ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

Algorithms and Methods for Distributed Storage Networks

11 Peer-to-Peer Storage (final version)

Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Rechnernetze und Telematik
Wintersemester 2007/08

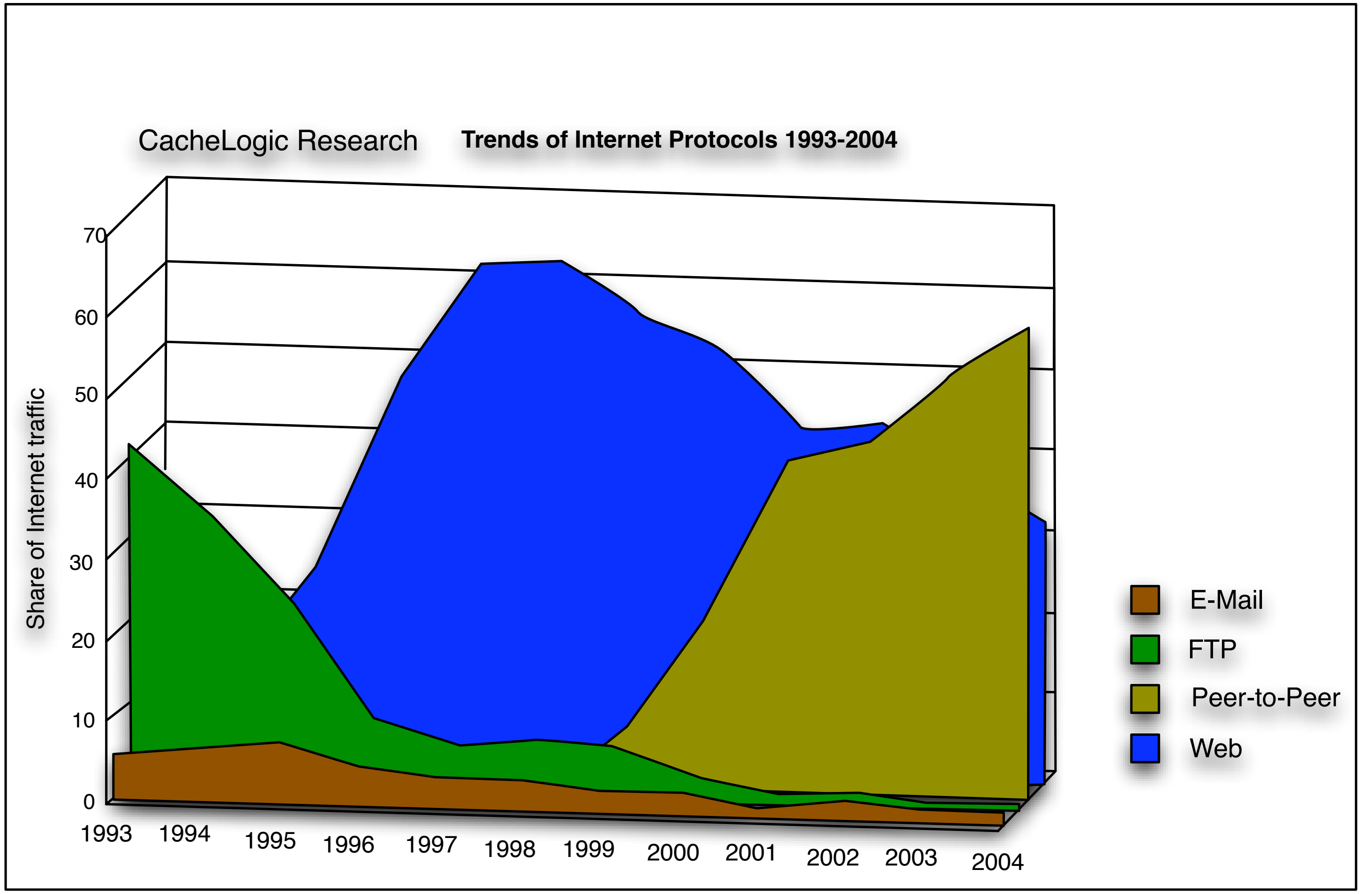
CoNe Freiburg

IIF
INSTITUT FÜR INFORMATIK FREIBURG

Outline

- **Principles and history**
- **Algorithms and Methods**
  - DHTs
  - Chord
  - Pastry and Tapestry
- **P2P Storage Systems**
  - PAST
  - Oceanstore
- **Further Issues**
  - Bandwidth
  - Anonymity, Security
  - Availability and Robustness

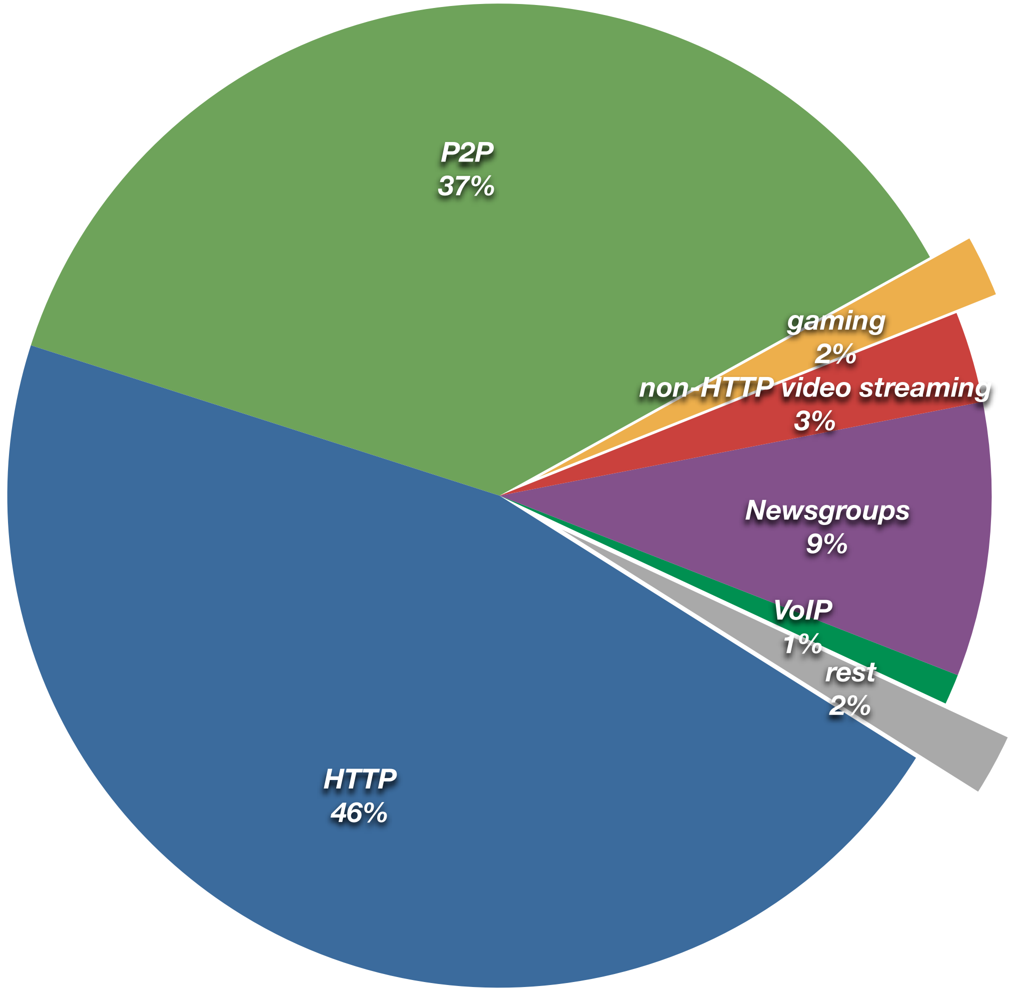
Global Internet Traffic Shares 1993-2004



Source: CacheLogic 2005

Global Internet Traffic 2007

- **Ellacoya report (June 2007)**
  - worldwide HTTP traffic volume overtakes P2P after four years continues record
- **Main reason: Youtube.com**



Milestones P2P Systems

- **Napster (1st version: 1999-2000)**
- **Gnutella (2000), Gnutella-2 (2002)**
- **Edonkey (2000)**
  - later: Overnet usese Kademlia
- **FreeNet (2000)**
  - Anonymized download
- **JXTA (2001)**
  - Open source P2P network platform
- **FastTrack (2001)**
  - known from KaZaa, Morpheus, Grokster
- **Bittorrent (2001)**
  - only download, no search
- **Skype (2003)**
  - VoIP (voice over IP), Chat, Video

Milestones Theory

- **Distributed Hash-Tables (DHT) (1997)**
  - introduced for load balancing between web-servers
- **CAN (2001)**
  - efficient distributed DHT data structure for P2P networks
- **Chord (2001)**
  - efficient distributed P2P network with logarithmic search time
- **Pastry/Tapestry (2001)**
  - efficient distributed P2P network using Plaxton routing
- **Kademlia (2002)**
  - P2P-Lookup based on XOr-Metrik
- **Many more exciting approaches**
  - Viceroy, Distance-Halving, Koorde, Skip-Net, P-Grid, ...
- **Recent developments**
  - Network Coding for P2P
  - Game theory in P2P
  - Anonymity, Security

What is a P2P Network?

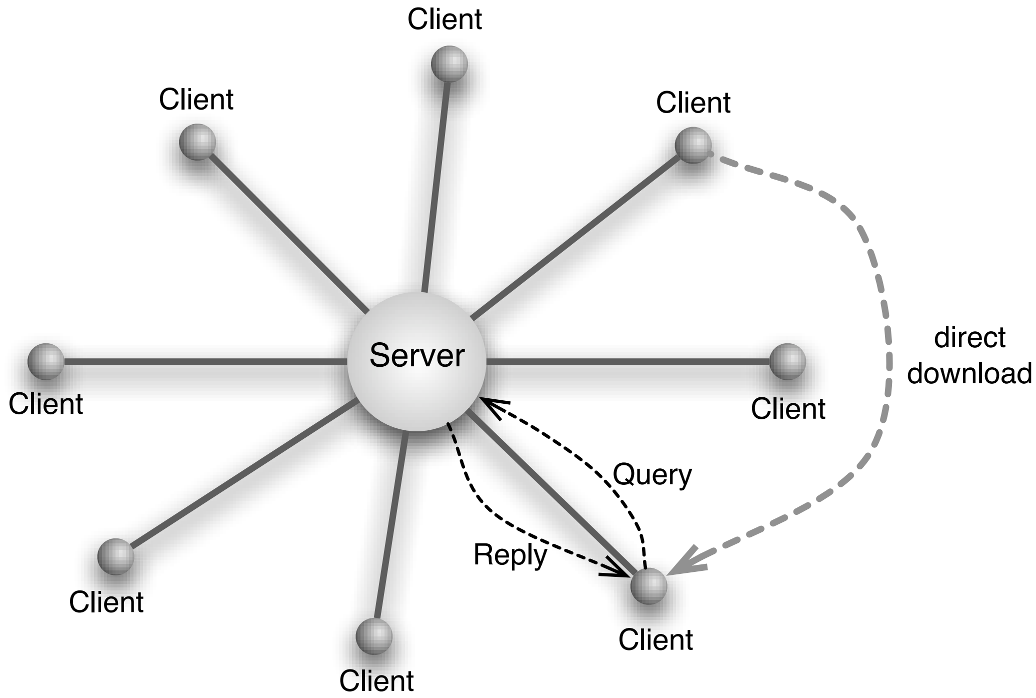
- **What is P2P NOT?**
  - a peer-to-peer network is ***not a client-server network***
- **Etymology: peer**
  - from latin par = equal
  - one that is of equal standing with another
  - P2P, Peer-to-Peer: a relationship between equal partners
- **Definition**
  - a Peer-to-Peer Network is a communication network between computers in the Internet
    - without central control
    - and without reliable partners
- **Observation**
  - the Internet can be seen as a large P2P network

Napster

- **Shawn (Napster) Fanning**
  - published 1999 his beta version of the now legendary Napster P2P network
  - File-sharing-System
  - Used as mp3 distribution system
  - In autumn 1999 Napster has been called download of the year
- **Copyright infringement lawsuit of the music industry in June 2000**
- **End of 2000: cooperation deal**
  - between Fanning and Bertelsmann Ecommerce
- **Since then Napster is a commercial file-sharing platform**

How Did Napster Work?

- **Client-Server**
- **Server stores**
  - Index with meta-data
    - file name, date, etc
  - table of connections of participating clients
  - table of all files of participants
- **Query**
  - client queries file name
  - server looks up corresponding clients
  - server replies the owner of the file
  - querying client downloads the file from the file owning client

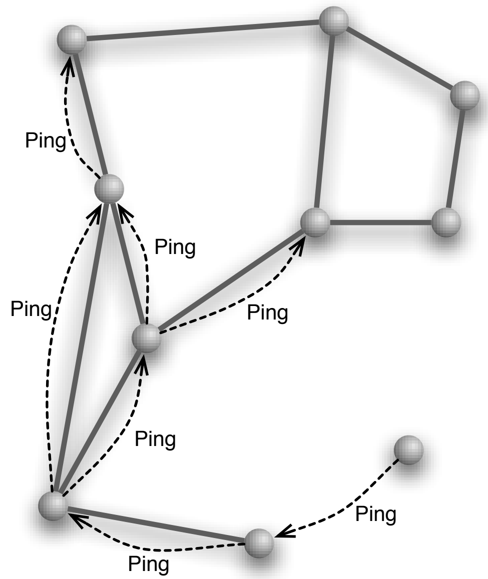


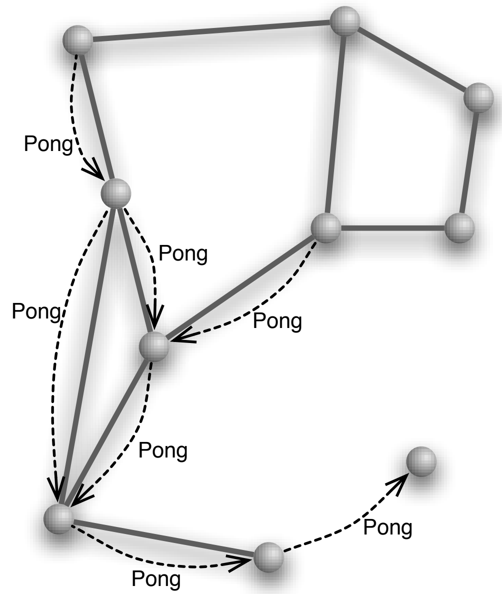
History of Gnutella

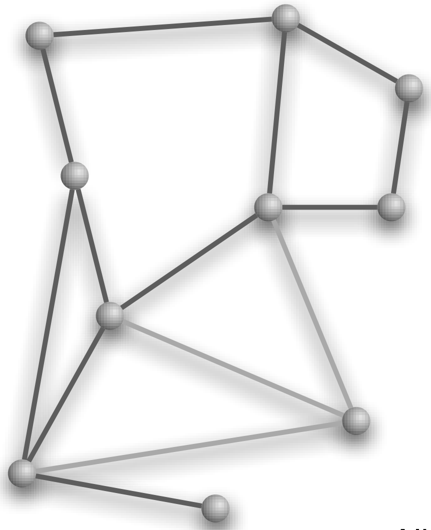
- **Gnutella**
  - was released in March 2000 by Justin Frankel and Tom Pepper from Nullsoft
  - Since 1999 Nullsoft is owned by AOL
- **File-Sharing system**
  - Same goal as Napster
  - But without any central structures

Gnutella — Connecting

- **Neighbor lists**
  - Gnutella connects directly with other clients
  - the client software includes a list of usually online clients
  - the clients checks these clients until an active node has been found
  - an active client publishes its neighbor list
  - the query (ping) is forwarded to other nodes
  - the answer (pong) is sent back
  - neighbor lists are extended and stored
  - the number of the forwarding is

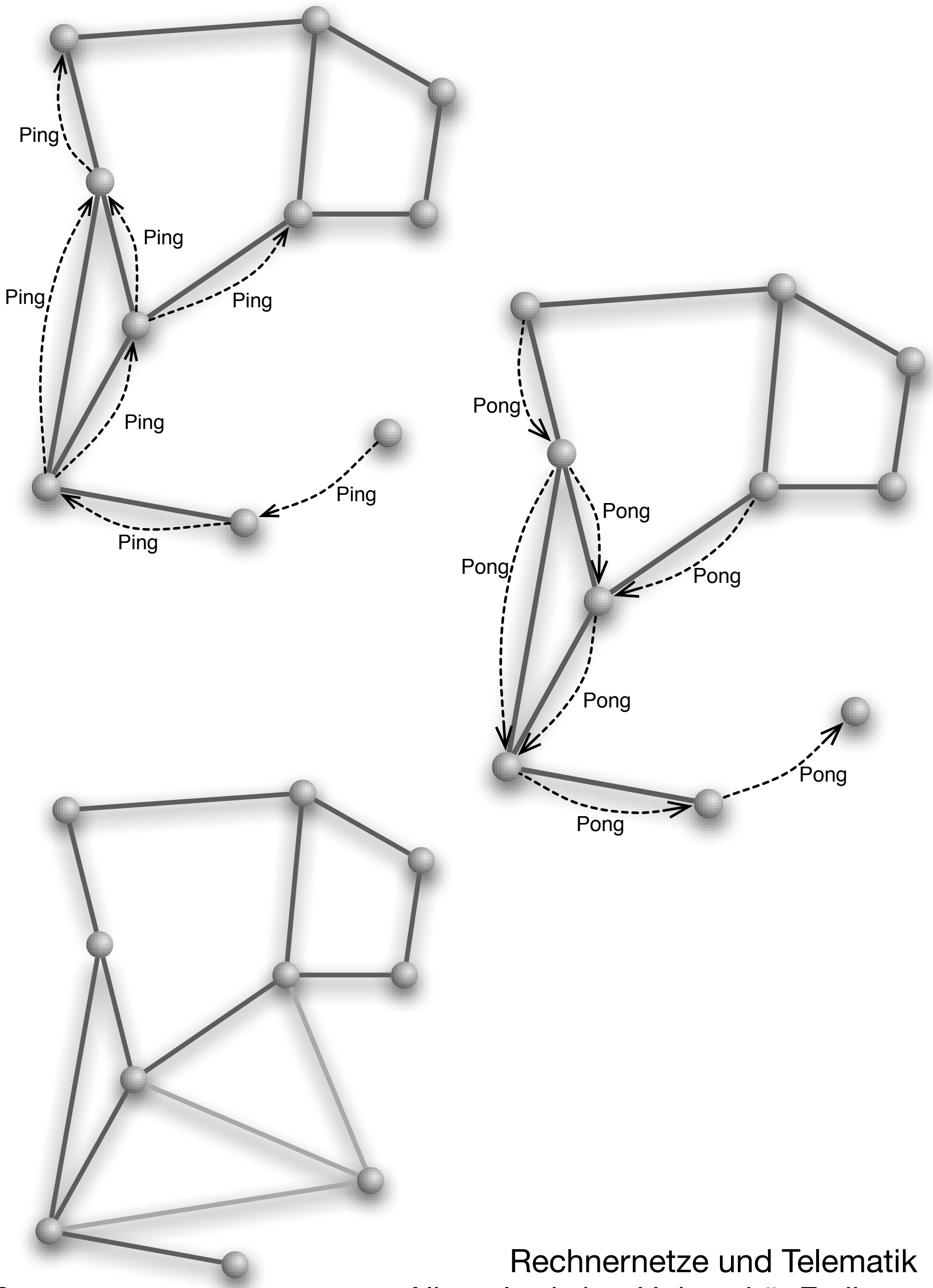






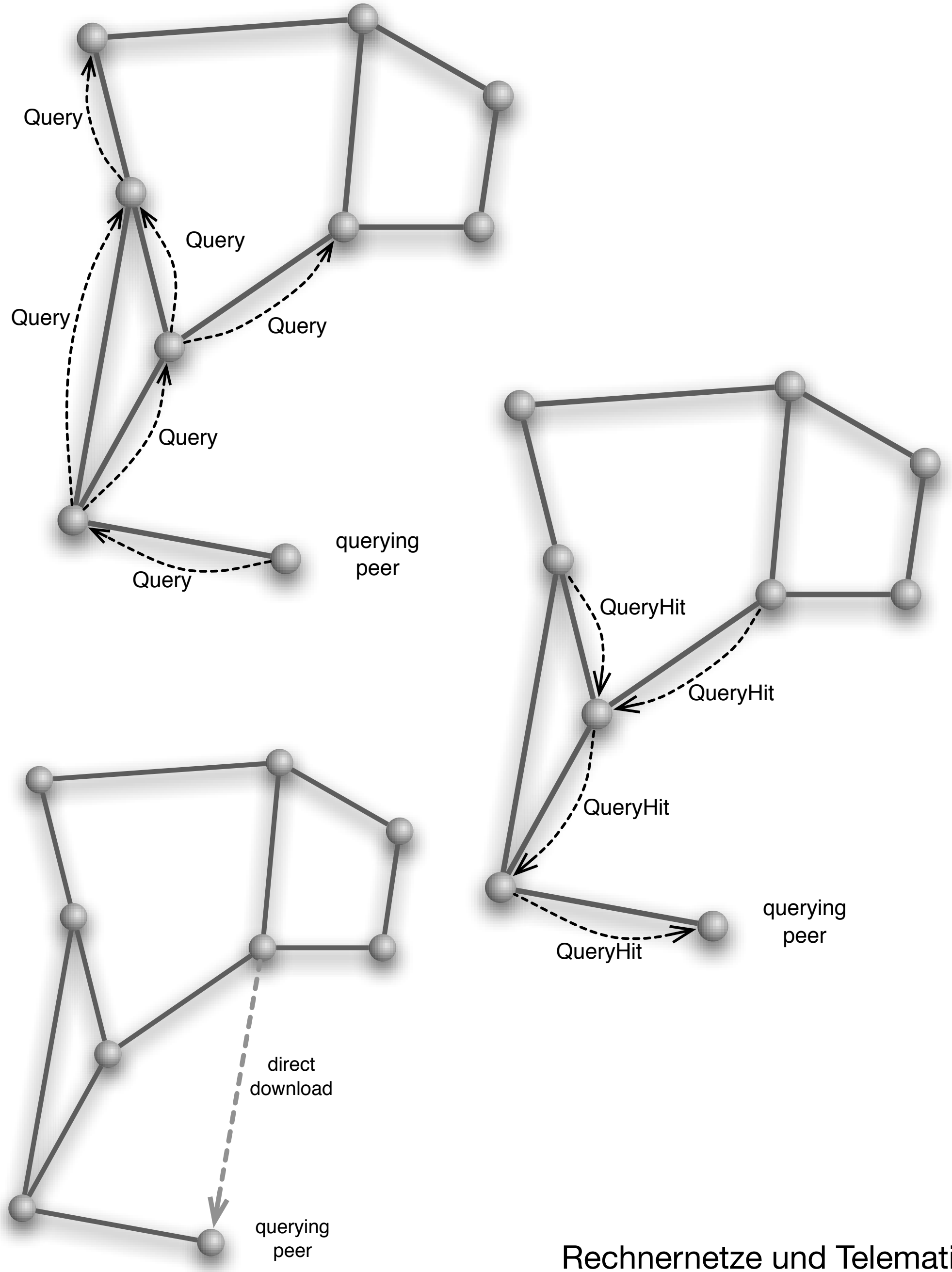
Gnutella — Connecting

- **Protokoll**
  - Ping
    - participants query for neighbors
    - are forwarded according for TTL steps (time to live)
  - Pong
    - answers Ping
    - is forwarded backward on the query path
    - reports IP and port adress (socket pair)
    - number and size of available files



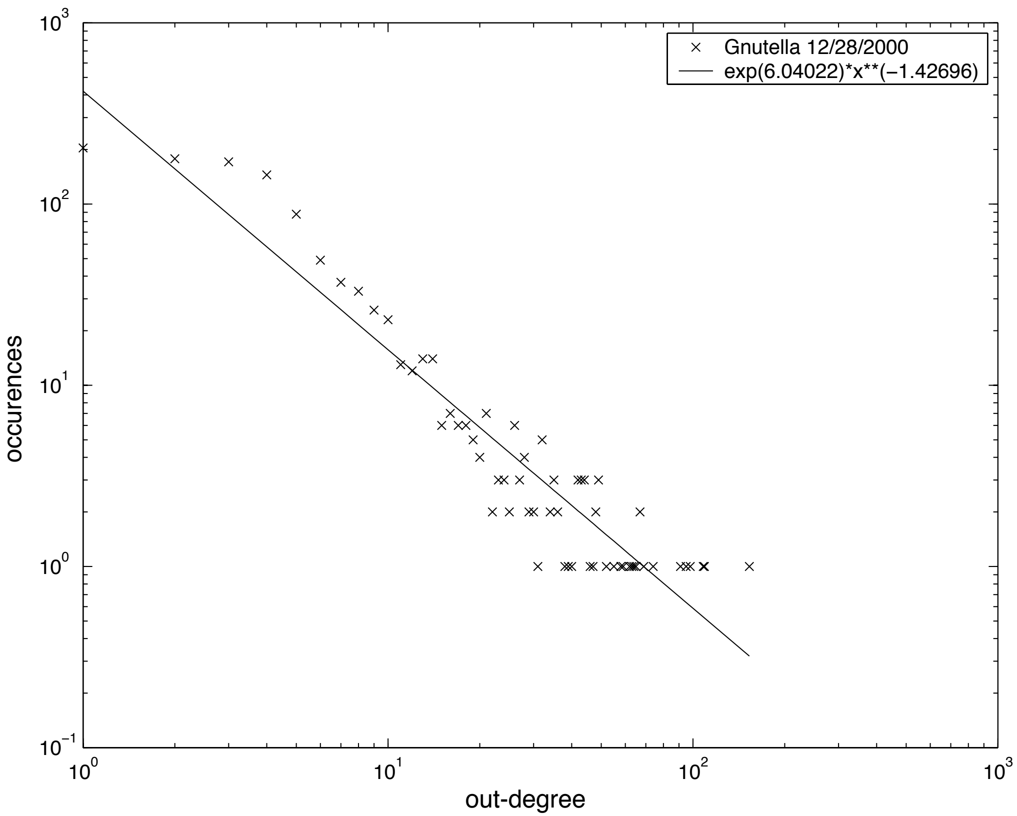
Gnutella — Query

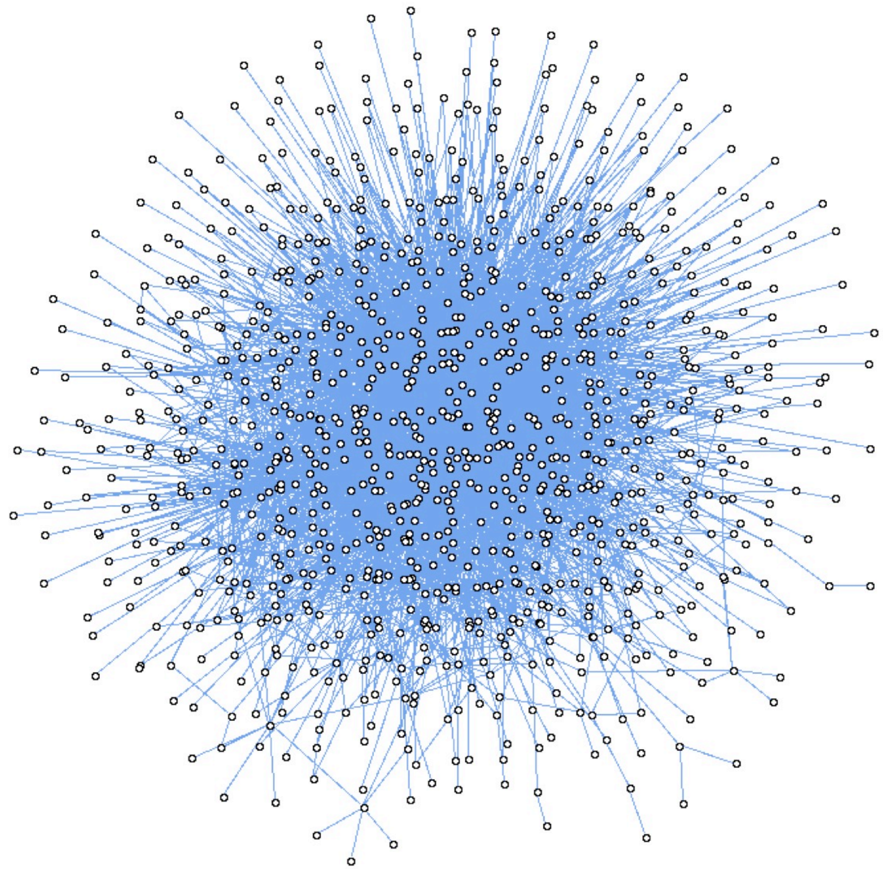
- **File Query**
  - are sent to all neighbors
  - Neighbors forward to all neighbors
  - until the maximum hop distance has been reached
    - TTL-entry (time to live)
- **Protocol**
  - Query
    - for file for at most TTL hops
  - Query-hits
    - answers on the path backwards
- **If file has been found, then initiate direct download**



Gnutella — Graph Structure

- **Graph structure**
  - constructed by random process
  - underlies power law
  - without control

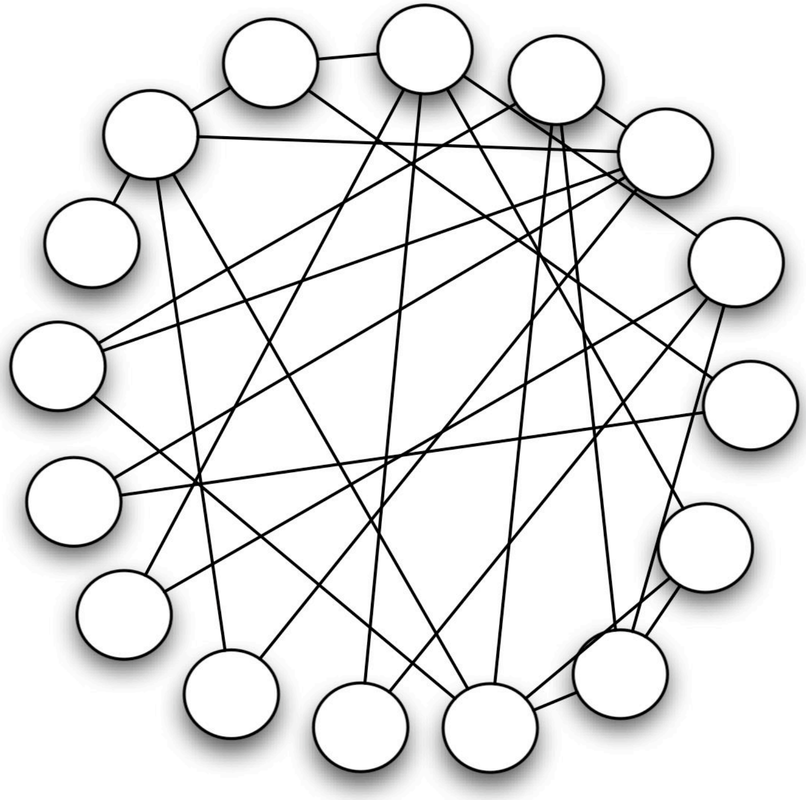




Gnutella snapshot in 2000

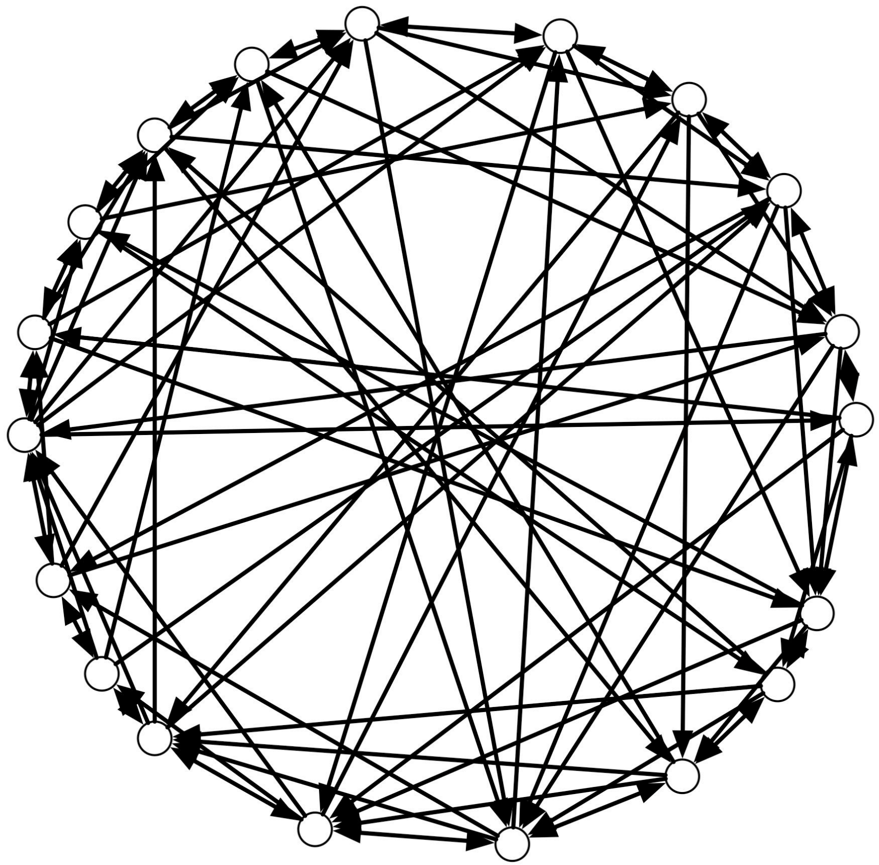
Why Gnutella Does Not Really Scale

- **Gnutella**
  - graph structure is random
  - degree of nodes is small
  - small diameter
  - strong connectivity
- **Lookup is expensive**
  - for finding an item the whole network must be searched
- **Gnutella‘s lookup does not scale**
  - reason: no structure within the index storage



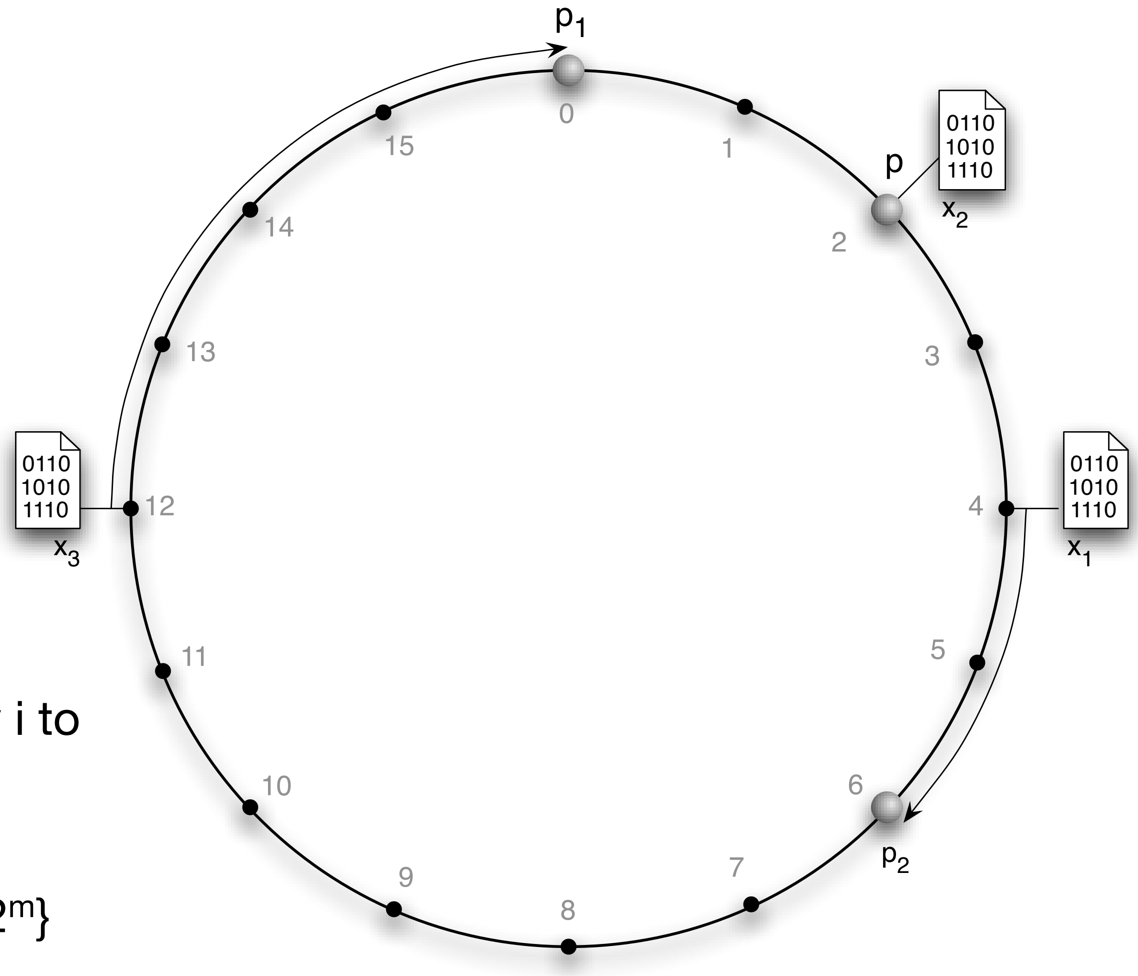
Chord

- **Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek and Hari Balakrishnan (2001)**
- **Distributed Hash Table**
  - range $\{0,..,2^m-1\}$
  - for sufficient large m
- **Network**
  - ring-wise connections
  - shortcuts with exponential increasing distance



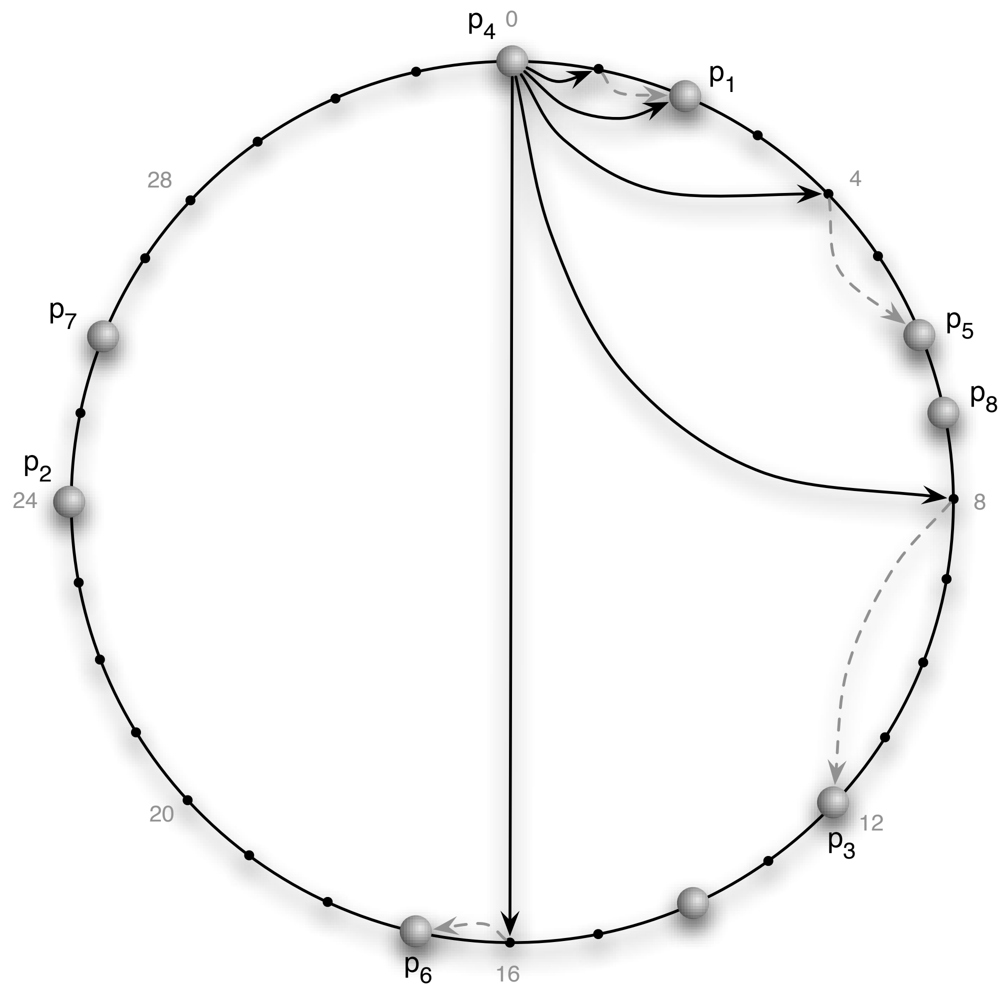
Chord as DHT

- **n number of peers**
- **V set of peers**
- **k number of data stored**
- **K set of stored data**
- **m: hash value length**
  - $m \geq 2 \log \max\{K,N\}$
- **Two hash functions mapping to $\{0,..,2^{m-1}\}$**
  - $r_V(b)$: maps peer to $\{0,..,2^{m-1}\}$
  - $r_K(i)$: maps index according to key i to $\{0,..,2^{m-1}\}$
- **Index i maps to peer $b = f_V(i)$**
  - $f_V(i) := \arg\min_{b \in V}\{(r_V(b) - r_K(i)) \bmod 2^m\}$



Pointer Structure of Chord

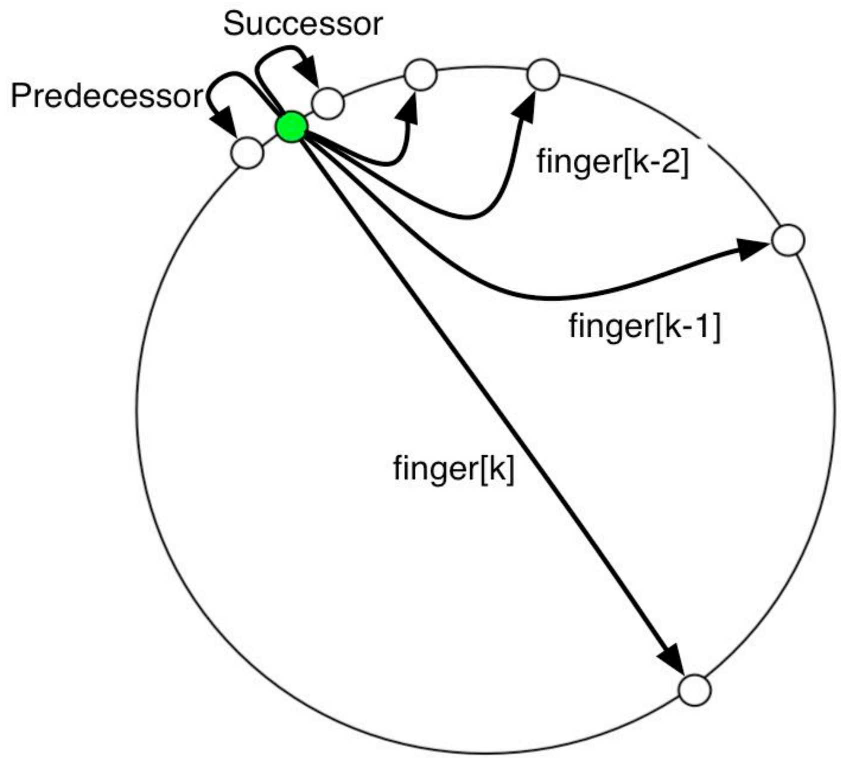
- **For each peer**
  - successor link on the ring
  - predecessor link on the ring
  - for all i ∈ {0,..,m-1}
    - Finger[i] := the peer following the value $r_V(b+2^i)$
- **For small i the finger entries are the same**
  - store only different entries
- **Lemma**
  - The number of different finger entries is O(log n) with high probability, i.e. 1-n^{-c}.



Rechnernetze und Telematik
Albert-Ludwigs-Universität Freiburg
Christian Schindelhauer

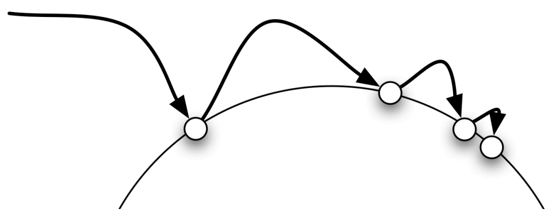
Data Structure of Chord

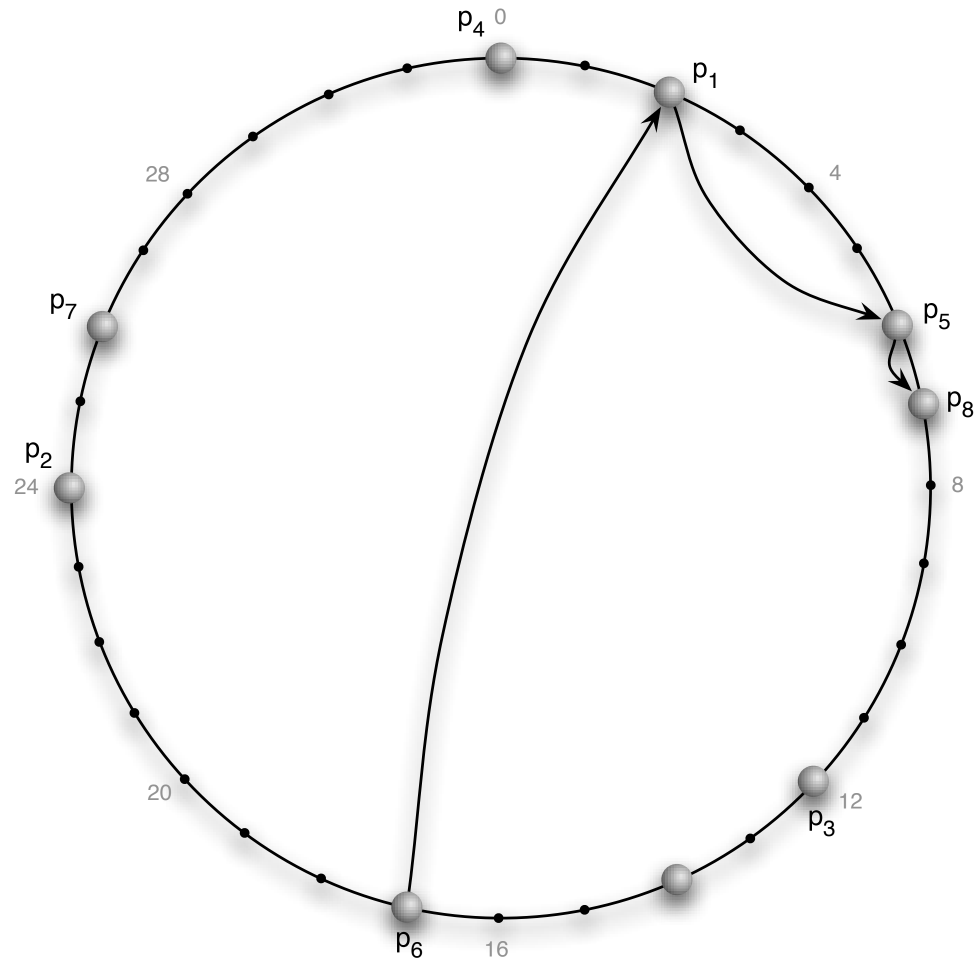
- **For each peer**
  - successor link on the ring
  - predecessor link on the ring
  - for all i ∈ {0,..,m-1}
    - Finger[i] := the peer following the value $r_V(b+2^i)$
- **For small i the finger entries are the same**
  - store only different entries
- **Chord**
  - needs O(log n) hops for lookup
  - needs $O(\log^2 n)$ messages for inserting and erasing of peers



Lookup in Chord

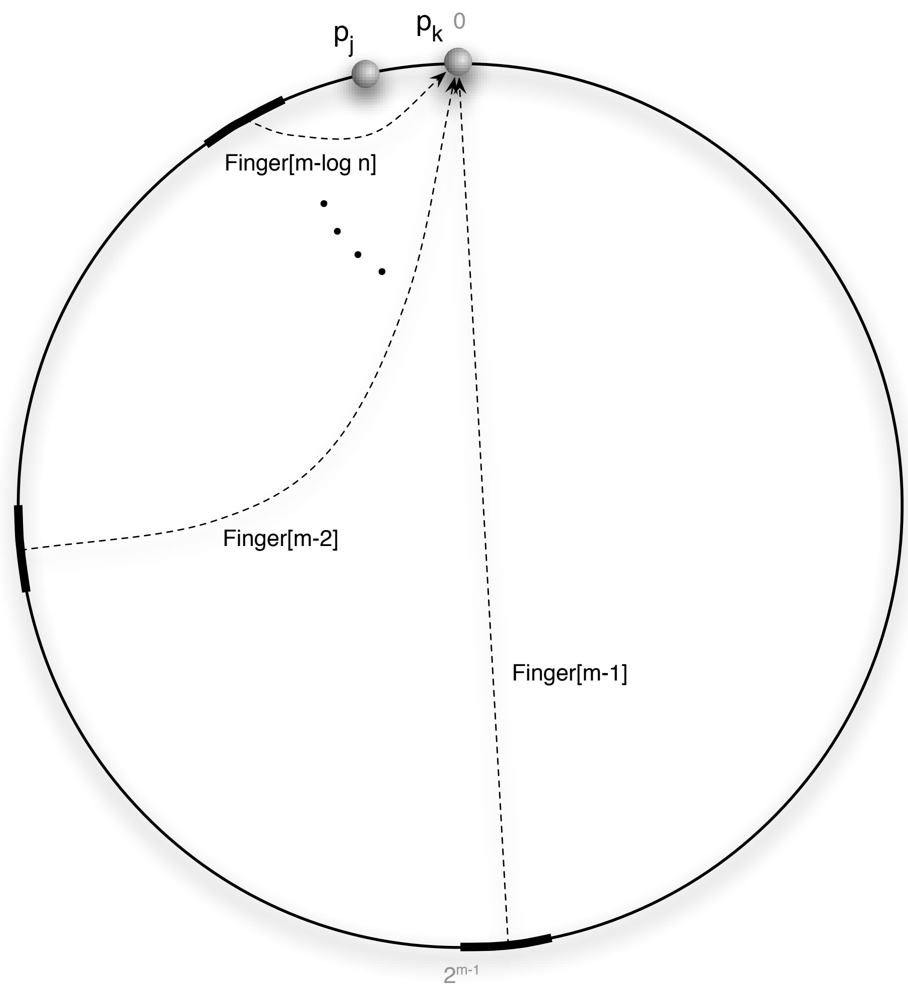
- **Theorem**
  - The Lookup in Chord needs O(log n) steps w.h.p.





How Many Fingers?

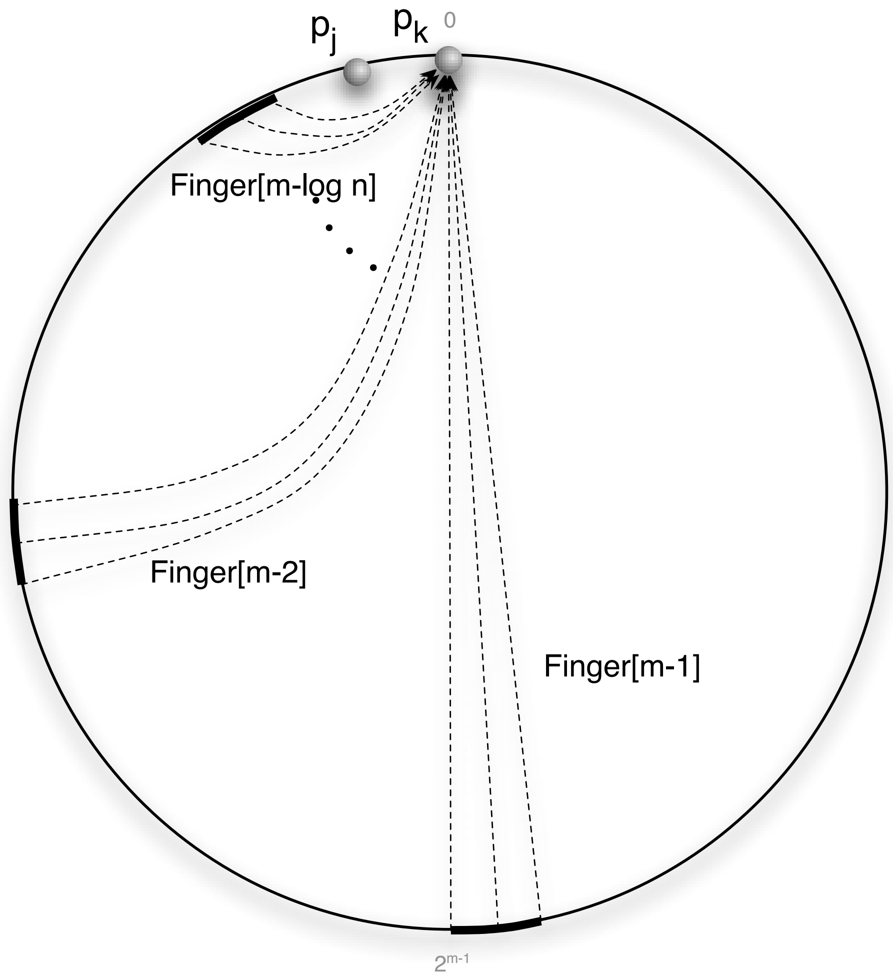
- **Lemma**
  - The out-degree in Chord is $O(\log n)$ w.h.p.
  - The in-degree in Chord is $O(\log^2 n)$ w.h.p.
- **Theorem**
  - For integrating a new peer into Chord only $O(\log^2 n)$ messages are necessary.



Rechnernetze und Telematik
Albert-Ludwigs-Universität Freiburg
Christian Schindelhauer

Adding a Peer

- **First find the target area in O(log n) steps**
- **The outgoing pointers are adopted from the predecessor and successor**
  - the pointers of at most O(log n) neighbored peers must be adapted
- **The in-degree of the new peer is $O(\log^2 n)$ w.h.p.**
  - Lookup time for each of them
  - There are O(log n) groups of neighb ored peers
  - Hence, only O(log n) lookup steps with at most costs O(log n) must be used
  - Each update of has constant cost



Peer-to-Peer Networks

Pastry

Pastry

- **Peter Druschel**
  - Rice University, Houston, Texas
  - now head of Max-Planck-Institute for Computer Science, Saarbrücken/ Kaiserslautern
- **Antony Rowstron**
  - Microsoft Research, Cambridge, GB
- **Developed in Cambridge (Microsoft Research)**
- **Pastry**
  - Scalable, decentralized object location and routing for large scale peer-to-peer-network
- **PAST**
  - A large-scale, persistent peer-to-peer storage utility
- **Two names one P2P network**
  - PAST is an application for Pastry enabling the full P2P data storage functionality
  - First, we concentrate on Pastry

Pastry Overview

- **Each peer has a 128-bit ID: nodeID**
  - unique and uniformly distributed
  - e.g. use cryptographic function applied to IP-address
- **Routing**
  - Keys are matched to $\{0,1\}^{128}$
  - According to a metric messages are distributed to the neighbor next to the target
- **Routing table has $O(2^b(\log n)/b) + \ell$ entries**
  - n: number of peers
  - ℓ: configuration parameter
  - b: word length
    - typical: b= 4 (base 16), $\ell = 16$
    - message delivery is guaranteed as long as less than $\ell/2$ neighbored peers fail
- **Inserting a peer and finding a key needs O((log n)/b) messages**

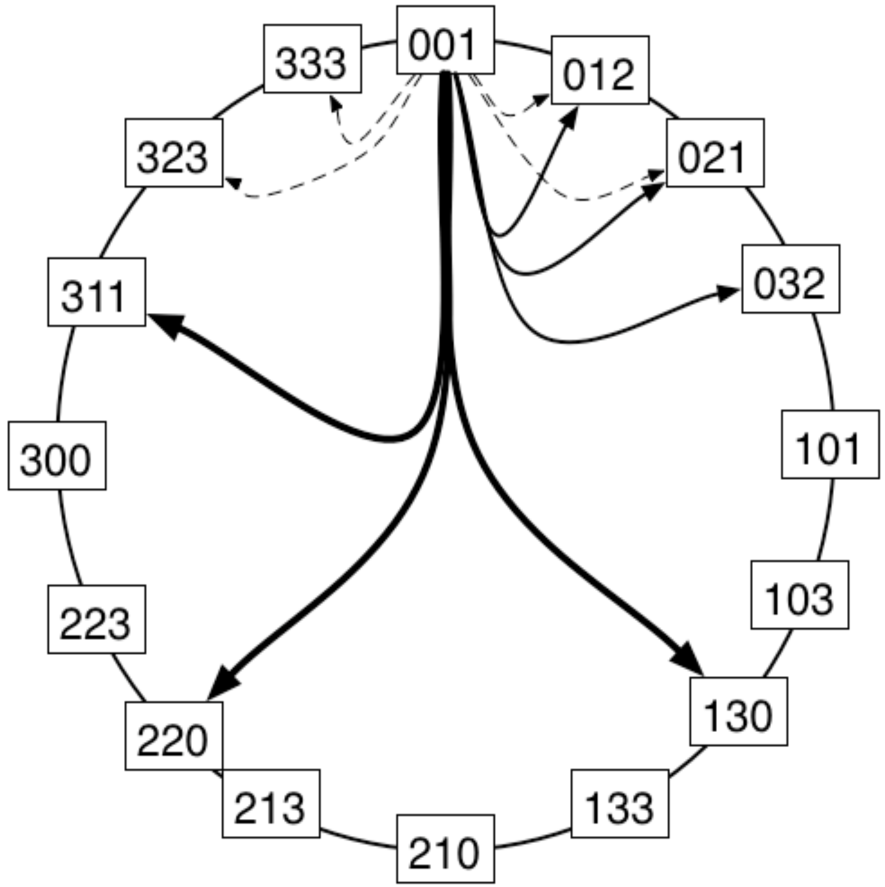
Routing Table

- **NodeId presented in base 2^b**
  - e.g. NodeID: 65A0BA13
- **For each prefix p and letter x ∈ {0,..,2^b-1} add an peer of form px* to the routing table of NodeID, e.g.**
  - b=4, 2^b=16
  - 15 entries for 0*,1*, .. F*
  - 15 entries for 60*, 61*,... 6F*
  - ...
  - if no peer of the form exists, then the entry remains empty
- **Choose next neighbor according to a distance metric**
  - metric results from the RTT (round trip time)
- **In addition choose ℓ neighbors**
  - $\ell/2$ with next higher ID
  - $\ell/2$ with next lower ID

0x	1x	2x	3x	4x	5x		7x	8x	9x	ax	bx	cx	dx	ex	fx
60x	61x	62x	63x	64x		66x	67x	68x	69x	6ax	6bx	6cx	6dx	6ex	6fx
650x	651x	652x	653x	654x	655x	656x	657x	658x	659x		65bx	65cx	65dx	65ex	65fx
65a0x		65a2x	65a3x	65a4x	65a5x	65a6x	65a7x	65a8x	65a9x	65aax	65abx	65acx	65adx	65aex	65afx

Routing Table

- **Example b=2**
- **Routing Table**
  - For each prefix p and letter $x \in \{0,..,2^b-1\}$ add an peer of form px* to the routing table of NodeID
- **In addition choose ℓ neighors**
  - $\ell/2$ with next higher ID
  - $\ell/2$ with next lower ID
- **Observation**
  - The leaf-set alone can be used to find a target
- **Theorem**
  - With high probability there are at most $O(2^b (\log n)/b)$ entries in each routing table



Routing Table

- **Theorem**
  - With high probability there are at most $O(2^b (\log n)/b)$ entries in each routing table
- **Proof**
  - The probability that a peer gets the same m-digit prefix is

$$2^{-bm}$$

  - The probability that a m-digit prefix is un

$$(1-2^{-bm})^n \le e^{-n/2^{bm}}$$

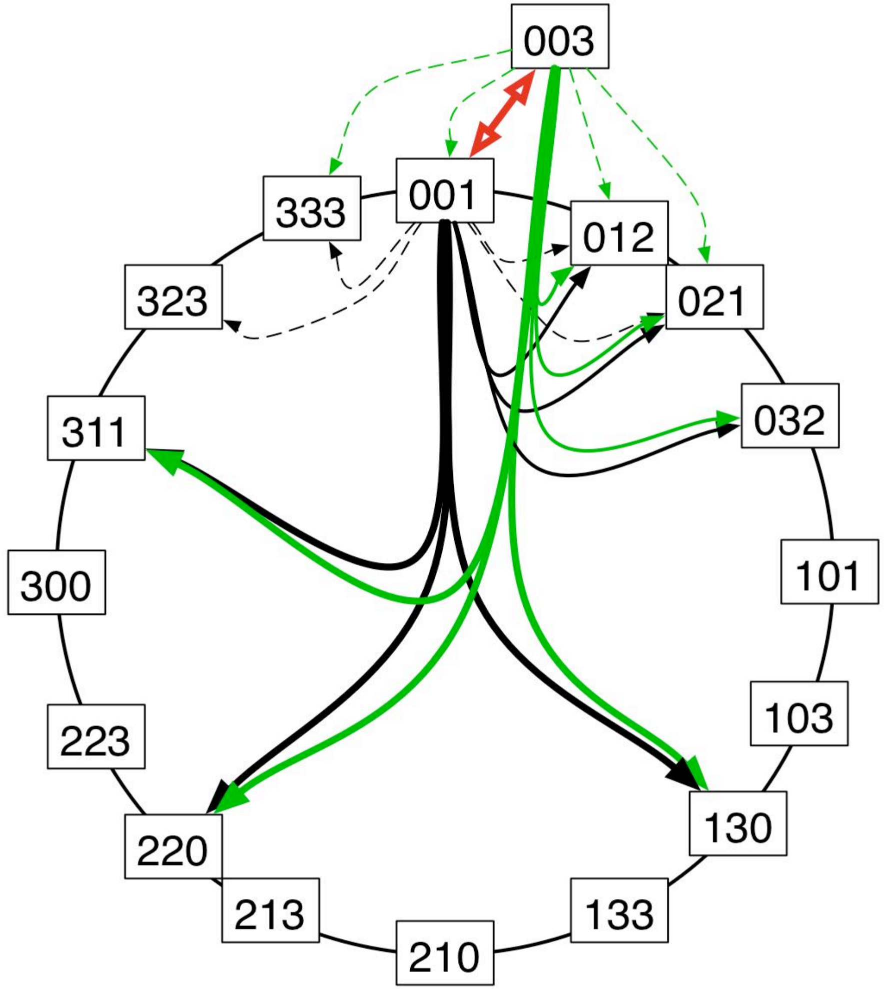
$$e^{-n/2^{bm}} \le e^{-n/2^{c \log n}} \le e^{-n/n^c} \le e^{-n^{c-1}}$$

- With (extremely) high probability there is no peer with the same prefix of length $(1+\varepsilon)(\log n)/b$
- Hence we have $(1+\varepsilon)(\log n)/b$ rows with 2^b-1 entries each

0x	1x	2x	3x	4x	5x		7x	8x	9x	ax	bx	cx	dx	ex	fx
60x	61x	62x	63x	64x		66x	67x	68x	69x	6ax	6bx	6cx	6dx	6ex	6fx
650x	651x	652x	653x	654x	655x	656x	657x	658x	659x		65bx	65cx	65dx	65ex	65fx
65a0x		65a2x	65a3x	65a4x	65a5x	65a6x	65a7x	65a8x	65a9x	65aax	65abx	65acx	65adx	65aex	65afx

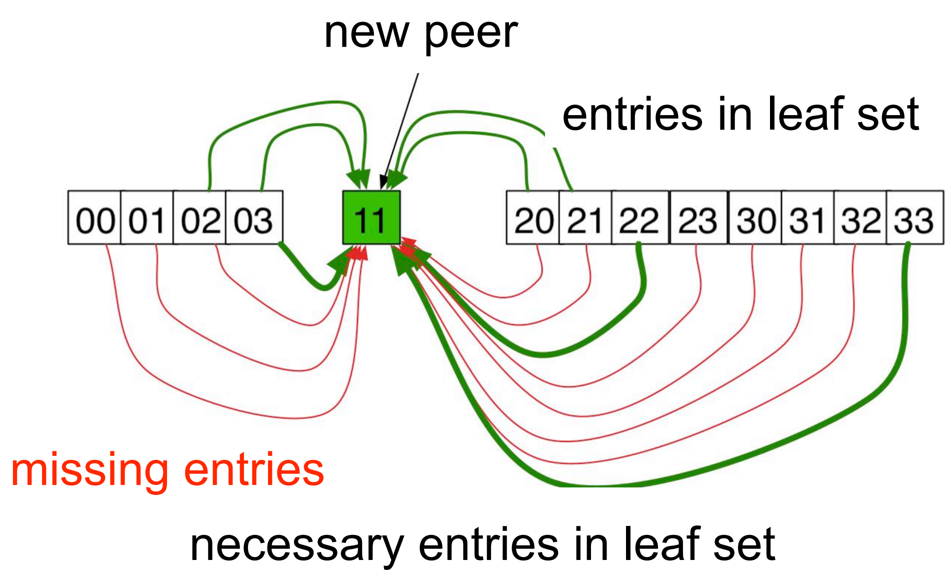
A Peer Enters

- **New node x sends message to the node z with the longest common prefix p**
- **x receives**
  - routing table of z
  - leaf set of z
- **z updates leaf-set**
- **x informs ℓ-leaf set**
- **x informs peers in routing table**
  - with same prefix p (if $\ell/2 < 2^b$)
- **Numbor of messages for adding a peer**
  - ℓ messages to the leaf-set
  - expected (2^b - $\ell/2$) messages to nodes with common prefix
  - one message to z with answer



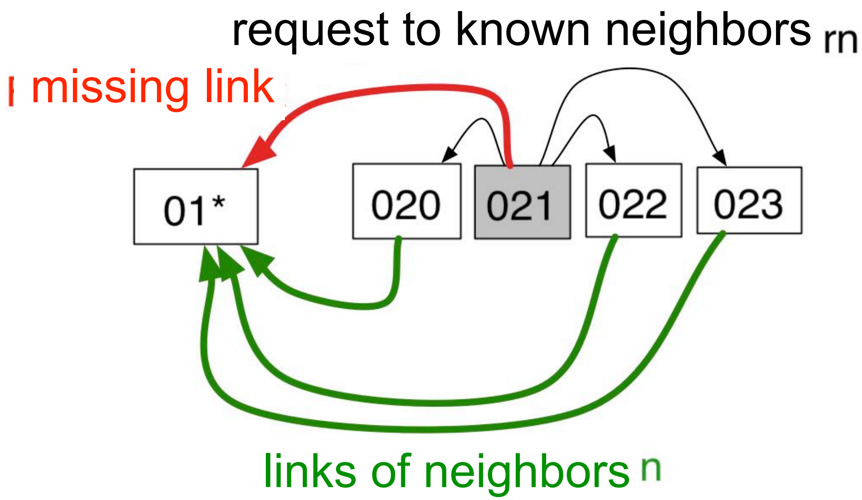
When the Entry-Operation Errs

- **Inheriting the next neighbor routing table does not allows work perfectly**
- **Example**
  - If no peer with 1* exists then all other peers have to point to the new node
  - Inserting 11
  - 03 knows from its routing table
    - 22,33
    - 00,01,02
  - 02 knows from the leaf-set
    - 01,02,20,21
- **11 cannot add all necessary links to the routing tables**



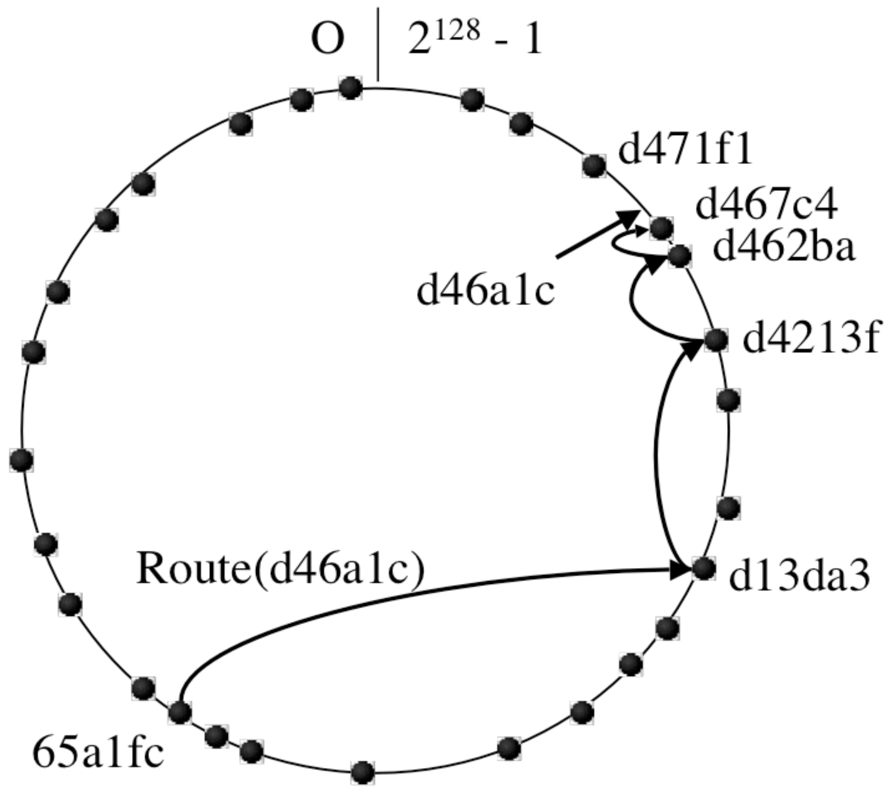
Missing Entries in the Routing Table

- **Assume the entry R_i^j is missing at peer D**
  - j-th row and i-th column of the routing table
- **This is noticed if a message of a peer with such a prefix is received**
- **This may also happen if a peer leaves the network**
- **Contact peers in the same row**
  - if they know a peer this address is copied
- **If this fails then perform routing to the missing link**



Lookup

- **Compute the target ID using the hash function**
- **If the address is within the ℓ-leaf set**
  - the message is sent directly
  - or it discovers that the target is missing
- **Else use the address in the routing table to forward the mesage**
- **If this fails take best fit from all addresses**



Lookup in Detail

- **L:** ℓ**-leafset**
- **R:** **routing table**
- **M:** **nodes in the vicinity of D (according to RTT)**
- **D:** **key**
- **A:** **nodeID of current peer**
- R^i_l**:** **j-th row and i-th column of the routing table**
- L_i**:** **numbering of the leaf set**
- D_i**:** **i-th digit of key D**
- **shl(A):** **length of the largest common prefix of A and D (shared header length)**

if ($L_{-\lfloor|L|/2\rfloor} \leq D \leq L_{\lfloor|L|/2\rfloor}$) {
    // D is within range of our leaf set
    forward to L_i, s.th. $|D - L_i|$ is minimal;
} else {
    // use the routing table
    Let $l = shl(D, A)$;
    if ($R_l^{D_l} \neq null$) {
        forward to $R_l^{D_l}$;
    }
    else {
        // rare case
        forward to $T \in L \cup R \cup M$, s.th.
            $shl(T, D) \geq l$,
            $|T - D| < |A - D|$
    }
}

Routing — Discussion

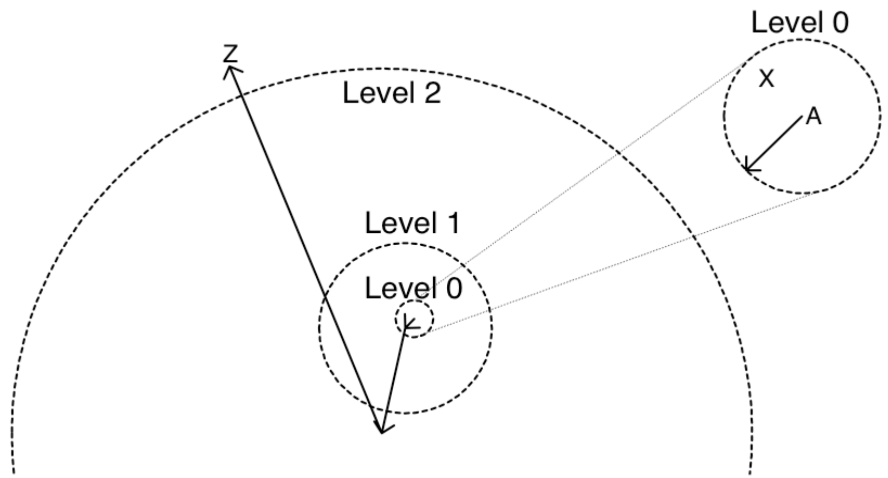
- **If the Routing-Table is correct**
  - routing needs O((log n)/b) messages
- **As long as the leaf-set is correct**
  - routing needs O(n/l) messages
  - unrealistic worst case since even damaged routing tables allow dramatic speedup
- **Routing does not use the real distances**
  - M is used only if errors in the routing table occur
  - using locality improvements are possible
- **Thus, Pastry uses heuristics for improving the lookup time**
  - these are applied to the last, most expensive, hops

Localization of the k Nearest Peers

- **Leaf-set peers are not near, e.g.**
  - New Zealand, California, India, ...
- **TCP protocol measures latency**
  - latencies (RTT) can define a metric
  - this forms the foundation for finding the nearest peers
- **All methods of Pastry are based on heuristics**
  - i.e. no rigorous (mathematical) proof of efficiency
- **Assumption: metric is Euclidean**

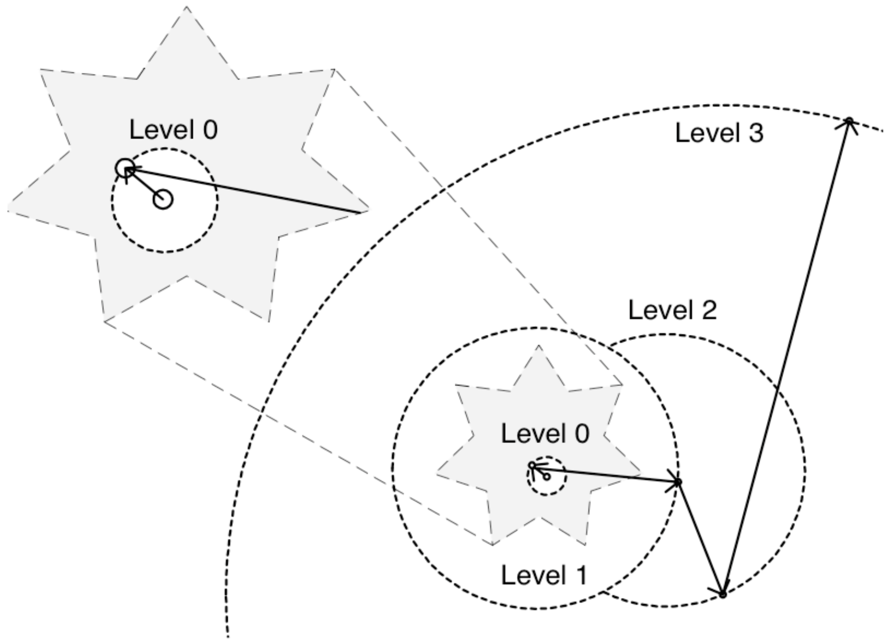
Locality in the Routing Table

- **Assumption**
  - When a peer is inserted the peers contacts a near peer
  - All peers have optimized routing tables
- **But:**
  - The first contact is not necessary near according to the node-ID
- **1st step**
  - Copy entries of the first row of the routing table of P
    - good approximation because of the triangle inequality (metric)
- **2nd step**
  - Contact fitting peer p‘ of p with the same first letter
  - Again the entries are relatively close
- **Repeat these steps until all entries are updated**



Locality in the Routing Table

- **In the best case**
  - each entry in the routing table is optimal w.r.t. distance metric
  - this does not lead to the shortest path
- **There is hope for short lookup times**
  - with the length of the common prefix the latency metric grows exponentially
  - the last hops are the most expensive ones
  - here the leaf-set entries help

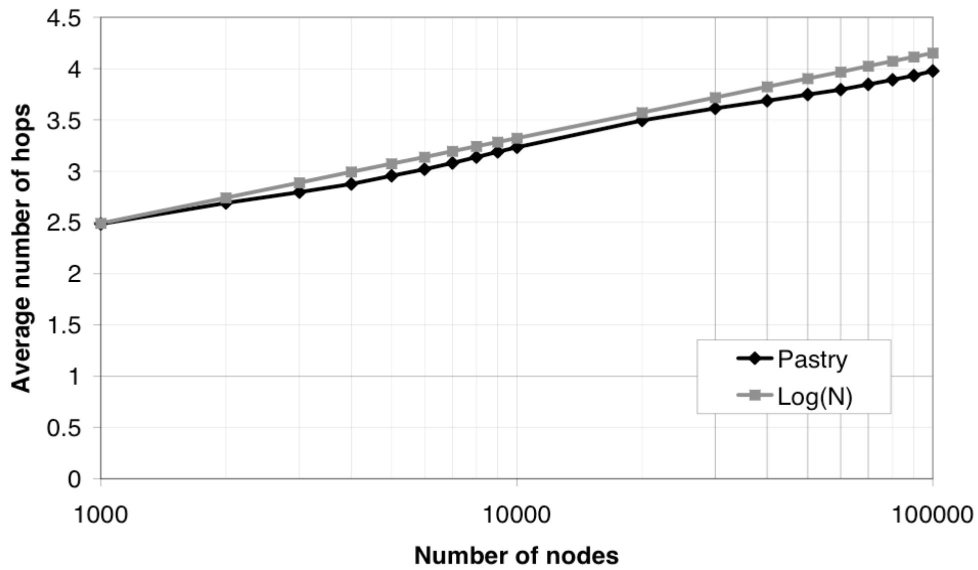


Localization of Near Nodes

- Node-ID metric and latency metric are not compatible
- If data is replicated on k peers then peers with similar Node-ID might be missed
- Here, a heuristic is used
- Experiments validate this approach

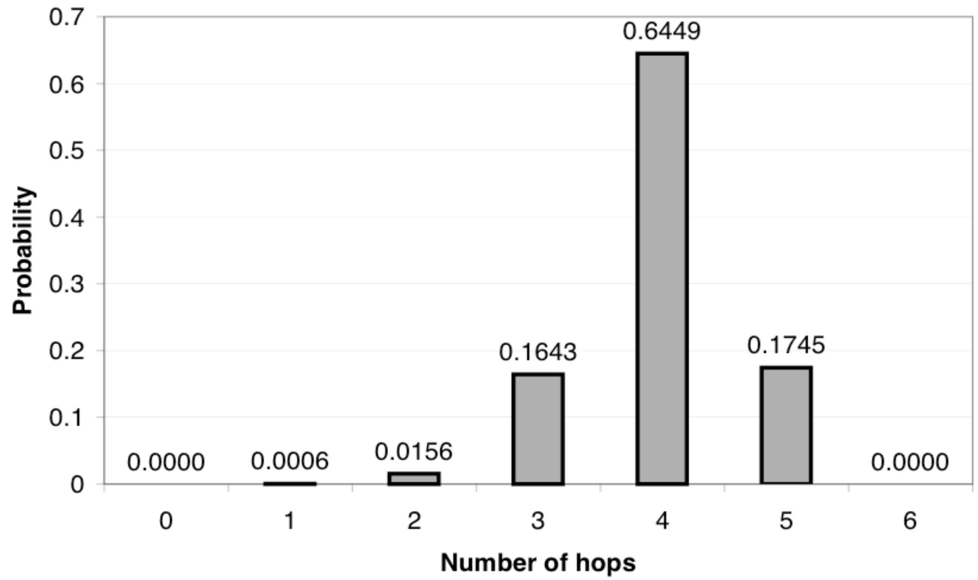
Experimental Results — Scalability

- **Parameter b=4, l=16, M=32**
- **In this experiment the hop distance grows logarithmically with the number of nodes**
- **The analysis predicts O(log n)**
- **Fits well**



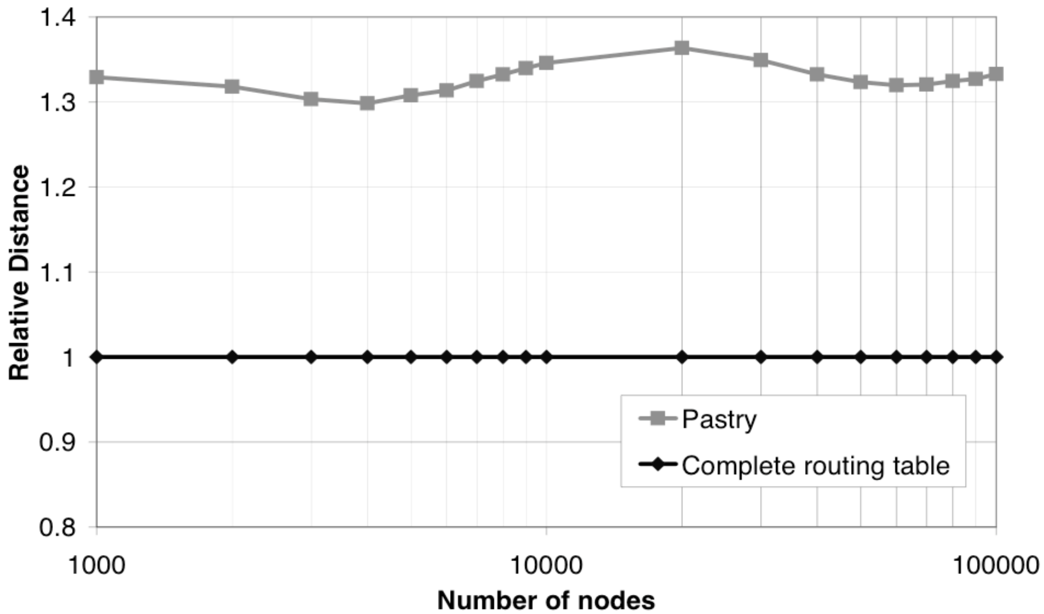
Experimental Results
Distribution of Hops

- **Parameter b=4, l=16, M=32, n = 100,000**
- **Result**
  - deviation from the expected hop distance is extremely small
- **Analysis predicts difference with extremely small probability**
  - fits well



Experimental Results — Latency

- **Parameter b=4, l=16, M=3**
- **Compared to the shortest path astonishingly small**
  - seems to be constant



Distributed Storage

Tapestry

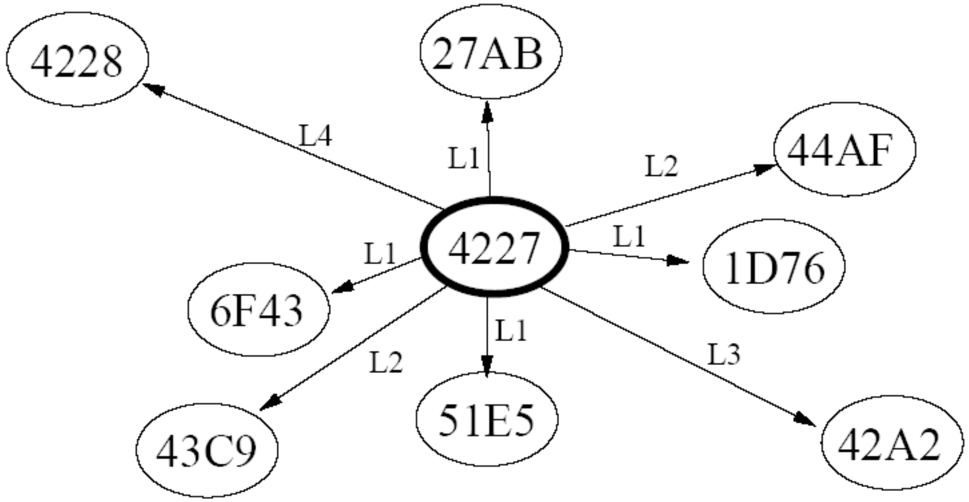
Zhao, Kubiatowicz und Joseph (2001)

Tapestry

- **Objects and Peers are identified by**
  - Objekt-IDs (Globally Unique Identifiers GUIDs) and
  - Peer-IDs
- **IDs**
  - are computed by hash functions
    - like CAN or Chord
  - are strings on basis B
    - B=16 (hexadecimal system)

Neighborhood of a Peer (1)

- **Every peer A maintains for each prefix x of the Peer-ID**
  - if a link to another peer sharing this Prefix x
  - i.e. peer with ID B=xy has a neighbor A, if xy´=A for some y, y´
- **Links sorted according levels**
  - the level denotes the length of the common prefix
  - Level L = |x|+1



Neighborhood Set (2)

- **For each prefix x and all letters j of the peer with ID A**
  - establish a link to a node with prefix xj within the neighboorhood set $N^{A}_{x,j}$
- **Peer with Node-ID A has b |A| neighborhood sets**
- **The neighborhood set of contains all nodes with prefix sj**
  - Nodes of this set are denoted by (x,j)

Example of Neighborhood Sets

Neighborhood set of node 4221

	Level 4	Level 3	Level 2	Level 1
j=0	4220	420?	40??	0???
j=1	4221	421?	41??	1???
.	4222	422?	42??	2???
.	4223	423?	43??	3???
.	4224	424?	44??	4???
.	4225	425?	45??	5???
.	4226	426?	46??	6???
j=7	4227	427?	47??	7???

Links

- **For each neighborhood set at most k Links are maintained**

$$k \geq 1 : \left| N^{A}_{x,j} \right| \leq k$$

- **Note:**
  - some neighborhood sets are empty

Properties of Neighborhood Sets

- **Consistency**
  - If $N^A_{x,j} = \emptyset$ for any A
    - then there are no (x,j) peers in the network
    - this is called a hole in the routing table of level |x|+1 with letter j
- **Network is always connected**
  - Routing can be done by following the letters of the ID $b_1 b_2 \ldots b_n$

N^{A}_{ϕ, b_1} 1st hop to node A_1

$N^{A_1}_{b_1, b_2}$ 2nd hop to node A_2

$N^{A_2}_{b_1 o b_2, b_3}$ 3rd hop to node A_3

...

Locality

- **Metric**
  - e.g. given by the latency between nodes
- **Primary node of a neighborhood set** $N_{x,j}^{A}$
  - The closest node (according to the metric) in the neighborhood set of A is called the primary node
- **Secondary node**
  - the second closest node in the neighborhood set
- **Routing table**
  - has primary and secondary node of the neighborhood table

Root Node

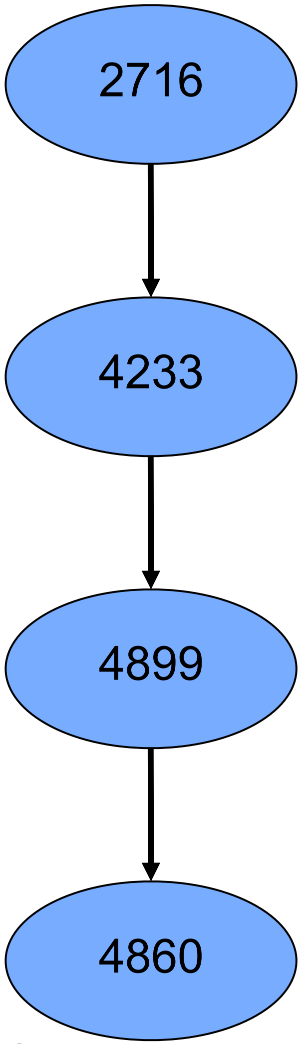
- **Object with ID Y should stored by a so-called Root Node with this ID**
- **If this ID does not exist then a deterministic choice computes the next best choice sharing the greatest commen prefix**

Surrogate Routing

- **Surrogate Routing**
  - compute a surrogate (replacement root node)
  - If (x,j) is a hole, then choose (x,j+1),(x,j+2),…,(x,B),(x, 0), ..., (x,j-1) until a node is found
  - Continue search in the next higher if no node has been found

Example: Surrogate Routing

- **Lookup of 4666 by peer 2716**



Level 1, j=4

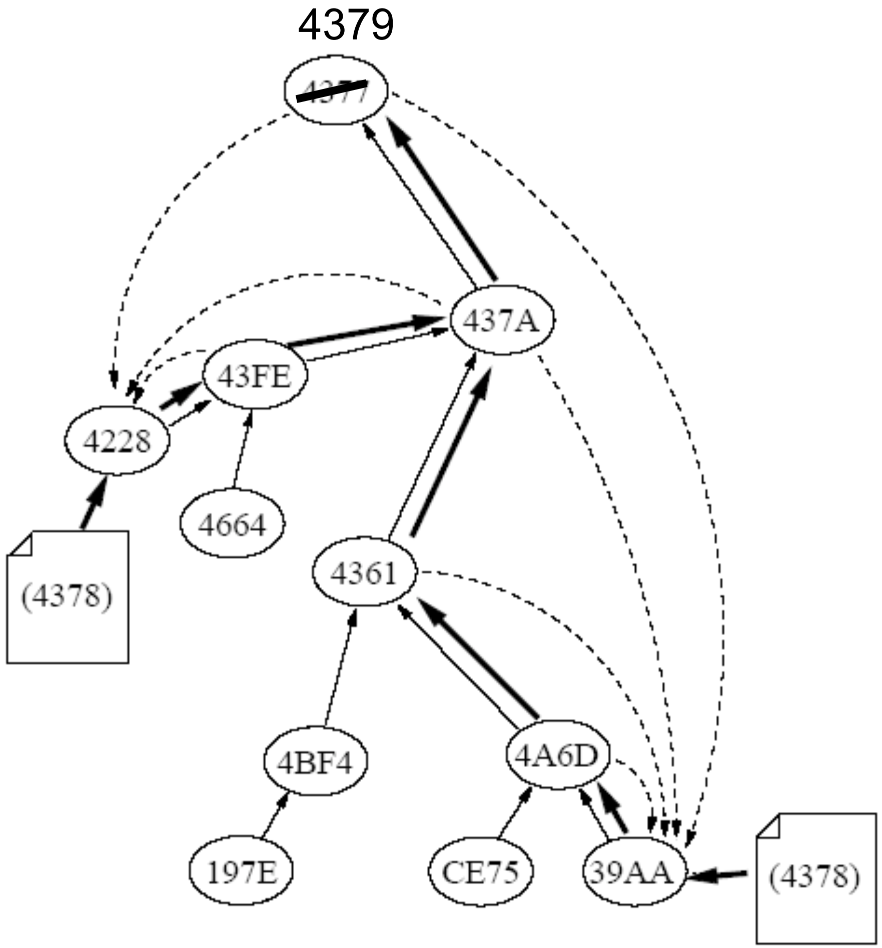
Level 2, j=6 does not exist, next link j=8

Level 3, j=6

Peer 4860 has no level 4 neighbors => end of search

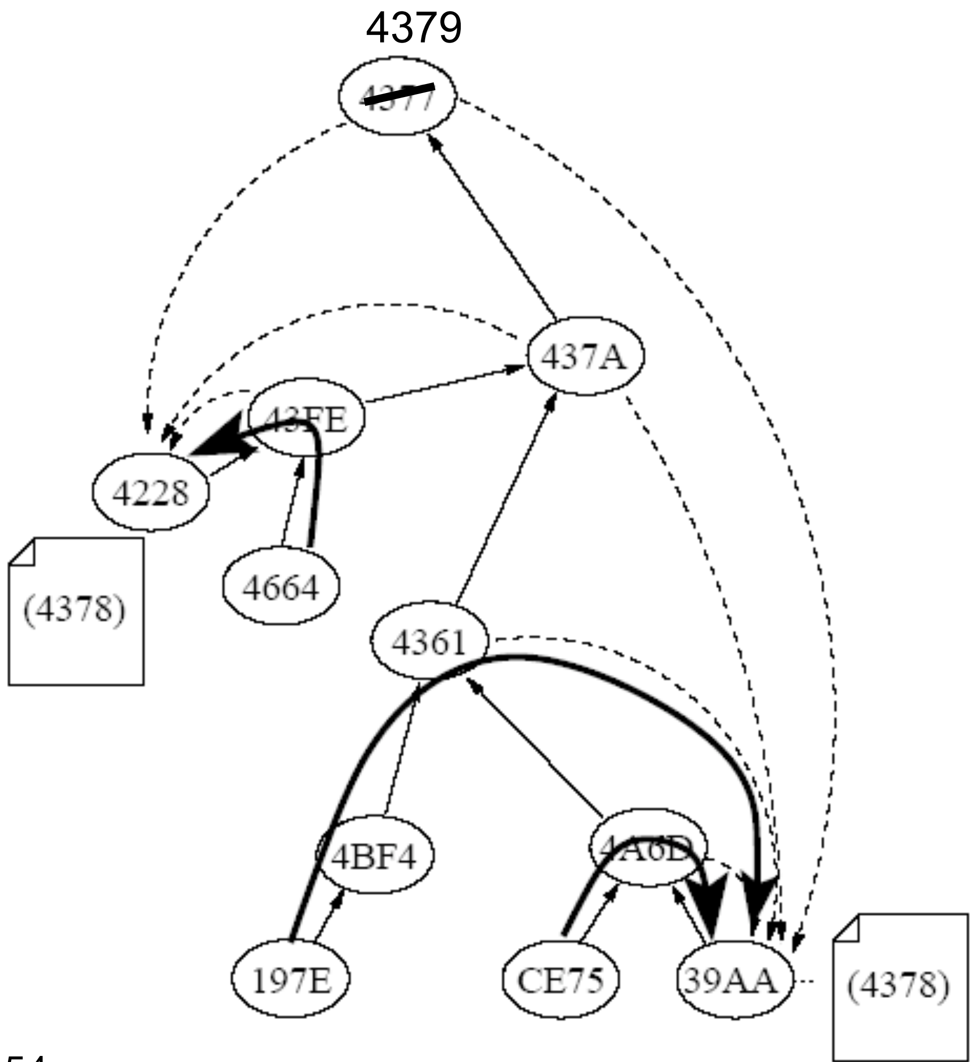
Publishing Objects

- **Peers offering an object (storage servers)**
  - send message to the root node
- **All nodes along the search path store object pointers to the storage server**



Lookup

- **Choose the root node of Y**
- **Send a message to this node**
  - using primary nodes
- **Abort search if an object link has been found**
  - then send message to the storage server



Fault Tolerance

- **Copies of object IDs**
  - use different hash functions for multiple root nodes for objects
  - failed searches can be repeated with different root nodes
- **Soft State Pointer**
  - links of objects are erased after a designated time
  - storage servers have to republish
    - prevents dead links
    - new peers receive fresh information

Surrogate Routing

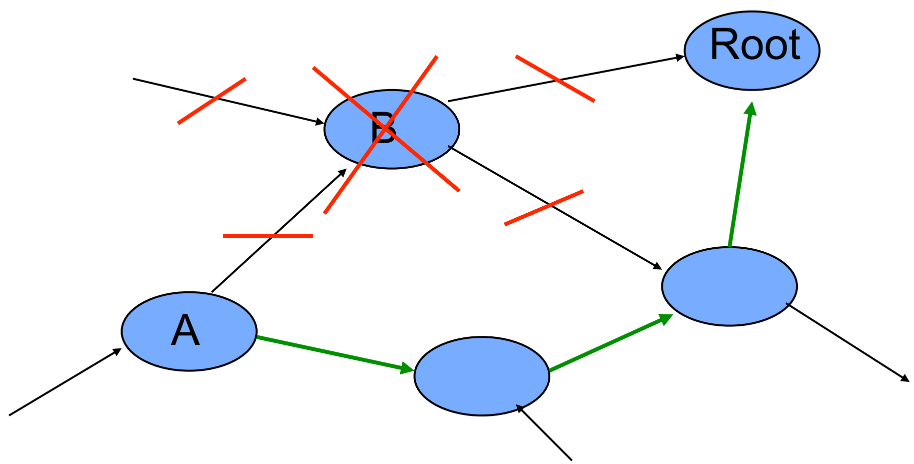
- **Theorem**
  - Routing in Tapestry needs O(log n) hops with high probability

Adding Peers

- **Perform lookup in the network for the own ID**
  - every message is acknowledged
  - send message to all neighbors with fitting prefix,
    - Acknowledged Multicast Algorithm
- **Copy neighborhood tables of surrogate peer**
- **Contact peers with holes in the routing tables**
  - so they can add the entry
  - for this perform multicast algorithm for finding such peers

Leaving of Peers

- **Peer A notices that peer B has left**
- **Erase B from routing table**
  - Problem holes in the network can occur
- **Solution: Acknowledged Multicast Algorithm**
- **Republish all object with next hop to root peer B**



Pastry versus Tapestry

- **Both use the same routing principle**
  - Plaxton, Rajamaran und Richa
  - Generalization of routing on the hyper-cube
- **Tapestry**
  - is not completely self-organizing
  - takes care of the consistency of routing table
  - is analytically understood and has provable performance
- **Pastry**
  - Heuristic methods to take care of leaving peers
  - More practical (less messages)
  - Leaf-sets provide also robustness

Distributed Storage

Past

Druschel, Rowstron

2001

PAST

- **PAST: A large-scale, persistent peer-to-peer storage utility**
  - by Peter Druschel (Rice University, Houston – now Max-Planck-Institut, Saarbrücken/Kaiserlautern)
  - and Antony Rowstron (Microsoft Research)
- **Literature**
  - A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility", 18th ACM SOSP'01, 2001.
    - all pictures from this paper
  - P. Druschel and A. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility", HotOS VIII, May 2001.

Goals of PAST

- **Peer-to-Peer based Internet Storage**
  - on top of Pastry
- **Goals**
  - File based storage
  - High availability of data
  - Persistent storage
  - Scalability
  - Efficient usage of resources

Motivation

- **Multiple, diverse nodes in the Internet can be used**
  - safety by different locations
- **No complicated backup**
  - No additional backup devices
  - No mirroring
  - No RAID or SAN systems with special hardware
- **Joint use of storage**
  - for sharing files
  - for publishing documents
- **Overcome local storage and data safety limitations**

Interface of PAST

- **Create:**
  `fileId = Insert(name, owner-credentials, k, file)`
  - stores a file at a user-specified number k of divers nodes within the PAST network
  - produces a 160 bit ID which identifies the file (via SHA-1)
- **Lookup:**
  `file = Lookup(fileId)`
  - reliably retrieves a copy of the file identified fileId
- **Reclaim:**
  `Reclaim(fileId, owner-credentials)`
  - reclaims the storage occupied by the k copies of the file identified by fileId
- **Other operations do not exist:**
  - No erase
    - to avoid complex agreement protocols
  - No write or rename
    - to avoid write conflicts
  - No group right management
    - to avoid user, group managements
  - No list files, file information, etc.
- **Such operations must be provided by additional layer**

Relevant Parts of Pastry

- **Leafset:**
  - Neighbors on the ring
- **Routing Table**
  - Nodes for each prefix + 1 other letter
- **Neighborhood set**
  - set of nodes which have small TTL

NodeId 10233102

Leaf set

SMALLER		LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Interfaces of Pastry

- **route(M, X):**
  - route message M to node with nodeId numerically closest to X
- **deliver(M):**
  - deliver message M to application
- **forwarding(M, X):**
  - message M is being forwarded towards key X
- **newLeaf(L):**
  - report change in leaf set L to application

Insert Request Operation

- **Compute fileId by hashing**
  - file name
  - public key of client
  - some random numbers, called salt
- **Storage (k x filesize)**
  - is debited against client‘s quota
- **File certificate**
  - is produced and signed with owner‘s private key
  - contains fileID, SHA-1 hash of file‘s content, replciation factor k, the random salt, creation date, etc.
- **File and certificate are routed via Pastry**
  - to node responsible for fileID
- **When it arrives in one node of the k nodes close to the fileId**
  - the node checks the validityof the file
  - it is duplicated to all other k-1 nodes numerically close to fileId
- **When all k nodes have accepted a copy**
  - Each nodes sends store receipt is send to the owner
- **If something goes wrong an error message is sent back**
  - and nothing stored

Lookup

- **Client sends message with requested fileId into the Pastry network**
- **The first node storing the file answers**
  - no further routing
- **The node sends back the file**
- **Locality property of Pastry helps to send a close-by copy of a file**

Reclaim

- **Client‘s nodes sends reclaim certificate**
  - allowing the storing nodes to check that the claim is authentificated
- **Each node sends a reclaim receipt**
- **The client sends this recept to the retrieve the storage from the quota management**

Security

- **Smartcard**
  - for PAST users which want to store files
  - generates and verifies all certificates
  - maintain the storage quotas
  - ensure the integrity of nodeID and fileID assignment
- **Users/nodes without smartcard**
  - can read and serve as storage servers
- **Randomized routing**
  - prevents intersection of messages
- **Malicious nodes only have local influence**

Storage Management

- **Goals**
  - Utilization of all storage
  - Storage balancing
  - Providing k file replicas
- **Methods**
  - Replica diversion
    - exception to storing replicas nodes in the leafset
  - File diversion
    - if the local nodes are full all replicas are stored at different locations

Causes of Storage Load Imbalance

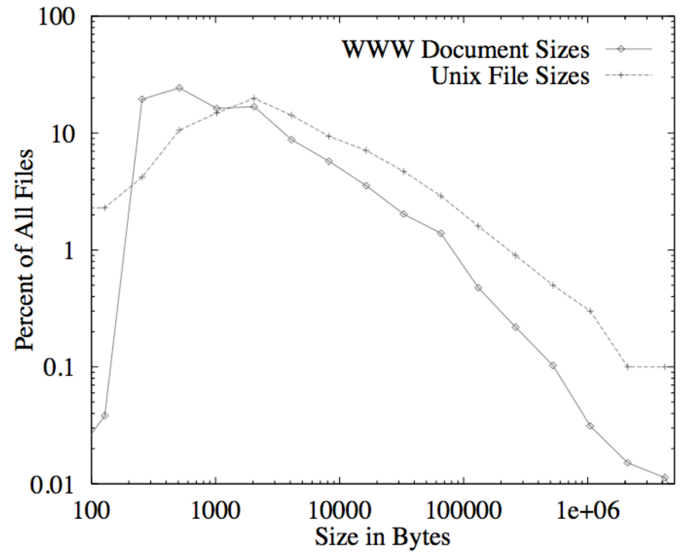
- **Statistical variation**
  - birthday paradoxon (on a weaker scale)
- **High variance of the size distribution**
  - Typical heavy-tail distribution, e.g. Pareto distribution
- **Different storage capacity of PAST nodes**

Heavy Tail Distribution

- **Discrete Pareto Distribution for x ∈ {1,2,3,…}**

$$\mathbf{P}[X = x] = \frac{1}{\zeta(\alpha) \cdot x^\alpha}$$

  - with constant factor $\zeta(\alpha) = \sum_{i=1}^{\infty} \frac{1}{i^\alpha}$
- **Heavy tail**
  - only for small k moments $E[X^k]$ are defined
  - Expectation is defined only if α>2
  - Variance and $E[X^2]$ only exist if α>3
  - $E[X^k]$ is defined ony if α>k+1
- **Often observed:**
  - Distribution of wealth, sizes of towns, frequency of words, length of molecules, ...,
  - file length, WWW documents
    - Heavy-Tailed Probability Distributions in the World Wide Web, Crovella et al. 1996



Per-Node Storage

- **Assumption:**
  - Storage of nodes differ by at most a factor of 100
- **Large scale storage**
  - must be inserted as multiple PAST nodes
- **Storage control:**
  - if a node storage is too large it is asked to split and rejoin
  - if a node storage is too small it is rejected

Replica Diversion

- **The first node close to the fileId checks whether it can store the file**
  - if yes, it does and sends the store receipt
- **If a node A cannot store the file, it tries replica diversion**
  - A chooses a node B in its leaf set which is not among the k closest asks B to store the copy
  - If B accepts, A stores a pointer to B and sends a store receipt
- **When A or B fails then the replica is inaccessible**
  - failure probability is doubled

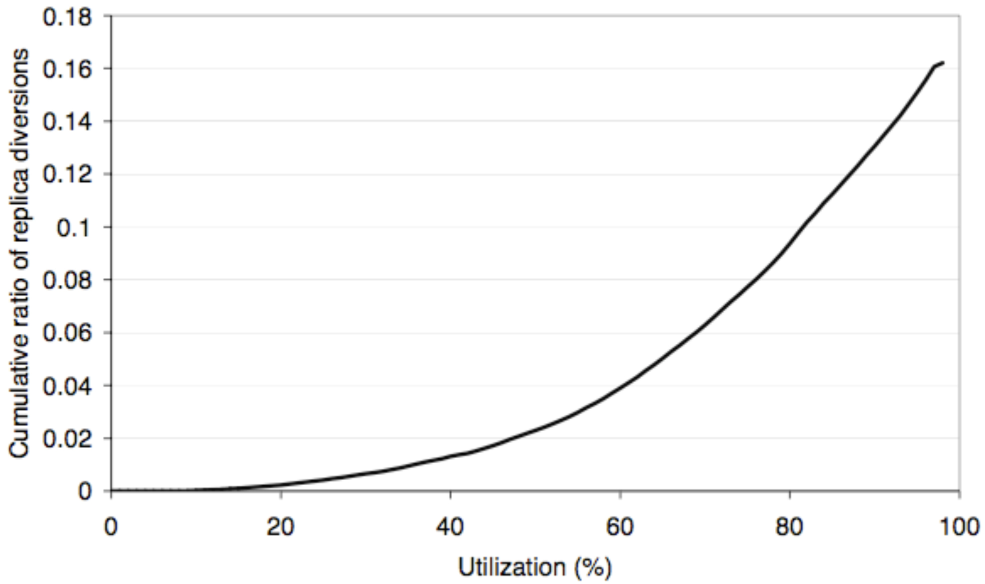


Figure 5: Cumulative ratio of replica diversions versus storage utilization, when $t_{pri} = 0.1$ and $t_{div} = 0.05$.

Policies for Replica Diversion

- **Acceptance of replicas at a node**
  - If (size of a file)/(remaining free space) > t then reject the file
    - for different t`s for close nodes (t_{pri}) and far nodes (t_{div}), where $t_{pri} > t_{div}$
  - discriminates large files and far storage
- **Selecting a node to store a diverted replica**
  - in the leaf set and
  - not in the k nodes closest to the fileId
  - do not hold a diverted replica of the same file
- **Deciding when to divert a file to different part of the Pastry ring**
  - If one of the k nodes does not find a proxy node
  - then it sends a reject message
  - and all nodes for the replicas discard the file

File Diversion

- **If k nodes close to the chosen fileId**
  - cannot store the file
  - nor divert the replicas locally in the leafset
- **then an error message is sent to the client**
- **The client generates a new fileId using different salt**
  - and repeats the insert operation up to 3 times
  - then the operation is aborted and a failure is reported to the application
- **Possibly the application retries with small fragments of the file**

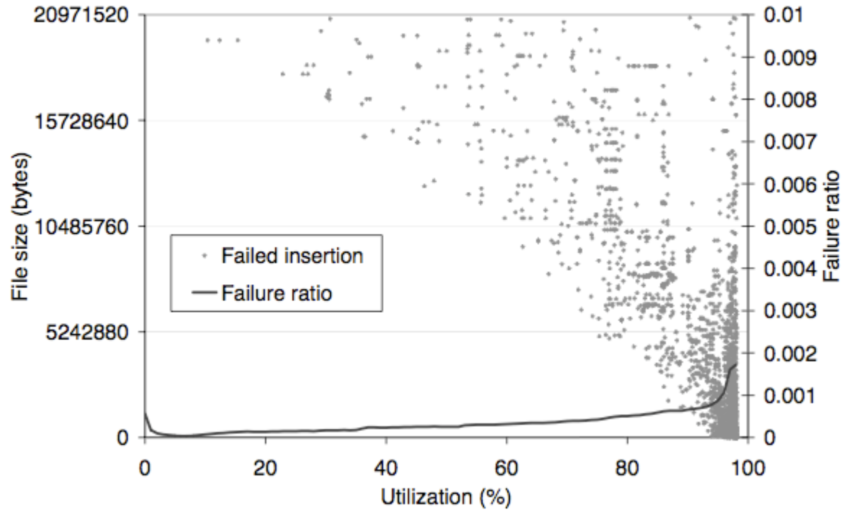


Figure 7: File insertion failures versus storage utilization for the filesystem workload, when $t_{pri} = 0.1$, $t_{div} = 0.05$.

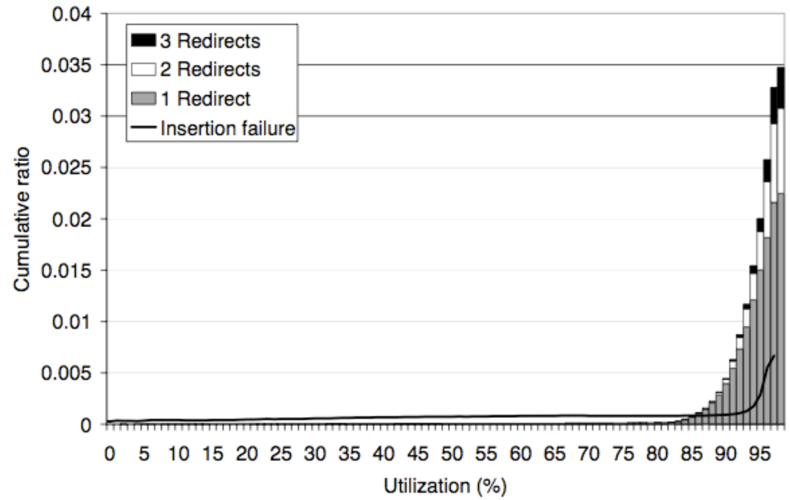


Figure 4: Ratio of file diversions and cumulative insertion failures versus storage utilization, $t_{pri} = 0.1$ and $t_{div} = 0.05$.

Maintaining Replicas

- **Pastry protocols checks leaf set periodically**
- **Node failure has been recognized**
    - if a node is unresponsive for some certain time
  - Pastry triggers adjustment of the leaf set
    - PAST redistributes replicas
  - if the new neighbor is too full, then other nodes in the nodes will be uses via replica diversion
- **When a new node arrives**
  - files are not moved, but pointers adjusted (replica diversion)
  - because of ratio of storage to bandwidth

File Encoding

- **k replicas is not the best redundancy strategy**
- **Using a Reed-Solomon encoding**
  - with m additional check sum blocks to n original data blocks
  - reduces the storage overhead to (m+n)/n times the file size
    - if all m+n shares are distributed over different nodes
  - possibly speeds upt the access spee
- **PAST**
  - does NOT use any such encoding techniques

Caching

- **Goal:**
  - Minimize fetch distance
  - Maximize query throughput
  - Balance the query load
- **Replicas provide these features**
  - Highly popular files may demand many more replicas
    - this is provided by cache management
- **PAST nodes use „unused“ portion to cache files**
  - cached copies can be erased at any time
    - e.g. for storing primary of redirected replicas
- **When a file is routed through a node during lookup or insert it is inserted into the local cache**
- **Cache replacement policy: GreedyDual-Size**
  - considers aging, file size and costs of a file

Experimental Results Caching

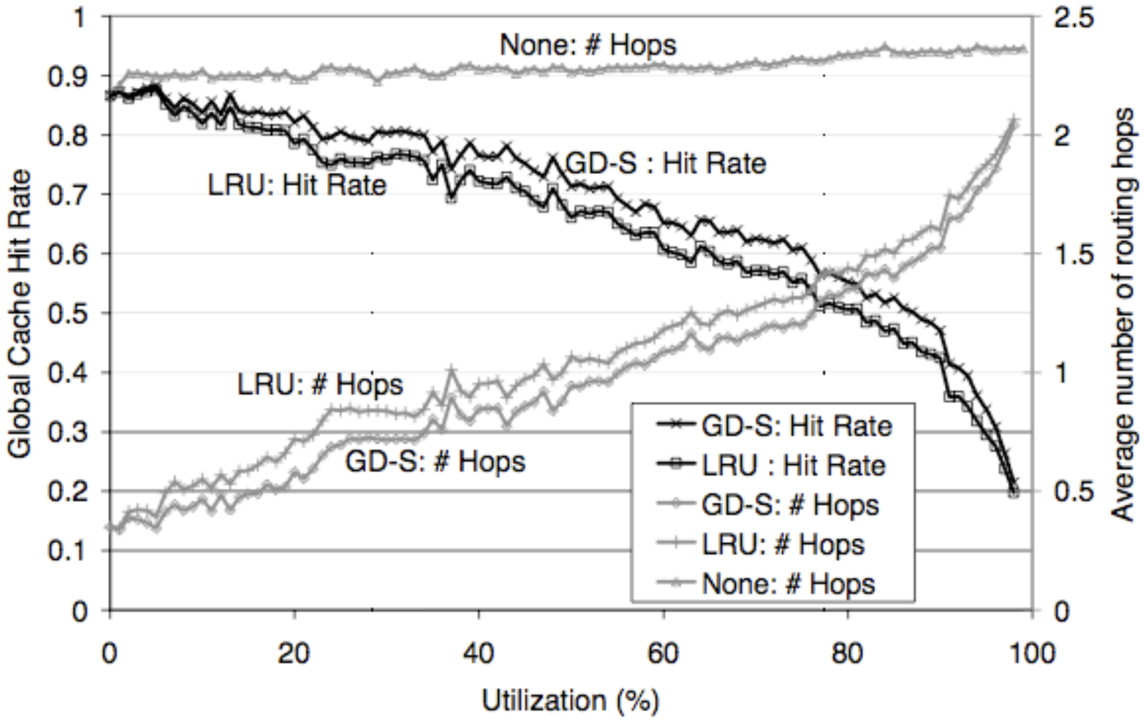


Figure 8: Global cache hit ratio and average number of message hops versus utilization using Least-Recently-Used (LRU), GreedyDual-Size (GD-S), and no caching, with $t_{pri} = 0.1$ and $t_{div} = 0.05$.

Summary

- **PAST provides a distributed storage system**
  - which allows full storage usage and locality features
- **Storage management**
  - based ond Smartcard system
    - provides a hardware restriction
  - utilization moderately increases failure rates and time behavior

Distributed Storage

Oceanstore

Kubiatowicz et al. 2000

Oceanstore

- **Global utility infrastructure providing continuous access to persistent information based on peer-to-peer network Tapestry**
- **Literature**
  - **OceanStore: An Extremely Wide-Area Storage System**
    - John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, Ben Zhao. U.C. Berkeley Technical Report UCB//CSD-00-1102, March 1999
  - **OceanStore: An Architecture for Global-Scale Persistent Storage**
    - John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, Ben Zhao.. ASPLOS 2000
  - **Extracting Guarantees from Chaos**,
    - John D. Kubiatowicz. Communications of the ACM, Vol 46, No. 2, February 2003
  - **Pond: the OceanStore Prototype**,
    - Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. FAST '03

Motivation of Oceanstore

- **Efficient distributed storage providing**
  - Availability
    - uninterrupted operation
  - Durability
    - information entered survives for some 1000 years
  - Access control
    - only authorized read/write
  - Authenticity
    - no publishing of forged documents
  - Robustness against attacks
    - e.g. denial of service

- **Goals**
  - Massive scalability
    - works with billions of clients
  - Anonymity
    - hard to determine producer and reader of a document
  - Deniability
    - users can deny knowledge of data
  - Resistance to censorship
- **Challenge**
  - coping with untrusted, unreliable, possibly evil peers

Example Applications

- **Storage server**
  - storing, retrieving, publishing documents
- **E-Mail**
  - distributed IMAP
- **Multimedia application**
  - with stream operations like append, truncate, etc.
- **Database Application**
  - ACID database semantics
    - i.e. atomicity, consistency, isolation, durability

First Goal

- **Work with untrusted infrastructure**
  - servers may crash without warning
  - network keeps on changing
  - may leak or spy on information
  - only clients can be trusted with cleartext
- **Assumption:**
  - servers work correctly most of the time
  - a certain class of servers can be trusted
    - regarding correctness
    - but may need read our data

2nd Goal

- **Data**
  - can be cached everywhere anytime
  - can float freely
- **Nomadic Data**
  - Information is separated from physical location
  - complicated data coherence and location
- **Introspective monitoring**
  - used to discover relationship of objects
  - information is used for *locality* management

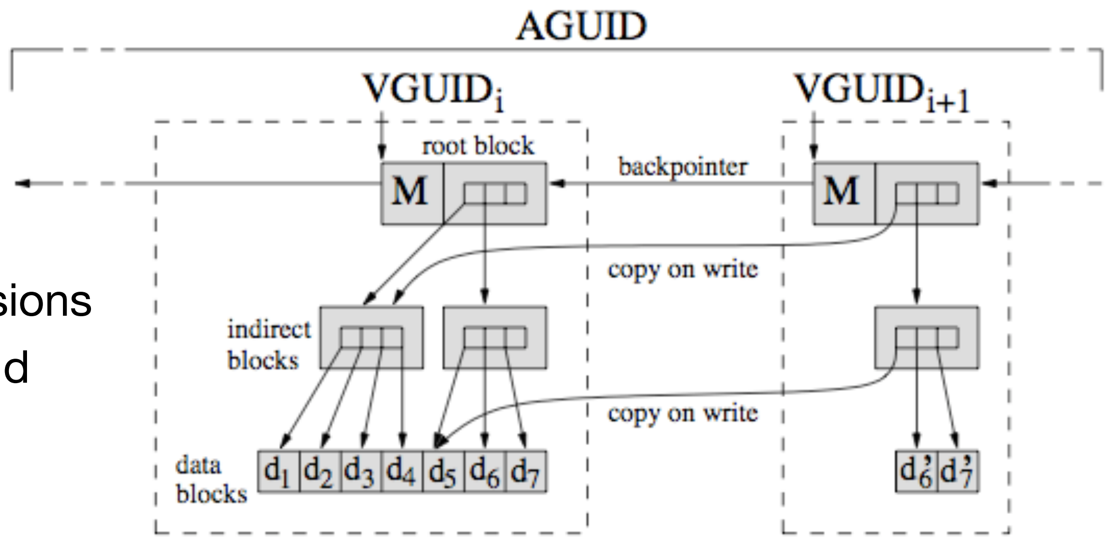
System Overview

- **Persistent object**
  - named by GUID (globally unique identifiers)
  - replicated and stored on multiple servers
  - replicas are independent from the server
    - floating replicas
- **Locating objects and replicas**
  - fast probabilistic algorithm for detecting nearby copies
  - slower deterministic algorithm for robust lookup
- **Modifying objects by updates**
  - every update creates a new version
  - consistency is based on versioning
  - cleaner recovery
  - supports permanent pointers
- **Active and archival forms of objects**
  - active form
    - latest version
  - archival form
    - permanent, read-only version
    - stored by erasure codes
    - spread over 100s or 1000s of servers
    - deep archival storage

Virtualization

- **Based on Tapestry**
- **Each peer has a GUID**
  - globally unique identifier
- **Decentralized object location and routing**
  - Tapestry as overlay networks provides it
  - Built upon TCP/IP
  - Addressing by GUID inside Tapestry, not by IP-address
- **Hosts**
  - publish the GUIDs of their resources
  - may unpublish or leave the network at any time

Data Model

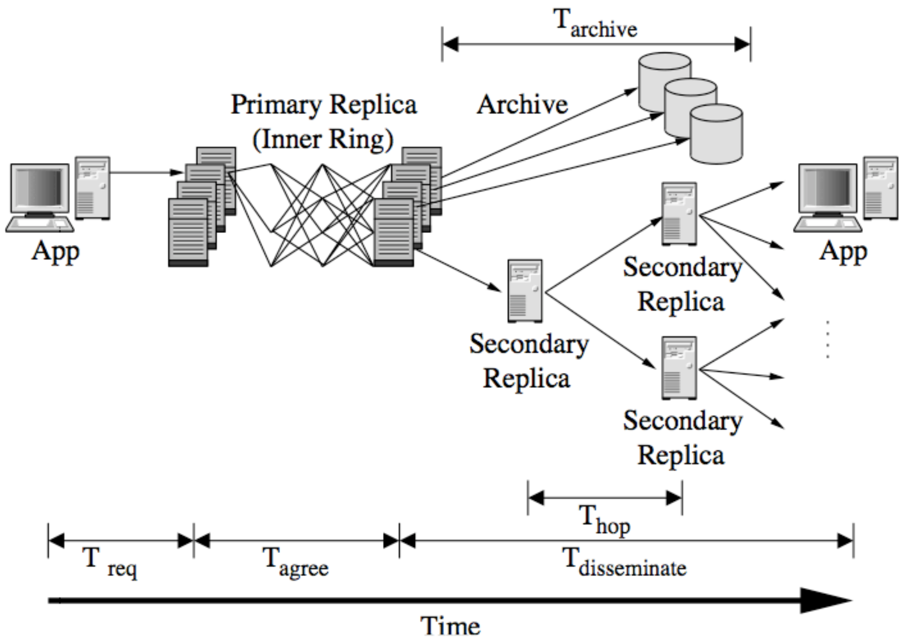


- **Data object**
  - analog of file
  - ordered sequence of read-only versions
  - allows „time travel“, i.e. revisiting old versions
  - allows recovering of deleted data
- **B-tree**
  - organizes blocks of a data objects
  - pointers reuse old blocks

- **BGUID**
  - block GUID
  - secure hash of a block of data
- **VGUID**
  - version GUID
  - BGUID of the root block of a version
- **AGUID**
  - active GUID
  - names a complete stream of versions

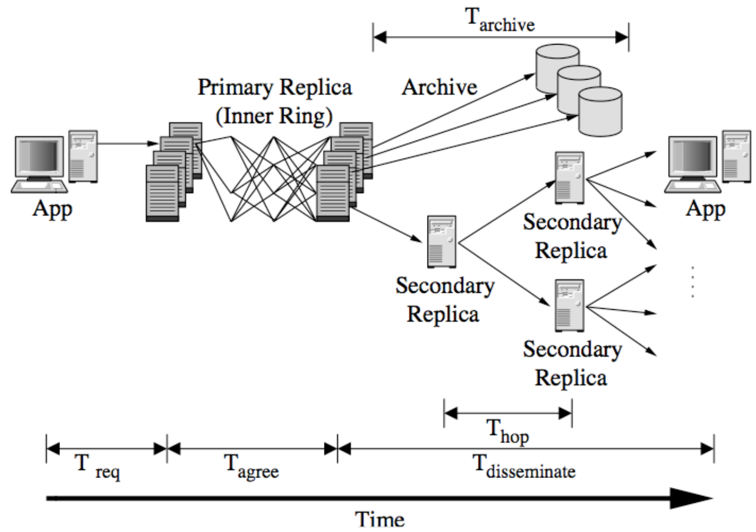
Replication

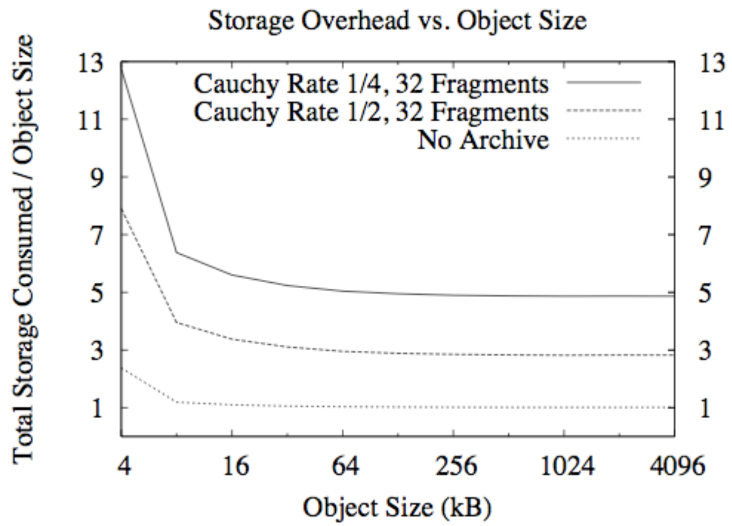
- **Primary replica**
  - unique first appearance of each object
  - addressed by AGUID
  - serializes and applies all updates to the object
  - enforces access control restrictions
- **Certificate**
  - called heartbeat
  - tuple containing AGUID, VGUID of most recent version, sequence number
- **Primary replicas are implemented on a set of servers**
  - Use Byzantine-fault-tolerant cryptographic protocol of Castro and Liskov



Replication: Archival Storage

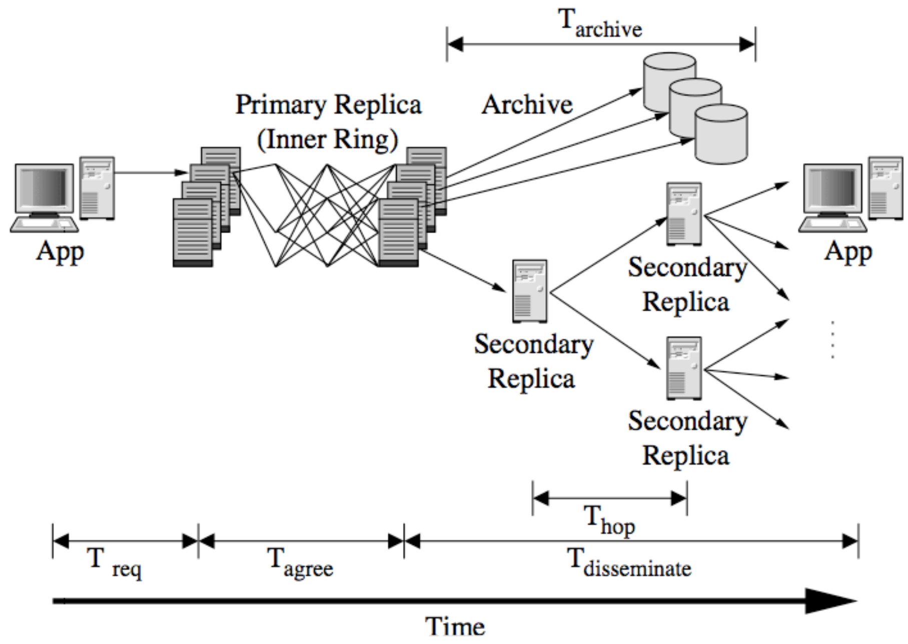
- **Uses Erasure Codes**
  - a block is divided in to m fragments
  - encoded into n>m fragments
    - e.g. by Reed-Solomon
  - r = m/n is rate of encoding
  - storage cost increases by a factor of 1/r
- **Reconstruction**
  - can be done from any m fragments
- **Prototype Pond uses**
  - rate 1/2-code with m=16 gives 32 fragments
  - provides higher fault tolerance
- **Each replica**
  - will be erasure-coded and stored using Tapestry within the network





Replication: Caching

- **Reconstruction of erasure codes is expensive**
- **Blocks are cached withoud encoding**
- **If a host queries Tapestry for a block**
  - Tapestry checks for cached blocks
  - If it does not exist, Tapestry performs decoding
  - Then Tapestry stores the copies
    - second replicas
  - Blocks are stored in soft-state
    - can be erased at any time
- **Caching in Oceanstore prototype uses Least-Recently-Used (LRU) strategy**



The Problem of Byzantine Generals

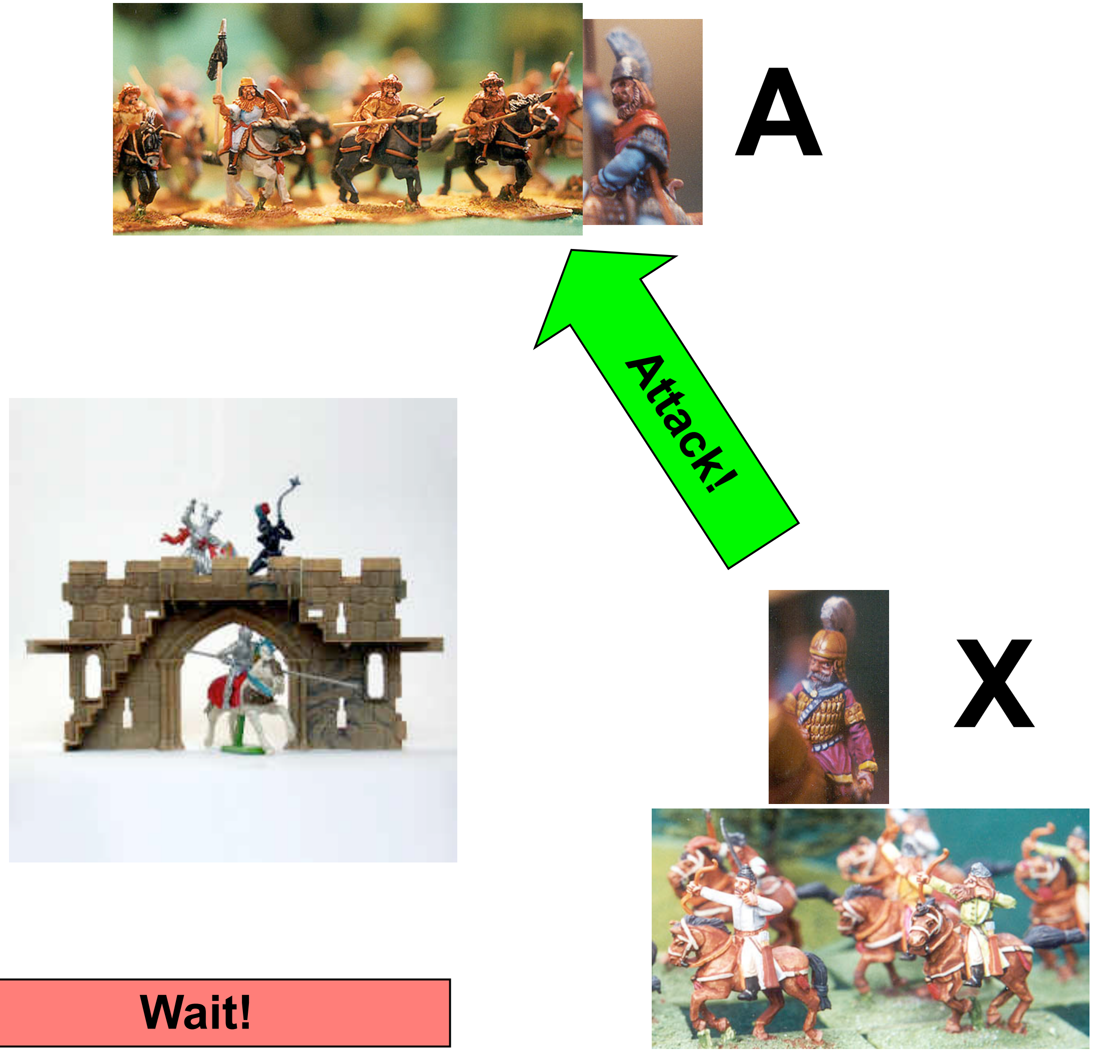
- **3 armies prepare to attack a castle**
- **They are separated and communicate by messengers**
- **If one army attacks alone, it loses**
- **If two armies attack, they win**
- **If nobody attacks the castle is besieged and they win**
- **One general is a renegade**
    - nobody knows who

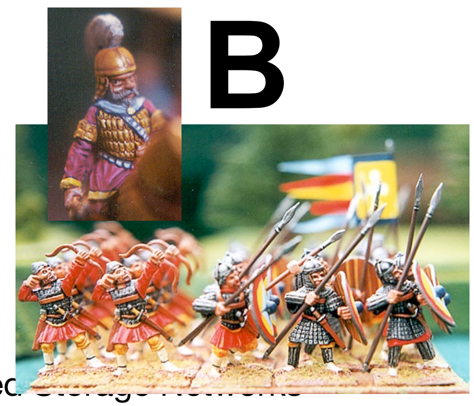




The Problem of Byzantine Generals

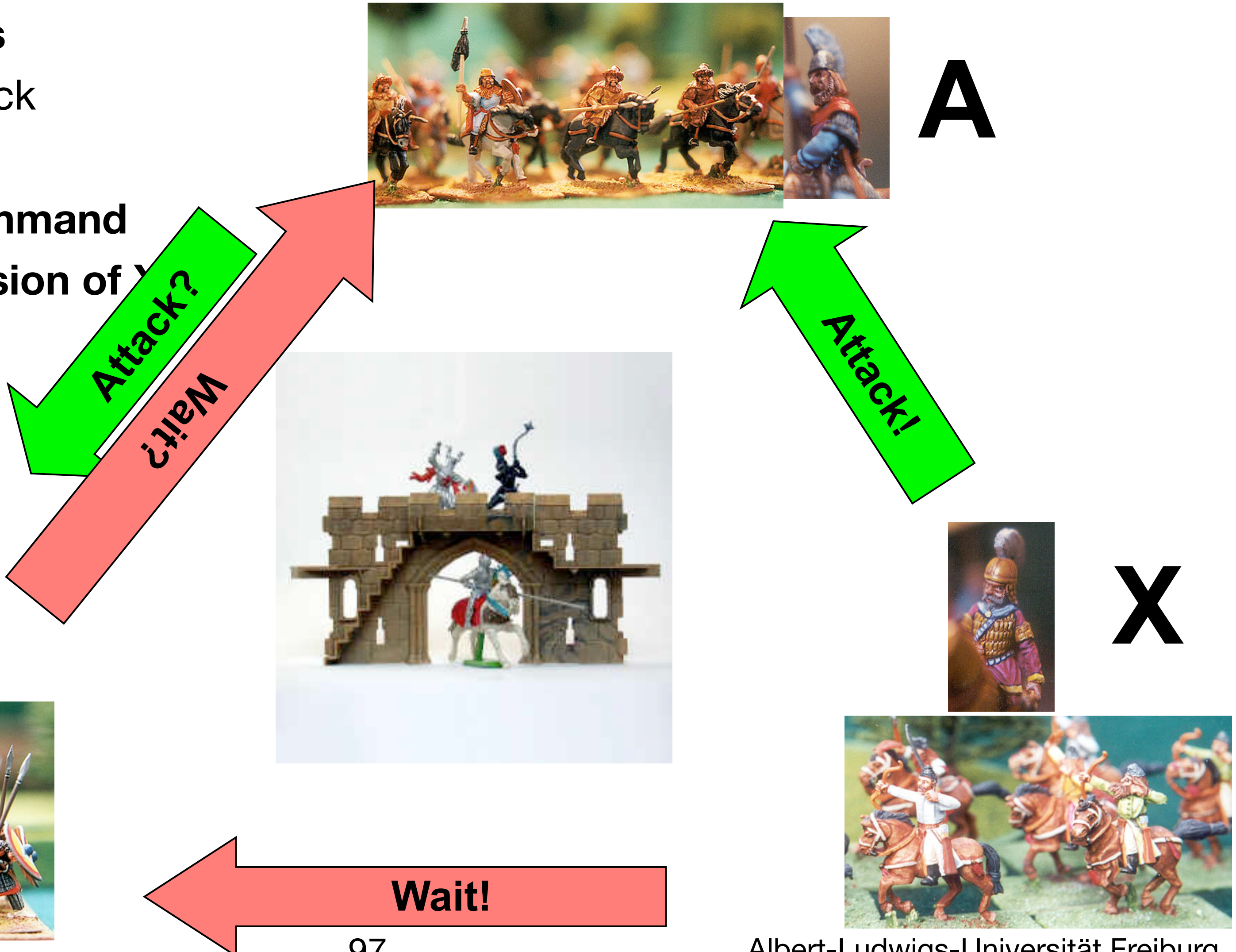
- **The evil general X tries**
  - to convince A to attack
  - to convince B to wait
- **A tells B about X‘s command**
- **B tells B about his version of X‘s command**
  - contradiction
- **But is A, B, or X lying?**

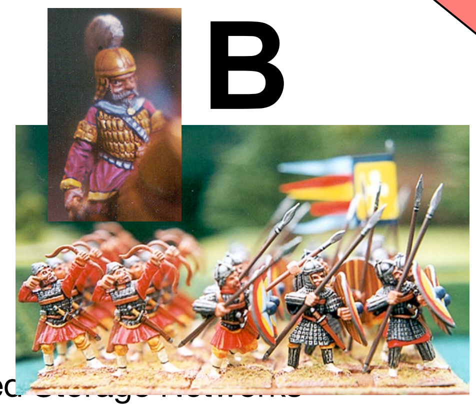




The Problem of Byzantine Generals

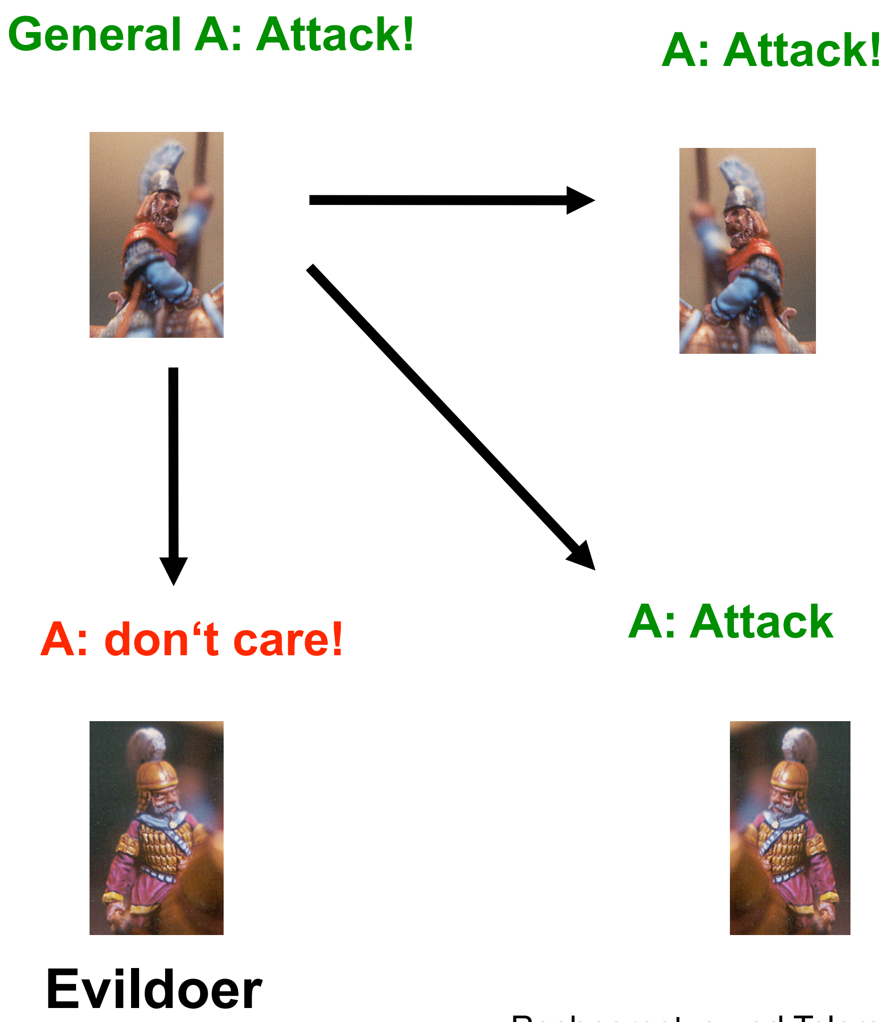
- **The evil general X tries**
  - to convince A to attack
  - to convince B to wait
- **A tells B about X‘s command**
- **B tells B about his version of X command**
  - contradiction
- **But is A, B, or X lying?**





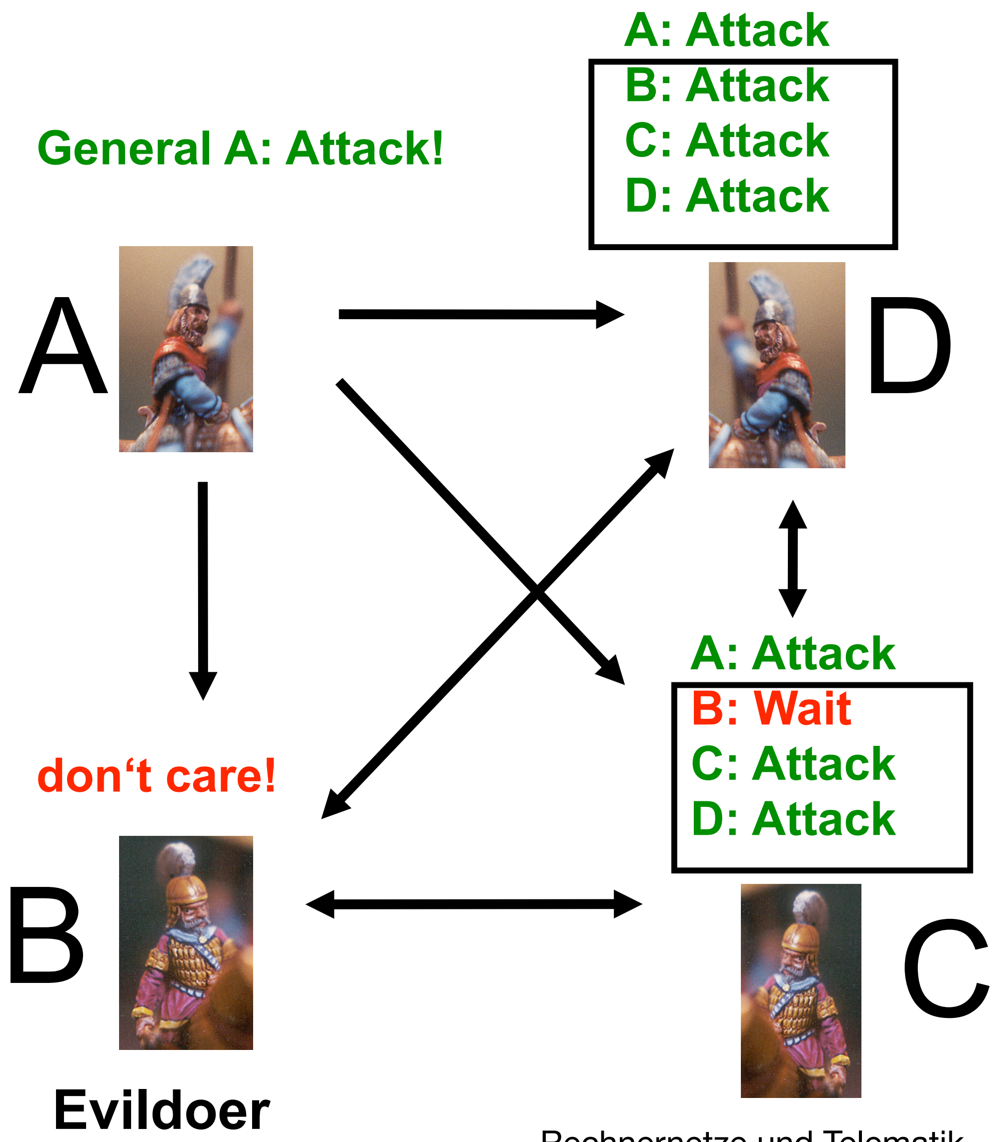
Byzantine Agreement

- **Theorem**
  - The problem of three byzantine generals cannot be solved (without cryptography)
  - It can be solved for 4 generals
- **Consider: 1 general, 3 officers problem**
  - If the general is loyal then all loyal officers will obey the command
  - In any case distribute the received commans to all fellow officers
  - What if the general is the renegade?



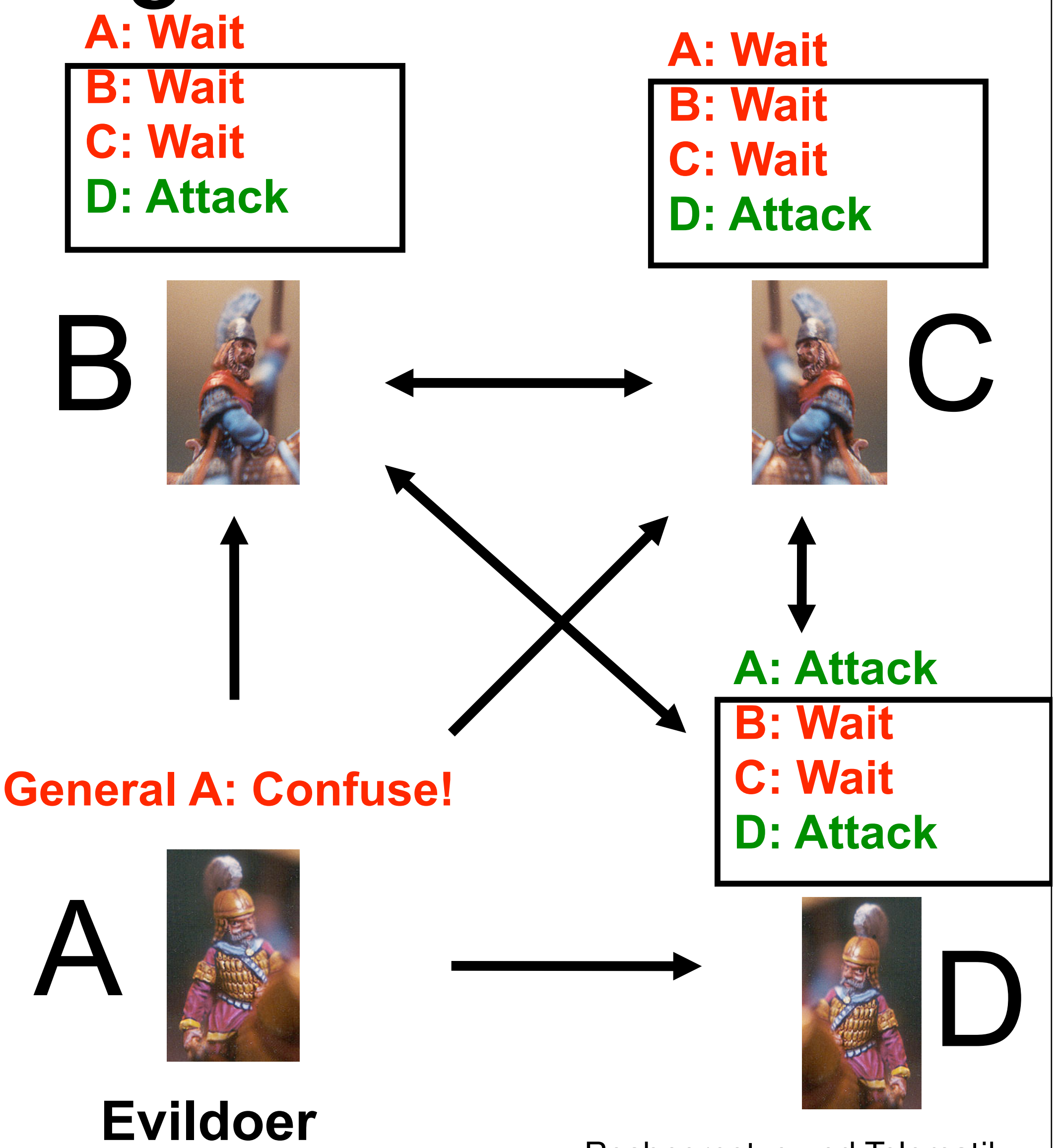
Byzantine Agreement

- **Theorem**
  - The problem of four byzantine generals can be solved (without cryptography)
- **Algorithm**
  - General A sends his command to all other generals
    - A sticks to his command if he is honest
  - All other generals forward the received command to all other generals
  - Every generals computes the majority decision of the received commands and follows this command



Byzantine Agreement

- **Theorem**
  - The problem of four byzantine generals can be solved (without cryptography)
- **Algorithm**
  - General A sends his command to all other generals
    - A sticks to his command if he is honest
  - All other generals forward the received command to all other generals
  - Every generals computes the majority decision of the received commands and follows this command
-



General Solution of Byzantine Agreement

- **Theorem**
  - If m generals are traitors then 2m+1 generals must be honest to get a Byzantine Agreement
- **This bound is sharp if one does not rely on cryptography**
- **Theorem**
  - If a digital signature scheme is working, then an arbitrarily large number of betraying generals can be dealt with
- **Solution**
  - Every general signs his command
  - All commands are shared together with the signature
  - Inconsistent commands can be detected
  - The evildoer can be exposed

Update Model

- **Updates are applied atomically**
  - represented as an array of potential actions and predicates
- **Example actions**
  - replacing a set of bytes in the objects
  - appending new data to the end of the object
  - truncating the object
  - checking latest version of the object

Introspection

- **Cycle of**
  - Observation
  - Optimization
  - Computation
- **Uses**
  - Cluster recognition
  - Replica management
  - Performance of routing structure, availability and durability of archival fragments, recognition of unreliably peers

Summary

- **Prototype of Oceanstore has been recently released**
  - Pond (presented 2003)
- **Plus**
  - Oceanstore provides more file system like structures
  - Efficient routing and caching
  - Consistent updates
  - Space efficient archival system
  - Access control
- **Contra**
  - complex design

ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

Algorithms and Methods for Distributed Storage Networks

11 Peer-to-Peer Storage (final version)

Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Rechnernetze und Telematik
Wintersemester 2007/08

CoNe Freiburg

IIF
INSTITUT FÜR INFORMATIK FREIBURG