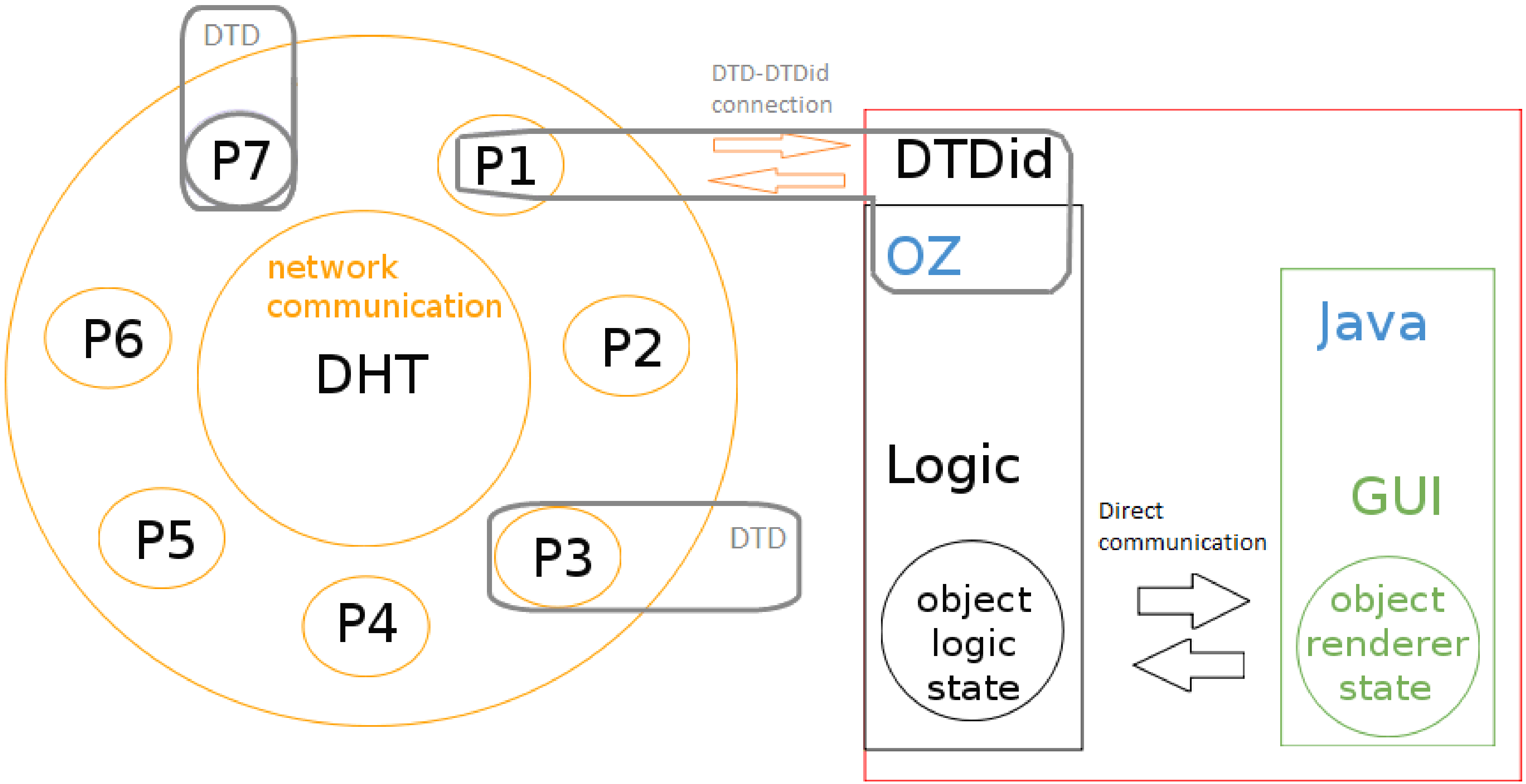
DeTransDraw – Propagating Update



European Commission

Information Society

DTD and DTDid architecture



THE ADVENTURES OF SELFMAN

Distributed Wikipedia with Scalaris

European Commission

Information Society

Wikipedia: A top 10 Web site

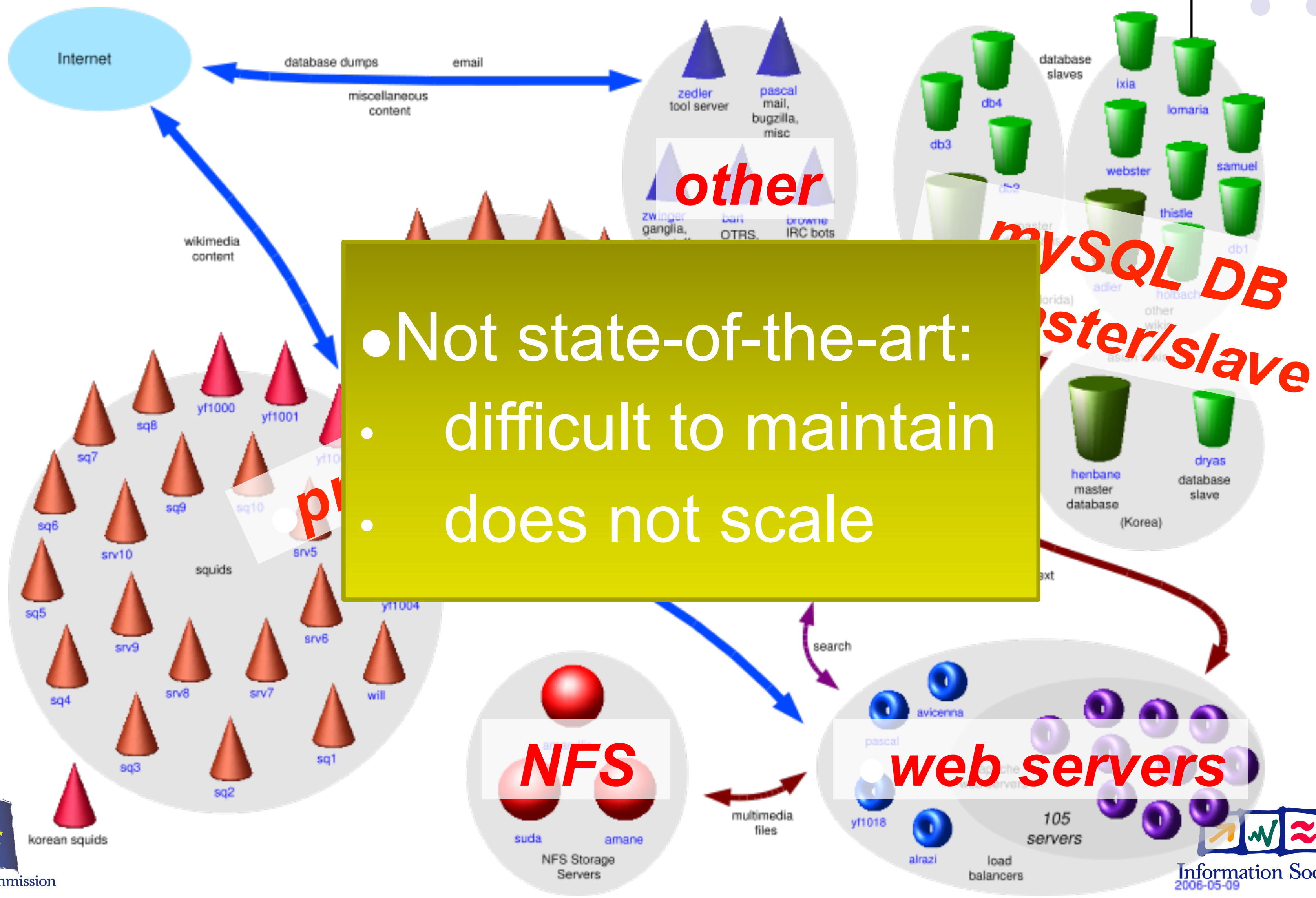
50.000 requests/sec

- 95% answered by squid proxies
  → **~18 squid servers**
- **2,000 req./sec** hit the backend
  → **12 MySQL DB, ~158 Apache servers**

→ Distributed Wikipedia built by ZIB using Scalaris (written in Erlang)

THE ADVENTURES OF SELFMAN

Wikipedia System Architecture



- Not state-of-the-art:
  - difficult to maintain
  - does not scale

European Commission

Information Society

2006-05-09

Data Model

Wikipedia

- SQL DB

Scalaris

- Key-Value Store

```
CREATE TABLE /*$wgDBprefix*/page (
page_id int unsigned NOT
        NULL auto_increment,
page_namespace int NOT NULL,
...
```

Map Relations to Key-Value Pairs

- (Title, List of Wikitext for all Versions)
- (CategoryName, List of Titles)
- (BackLinkTitle, List of Titles)

THE ADVENTURES OF SELFMAN

Data Model (Simple Query Layer)

```
void updatePage(string title, int oldVersion, string newText)
{
    //new transaction
    Transaction t = new Transaction();
    //read old version
    Page p = t.read(title);
    //check for concurrent update
    if(p.currentVersion != oldVersion)
        t.abort();
    else{
        //write new text
        t.write(p.add(newText));
        //update categories
        foreach(Category c in p)
            t.write(t.read(c.name).add(title));
        //commit
        t.commit();
    }
}
```

European Commission

Information Society

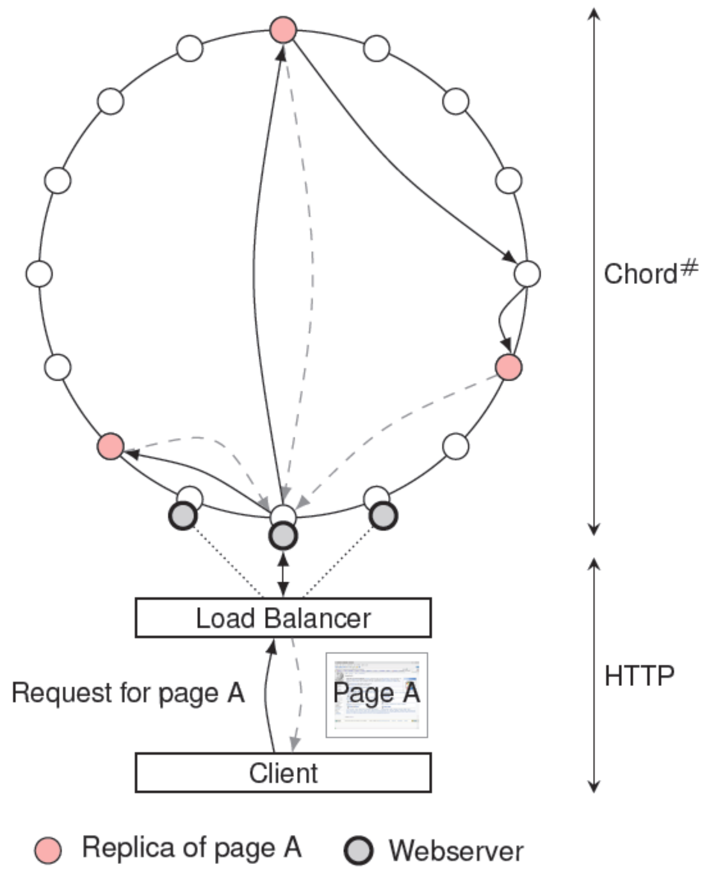
Self-* Architecture

Database:

- Chord#
- Mapping
  - Wiki -> Key-Value Store

Renderer:

- Java
  - Tomcat
  - Plog4u
- Jinterface
  - Interface to Erlang



Our Approach: P2P with Transaction Layer

Benefits

- distributed
- scalable
  - because of peer concept
- fault tolerance
  - because of replication

Challenges

- need synchronization
  - concurrency control
- need atomicity
  - in face of churn
- need transactions

DHT + Transactions = Scalable, Reliable, Efficient Key/Value Store

System Solutions

- Wikipedia

Map Wiki to key / value,
render wiki text to HTML

Written in Erlang

Key/Value Store
(simple DBMS)

Simple data read/write interface

- Transaction

Adapted Paxos Algorithm

Read → 1 access to majority of replicas
Write → 3 rounds accessing the replicas

- Replication

Symmetric Replication in P2P

Replica locations can be calculated locally

- DHT

Chord#, log (N) routing,
no hashing, range queries

Demonstration

Two independent instances are set up:

Cluster:

640 peers on 20 x 8 cores

PlanetLab:

about 150 peers
distributed worldwide

THE ADVENTURES OF SELFMAN

Boot-Server: P2P management interface

- store keys
- search keys
- see the P2P ring
- statistics
- debug data

THE ADVENTURES OF SELFMAN

Wikipedia Frontend

- Wikipedia on top of **scalable key/value store**
- installed a dump of **Simple English**
- interface language is static (Bavarian)
- **no images**
  URLs not in dump
- **browse links**
- **no fulltext search**

European Commission

Information Society

Outlook

Conclusions

- DHTs are a good foundation for large-scale distributed applications
  - Horizontally scalable distributed transaction store
- Scalaris and Beernet
  - Robust implementations with applications
  - Written in Erlang (Scalaris) and Oz (Beernet)
    - Support for fine-grain concurrency, message passing, and transparent distribution
- Some applications
  - DeTransDraw
  - Distributed Wikipedia

THE ADVENTURES OF SELFMAN

Some future directions

- Support mobile applications with large numbers of collaborators
  - Some form of consistency is important
  - Transactional DHT can be a good foundation

WISEMAN proposal (ANR)

- Combine cloud computing and data-intensive applications
  - Horizontal scalability makes it a perfect fit
  - Elasticity enables new kinds of applications
  - DHTs support elasticity very well
- New language to simplify programming large-scale applications
  - In course project, students complained Beernet is too easy ☺
  - Program for the whole system, not for single machines
- Design for global behavior?
  - Partitions, failures, security
  - Design with the CAP theorem, not against the CAP theorem

European Commission

Information Society

Opinion

Data-intensive applications

- Computing science is changing fundamentally
- It is becoming focused on programming with large data sets
  - Elastic data-intensive algorithms running on clouds are realizing one by one the old dreams of artificial intelligence
  - The canonical example is Google Search using PageRank
    - It extracts useful information from the Web link graph
  - Many other applications are now following this path: data mining (e.g., recommendation systems), machine learning, statistical language translation, image recognition, visualization, complex problem solving, etc.
- This is where most of the innovation will happen in Internet applications in the next decade
  - Elastic data-intensive algorithms on clouds and P2P systems
  - Domain knowledge is the key!