Peer-to-Peer Systems and Gossip Protocols

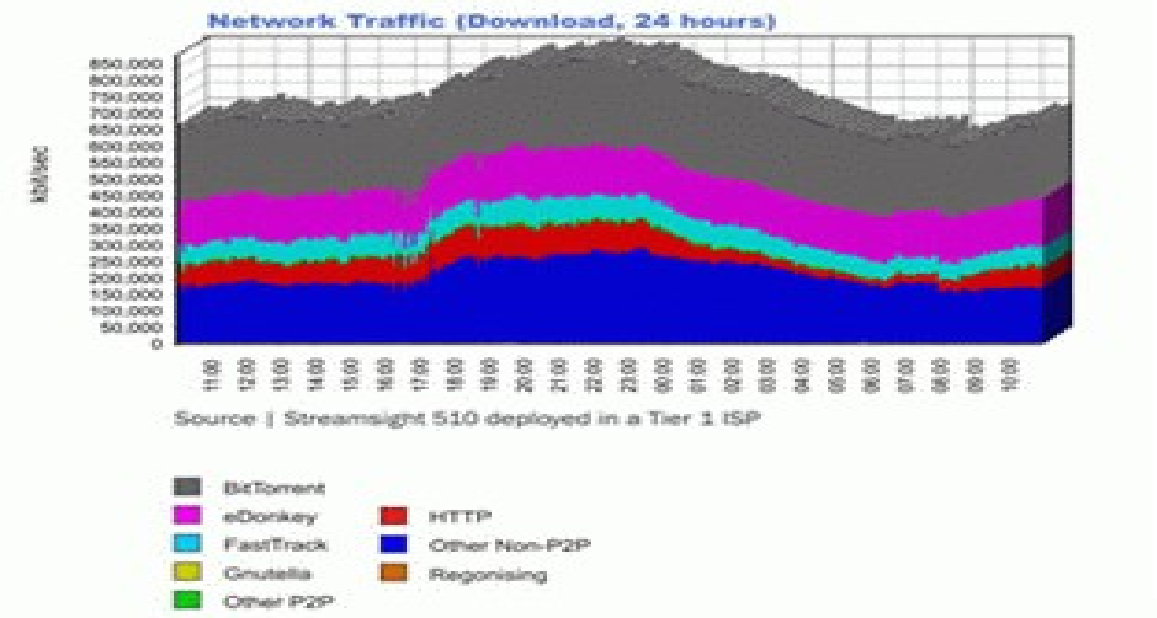
Márk Jelasity

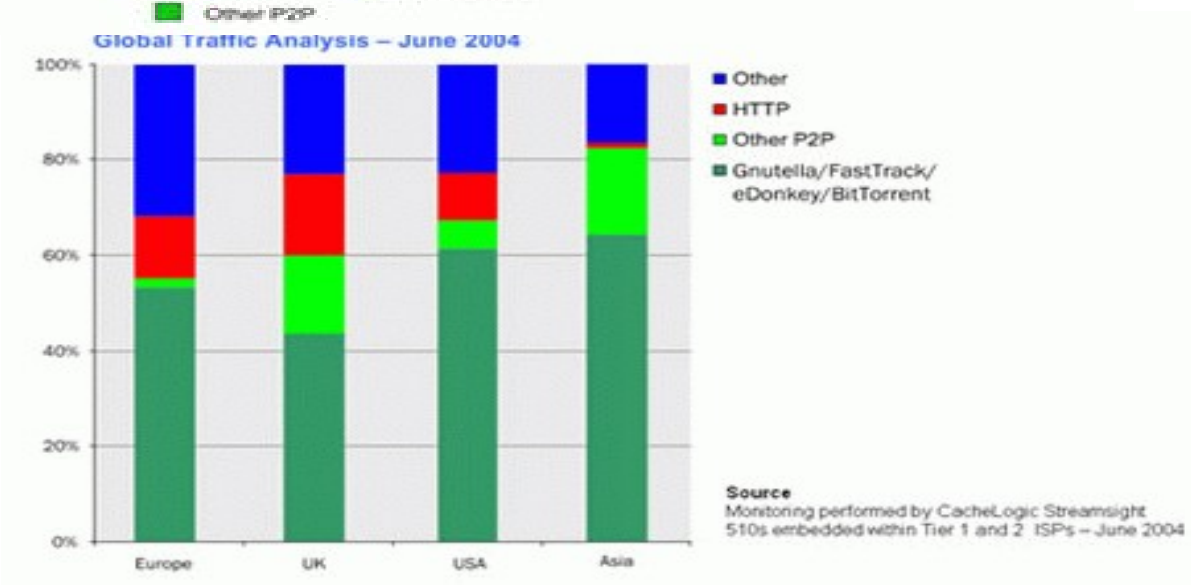
Hungarian Academy of Sciences and
University of Szeged, Hungary

Motivation and Introduction

P2P as bandwidth heavyweight

- CacheLogic's 2004 measurements: majority of Internet traffic is P2P
- Currently streaming video (YouTube, etc) is gaining weight, but P2P still leads





P2P as social/economic/security heavyweight

- User base
  - Tens of millions of users in various P2P networks at any point in time
  - much more log in every now and then
- Social Aspects
  - free flow of information bypasses censorship and content filters
  - anonymous access to services through P2P networks

P2P as social/economic/security heavyweight

- Economic Aspects
  - Gradually puts traditional telecommunication (land line, and soon also mobile) companies out of business
  - Major impact on music and movie industry: hurts sales through traditional channels, bypasses major labels in both distribution and promotion
  - About to change the economics of Internet access
- Major security threat
  - P2P botnets are emerging in the hands of organized crime

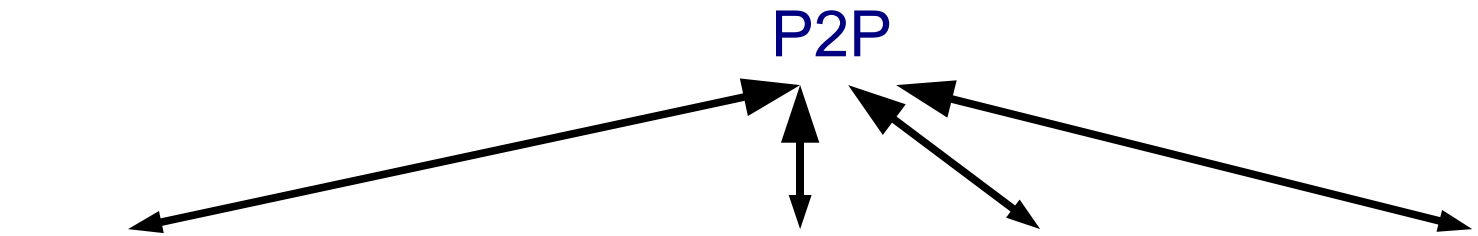
Conclusion

- P2P is one of the most interesting and far-reaching phenomena of today's information technology: would be nice to know
  - how it works
  - how it can be made work better
  - how it can be made work worse (!)
- Lots of food for thought for everyone from computer science, sociology, economics, etc
- Here we will look at abstract algorithms not actual systems

From the pre-P2P era: centralized vs decentralized

- Internet itself was/is P2P
- Other services such as news (NTTP) and DNS involve a decentralized approach at some level, email, telnet, etc are also decentralized
- Early 90-s: emphasis on centralized (client/server) applications (web, etc)
- P2P: from around 2000 the people take the Internet back
- currently centralized services are making a strong comeback (WEB 2.0)
- centralized/decentralized seems to swing in other areas as well (eg mainframe vs PC, GRID vs datacenter, network computers, etc)

Evolution of P2P applications



GRID

Seti@Home
distributed.net
etc

BOINC

file sharing

Napster

Gnutella

FastTrack

various hybrid technologies and networks

content distribution

Akamai

Bittorrent
Skype

worms, botnets

IRC based botnets

P2P botnets

So, what is P2P in academic research?

- We focus on algorithmic and systems research aspects, so we need to define the scope
- A checklist of p2p systems
  - fully decentralized
  - dynamic and unreliable network links
  - asynchronous message passing model
  - very large scale (millions or more)
  - unreliable, selfish and perhaps byzantine nodes

So, what is P2P in academic research?

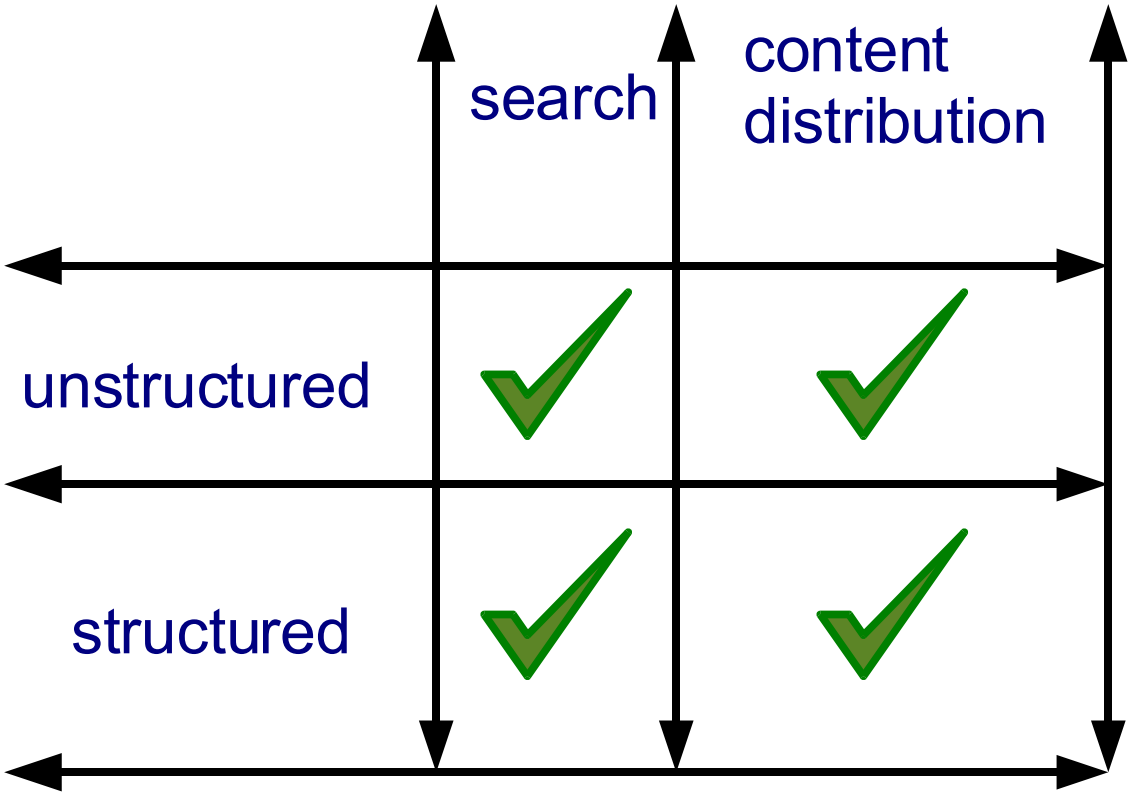
- Not all features need to be present
- Most important is decentralization (no server)
  - Some systems (Napster, seti@home, IRC based botnets) have a server: not realy P2P
- Decentralization might have different reasons
  - legal or political pressure: avoid to have a single point of failure (Napster vs FastTrack, anonymous networks, and P2P botnets)
  - efficiency: (BitTorrent, esp serverless versions, media streaming, application layer IP routing)
  - cost: not having to maintain a server is cheap

Basic concepts

- Due to decentralization, we always work with an overlay network (the structure)
  - defines who can pass messages to whom
  - structure is given, or needs to be built and maintained appropriately (self-organizing)
- We implement functions on a given network
- Algorithms and networks go hand-in-hand, like in the case of data structures
- In the following we look at various functions and structures and their relationships

Outline (P2P)

- function
  - search
  - content distribution
- structure
  - “unstructured”
  - “structured”
- selected issues
  - incentives, security, research methodology



Outline (gossip)

- function
  - main function: application level multicast
  - data aggregation
  - overlay topology maintenance
- structure
  - fully connected and random (unstructured)
  - structured

References

- Nelson Minar and Marc Hedlund. A network of peers: Peer-to-peer models through the history of the internet. In Andy Oram, editor, Peer-to-Peer: Harnessing the Power of Disruptive Technologies, chapter 1. O'Reilly, 2001.
- CacheLogic Inc. P2P in 2005. http://www.cachelogic.com/home/pages/studies/2005_01.php , a survey on the volume and composition of P2P traffic on the Internet.
- Thomas Mennecke. P2P Remains Dominant Protocol. http://www.slyck.com/story1502.html Slyck. 2007

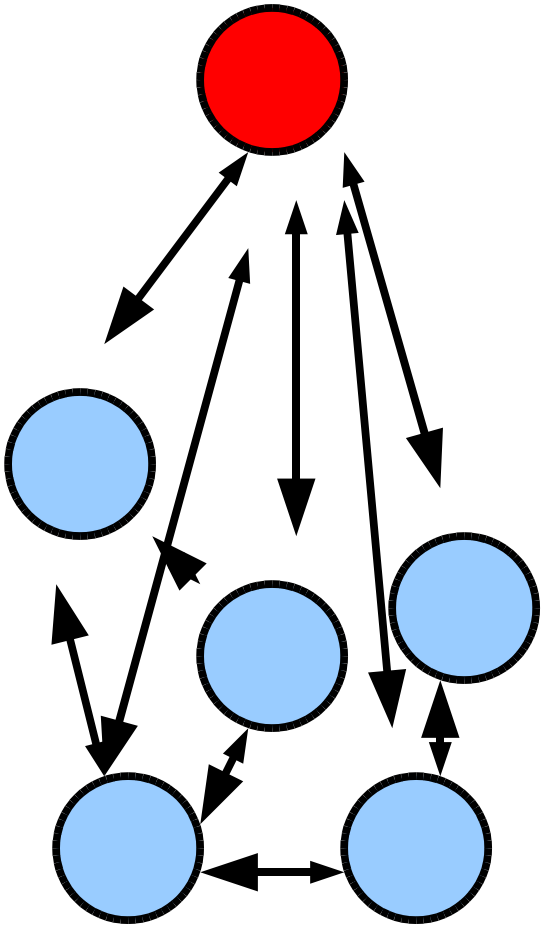
Search in Unstructured Networks

Outline

- Motivation for decentralized networks
- The Gnutella network: how it worked and looked like
  - Some surprises about the emergent network structure
- Search algorithms in unstructured networks
  - Random walk search in power law networks
  - Random walk search in random networks
  - Replication strategies
  - GIA: a prominent algorithm

Central index

- Index is stored on central servers: search is centralized
- Download is P2P
- For example, Napster
  - Works well, but not scalable
    - **Major investments needed if networks grows**
    - **Eg Google: 100,000+ servers already**
  - Not robust to attacks (legal and malicious)
- Incentive to go decentralized



First attempt to go decentralized: Gnutella

- Nullsoft (Justin Frankel)
- First client is spread via gossip...
  - AOL shuts down Nullsoft servers the day after the release
- Initially no explicit attempt to control overlay topology
- Naive approach to search: flooding
- All communication (queries) are via flooding too

How does Gnutella work?

- Gnutella protocol: flooding of queries
  - Ping, pong
    - **peer discovery at join and also continuously**
  - Query, query hit:
    - **Search hits are propagated back on the path of the search query**
- Join procedure
  - Find any member
  - Send ping message and collect pong messages

What was the Gnutella overlay supposed to look like?

- We don't know for sure but probably designers had random networks in mind (if anything)
- What properties does a random network have?
  - Is it good for search, is it robust, connected, etc?
- Mathematics has some answers for a number of special models of random networks. We briefly overview one: The Erdős-Rényi model.

The model

- Simple undirected graph $G_{N,p}$
- Parameters
  - N: number of nodes
  - p: probability of connecting any pairs of nodes
- Algorithm
  - Start with empty graph of N nodes
  - Draw all N(N-1)/2 possible edges with probability p
- Stats of degree of a fixed node i
  - $<k_i>=p(N-1)$, k_i has binomial distr, approx Poisson

Probabilistic properties

- Usual question: P(Q) over a probability space of graphs
  - Q can be eg “connected”, or “contains a triangle”, etc
- Usually P(Q) depends on N and p
- We are interested in “almost always” Q:

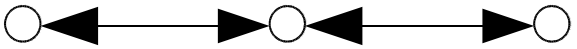
$$P_{N,p}(Q) \to 1 \quad (N \to \infty)$$

Probabilistic properties

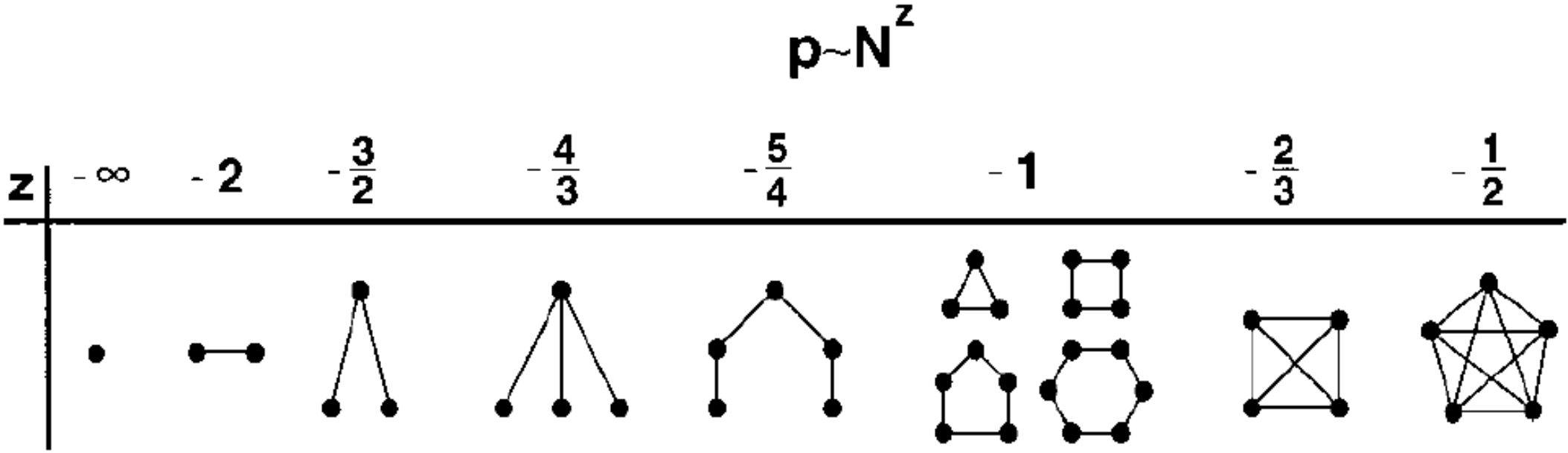
- Often there is a critical probability p_c such that

$$\lim_{N\to\infty} P_{N,p}(Q) = \begin{cases} 0 & \dfrac{p(N)}{p_c(N)} \to 0 \\ 1 & \dfrac{p(N)}{p_c(N)} \to \infty \end{cases}$$

- We are interested in p_c for different Q-s
- Example: $G_{N,p}$ has a subgraph



Critical pr. for small subgraphs



- Note the case p~1/N where cycles of all order appear
- Note that this is understood as N tends to ∞

Connectivity

- Let's look at connectivity as a function of p
  - AKA "graph evolution": when we keep adding edges
- Note that if p grows slower than 1/N, the graph is a disconnected collection of small (constant size) components
- If p~1/N, avg node degree <k> is constant, cycles of all order have finite probability
  - What's going on if <k> is constant?

The case when p~1/N

- 0< <k> <1
  - One cycle, otherwise trees, the larges being O(ln N) size
  - The number of clusters is N-n (ie each new edge connects two clusters)
- <k>=1
  - Critical value: largest cluster is suddenly $O(N^{2/3})$, with complex structure
- <k> >1
  - The largest cluster is of size (1-f(<k>))N nodes where f decreases exponentially
- [If <k> >= ln N, completely connected (but here the avg degree grows with N)]

Degree distribution

- k_i the degree of fixed node
  - k_i is binomial (Bin(N-1,p))
- Degree distribution: the degree of a random node from a random graph
  - x_k: number of nodes with degree k
  - $<x_k>=NP(k_i=k)$
  - Distribution of x_k has very low variance
  - So it is a reasonable assumption to say that a random graph $G_{N,p}$ has very close to binomial degree distribution

Diameter

- The longest shortest path
- $L = \ln N / \ln \langle k \rangle = \log_{\langle k \rangle} N$
- Intuitively, the reason is that these graphs are locally like trees
- The average path length (l) grows also as $\log_{\langle k \rangle} N$

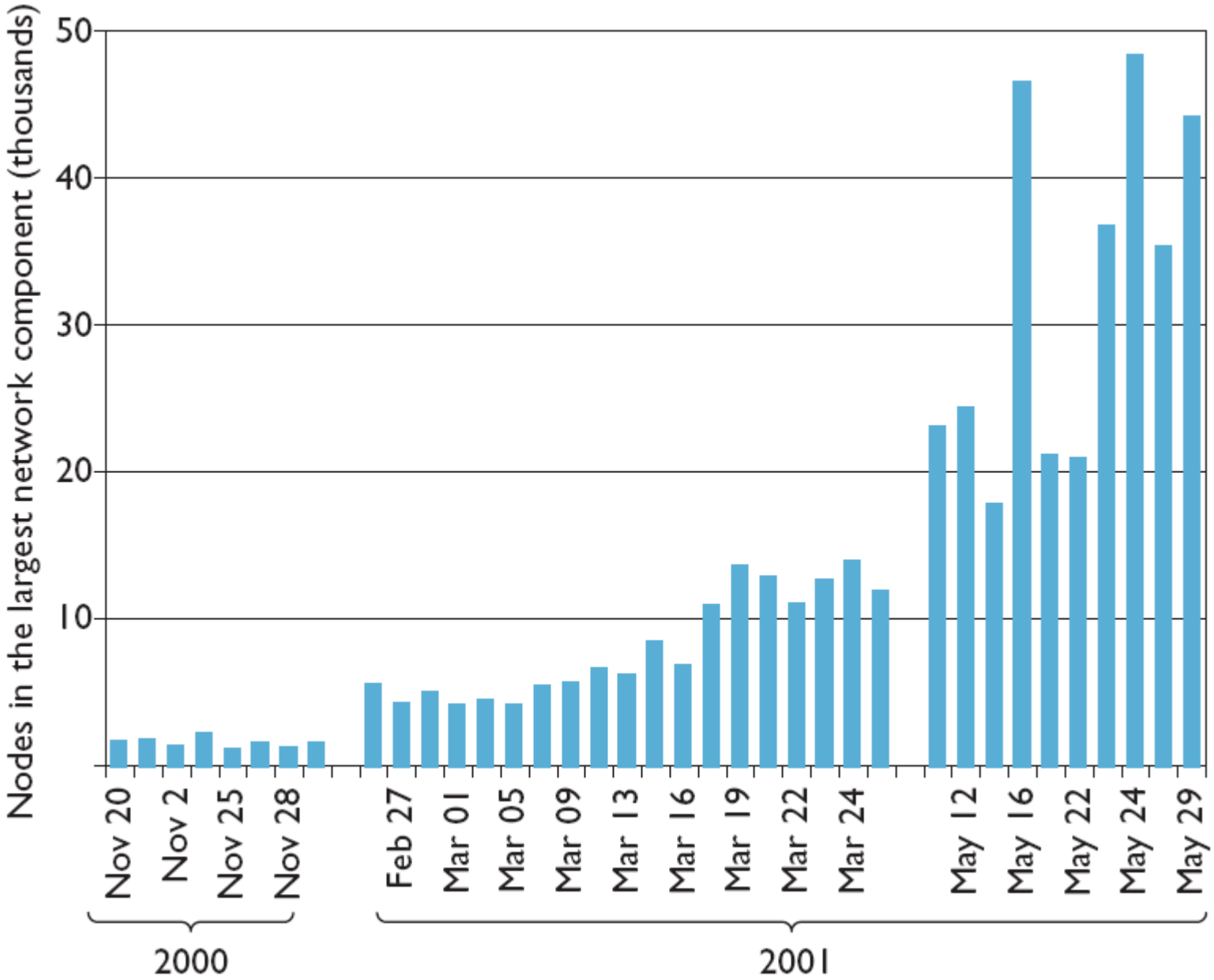
Some other similar models

- $G_{r\text{-}reg}$: probability space is the set of r-regular graphs with equal probability
  - $G_{3\text{-}reg}$ is Hamiltonian
  - Note that $G_{3/(N-1),N}$ is not even connected
- $G_{r\text{-}out}$: we generate a random graph by adding 3 edges from all nodes
  - $G_{4\text{-}out}$ is Hamiltonian
  - It is believed that $G_{3\text{-}out}$ is also Hamiltonian
- So we need to be careful! When there is guarantee that all nodes have some edges, things are radically different

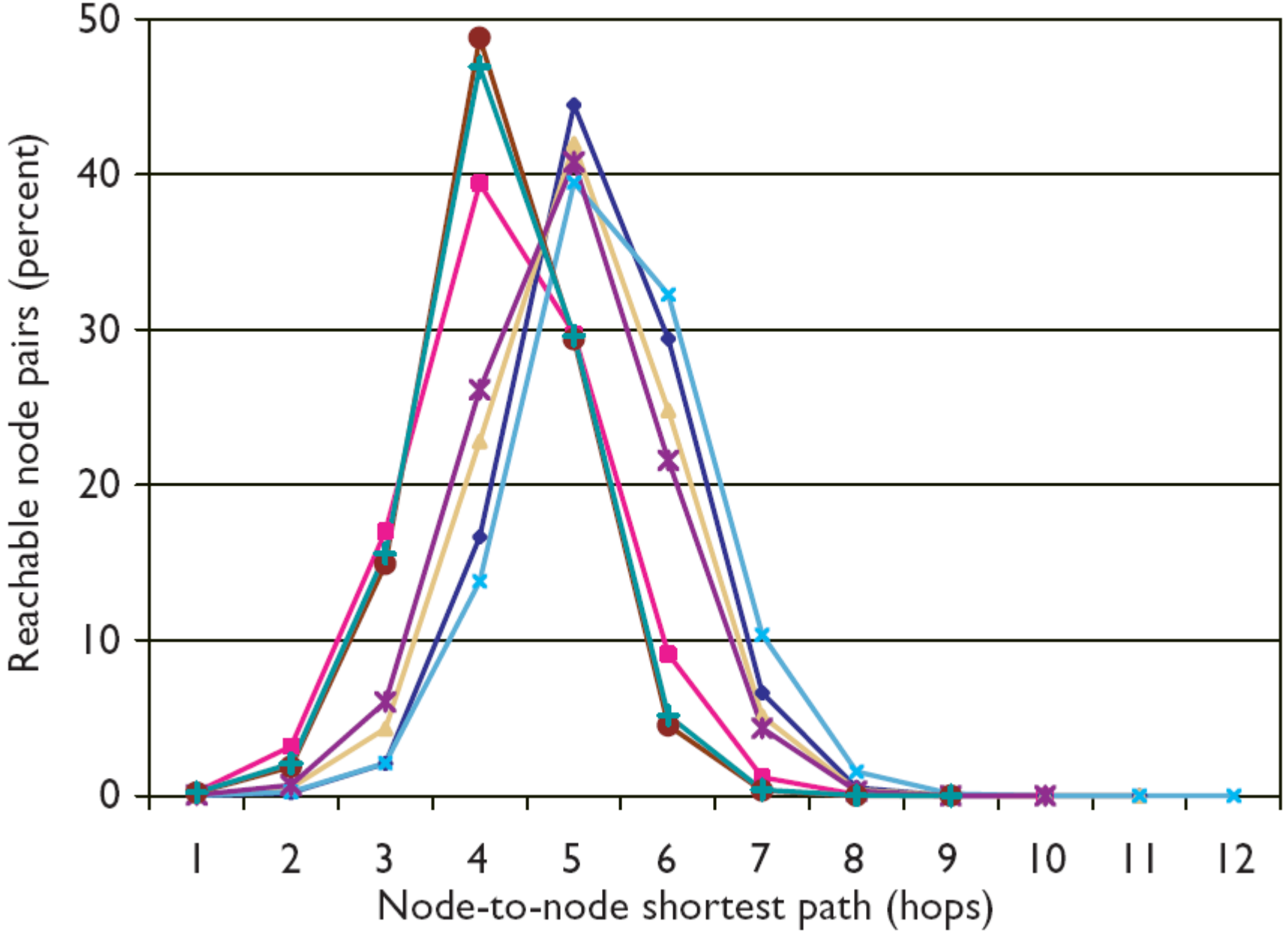
What did the Gnutella overlay actually look like?

- Measurements by Ripeanu et al.
- Distributed Gnutella crawler collecting snapshots of size in the order of 50,000 for a year
- They discover complex network structure and highly dynamical composition: churn
  - 40% spend less than 4 hours in the network
  - 25% spend more than 24 hours

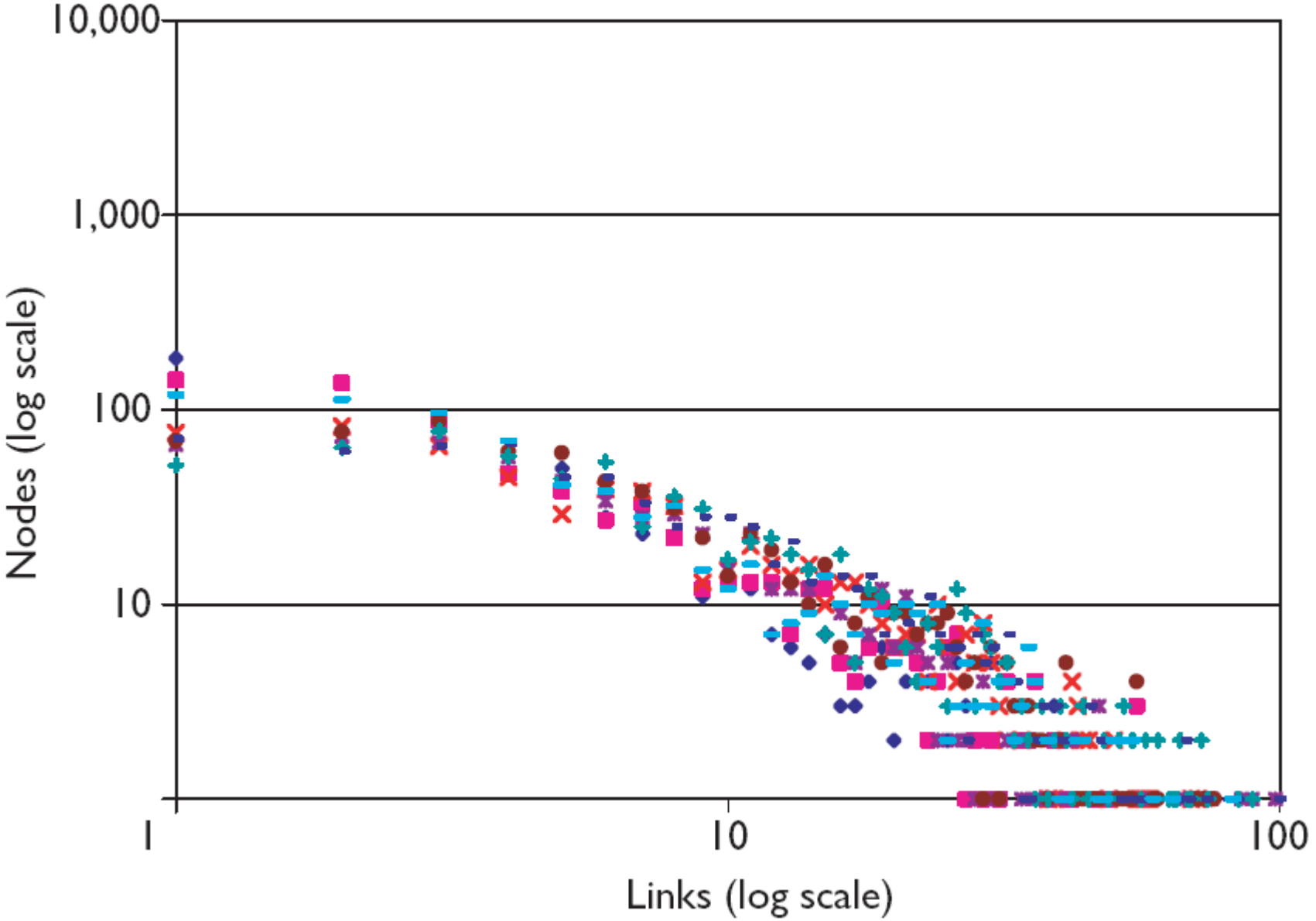
Growth of the network



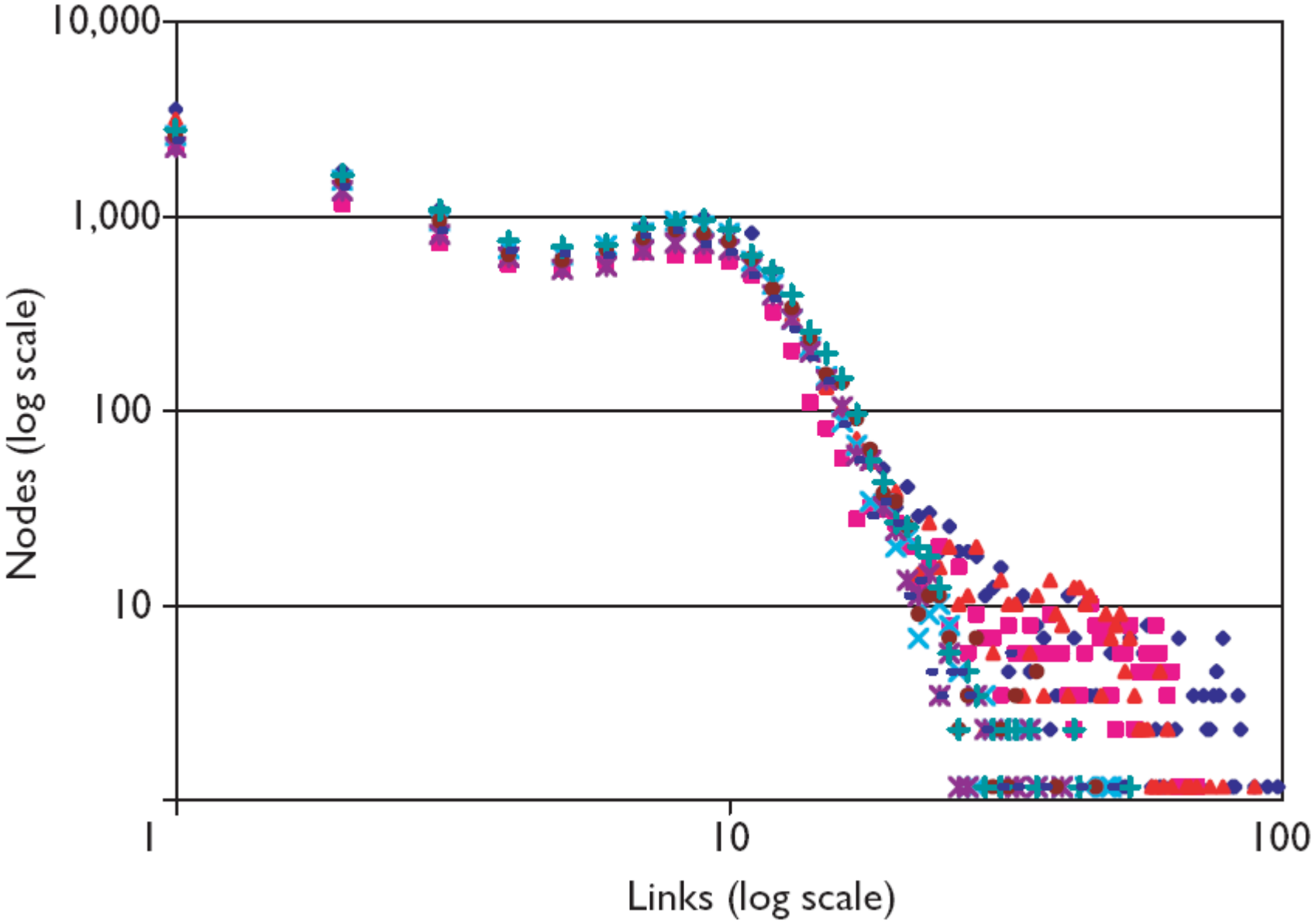
Path lengths



Degree distribution 2000 November

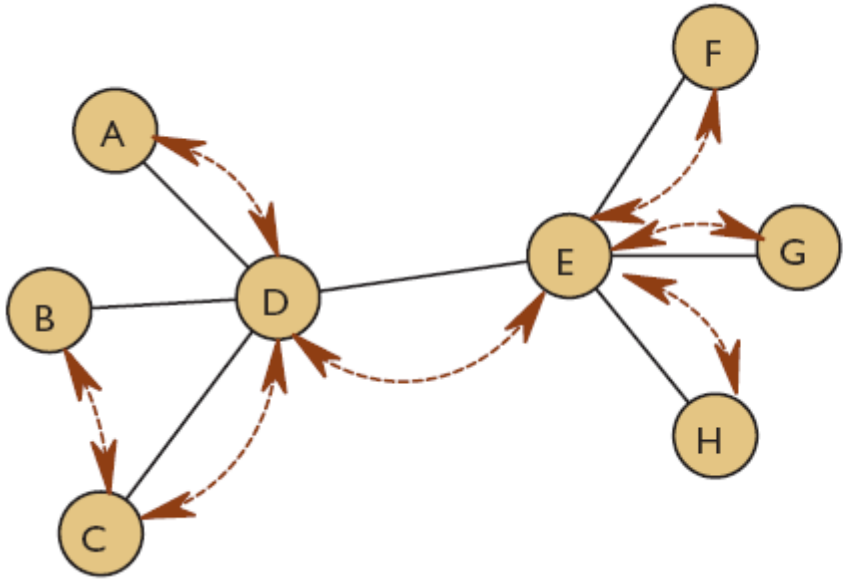


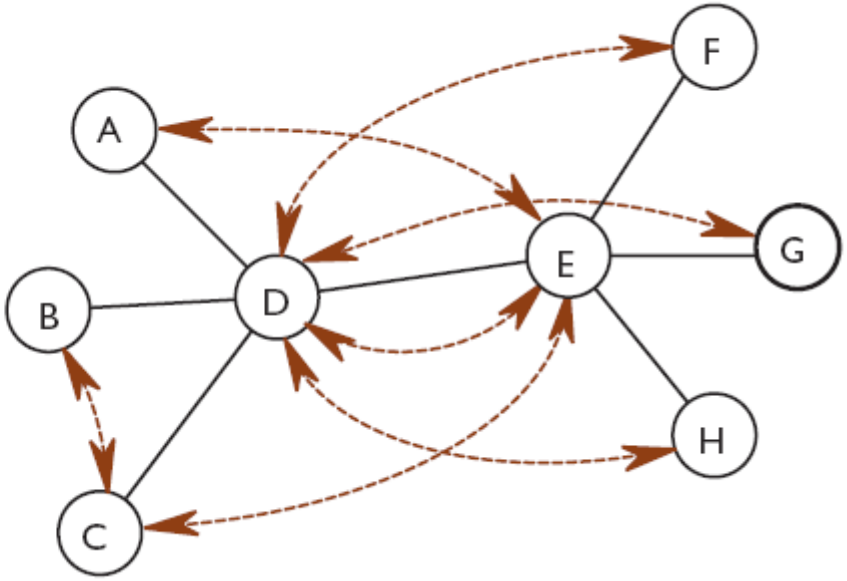
Degree distribution 2001 May



Underlying topology

- We have seen the that Internet is also power law
- Is there correlation between the overlay and the Internet?
- Ripeanu et al find that there is none





What's going on?

- Path length is as in the ER model, but degree distribution is heavy tail
  - $P(k) \sim k^{-\gamma}$ (maybe with some cutoff, eg $P(k) \sim k^{-\gamma} e^{-\kappa/\gamma}$)
- Without cutoff
  - No expectation value (ie $\langle k \rangle = \infty$) if $\gamma <= 2$
  - No variance (ie $Var(k) = \infty$) if $\gamma <= 3$, etc
- Called scale-free because of fractals
- How do such networks form and why?

Observed scale free networks

Network	Size	$\langle k \rangle$	κ	γ_{out}	γ_{in}	ℓ_{real}	ℓ_{rand}	ℓ_{pow}	Reference	Nr.
WWW	325 729	4.51	900	2.45	2.1	11.2	8.32	4.77	Albert, Jeong, and Barabási 1999	1
WWW	4×10^7	7		2.38	2.1				Kumar *et al.*, 1999	2
WWW	2×10^8	7.5	4000	2.72	2.1	16	8.85	7.61	Broder *et al.*, 2000	3
WWW, site	260 000				1.94				Huberman and Adamic, 2000	4
Internet, domain*	3015–4389	3.42–3.76	30–40	2.1–2.2	2.1–2.2	4	6.3	5.2	Faloutsos, 1999	5
Internet, router*	3888	2.57	30	2.48	2.48	12.15	8.75	7.67	Faloutsos, 1999	6
Internet, router*	150 000	2.66	60	2.4	2.4	11	12.8	7.47	Govindan, 2000	7
Movie actors*	212 250	28.78	900	2.3	2.3	4.54	3.65	4.01	Barabási and Albert, 1999	8
Co-authors, SPIRES*	56 627	173	1100	1.2	1.2	4	2.12	1.95	Newman, 2001b	9
Co-authors, neuro.*	209 293	11.54	400	2.1	2.1	6	5.01	3.86	Barabási *et al.*, 2001	10
Co-authors, math.*	70 975	3.9	120	2.5	2.5	9.5	8.2	6.53	Barabási *et al.*, 2001	11
Sexual contacts*	2810			3.4	3.4				Liljeros *et al.*, 2001	12
Metabolic, *E. coli*	778	7.4	110	2.2	2.2	3.2	3.32	2.89	Jeong *et al.*, 2000	13
Protein, *S. cerev.**	1870	2.39		2.4	2.4				Jeong, Mason, *et al.*, 2001	14
Ythan estuary*	134	8.7	35	1.05	1.05	2.43	2.26	1.71	Montoya and Solé, 2000	14
Silwood Park*	154	4.75	27	1.13	1.13	3.4	3.23	2	Montoya and Solé, 2000	16
Citation	783 339	8.57			3				Redner, 1998	17
Phone call	53×10^6	3.16		2.1	2.1				Aiello *et al.*, 2000	18
Words, co-occurrence*	460 902	70.13		2.7	2.7				Ferrer i Cancho and Solé, 2001	19
Words, synonyms*	22 311	13.48		2.8	2.8				Yook *et al.*, 2001b	20

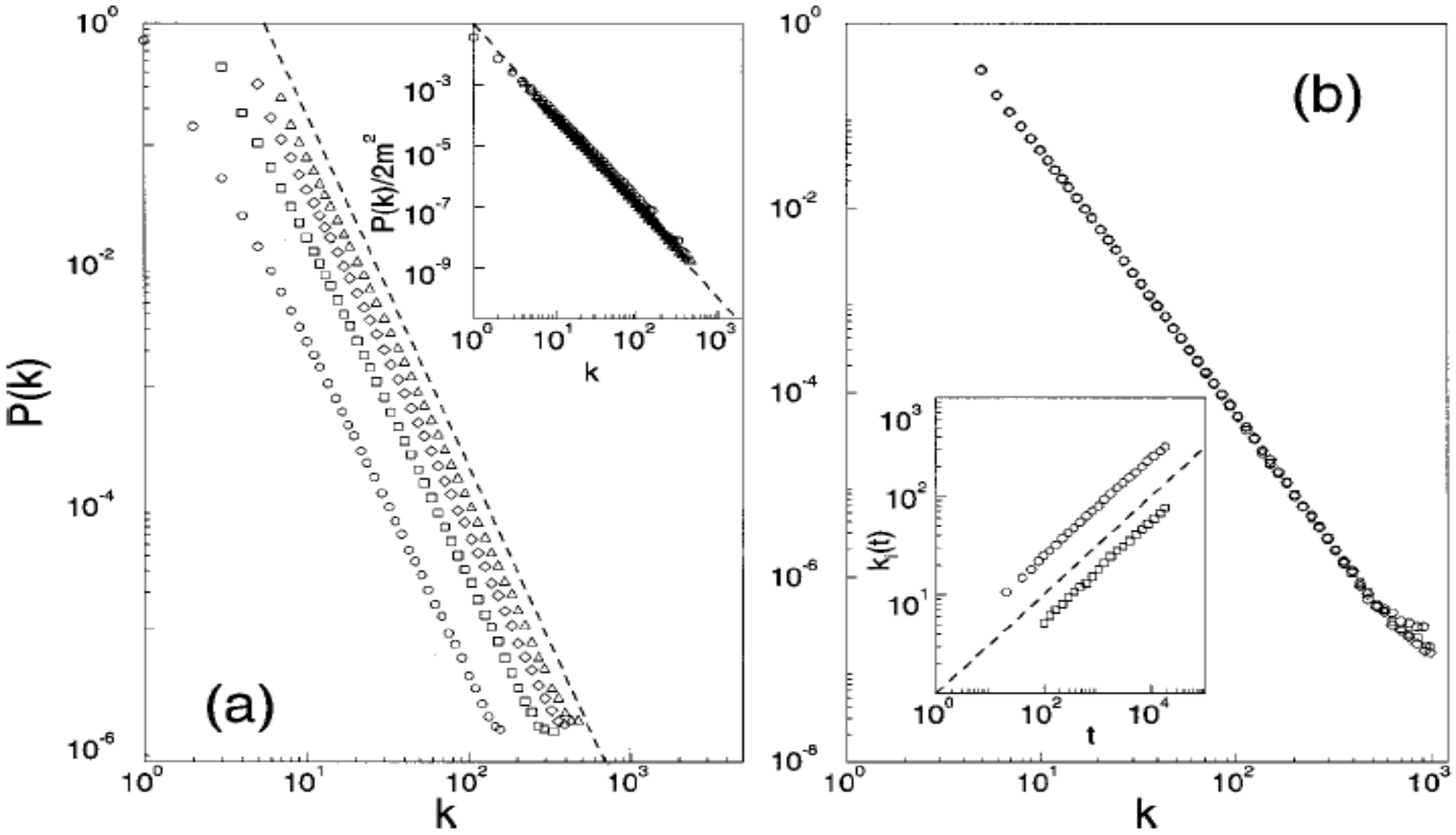
Barabási-Albert model

- The rich get richer principle: growth models
- Preferential attachment rule
  - Start with a small number (m_0) of nodes
  - Repeat adding a new node with m<=m0 links, where each linked is linked to node i according to

$$\Pi(k_i) = \frac{k_i}{\sum_j k_j}$$

- T time step, t+m_0 nodes, mt edges
- Converges to exponent γ=3
- Average path length
  - L~ln N/ ln ln N (somewhat smaller than ER model)

Empirical results with BA model



Search: flooding

- The default search model is flooding
  - Query is sent with a TTL, typically TTL=7
  - Query hits are propagated back on the path of the query
- Serious problems
  - Extremely wasteful with bandwidth
    - **A large (linear) part of the network is covered irrespective of hits found**
    - **Enormous number of redundant messages**
    - **All users do this in parallel: local load grows linearly with size**

Questions

- Does the scale-free topology has an effect on search protocols
  - Can we exploit it, or is it a disadvantage
  - What is the optimal search protocol for it
- In general, what search protocols can we come up with in an unstructured network
- What other techniques can we apply
  - Controlling topology to allow for better search
  - Controlling placement of objects (replication)

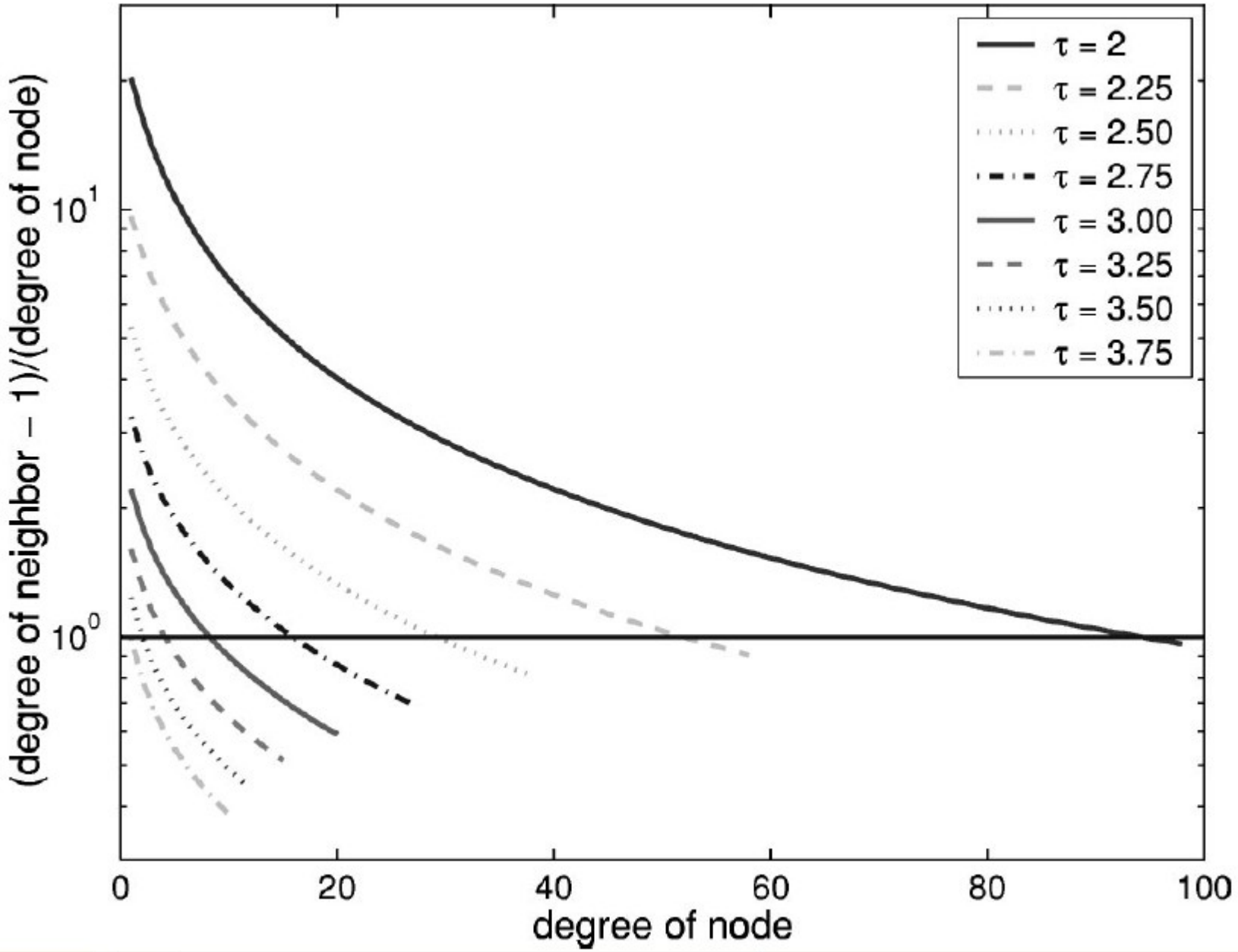
Search in scale-free networks

- Basic observations
  - In certain models if degree distribution is p_k then the distribution of the degree of a neighbor is proportional to kp_k (very important observation)
  - Nodes can easily store index of objects stored by their neighbors
- So in scale-free: high degree nodes are easy to find by (biased) random walk
- And high degree nodes can store the index about a large portion of the network
- Hint: a bit like the star topology

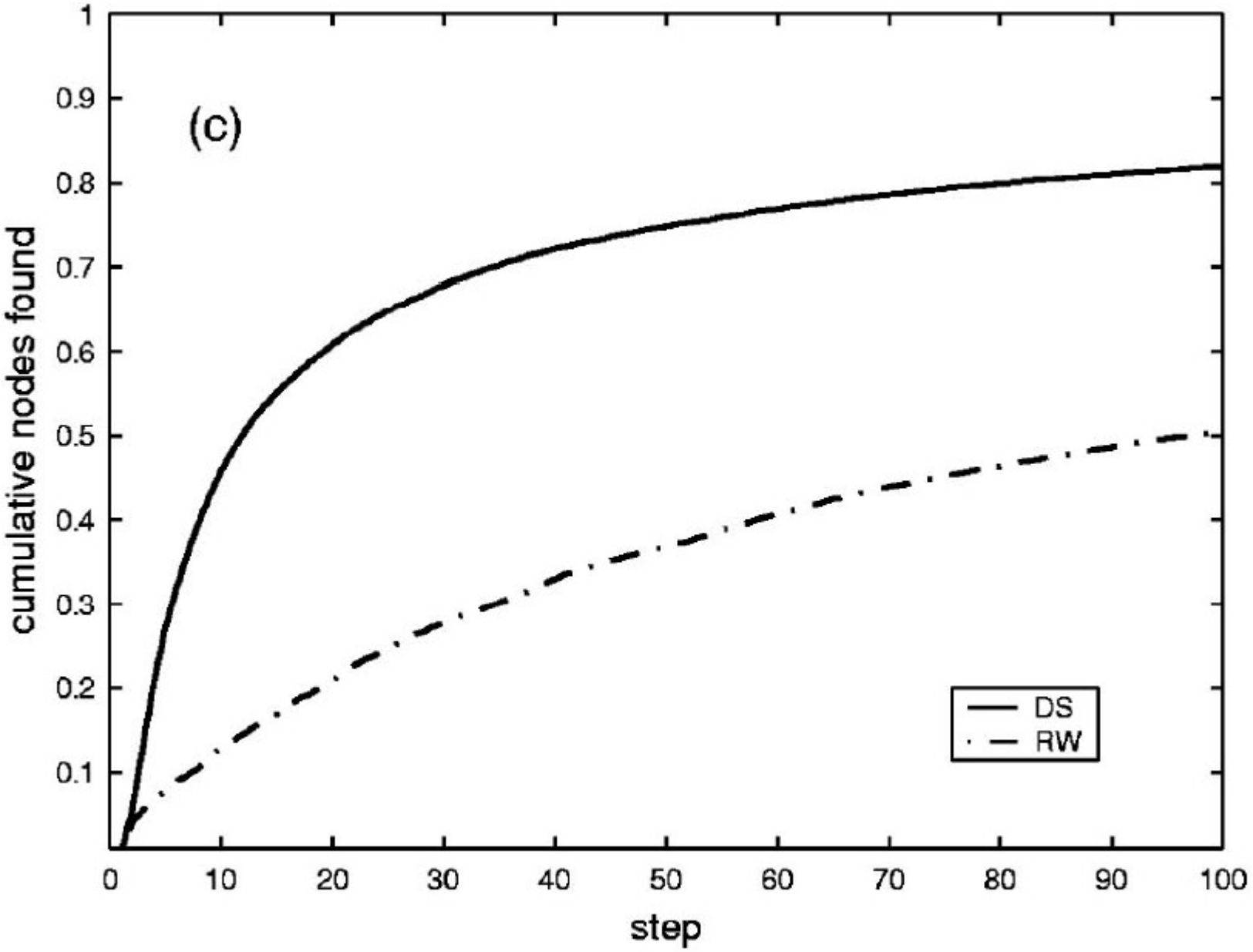
Search in scale-free networks

- Proposed algorithm variants
  - Random walk (RW)
    - **avoiding the visit of last visited node**
  - Degree-biased random walk (DS)
    - **Select highest degree node, that has not been visited**
    - **This first climbs to highest degree node, then climbs down on the degree sequence**
    - **Provably optimal coverage**
- Examined networks
  - Scale-free network with γ=2.1, abrupt cutoff
  - ER graphs
  - Different sizes, but N=10,000 if not specified

Climbing up the degree sequence

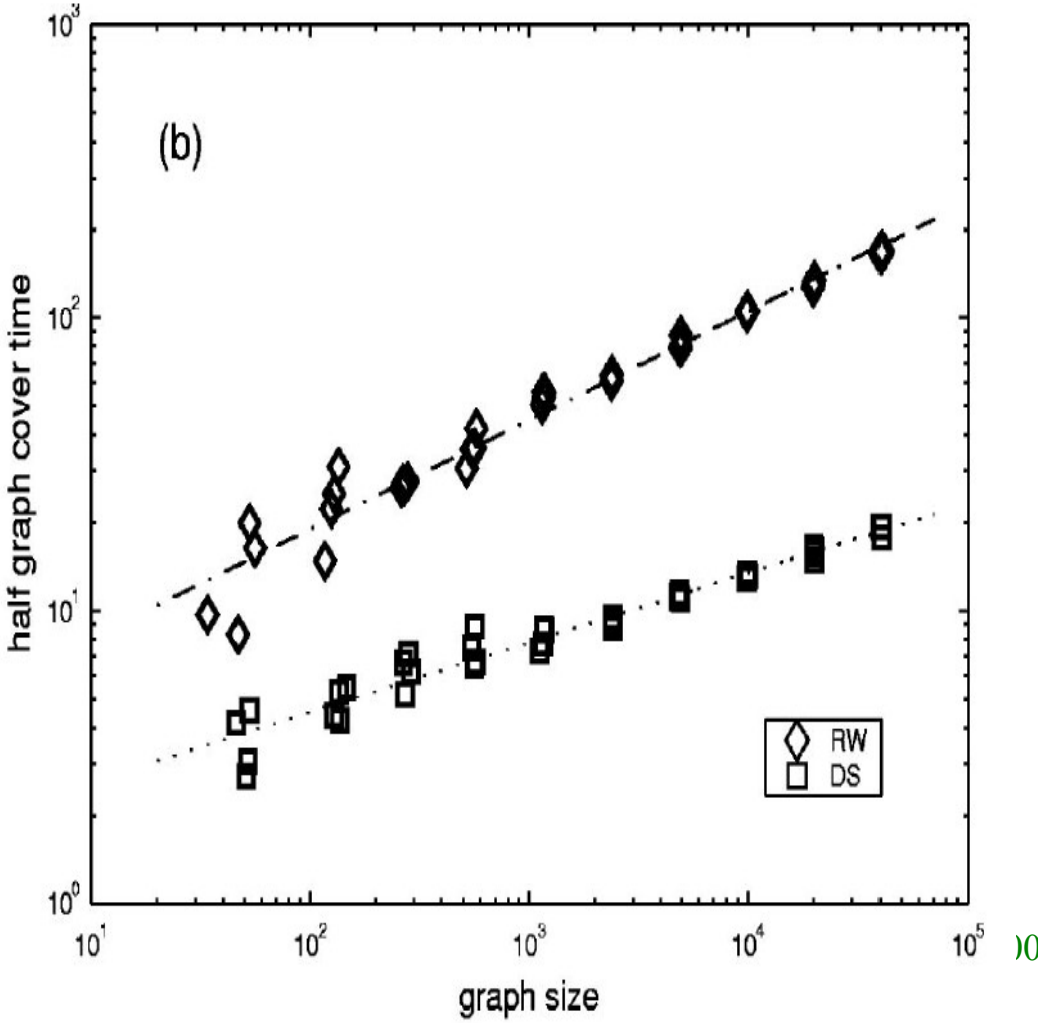


Speed of coverage

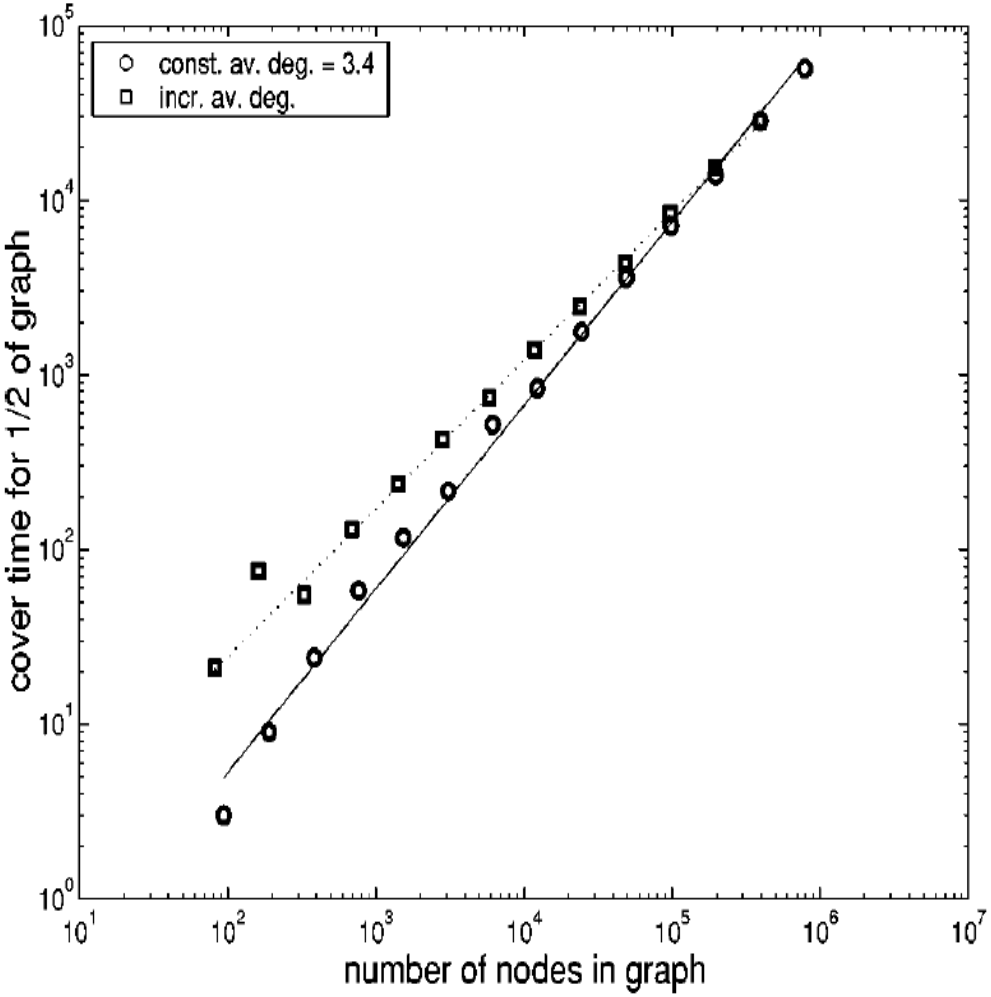


Half graph cover time

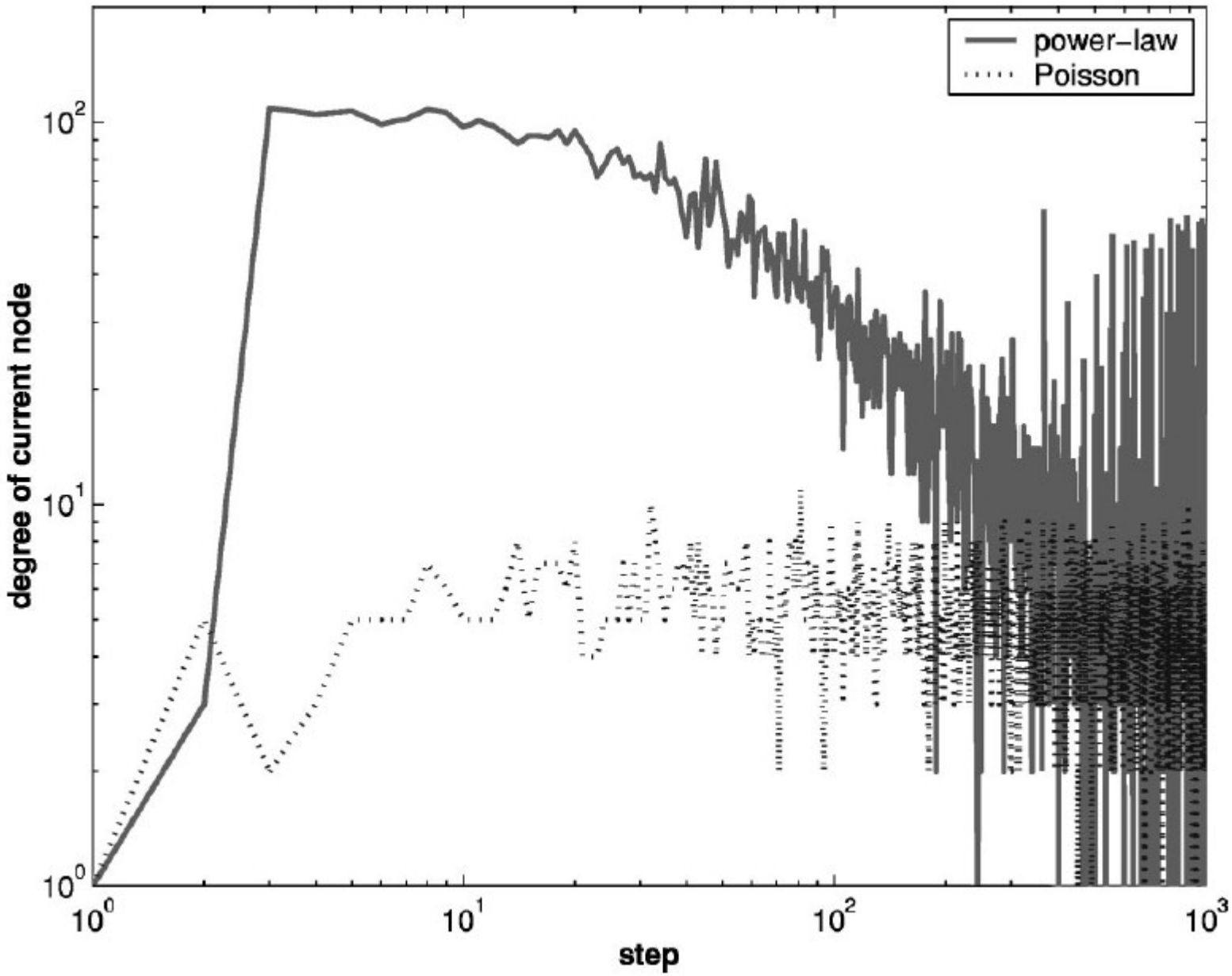
Scale free graph



ER graph



Visited node degrees



Conclusions

- Advantages
  - Takes advantage of scale-free distribution and speeds up search relative to ER graphs
  - Search time complexity is sublinear
- Disadvantages
  - Difficulty with rare objects (but this is a common problem of unstructured search)
  - Places very high load on high degree nodes
- Keeping this in mind, let's look at other topologies and see if they are better

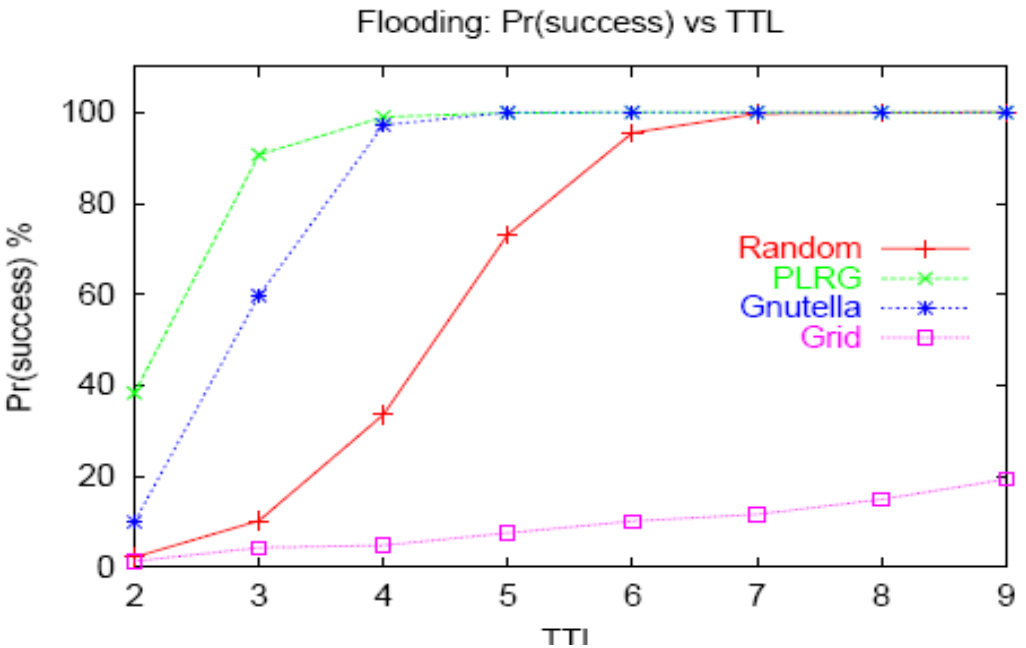
More search algorithms

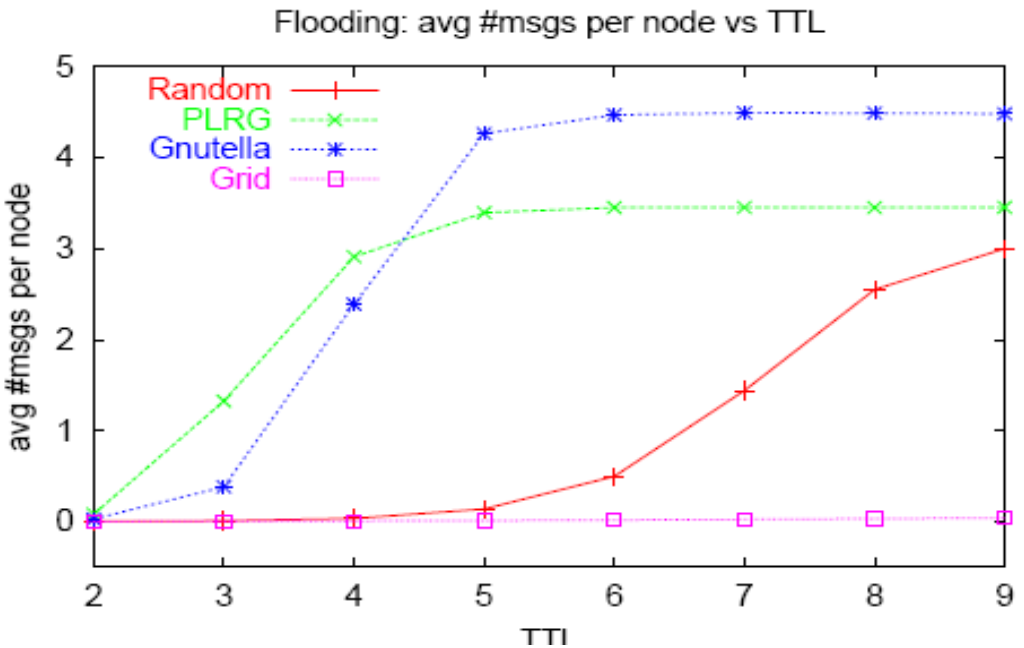
- Expanding ring
  - Flooding with increasing TTL until result is found
  - The point is to avoid a fixed TTL
- K-walker
  - K independent random walks, to avoid message duplication in flooding and expanded ring
    - **With checking: in every 4 steps all walks check back if they need to go on or not**
    - **With state keeping: to implement self-avoiding walks**

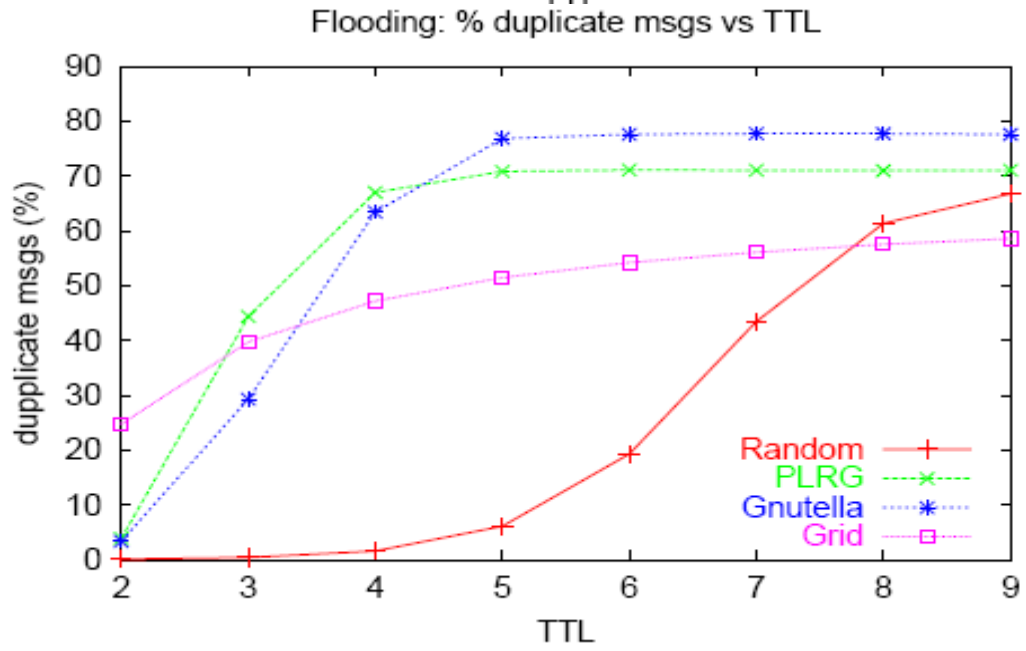
Evaluation of search algorithms

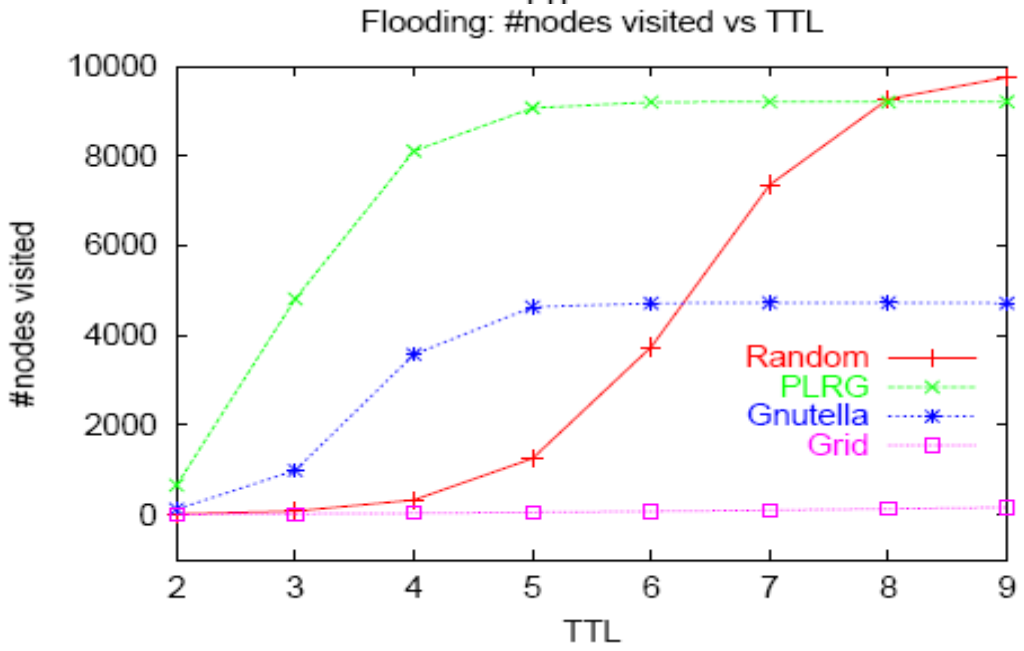
- So far simplified model
  - ignored query and replication distribution, focused on coverage
- Three main components
  - Overlay network, Query modeling, Replication strategies
- Overlay networks
  - ER graph, avg. degree 4, N=10000
  - Power law (scale-free) graph, N=10000
  - Gnutella snapshot 2000 Oct, N=4000
  - 2-dim 100x100 grid

Problems with flooding









Evaluation of search algorithms

- Query distributions
  - q_i: the proportion of queries for object i
  - Uniform: all objects receive the same amount of queries
  - Power law: a few objects are very popular, many objects are not so much (heavy tail)
- Replication plays a role too
  - Spread copies of objects to peers: more popular objects can be found easier
  - File-sharing networks show an emergent replication behavior

Evaluation of search algorithms

- Object replication
  - Replication of object i typically proportional to q_i
  - Uniform: all objects receive the same amount of copies
  - Proportional: proportional to q_i
  - Square-root: proportional to square-root q_i
    - **Can be proven to be optimal in certain cases (see later)**
- Meaningful combinations of query/replication
  - uniform/uniform, power-law/proportional, power-law/square-root

Some results

distribution model		50 % (queries for hot objects)				100 % (all queries)			
query/replication	metrics	flood	ring	check	state	flood	ring	check	state
Uniform / Uniform	#hops	3.40	5.77	10.30	7.00	3.40	5.77	10.30	7.00
	#msgs per node	2.509	0.062	0.031	0.024	2.509	0.061	0.031	0.024
	#nodes visited	9220	536	149	163	9220	536	149	163
	peak #msgs	6.37	0.26	0.22	0.19	6.37	0.26	0.22	0.19
Zipf-like / Proportional	#hops	1.60	2.08	1.72	1.64	2.51	4.03	9.12	6.66
	#msgs per node	1.265	0.004	0.010	0.010	1.863	0.053	0.027	0.022
	#nodes visited	6515	36	33	47	7847	396	132	150
	peak #msgs	4.01	0.02	0.11	0.10	5.23	0.20	0.17	0.14
Zipf-like / Square root	#hops	2.23	3.19	2.82	2.51	2.70	4.24	5.74	4.43
	#msgs per node	2.154	0.010	0.014	0.013	2.308	0.031	0.021	0.018
	#nodes visited	8780	92	50	69	8983	269	89	109
	peak #msgs	5.88	0.04	0.16	0.16	6.14	0.12	0.17	0.16

ER graph

distribution model		50 % (queries for hot objects)				100 % (all queries)			
query/replication	metrics	flood	ring	check	state	flood	ring	check	state
Uniform / Uniform	#hops	2.37	3.50	8.95	8.47	2.37	3.50	8.95	8.47
	#msgs per node	3.331	1.325	0.030	0.029	3.331	1.325	0.030	0.029
	#nodes visited	8935	4874	147	158	8935	4874	147	158
	peak #msgs	510.4	132.7	12.3	11.7	510.4	132.7	12.3	11.7
Zipf-like / Proportional	#hops	1.74	2.36	1.81	1.82	2.07	2.93	9.85	8.98
	#msgs per node	2.397	0.593	0.011	0.011	2.850	0.961	0.031	0.029
	#nodes visited	6969	2432	43	49	7923	3631	136	145
	peak #msgs	412.7	58.3	4.9	5.1	464.3	98.9	12.7	11.7
Zipf-like / Square root	#hops	2.07	2.94	2.65	2.49	2.21	3.17	5.37	4.79
	#msgs per node	3.079	0.967	0.014	0.014	3.199	1.115	0.021	0.020
	#nodes visited	8434	3750	62	69	8674	4200	97	103
	peak #msgs	496.0	93.7	6.3	6.3	499.6	111.7	8.9	8.4

power-law graph

Notes for the experiments

- Parameters
  - 100 objects, avg replication ratio 1%
  - ER graph: TTL for flooding is 8, “check” and “state” are 32-walkers, γ=1.2 for query distribution
  - Power-law graph: same, but TTL=5
- Algorithms
  - Check: 32-walker with checking for termination
  - State: same as 32-walker, but also self-avoiding

Conclusions

- Fixed TTL must be avoided, be adaptive instead
- Avoid exponential spreading of queries
  - Note that this assumes that each object is replicated enough, otherwise search takes too long
- Message duplication must be avoided
  - ER random graph is best for this
  - So now: is scale-free good or bad?
- Square-root replication is optimal
  - How about dynamic methods for achieving that?

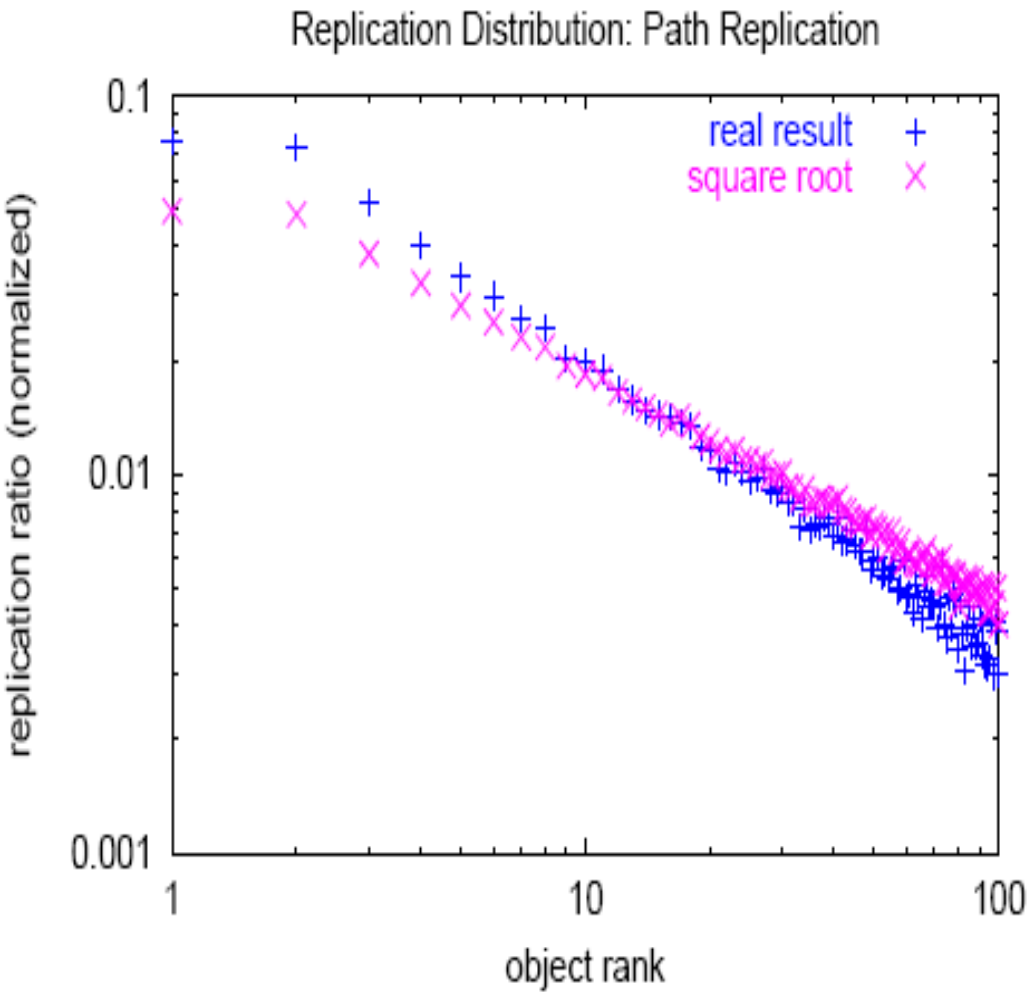
Replication strategies

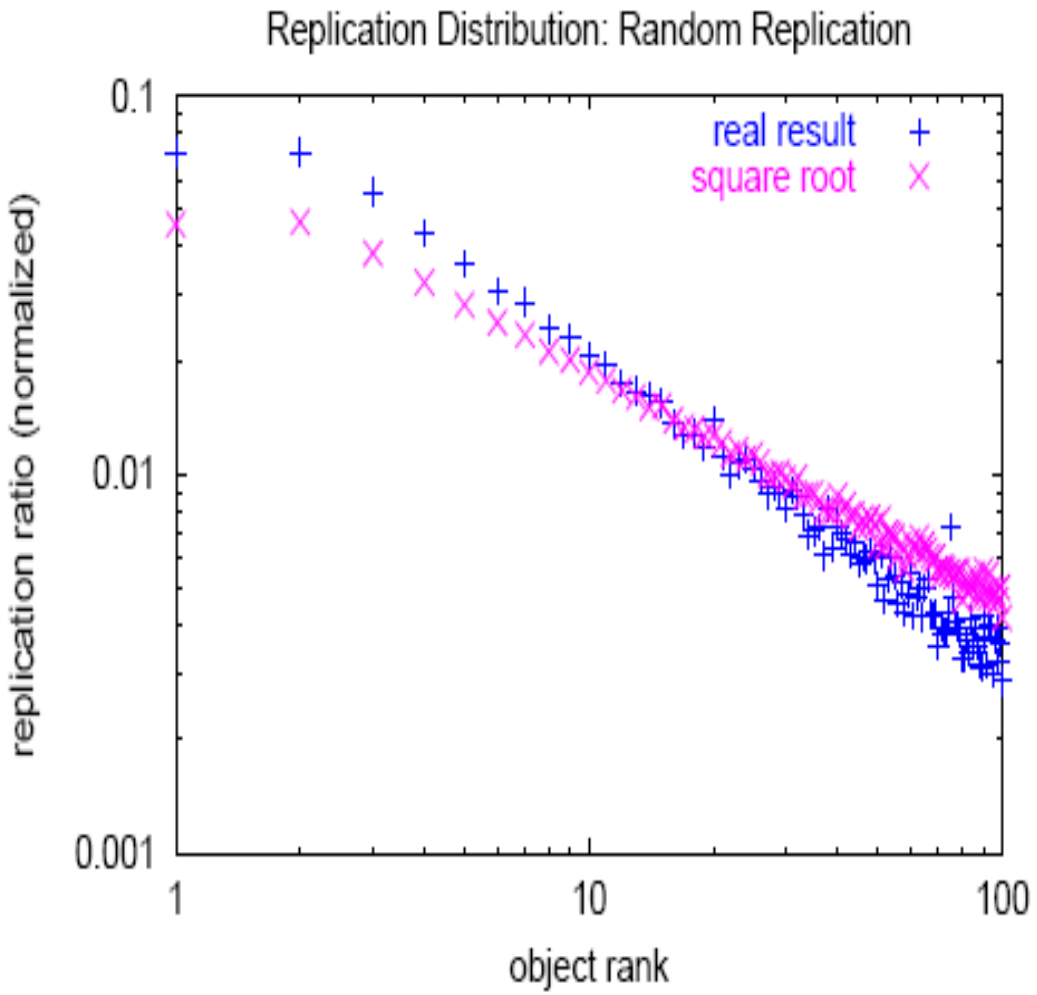
- Average search size
  - The uniform and proportional strategies result in the same avg search size (avg number of random probes to find an object)
  - Avg search sizes for individual objects differ with the proportional strategy
  - Square-root can reduce avg search size
- Utilization ratio
  - Avg utilization ratio is 1 if we run each search until success
  - Variance is quite different with different strategies

Achieving good replication

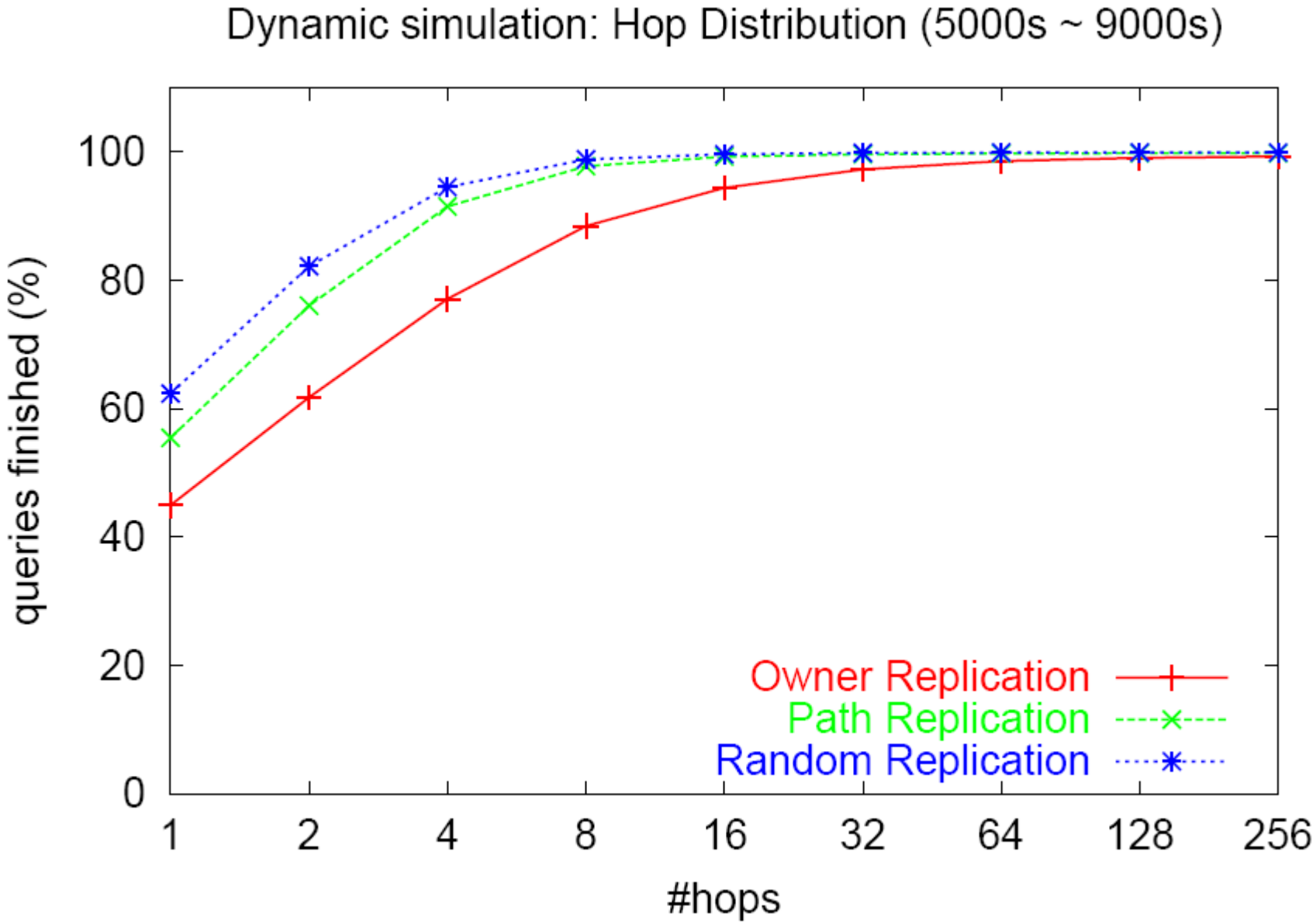
- Owner replication
  - Results in proportional replication
- Path replication
  - Results in square root replication
- Random replication
  - Same as path replication, only using the given number of random nodes, not the path
- Removal strategy
  - Must be random or based on fixed time

Achieved replication distribution





Performance of different replications



GIA: motivation

- Unstructured networks are good
  - Fault tolerant, robust
  - Support arbitrary keyword queries
- Flooding is not good
- Random walks are better but not perfect
  - They are too blind without some help, such as biased walk (see scale-free nets)
  - Load balancing can be a problem esp in heterogeneous networks under high query load

GIA motivation

- Major problem seems to be poor load balancing
- So let us now make they query “throughput” of the system the main evaluation criterion
  - Load balancing is the major thing to optimize here
- We know networks are heterogeneous
- This means we must make sure nodes process queries proportional to their bandwidth
  - Topology: Let's adapt the topology so that all nodes have the right amount of neighbors
  - Flow control: Let's cleverly limit the number of forwarded queries to neighbors

Components

- One hop replication
  - Pointers to objects are replicated on neighbors
- Topology adaptation
  - Put most nodes within short reach of high capacity nodes
- Flow control
- Search protocol
  - Random walk biased towards high capacity (not high degree) nodes
  - Note that without topology adaptation, capacity and degree do not necessarily correlate

Topology adaptation

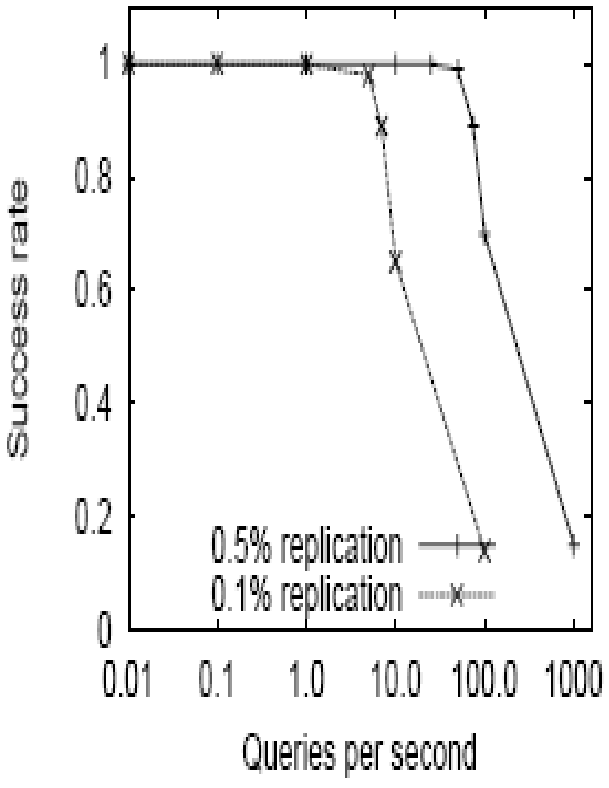
- All nodes keep trying to improve their neighbor set until possible (satisfaction function)
  - Candidates in “host cache”
  - Using candidates, we continuously want to
    - **increase the capacity of our neighbors**
    - **decrease the number of neighbors of our neighbors**
- Topology is undirected: handshake mechanism
  - We need to ask nodes to accept us as a neighbor
  - They might need to drop neighbors

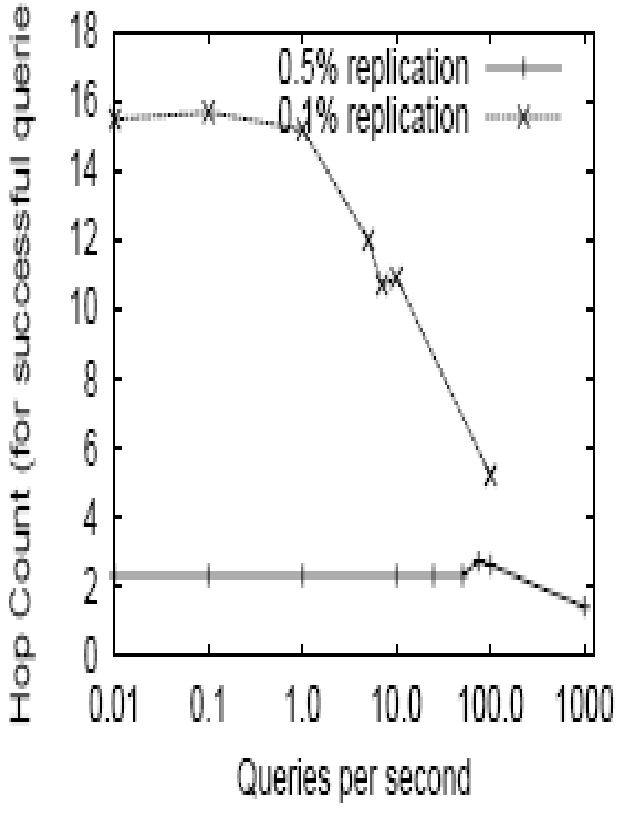
Flow control

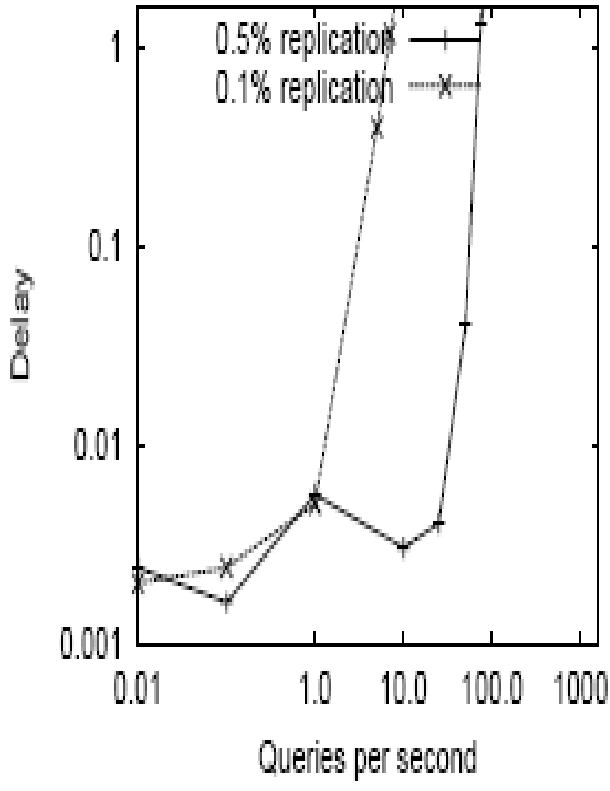
- Nodes assign tokens to their neighbors proportional to their capacity
- More tokens are assigned to higher capacity nodes (incentive to be honest when reporting capacity)
- Search protocol
  - Picks highest capacity neighbor to forward query, for which there is a token available

Performance measures

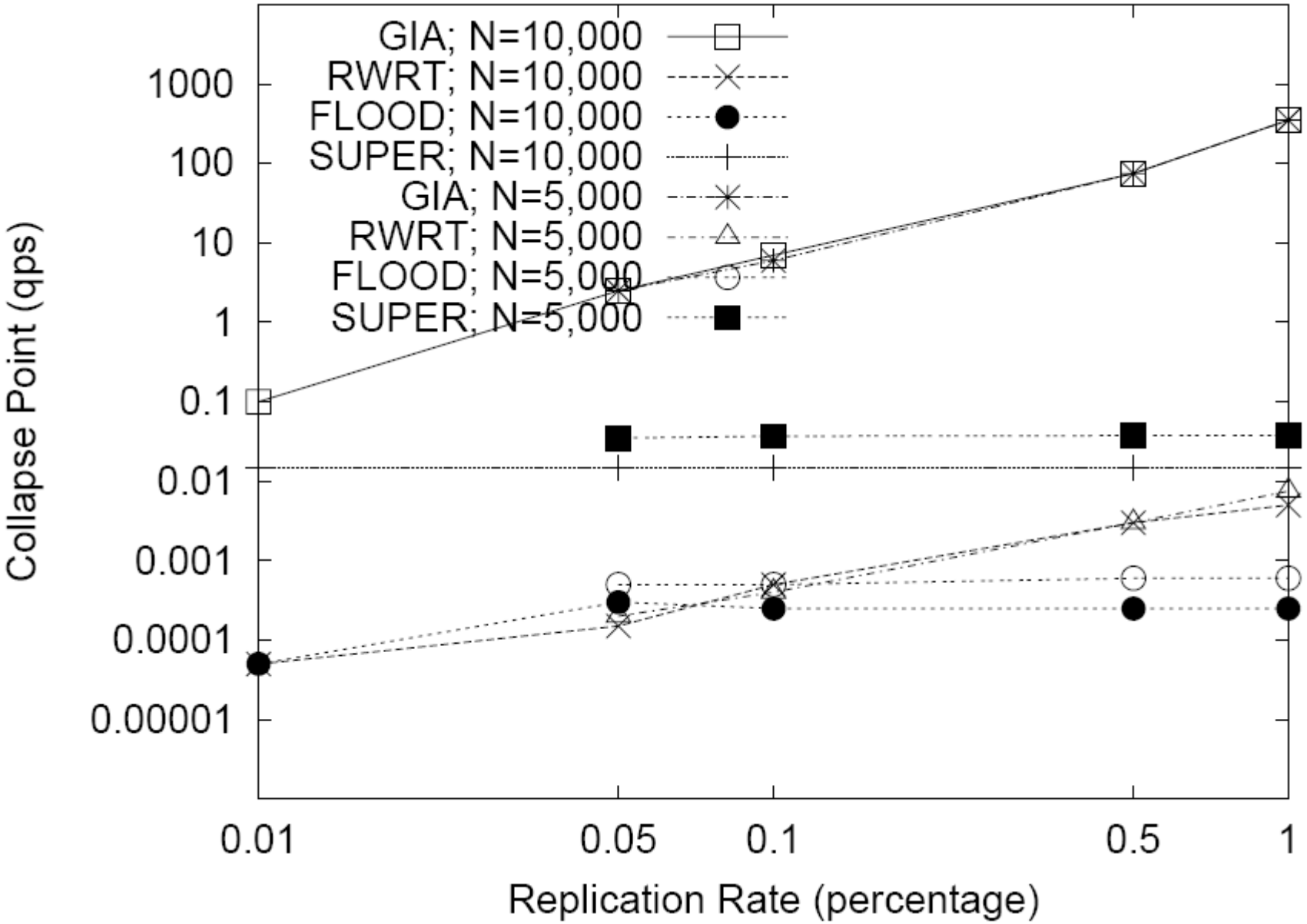
- Main focus is system load, and metrics as a function of that
- Behavior is captured by “collapse point”: success rate passes 90%



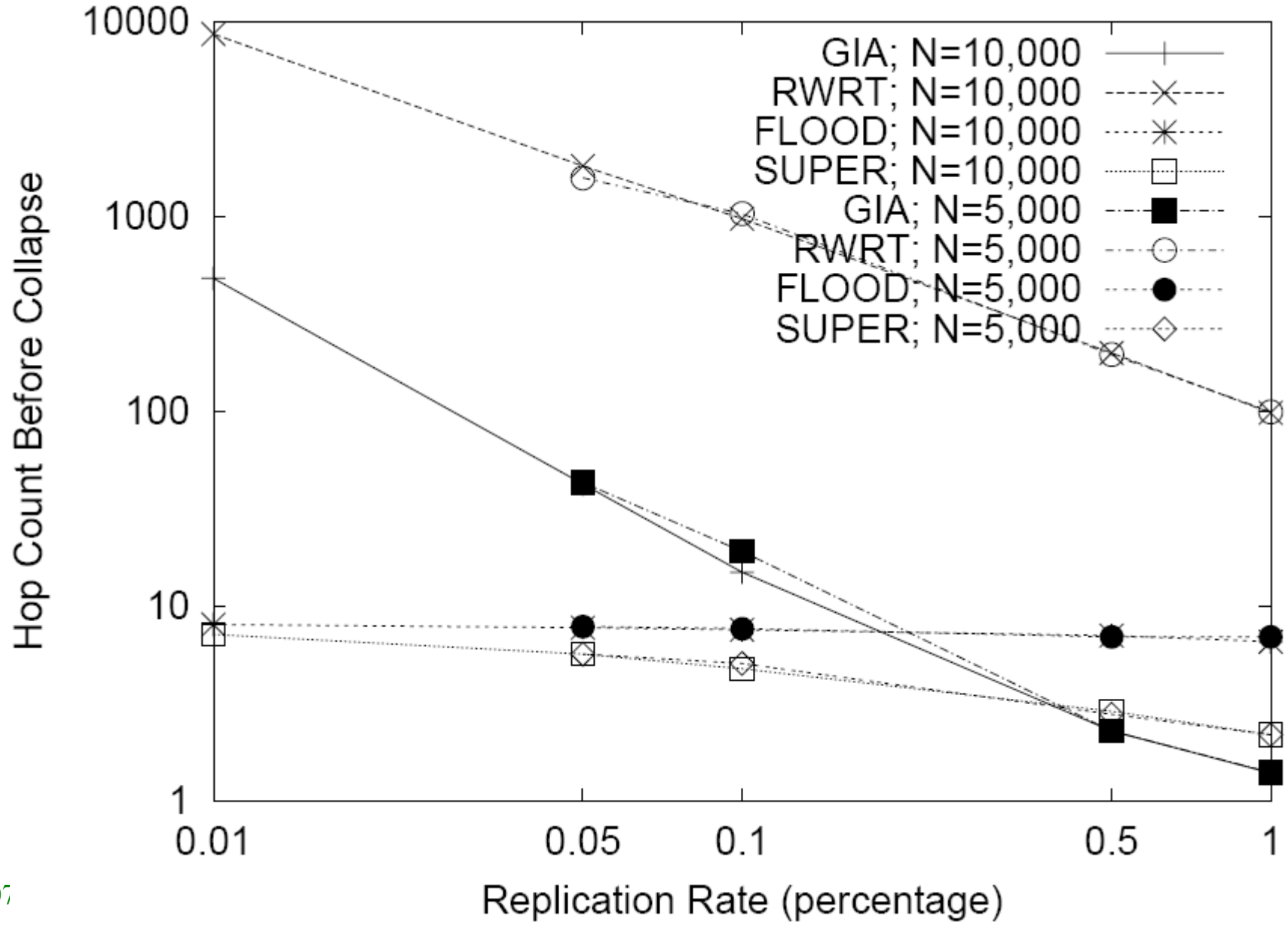




Results: collapse points



Results: hop count before collapse



Factor analysis of components

- 10,000 nodes, 0.1% replication
- Only all components together achieve the desired effect

Algorithm	Collapse Point	Hop-count
GIA	7	15.0
GIA – OHR	0.004	8570
GIA – BIAS	6	24.0
GIA – TADAPT	0.2	133.7
GIA – FLWCTL	2	15.1

Algorithm	Collapse Point	Hop-count
RWRT	0.0005	978
RWRT + OHR	0.005	134
RWRT + BIAS	0.0015	997
RWRT + TADAPT	0.001	1129
RWRT + FLWCTL	0.0006	957

Summary

- Major components are
  - Search algorithm
  - Overlay topology
  - Replication strategies (pointer and object)
  - Flow control
- All of these can (and should) be adapted cleverly!
- At least topology and replication can be emergent as well (that is, influenced by aggregate user behavior)
- Problem of poor performance on rare files still exists

References

- Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In Proceedings of ACM SIGCOMM 2003, pages 407–418, 2003.
- Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In Proceedings of the 16th ACM International Conference on Supercomputing (ICS'02), 2002.
- Matei Ripeanu, Adriana Iamnitchi, and Ian Foster. Mapping the gnutella network. IEEE Internet Computing, 6(1):50–57, 2002. (doi:10.1109/4236.978369)
- Lada A. Adamic, Rajan M. Lukose, Amit R. Puniyani, and Bernardo A. Huberman. Search in power-law networks. Physical Review E, 64:046135, 2001.
- Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47-97, January 2002.
- Mark E. J. Newman. Random graphs as models of networks. In Stefan Bornholdt and Heinz G. Schuster, editors, *Handbook of Graphs and Networks: From the Genome to the Internet*, chapter 2. John Wiley, New York, NY, 2002.

Search in Structured Networks

Outline

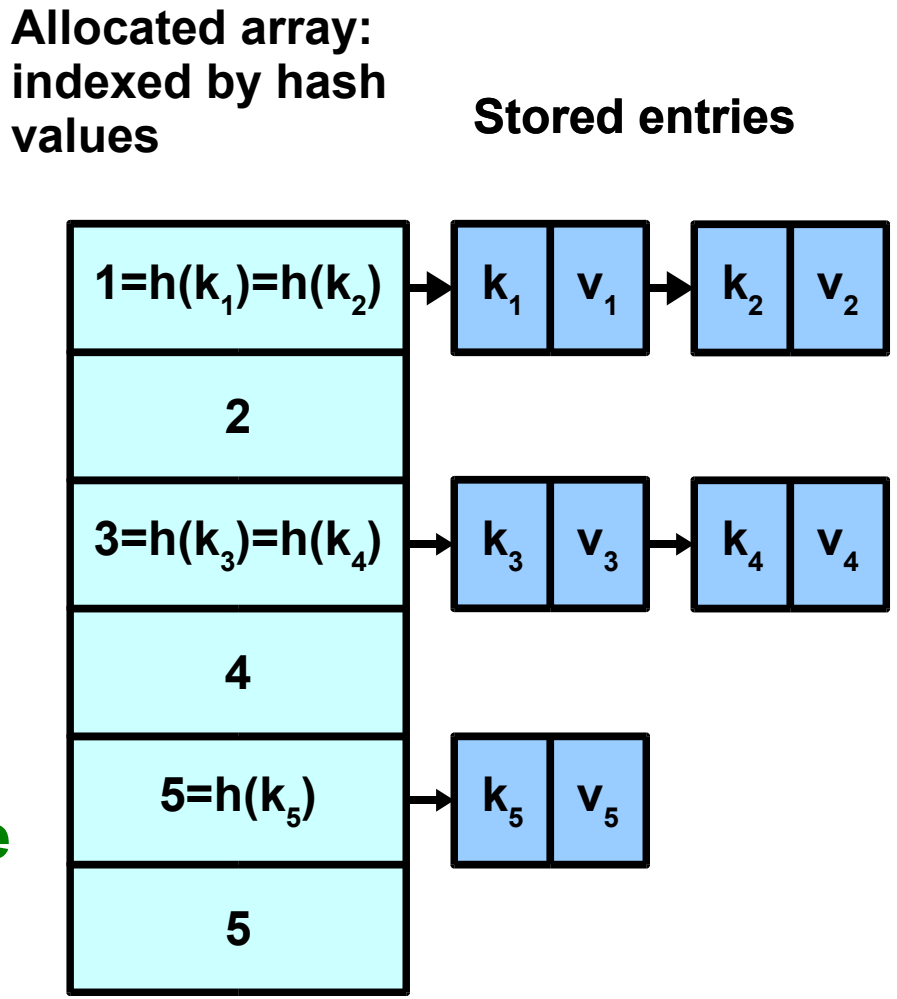
- Hash tables and distributed hash tables (DHT): the abstraction
- An example implementation: Chord
- Implementing keyword search on a DHT
- Some other other DHTs: Pastry and CAN
- Summary of DHT complexity results
- Hybrid (structured/unstructured) approaches to search

Motivation

- We have seen search does well in unstructured networks except when items are rare
- Can we come up with a technique that provides efficient search (lookup) for rare items?
  - Yes: distributed hash tables (DHT)
- What is the ultimate solution that is robust, cheap and works for popular and rare items too?
  - Hybrid solutions?
  - Something not yet invented?
- DHTs are good for other things too

Hash tables

- Store arbitrary keys and satellite data (value)
  - **put(key,value)**
  - **value = get(key)**
- Lookup must be fast
  - **Calculate hash function h() on key that returns a storage cell**
  - **Chained hash table: Store key (and optional value) there**



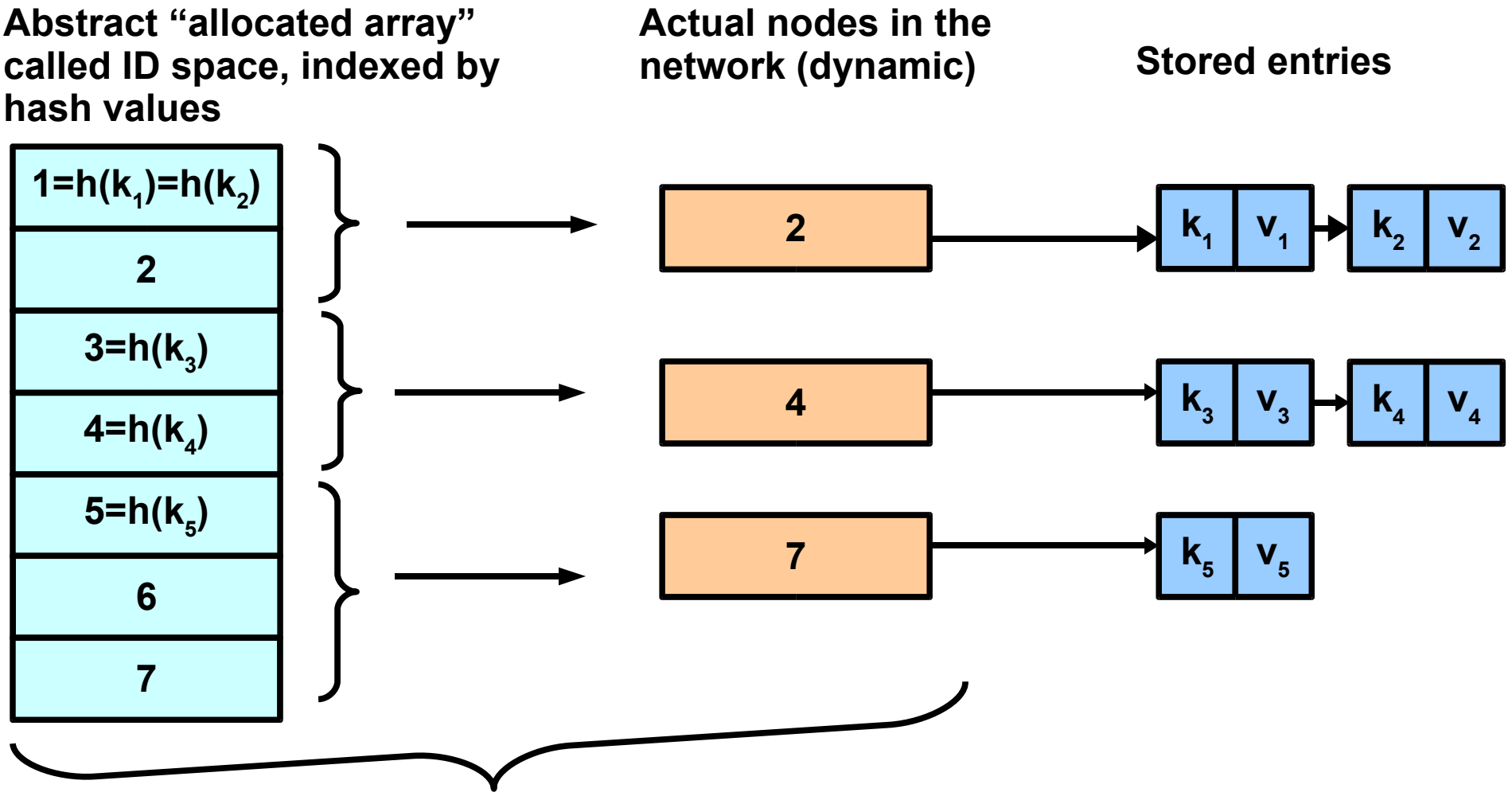
Why a hash table?

- Most often the point of a hash table is fast and cheap lookup of data indexed by a key
- When used for search, the issue of query richness comes up
  - In random walk/flooding, a query can be arbitrarily complex (even full text search with regular expressions).
  - If we use only key based lookup, we must be creative and work more to allow for non-trivial queries
    - **Inverse indexing, etc**
- The idea is trading some flexibility and simplicity off for efficiency and effectiveness

Distributed hash table

- We want hash table functionality in a p2p network: lookup of data indexed by keys
- Assume the storage space is a distributed set of nodes (not an array)
  - Note that in all cases we will have an overlay network that connects these nodes in tricky ways
  - The exact set of nodes is not known locally and can change all the time
  - We work with an idealized storage space,
    - **Hash function maps to this ideal space**
    - **We assign parts of the space to nodes in a distributed way dynamically: extra complications**

Distributed hash tables



Distributed has tables: main functions

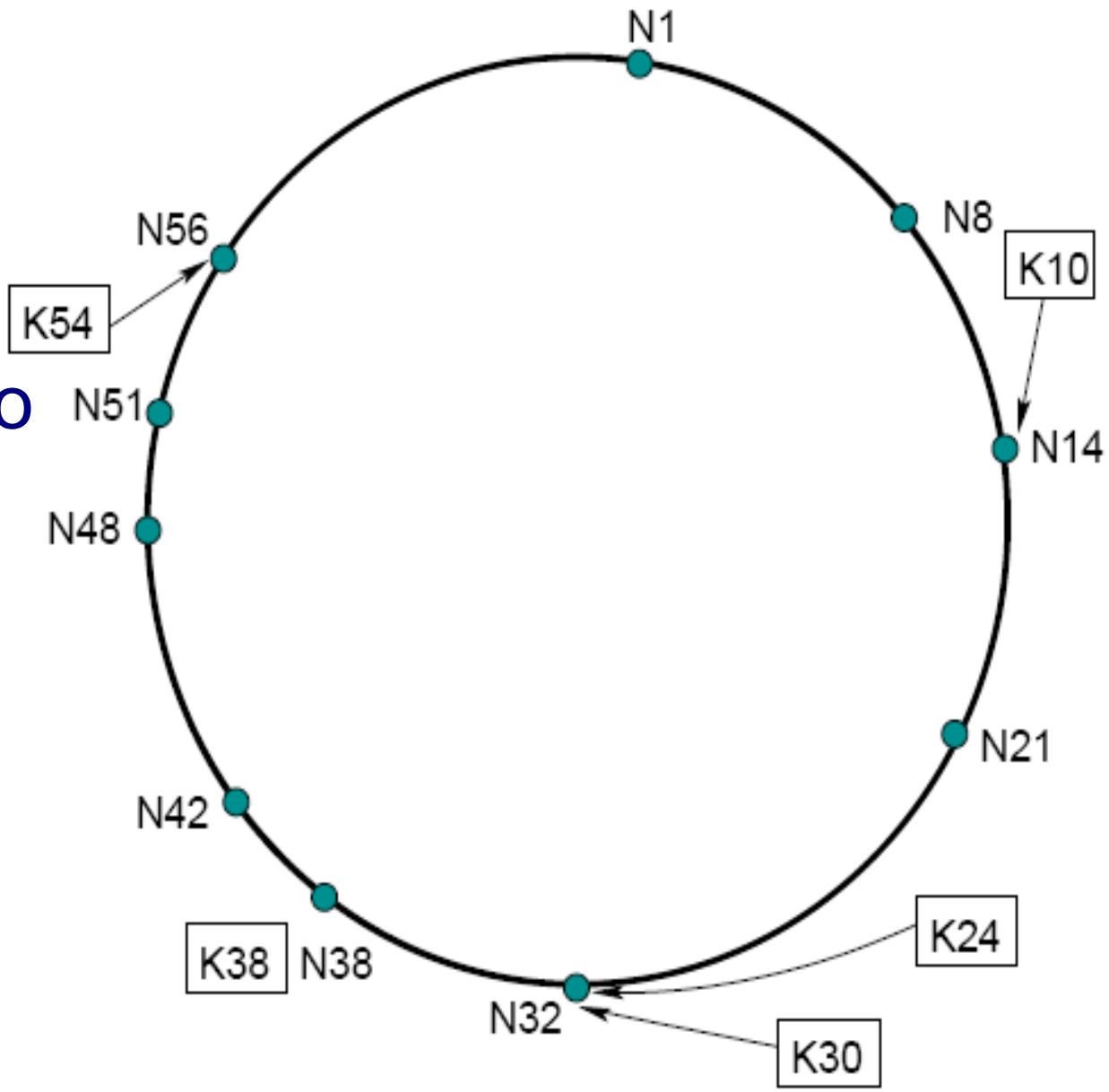
- Key-hash $\leftrightarrow$ node mapping
  - Assign a unique live node to a key
  - Find this node in the overlay network quickly and cheaply (routing)
- Maintenance, optimizations
  - Implement DHT API on top of routing
  - Load balancing: maybe even change the key-hash $\leftrightarrow$node mapping on the fly
  - Replicate entries on more nodes to increase robustness
  - etc

Chord

- Most cited DHT implementation (3000+ citations to date!!!)
- Advantages
  - Simple
  - Good storage and message complexity
- Consistent hashing based on an ordered ring overlay
  - This is why it is “structured”

Hashing in the Chord ring

- Identifier circle
  - 10 nodes
  - 5 keys
- Both keys and nodes are hashed to 160 bit IDs (SHA-1)
- Then keys are assigned to nodes using consistent hashing
  - Successor in ID space

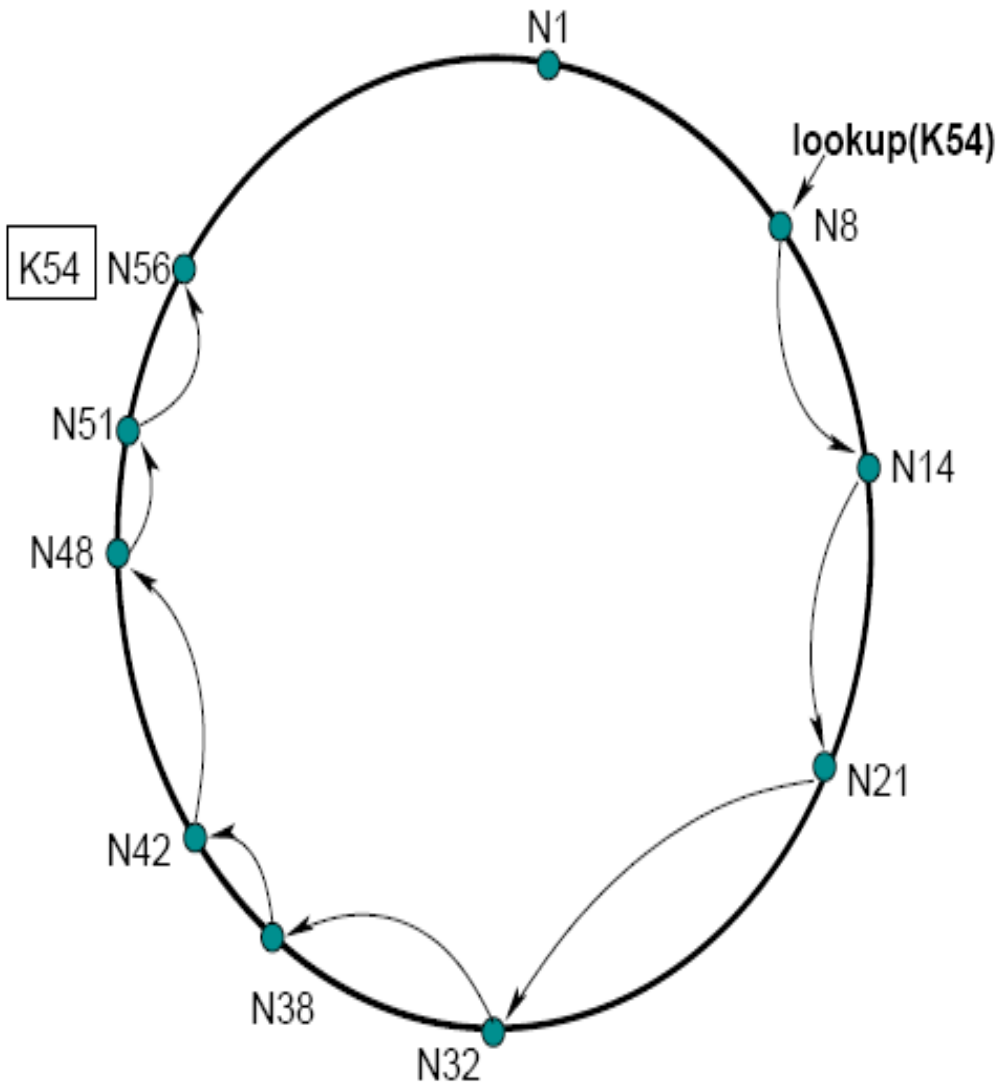


Chord hashing properties

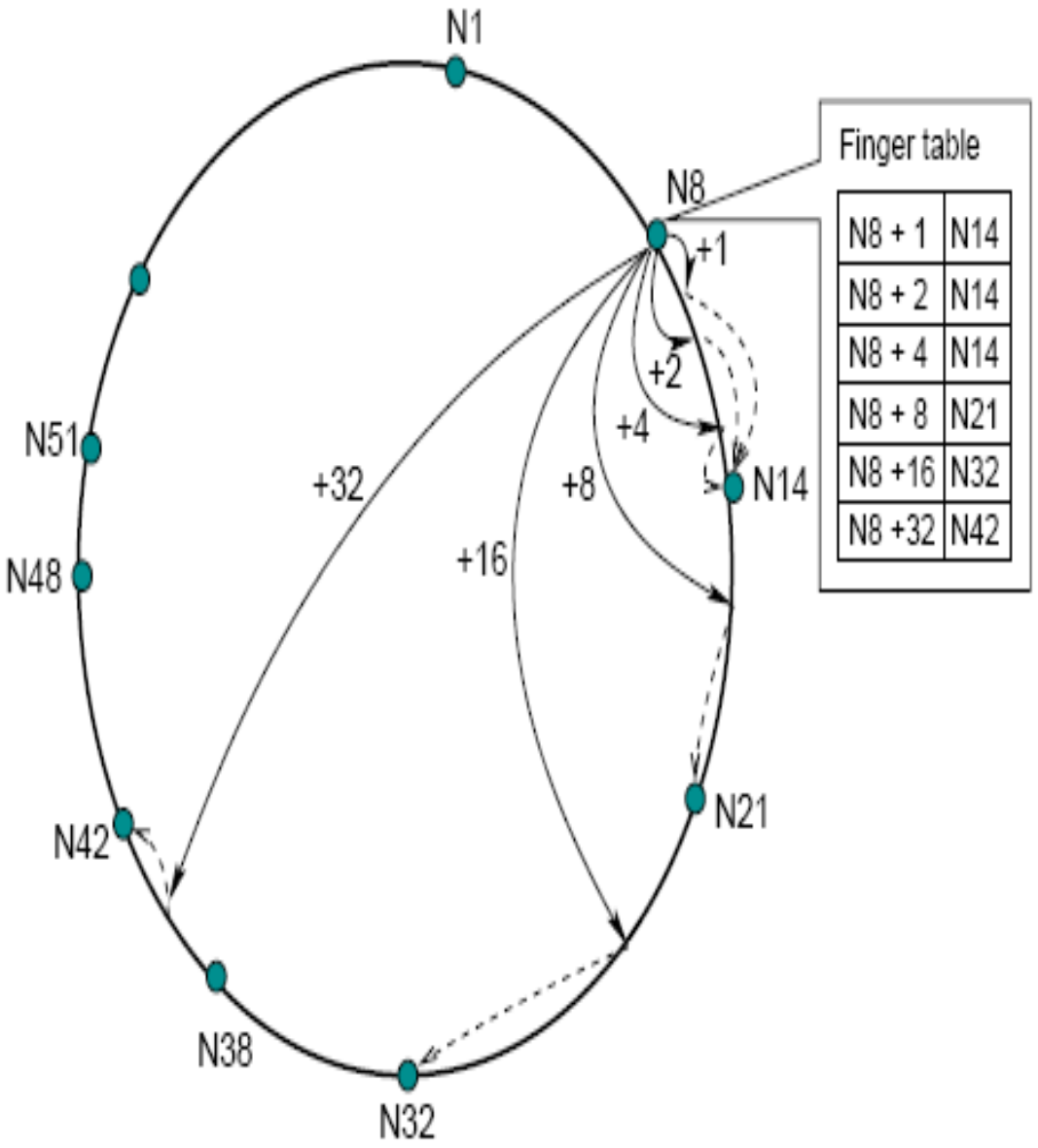
- Consistent hashing
  - randomized
    - **All nodes receive roughly equal share of load**
  - Local
    - **Adding or removing a node involves an O(1/N) fraction of the keys getting new locations**
- Actual lookup
  - Chord needs to know only O(log N) nodes in addition to successor and predecessor to achieve O(log N) message complexity for lookup

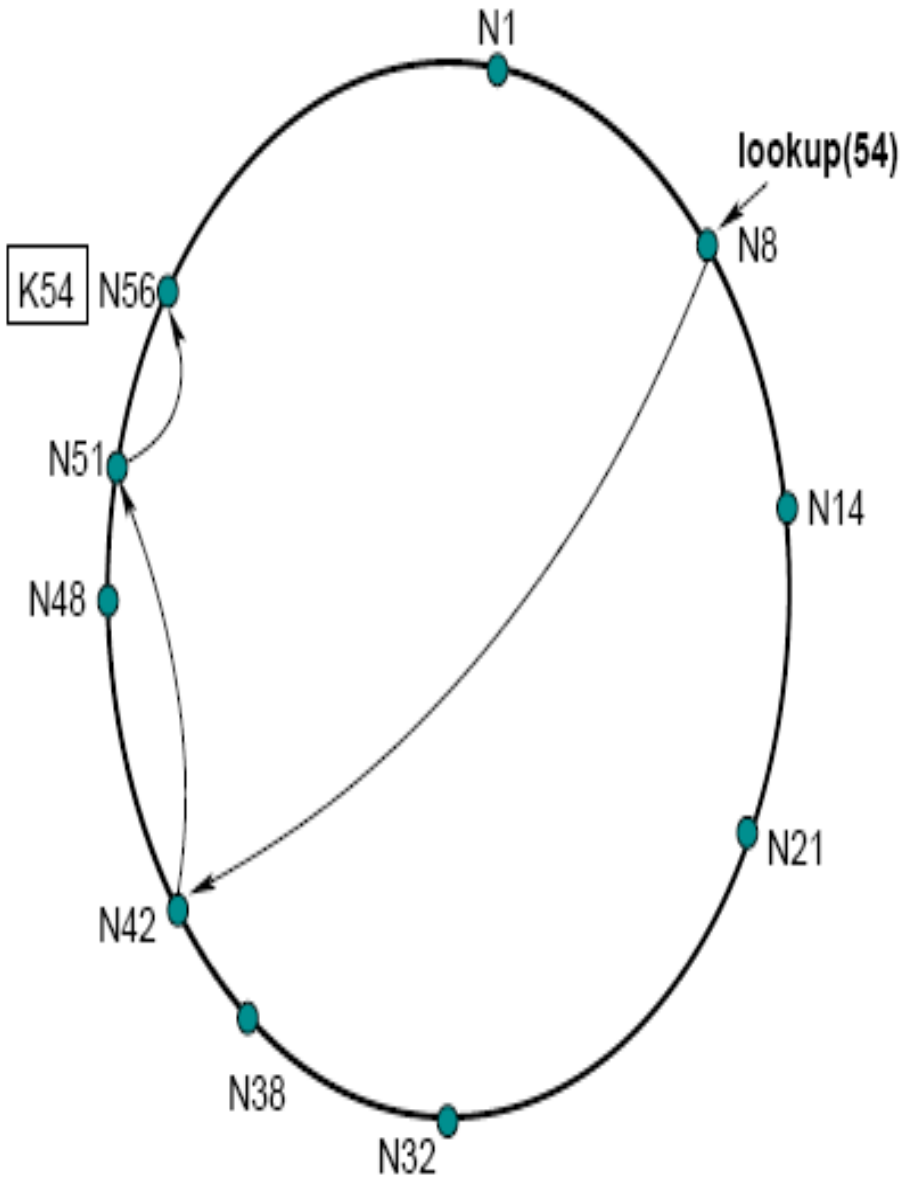
A primitive lookup algorithm

```
// ask node n to find the successor of id
n.find_successor(id)
  if (id ∈ (n, successor])
    return successor;
  else
    // forward the query
    // around the circle
    return successor.find_successor(id);
```



A scalable lookup algorithm





A scalable lookup algorithm

```
// ask node n to find the successor of id
n.find_successor(id)
  n' = find_predecessor(id);
  return n'.successor;
```

```
// ask node n to find the predecessor of id
n.find_predecessor(id)
  n' = n;
  while (id ∉ (n', n'.successor])
    n' = n'.closest_preceeding_finger(id);
  return n'
```

- Jump to the closest preceeding finger
- O(logN) jumps
- O(logN) neighbors stored at each node
- This formulation assumes one node coordinates the lookup (not recursive) but could be

Join: an expensive approach

- A new node has to
  - Fill its own successor, predecessor and fingers
  - Notify other nodes for which it can be a successor, predecessor of finger
- With several optimizations this can be done in O(logN) time
- But the resulting protocol is complex
- Can be done simpler, using a relaxed and simple stabilization protocol, used also for error correction

Join: a relaxed approach

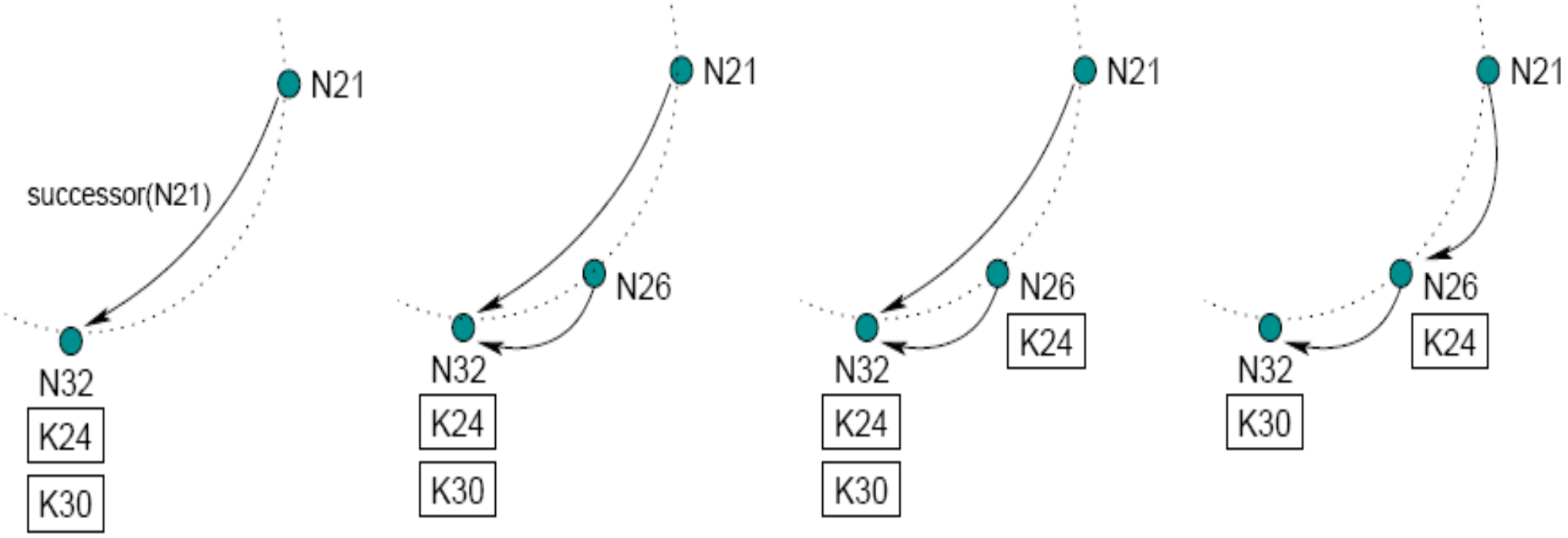
- If the ring is correct, then routing is correct, fingers are needed for the speed only
- Stabilization
  - Each node periodically runs the stabilization rutine
  - Each node refreshes all fingers by periodically calling find_successor($n+2^{i-1}$) for a random i
  - Periodic cost is O(logN) per node due to finger refresh

```
n.stabilize()
  x = sucessor.predecessor;
  if (x ∈ (n, successor) )
      successor = x;
  successor.notify(n);
```

```
n.join(n')
  predecessor = nil;
  sucessor =
          n'.find_successor(n);
```

Join: a relaxed approach

- Node join: find successor and then stabilize
  - Ring is immediately joined: routing works
  - Routing also fast enough if not too many nodes join concurrently, but eventually fingers will be ok too

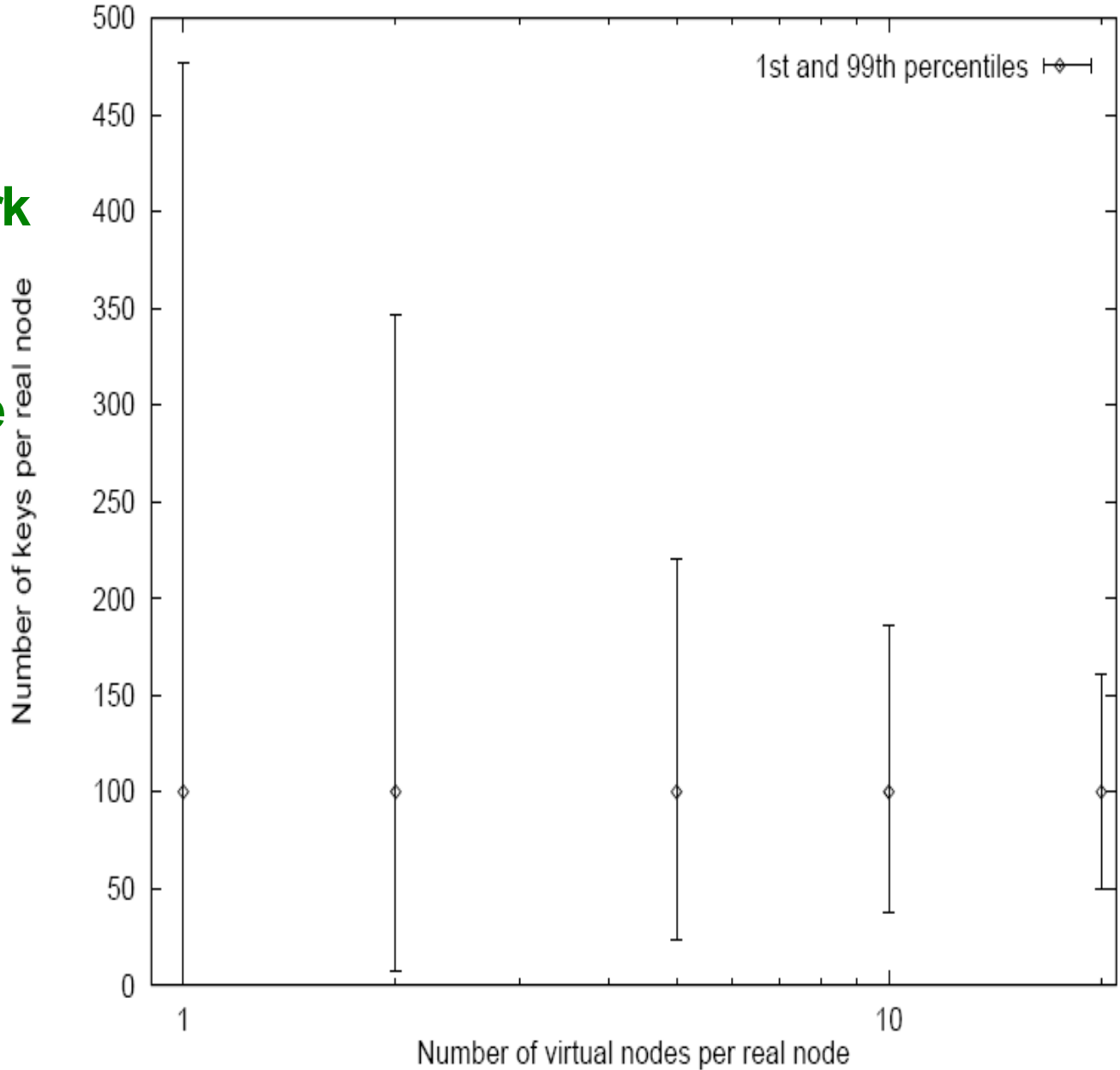


Failure and replication

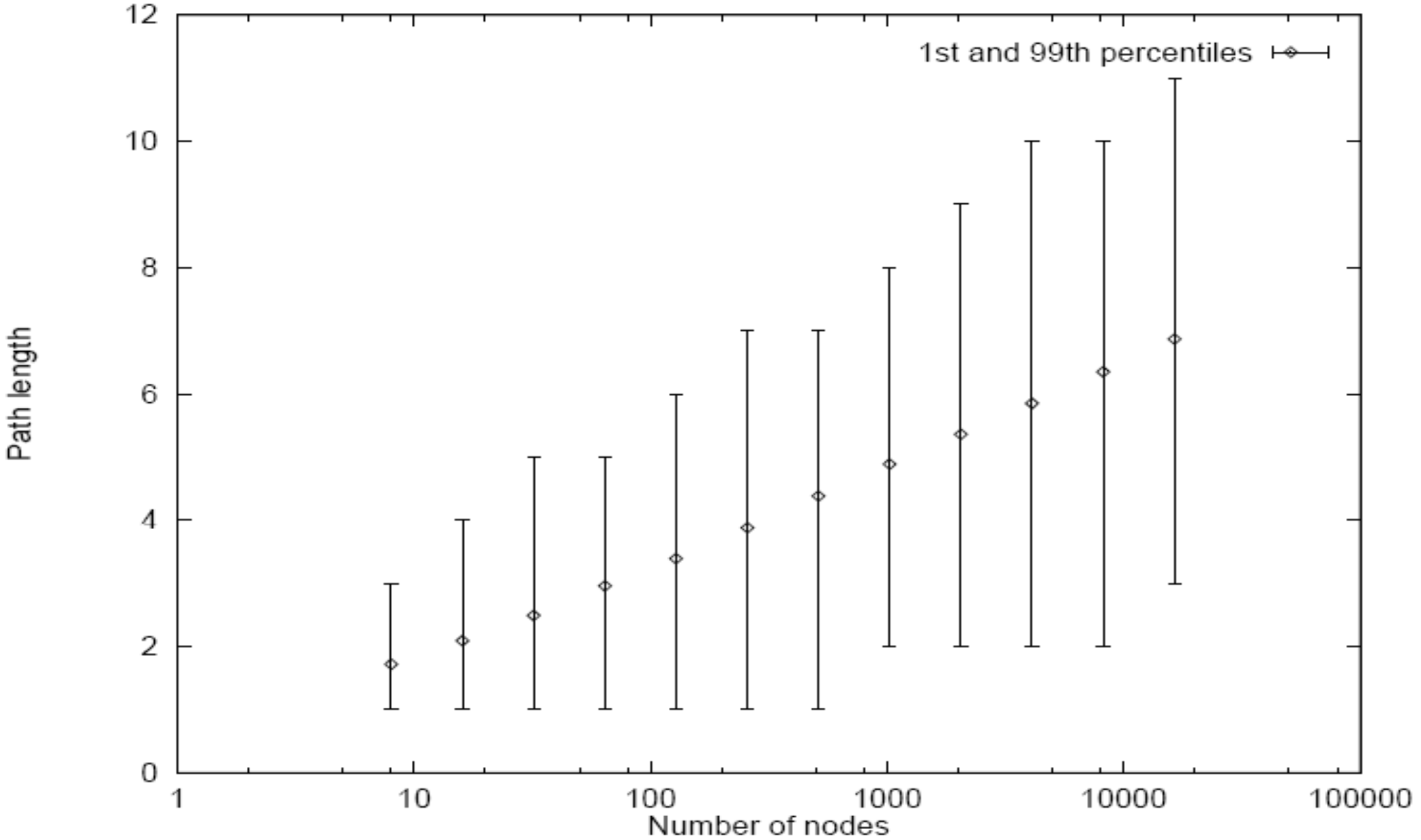
- Failed nodes are handled by
  - Replication: instead of one successor, we keep r successors
    - **More robust to node failure (we can find our new successor if the old one failed)**
  - Alternate paths while routing
    - **If a finger does not respond, take the previous finger, or the replicas, if close enough**
- At the DHT level, we can replicate keys on the r successor nodes
  - The stored data becomes equally more robust

Virtual nodes

- A physical node acts as if it was many nodes
  - **The Chord network appears to be larger**
  - **One phisical node gets a much more balanced number of keys**
  - **Maintenance cost grows**
  - **Path length does not grow significantly**



Path length in simulations



Conclusions

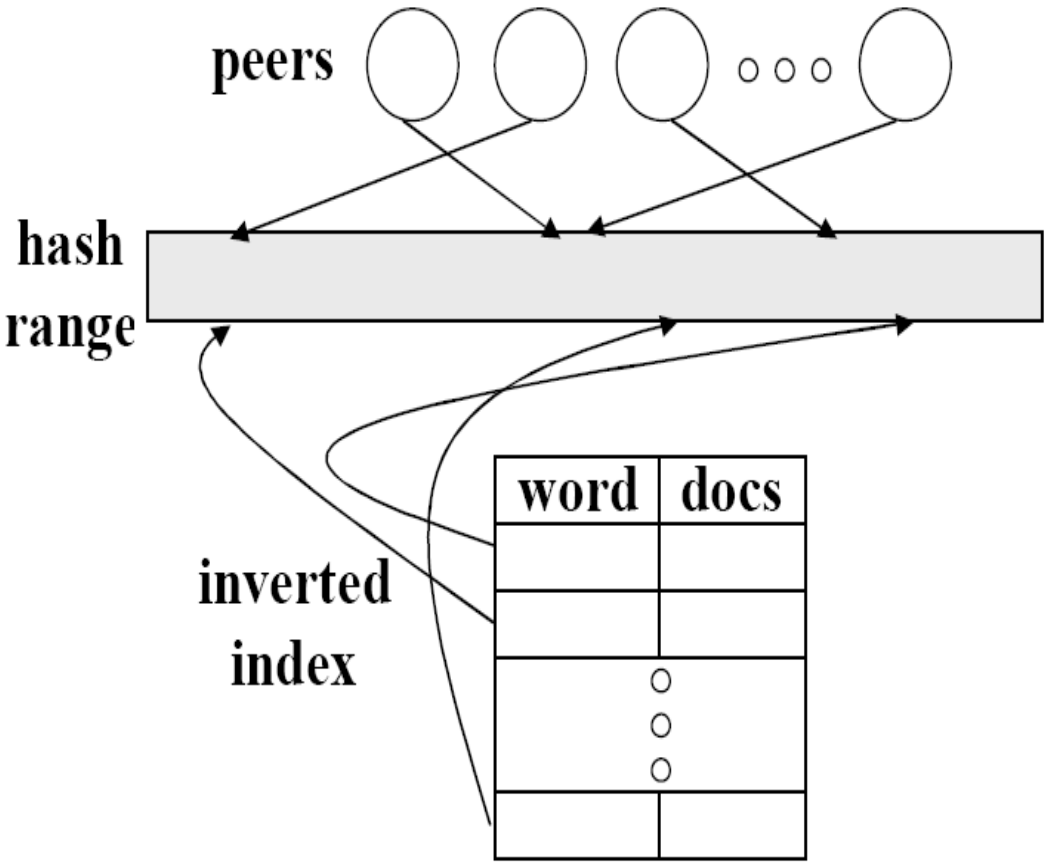
- The DHT abstraction can be implemented in a fairly simple and efficient way
- All implementations are based on a distributed data structure, a so called “structured overlay”
  - Chord used an ordered ring, with fingers (shortcuts)
- Some remaining issues to consider
  - Can more complex and more flexible applications be implemented such as keyword search (yes)
  - Can the storage or message complexity improved (yes)
  - So, what is the best way to implement a file sharing system?

Keyword search in DHTs

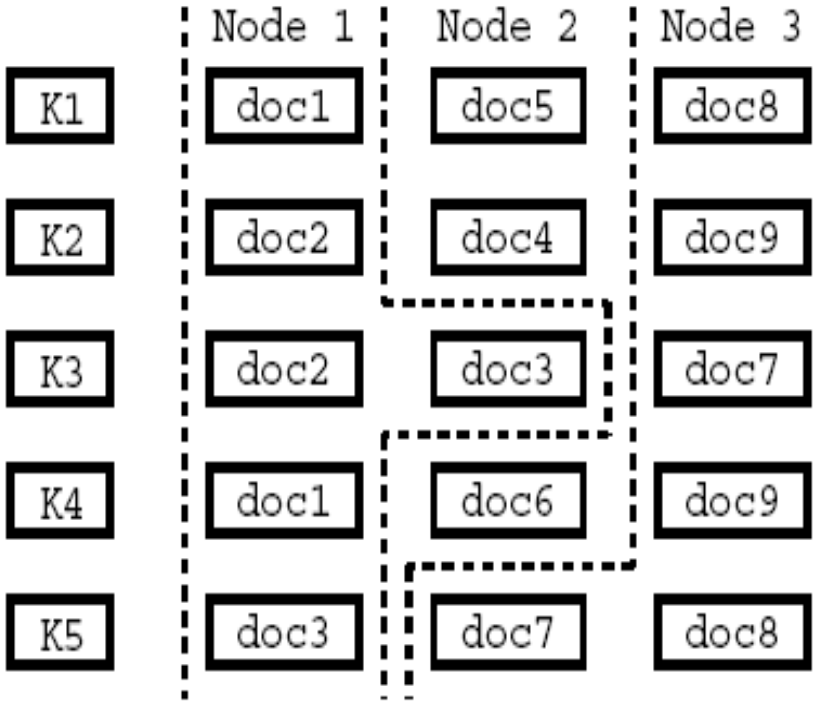
- DHTs support only key lookup by default
- We need to perform complex queries as in unstructured networks
- We need to be creative: here we discuss an inverted index-based approach
  - Document identifiers are stored in a DHT with all contained keywords as keys
  - All keywords are looked up and the intersection of matches is calculated
  - A few techniques to optimize the cost of all this

Inverted index approach

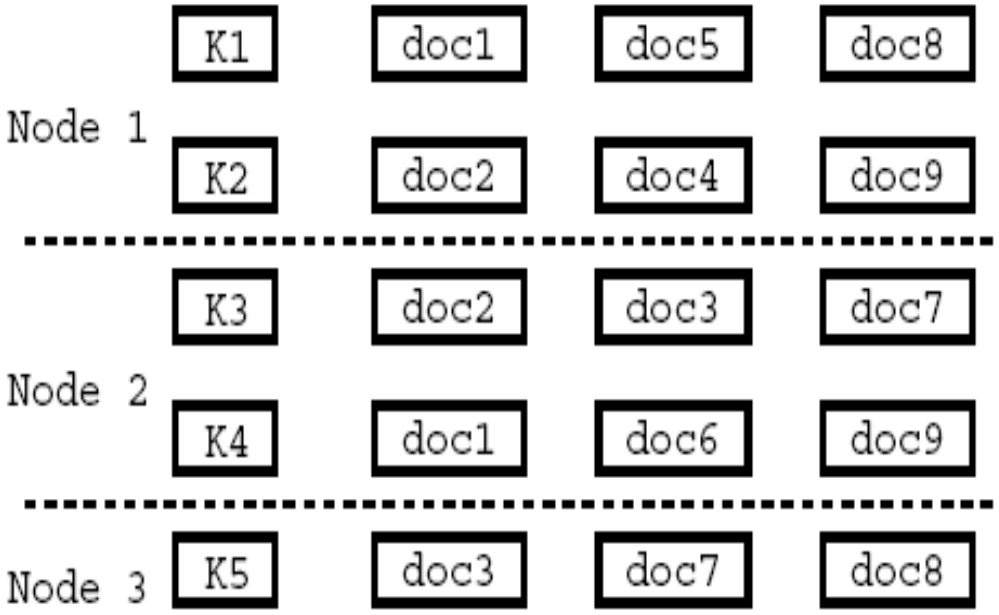
- Inverted index usual in search engines
  - For all keywords collect the documents that contain that keyword
  - Create intersection, union, etc, base on keyword based query
- Do that P2P style



Distributing the inverted indices



Horizontal partitioning



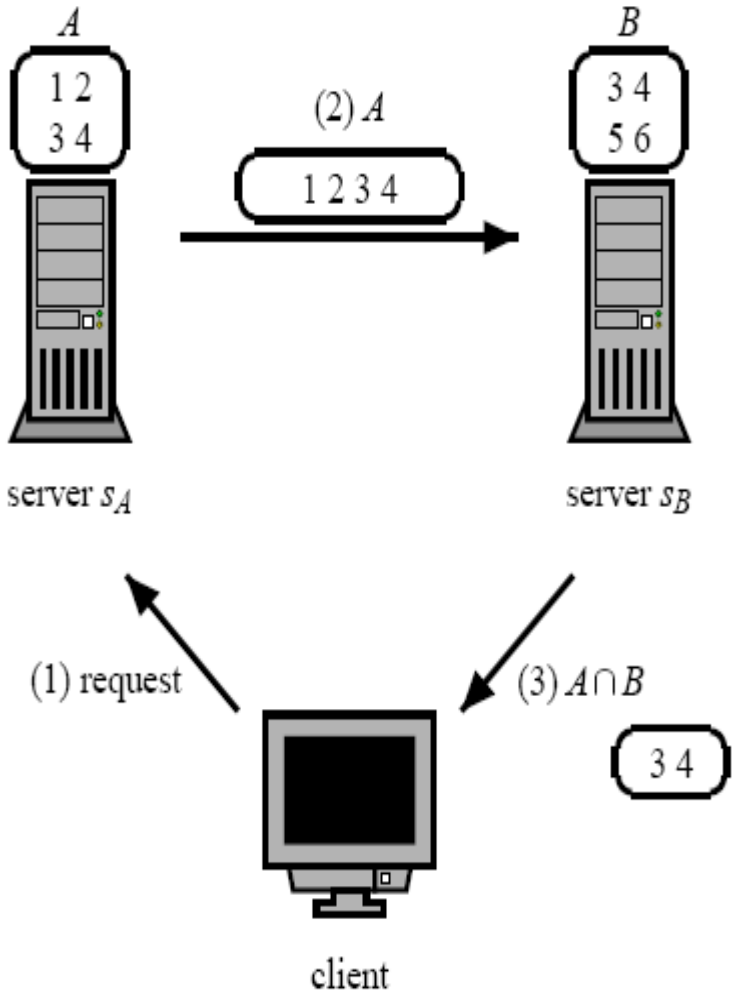
Vertical partitioning

Mainly centralized services
cheap update, expensive lookup

Better if update is rare but
communication is expensive

DHT for storing documents sets

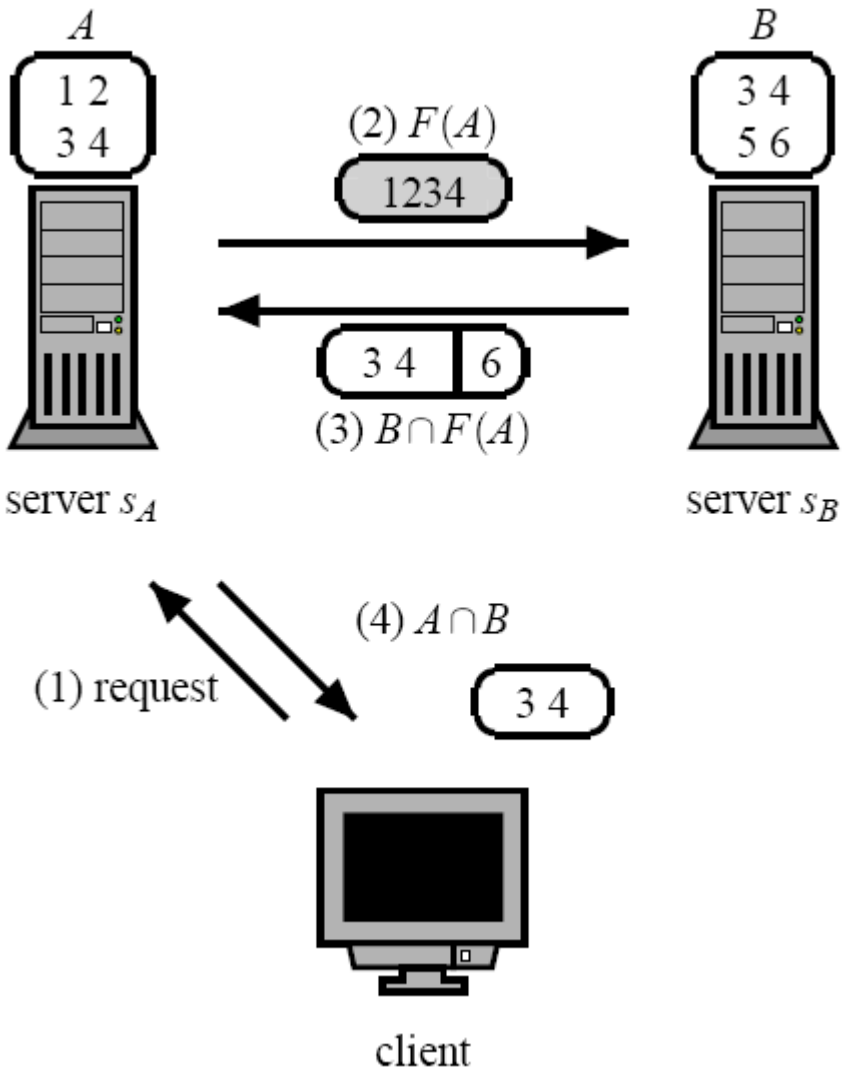
- A DHT is used to map keywords to nodes
  - A node is assigned a set of keywords, and stores sets of pointers to documents that contain the given keyword
- The retreival procedure needs to AND sets
  - Naive procedure shown
    - **Set A on server s_A contains documents that have keyword k_A**



Request is "k_A & k_B"

Optimizations: Bloom filters

- Bloom filter of A is sent to s_B (2)
- s_A removes false positives ("6" in this example)
- It saves bandwidth if set is large enough
  - We use filters for more than 300 elements only
- Smaller set should be visited first (natural thing)
- Works for more keywords too
  - All servers need to see the final result to remove false positives



Optimizations: Caches

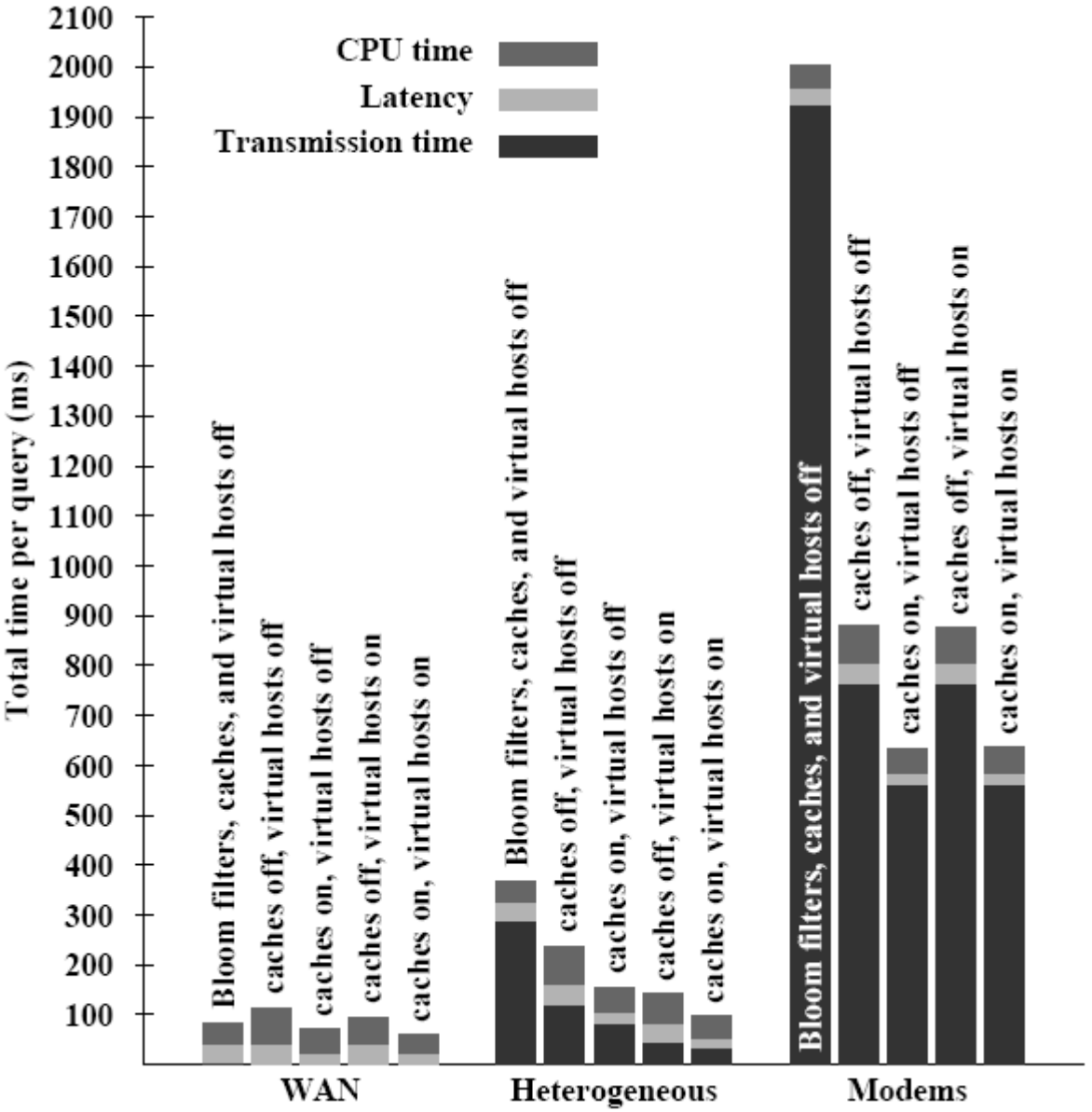
- Bloom filters or unencoded keyword match sets can be cached
  - Some measurements indicate there are very popular keywords (power law distr) so hit rate can be good
- Utilization of caches
  - A server checks if it has cached info on a next keyword to be intersected
  - If yes, performs intersection locally, skips the corresponing server

Optimizations: virtual nodes

- Same idea as in Chord
- Assign virtual nodes proportional to capacity
  - Number of keywords proportional to capacity
  - Variance due to random hashing is reduced (as in Chord)
- Load balancing still a problem
  - Keyword popularity is not equal
    - **Number of keywords is not a good measure, popularity needs to be considered too**

Experiments

- Network types
  - All backbone, all modem, and gnutella trace
- Search trace: IRCache log file
- Parameters
  - Bloom filter threshold 300, Bloom filter size: 18/24 with cache on/off



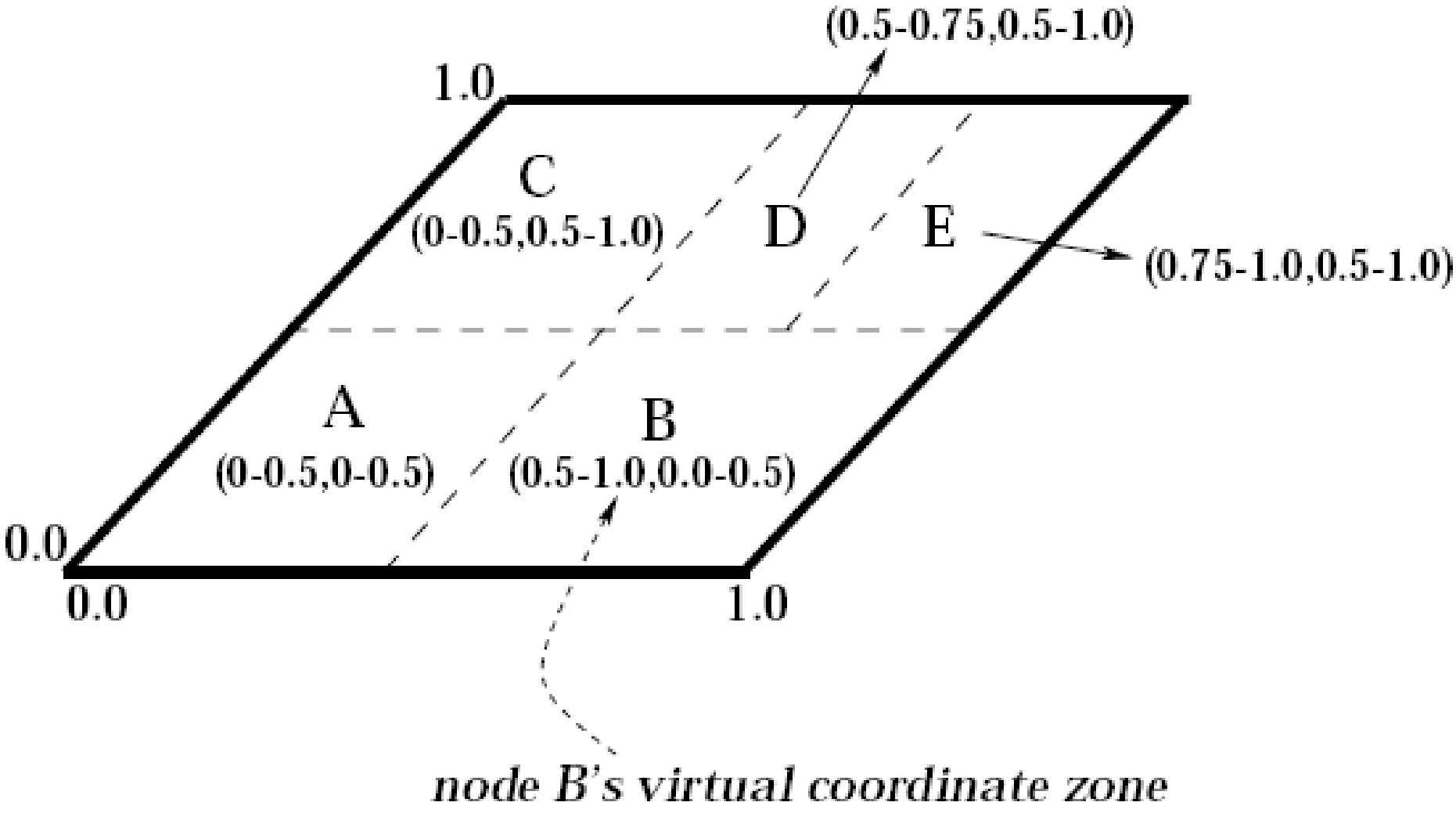
Other DHT designs

- A DHT is an abstraction
  - Eg previous keyword search technique used a generic DHT
- A DHT has many popular implementations, we review two briefly: CAN and Pastry
- Different implementations have different tradeoffs and complexity properties, we review these

Content addressable network (CAN)

- CAN became the name of a specific algorithm, although it is in fact a synonim to DHT
- Logical space to which keys are mapped by a hash function
  - D-dimensional real space $[0,1]^d$
- All nodes are assigned a partition of this space
  - At any point in time the set of current nodes cover the space
- Compare with Chord!
  - Logical space is different; partitioning of this space is implicit (but nevertheless well defined)

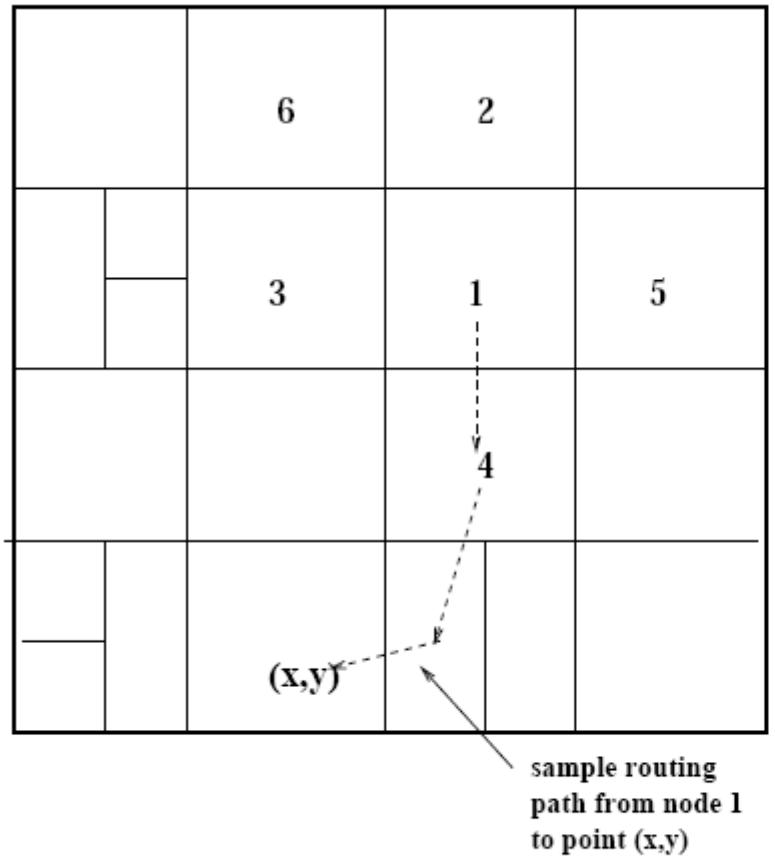
CAN logical space



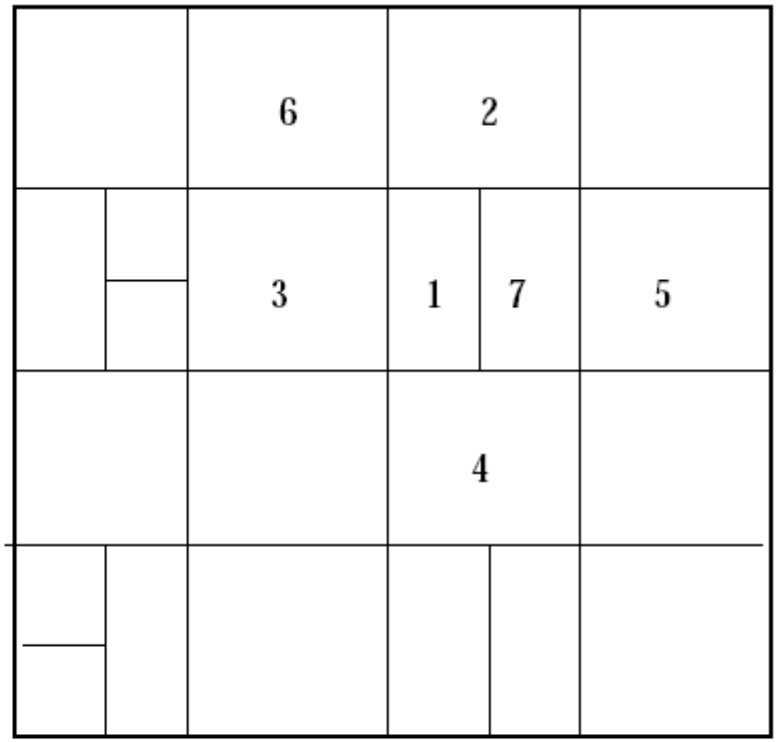
Routing and node join

- Greedy routing to neighbor that is closest to destination
  - Hop count is $O(dN^{1/d})$
  - Number of neighbors is O(d)
  - If d=O(logN), then roughly same as Chord
- Join
  - Create random point in virtual space
  - Find the node that is responsible for that point
  - Split the block of that node and update neighbors appropriately

Node join in CAN



1's coordinate neighbor set = {2,3,4,5}
7's coordinate neighbor set = { }



1's coordinate neighbor set = {2,3,4,7}
7's coordinate neighbor set = {1,2,4,5}

Node departure and recovery

- Failure detection through missing heartbeat
- Neighbors of failed node independently try to take over the zone of the failed node
- The winning node merges the failed zone if possible, or simply holds it if not possible
- Background repair mechanism reassigns zones to prevent fractioning
- Perhaps this is the weakest point of CAN
  - Possibility for incosistency, complex repair and failure handling procedure

Optimizations

- Increasing d
  - Shorter path length, more fault tolerance (more paths) but more neighbors
- More realities
  - Maintain many virtual spaces (CANs) in parallel
  - Replicate stored data on all realities
  - Improves path lengths (jumps inside a node) and fault tolerance (replication, more paths)
- Uniform partitioning: more balanced zone sizes
  - When joining, the selected random node replaces itself with the neighbor with the largest zone

Optimizations

- Improved routing taking proximity into account
  - When selecting a neighbor, use network latency also
- Overloading zones: more nodes in the same zone
  - When joining, zones are not split, only if enough nodes are in the zone
  - Reduces path length (fewer zones)
  - Reduces latency (possibility to select neighbor that has smallest latency)
  - Improved fault tolerance due to redundancy

Pastry: another DHT

- Applies a sorted ring in ID space like Chord
- Virtual space: same as Chord
  - We interpret IDs as sequeces of digits with base 2^b
- Applies Finger-like shortcuts to speed up routing
- The node that is reponsible for a key is the numerically closest (not the successor)
  - Pastry is bidirectional and uses numeric distance

Pastry routing

- If destination is among the leafs, stop
- Otherwise Pastry either forwards the message to a node which
  - has a longer common prefix with the destination or
  - has an equally long prefix but is numerically closer
- Routing is succesful if no L/2 consequtive nodes fail (ring is intact)

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	

Pastry maintenance

- Join
  - Use routing to find numerically closest node already in network
  - Ask state from all nodes on the route and initialize own state
- Error correction
  - Failed leaf node: contact a leaf node on the side of the failed node and add appropriate new neighbor
  - Failed table entry: contact a live entry with same prefix as failed entry until new live entry found, if none found, keep trying with longer prefix table entries

Proximity in Pastry

- All routing table entries are drawn from rather large sets (unlike with Chord)
  - Pastry puts emphasis on optimizing the actual entry based on proximity
  - Entries can be selected based on other criteria as well (semantic proximity, capacity, etc)
- The shorter the common prefix, the larger the set of potential entries (exponentially)
- Original Pastry approach for actually implementing the proximity bias can be improved (not discussed here)

Are Pastry and Chord a different protocol?

- Chord and Pastry are variations of the same idea and can be transformed into each other smoothly
- What is not different
  - Basic idea: ring + shortcuts to exponentially increasing distance
  - Leaf set/successor list: Chord also uses r successors/predecessors
  - Chord can also use more fingers to achieve the same hop count and model a b letter alphabet ID space
  - Same lazy repair protocol for leafs/successors

Are Pastry and Chord a different protocol?

- What is different?
  - A Chord finger is a unique node, whereas with Pastry a routing table entry can come from a large set
    - **Chord could define fingers more loosely, but that needs a different update protocol for fingers**
  - Chord routing is unidirectional, Pastry is direction independent
    - **Chord could easily be bidirectional too with fingers into two directions**

A final note on complexity

- Chord and Pastry have O(logN) storage and hop count complexity
- CAN has $O(dN^{1/d})$ hop count complexity and O(d) storage
- It is possible to have O(1) storage complexity with O(logN) hop count (Viceroy) or with $O(\log^2 N)$ hop count (Symphony)
  - Sounds good but more complex protocols, less reliability and logN is small enough: is it worth it?

So, how to implement filesharing?

- Get the best of both worlds: hybrid approaches
- Use DHT for rare items, random walk for popular items
- What about the topology of the overlay network?
  - Unstructured networks are easy to build and maintain, and robust to churn
  - Are DHT-s realy more complicated or expensive or less robust? Not necessarily
- We overview two hybrid approaches along the lines above

Gnutella: observing the long tail

- Gnutella (latest version with ultrapeers and dynamic query) is excellent for locating popular items (reliable, fast)
- Gnutella is not so good at locating rare items
  - 41% of queries receive <10 results, 18% none at all
  - Queries that return a single result take 73s on average, and for <10 results, first is 50s on average
  - Very often results are not found that actually exist (eg the 18% failure can be reduced to 6%)
- Lots of room (we knew that) and need (this is new info) for improvement for rare items

Hybrid approach

- Inverted index for popular keywords is
  - expensive to compute (many messages to the responsible node)
  - Expensive to use (the distributed join (ie intersection of matches for keywords in query) is expensive)
- For rare keywords all that is cheap
  - We need to identify rare files and rare keywords and publish those to the DHT
  - When a query has no result for some tome (~30s), we ask the DHT
  - Rarity can be determined by seeing a file in a small result set, and by other heuristics

Another kind of hybrid

- Common wisdom
  - Structured overlays are more expensive and less robust to churn and failures
- Is this true?
  - Comparison is very difficult: too many factors, not clear how to be fair
  - But there are indications it is NOT necessarily true
- If it is indeed not true, they are actually (much) better to support “unstructured” search algorithms, such as flooding and random walks

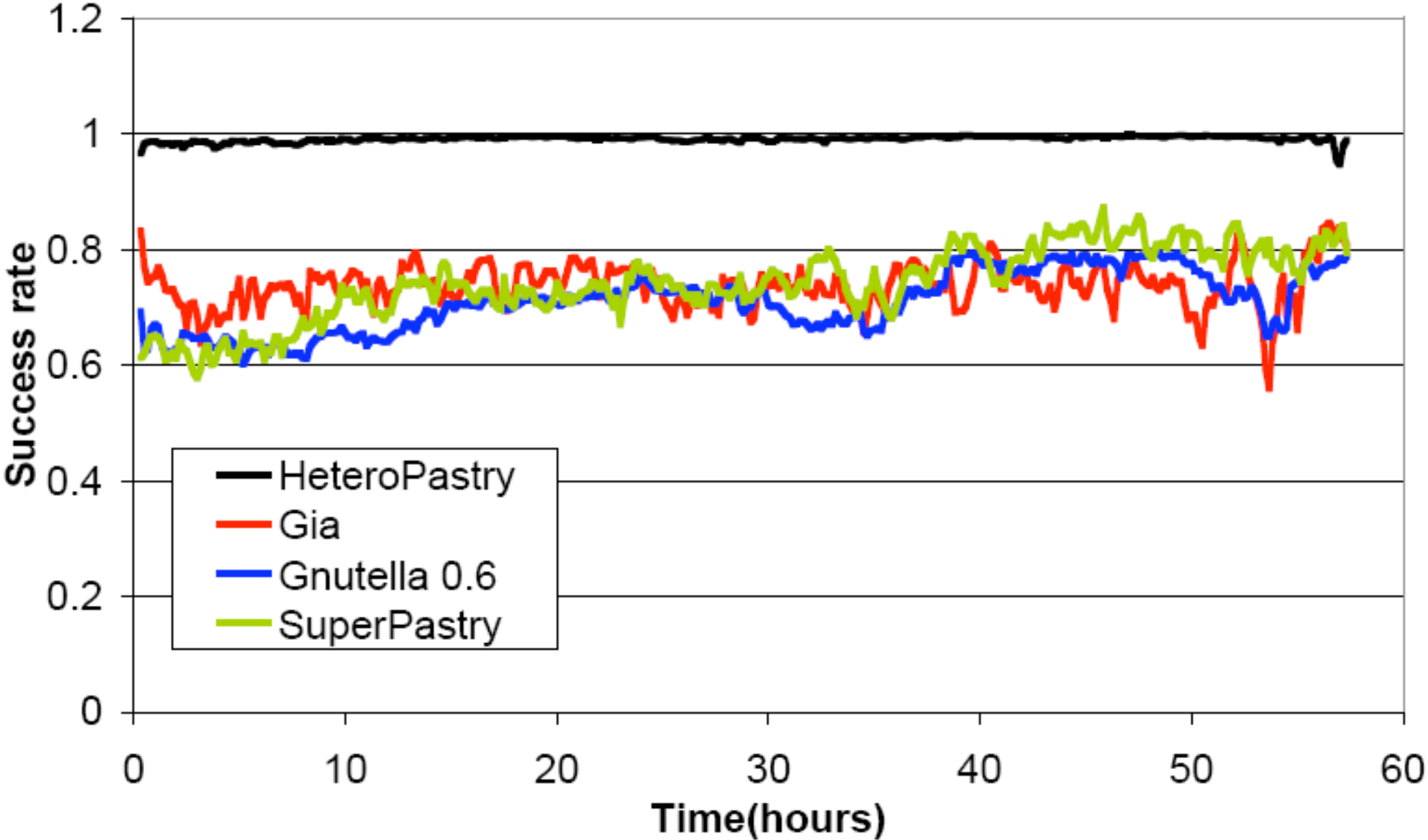
Busting a myth?

- On some real traces maintenance cost of MS Pastry appears to be better than that of Gnutella
  - Heartbeat messages only to one node: the left neighbor in ring (as opposed to gnutella)
- Heterogeneity can also be captured
  - Super Pastry: similar to Gnutella, but ultrapeers form a Pastry network
  - Hetero Pastry: similar to GIA: routing table entries are optimized to prefer high capacity nodes, and a bound on the in-degree can also be set
  - Maintenance overhead is still fine here

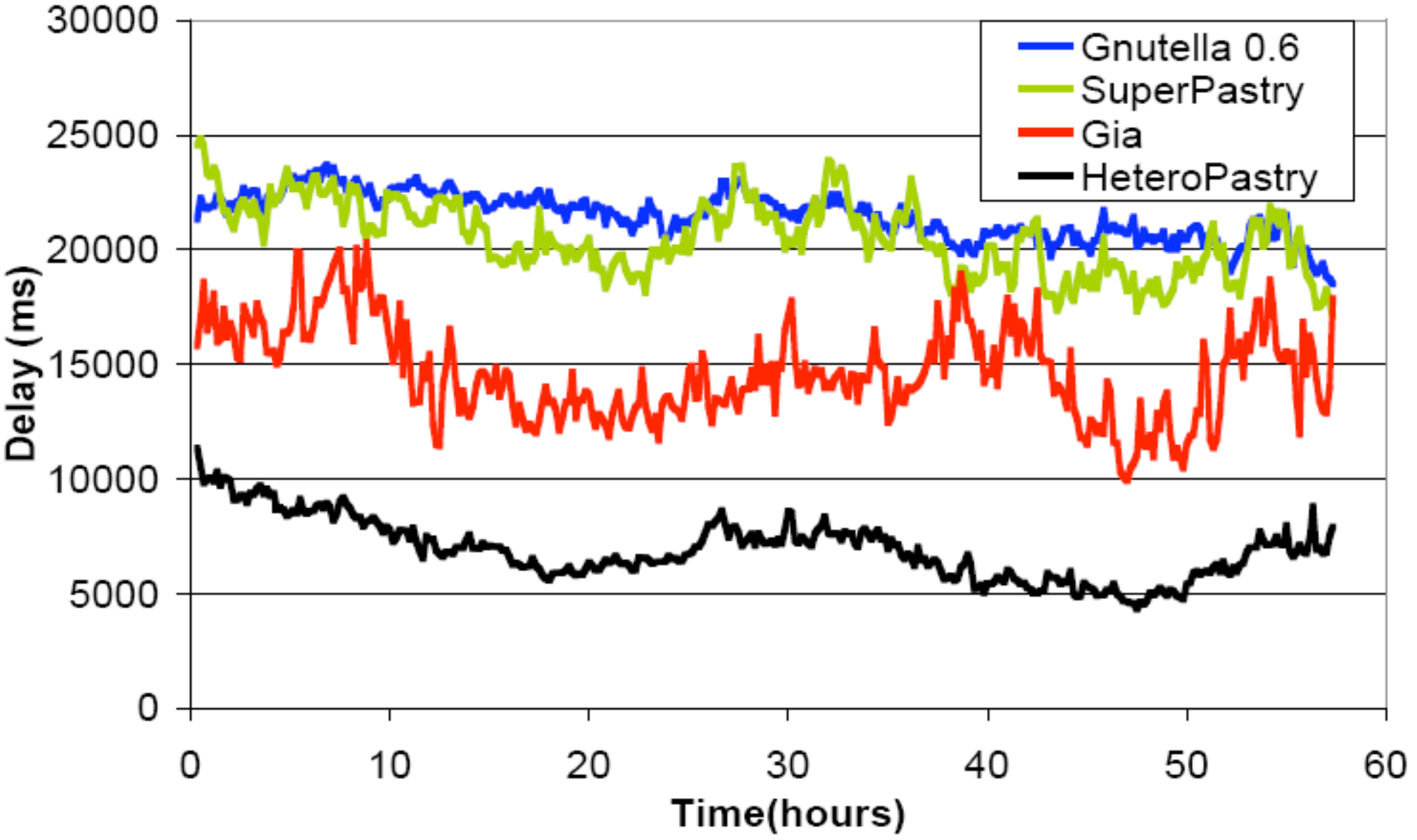
Flooding and random walk in structured networks

- Exploiting the structure of the overlay, broadcast can be optimized to have almost no wasted traffic
- Restricted flooding: a given number of nodes can be visited effectively in parallel
  - Same mechanism for random walk: sequential instead of parallel traversial
- Compare some algorithms
  - using an eDonkey trace
  - max 128 node random walk, one hop replication in all cases (in Pastry, on routing table entries)

Experimental results



Experimental results



Conclusions

- DHTs are an alternative to support search
  - They are very efficient
  - They support key based lookup but
  - They can be adapted to support more complex queries as well
- Restricted flooding and random walk is still better for not-so-rare items
- Hybrid approcahes
  - Use DHT for rare items only
  - Use structured network to support flooding-style queries instead of random network

References

- Miguel Castro, Manuel Costa, and Antony Rowstron. Peer-to-peer overlays: structured, unstructured, or both?. Technical Report MSR-TR-2004-73, Microsoft Research, Cambridge, UK, 2004.
- Boon Thau Loo, Ryan Huebsch, Ion Stoica, and Joseph M. Hellerstein. The case for a hybrid P2P search infrastructure. In Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS'04), San Diego, CA, USA, 2004.
- Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In Middleware 2003, volume 2672 of Lecture Notes in Computer Science, pages 21–40. Springer-Verlag, 2003.
- Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM), pages 161–172, San Diego, CA, 2001. ACM, ACM Press.
- Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In Rachid Guerraoui, editor, Middleware 2001, LNCS 2218, pages 329–350. Springer-Verlag, 2001.
- Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In Proceedings of the 2001 SIGCOMM, pages 149–160, San Diego, CA, 2001. ACM, ACM Press.

Cooperative End-to-End Content Distribution

Content distribution

- So far we looked at search
- Content distribution is about allowing clients (peers) to actually get a file or other data after it has been located
- Different types of content require different techniques
  - Downloading huge files (dvd-s, linux distributions, etc)
  - Streaming media

Content distribution networks

- System organization
  - Centralized
    - **Server farms behind single domain name, load balancing**
  - Dedicated CDN
    - **CDN is an independent system for typically many providers, that clients only download from (use it as a service); typically http**
  - End-to-end (p2p)
    - **special client is needed and clients self-organize to form the system themselves (as usual in p2p)**

Outline

- Large file distribution
  - Dedicated CDN
    - **Akamai: privately owned CDN**
    - **CoralCDN: similar idea to Akamai, only cooperative p2p technology is used**
  - End-to-end p2p CDN
    - **Bittorrent**
    - **The network coding approach**
- Media streaming
  - SplitStream, bullet

Akamai

- Provider (eg CNN, BBC, etc) allows Akamai to handle a subset of its domains (authoritive DNS)
- Http requests for these domains are redirected to nearby proxies using DNS
  - Akamai DNS servers use extensive monitoring info to specify best proxy: adaptive to actual load, outages, etc
- Currently 20,000+ servers worldwide, claimed 10-20% of overall Internet traffic is Akamai
- Wide area of services based on this architecture
  - availability, load balancing, web based applications, etc

CoralCDN: motivation

- Commercial CDN-s are good but expensive: small sites with low bandwidth can't afford them
- Small sites are more vulnerable to flash crowds and any fluctuation of traffic in general
- P2P filesharing has shown willingness to provide bandwidth for popular content
- Let's build a P2P CDN to support (popular but) small, underprovisioned websites
- [motivation is shaky, but interesting technology anyway]

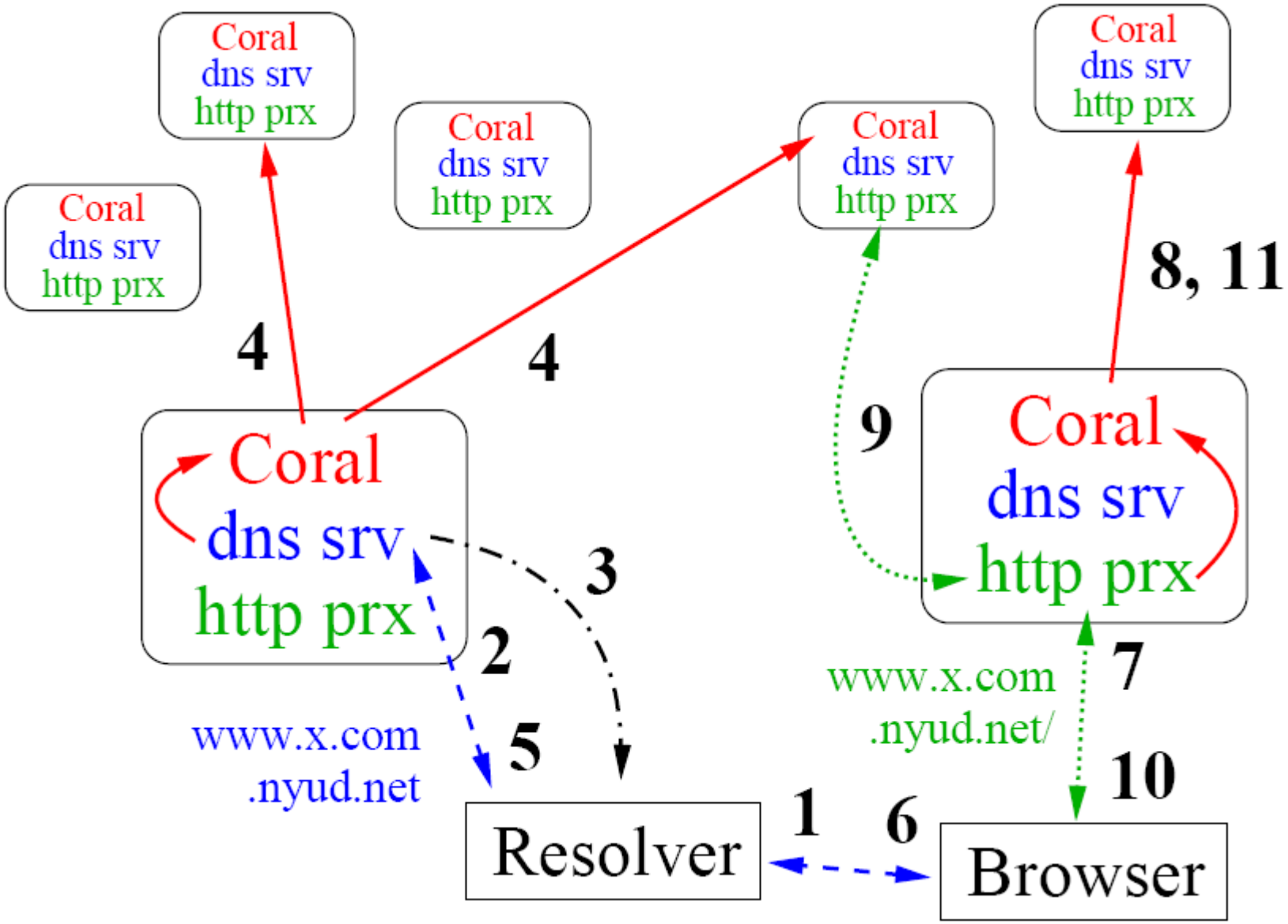
CoralCDN

- Participating peers form
  - an indexing infrastucture (DHT-like)
  - a network of HTTP proxies
  - a network of DNS servers
- How to use the system?
  - Publishers can “coralize” urls by appending the domain “.nyud.net:8090” to the name of the server, eg http://www.inf.u-szeged.hu.nyud.net:8090/
  - In email, usenet, etc messages any url-s can be coralized the same way (thereby preventing the “slashdotting” of the site in question)

How it works

- Clients tries to resolve www.inf.u-szeged.hu.nyud.net
- Coral DNS server probes the client for RTT and looks for coral DNS and HTTP servers nearby
- Coral DNS returns DNS and HTTP servers for www.inf.u-szeged.hu.nyud.net
- Clients send HTTP request to http://www.inf.u-szeged.hu.nyud.net:8090/
- If given coral serever has the page, sends it.
- Otherwise looks up the URL in Coral, and if it is available, caches it from within Coral, and sends it
- Otherwise it fetches the page from original location http://www.inf.u-szeged.hu/
- The coral server notifies the system that it now caches the URL

Overview of CoralCDN



Distributed sloppy hash table

- Sloppy: keys can be stored not only on nodes that are the closest, but also in nodes that are close enough: better load balancing
- Inserting a key
  - Approach the reposnsible node through routing as in DHT, but stop sooner, if nodes that are close are “full” and “loaded” (load balancing technique)
- Retrieval
  - Approach the responsible node for the key, and stop when finding the first node storing the key

Clustering

- Many DSHT-s in parallel: hierarchical clustering
  - 3 levels: according to RTT among cluster members
  - All nodes have same ID in all clusters, level 0 cluster covers full network
- Implementation of clustering
  - Storing hints in the DSHT: key: IP address of router and subnet prefix: value: node
    - **Joining nodes quickly find other nodes in the same subnets**
  - Collect RTT info in all contacts: if other cluster seems closer, change cluster

Exploiting clustering

- Retrieval is biased towards lower levels, so nearby HTTP and DNS servers can be located
  - Start routing protocol at level 2 (closest nodes)
  - If no key found, go to level 1 and simply continue the routing (the nodes level 2 cluster is subset of its level 1 cluster)
  - Go until reaching level 0
- Clusters do not increase lookup time (roughly the same as a simple routing at level 0)

Some notes

- CoralCDN is deployed on PlanetLab
  - 750 nodes
  - 1,500 Gbytes for 700,000 IP addresses a day
- It is only a proof of concept, but wide scale deployment is a question
  - If it is single administrative domain, why not more control, why the p2p approach?
  - If it is voluntary, multiple admin domain, who would want to join voluntarily without restricting content? What kind of content? Etc
  - DNS and other overhead makes it rather slow

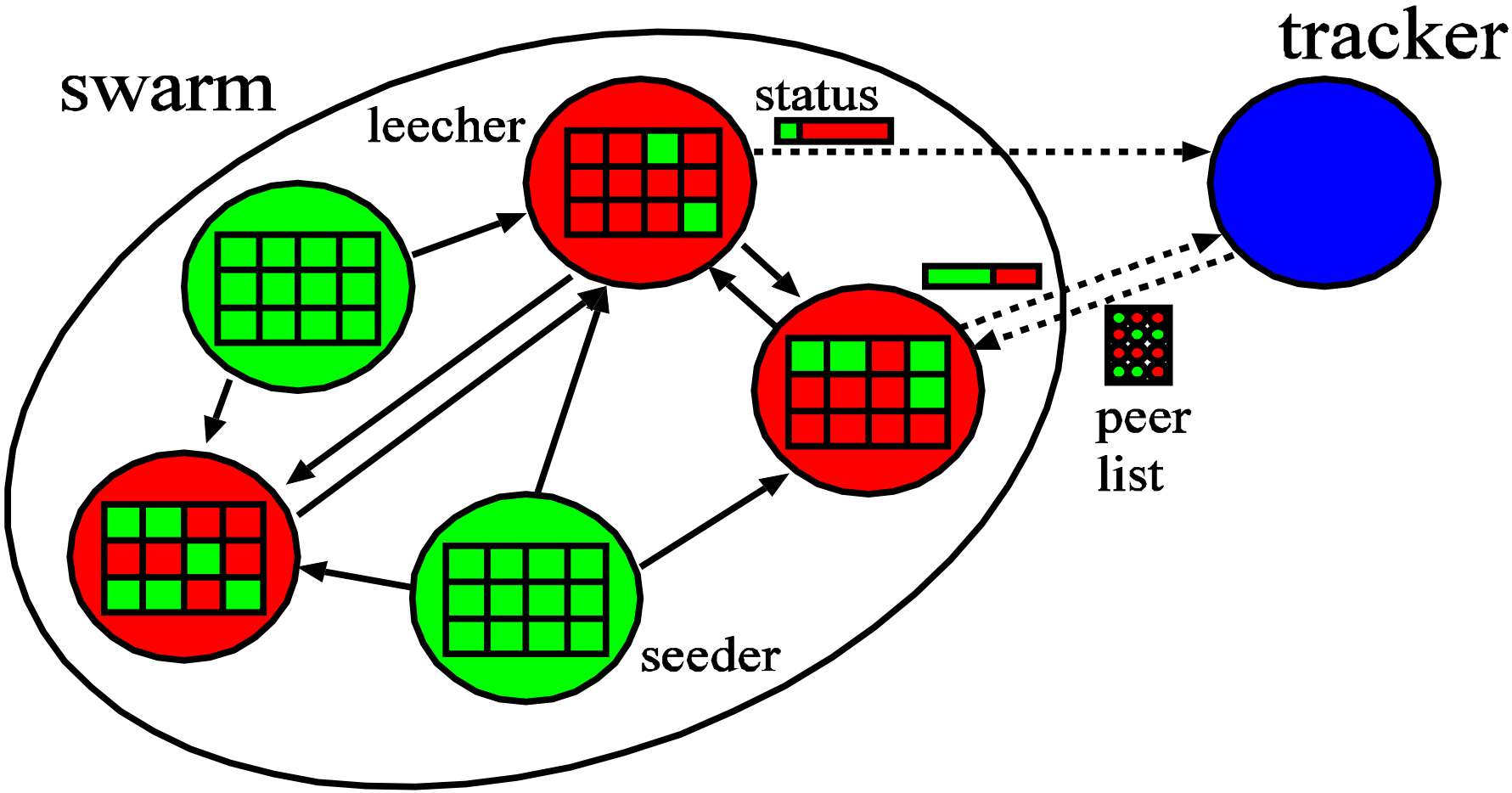
End-to-end P2P CDN: BitTorrent

- Invented by Bram Cohen
- Currently 20-50% of Internet traffic is BitTorrent
- Special client software is needed
- Basic idea
  - Clients that download a file at the same time help each other (ie, also upload chunks to each other)
  - BitTorrent clients form a swarm: a random overlay network

BitTorrent

- Publishing a file
  - Put a ".torrent" file on the web: it contains the address of the tracker, and information about the published file: eg chunk hashes (256M chunks)
  - Start a tracker, a server that
    - **Gives joining downloaders random peers to download from and to**
    - **Collects statistics about the swarm**
  - [Note that there are "trackerless" implementations already]
- Download a file
  - Install a bittorrent client and click on a ".torrent" file

BitTorrent overview



BitTorrent client

- Client first asks 50 random peers from tracker
  - Also learns about what chunks they have
- Picks a chunk and tries to download its pieces (16K) from the neighbors that have them
  - Download does not work if neighbor is disconnected or denies download (choking)
- Allows only 4 neighbors to download (unchoked neighbors)
  - Periodically (30s) does optimistic unchoke: allows download to random peer (important for bootstrapping and optimization (exploration))
  - Otherwise unchokes peer that allows the most download (each 10s)

tit-for-tat

- Tit-for-tat in iterated prisoners dilemma
  - Cooperate first, then do what the opponent did in the previous game
  - Very good strategy (Axelrod)
- BitTorrent is a kind of tit-for-tat
  - We unchoke peers (allow them to download) that allowed us to download from them
  - Optimistic unchoking is the initial cooperation step to bootstrap the thing
- How about hacked clients? Why don't they spread and kill BitTorrent?

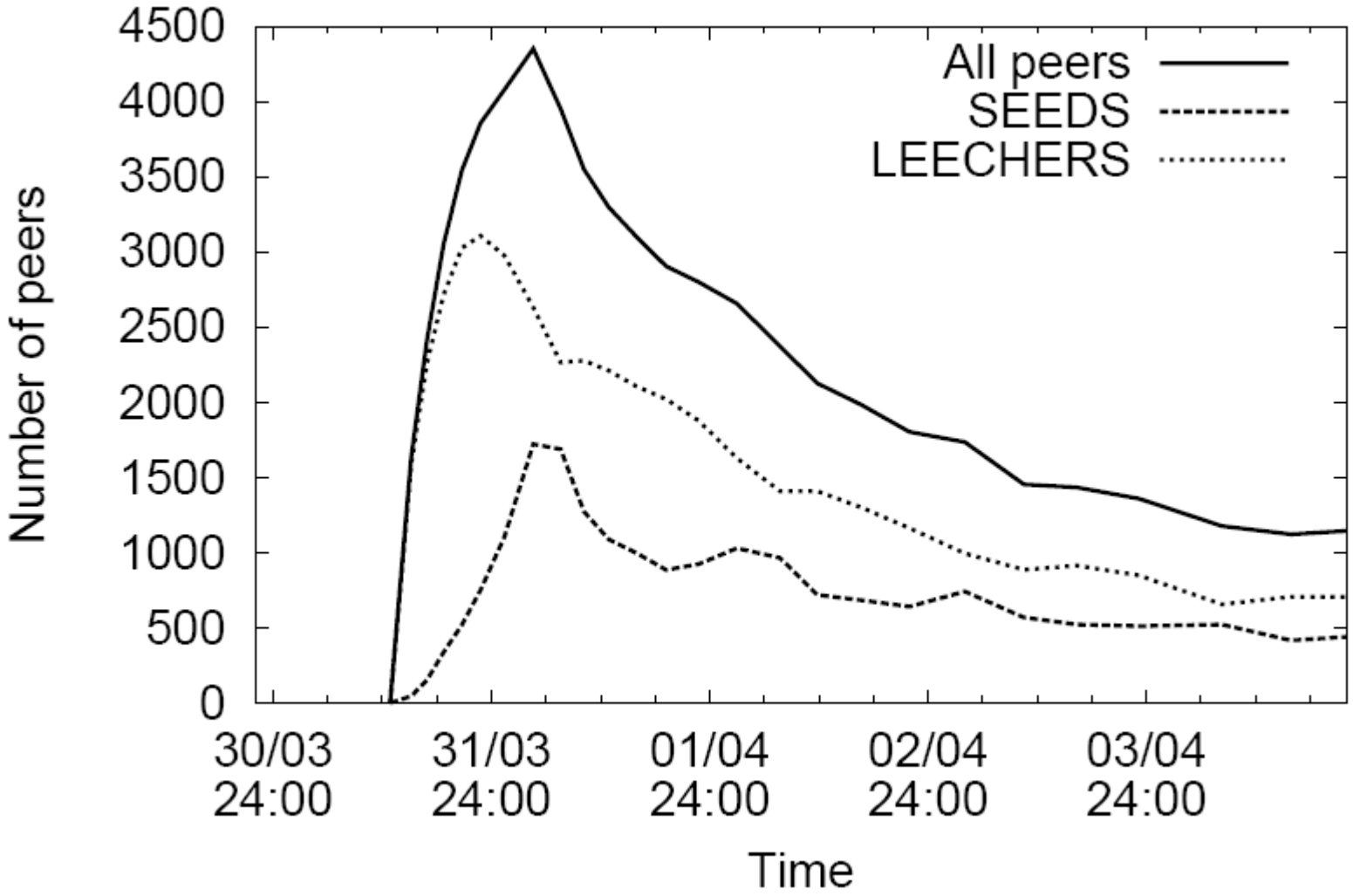
Chunk selection

- Another very important question is what chunk to select to download?
- Clients select the chunk that is rarest among the neighbors (local decision)
  - Keeps all chunks equally represented
  - This is good because no chunks get lost, and it is likely that peers find chunks they don't have
- Exception is first chunk
  - Select a random one (to make it fast: many neighbors must have it)

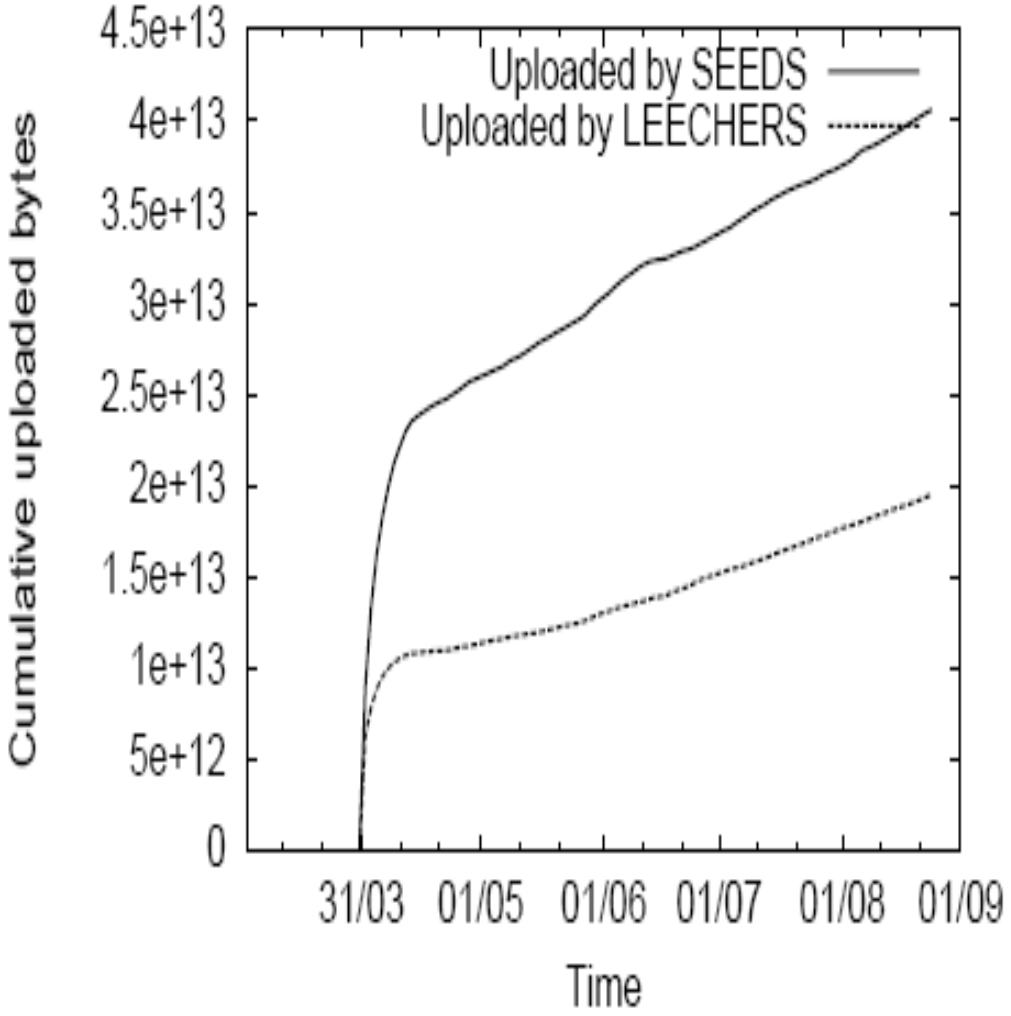
Measurements

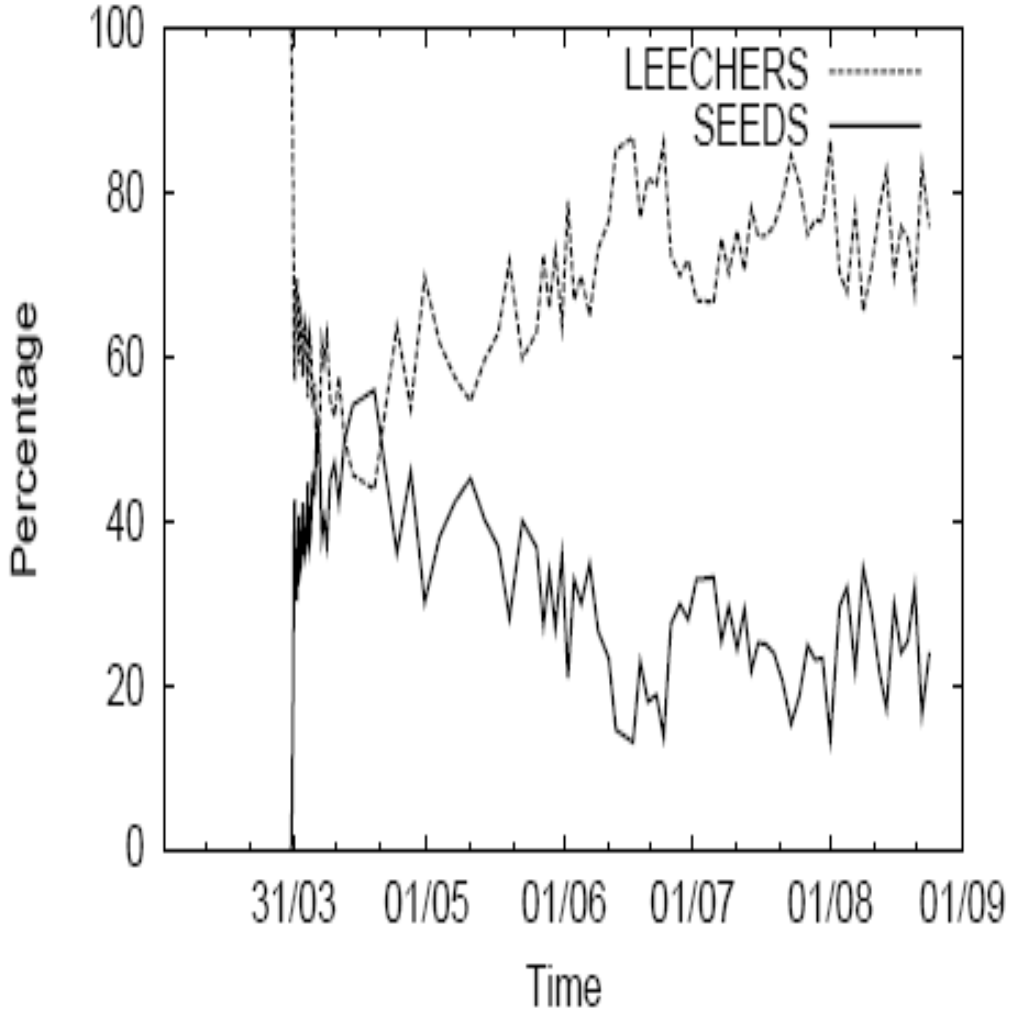
- 5 month trace of the 1.77GB RedHat ISO image
- Two sources of data
  - Tracker statistics
  - Modified client participating in the swarm
- 180,000 clients total
- 50,000 clients in the first five days
  - Flash crowd

Initial flash crowd



Seeds and leechers: altruism





Some statistics

- Average download rate is 500kb/s, during flash crowd, active clients averged at 800kb/s
- 5% of sessions is “seed session”
  - Joining peer already has to complete file, joins only to share it
- About 50% of sessions (peer joins) belong to peers that spend little time in the network and down/upload little data
  - Maybe disappointed users behind slow links

Summary

- BitTorrent: simple (by design and also to use), almost optimal and works $\rightarrow$ it is popular
- The devil is in the details too (good efficient client)
- Only slight problem: endgame
  - Last chunks in endgame mode: aggressive parallel downloads to maximize speed
  - Does not result in very significant overhead

Network coding

- In bittorrent: chunk selection and peer selection are important to make sure that
  - All chunks are represented equally
  - We have a random network
- We can get rid of these using coding theory
  - Works even if overlay has bottlenecks
  - No need to worry about chunk selection

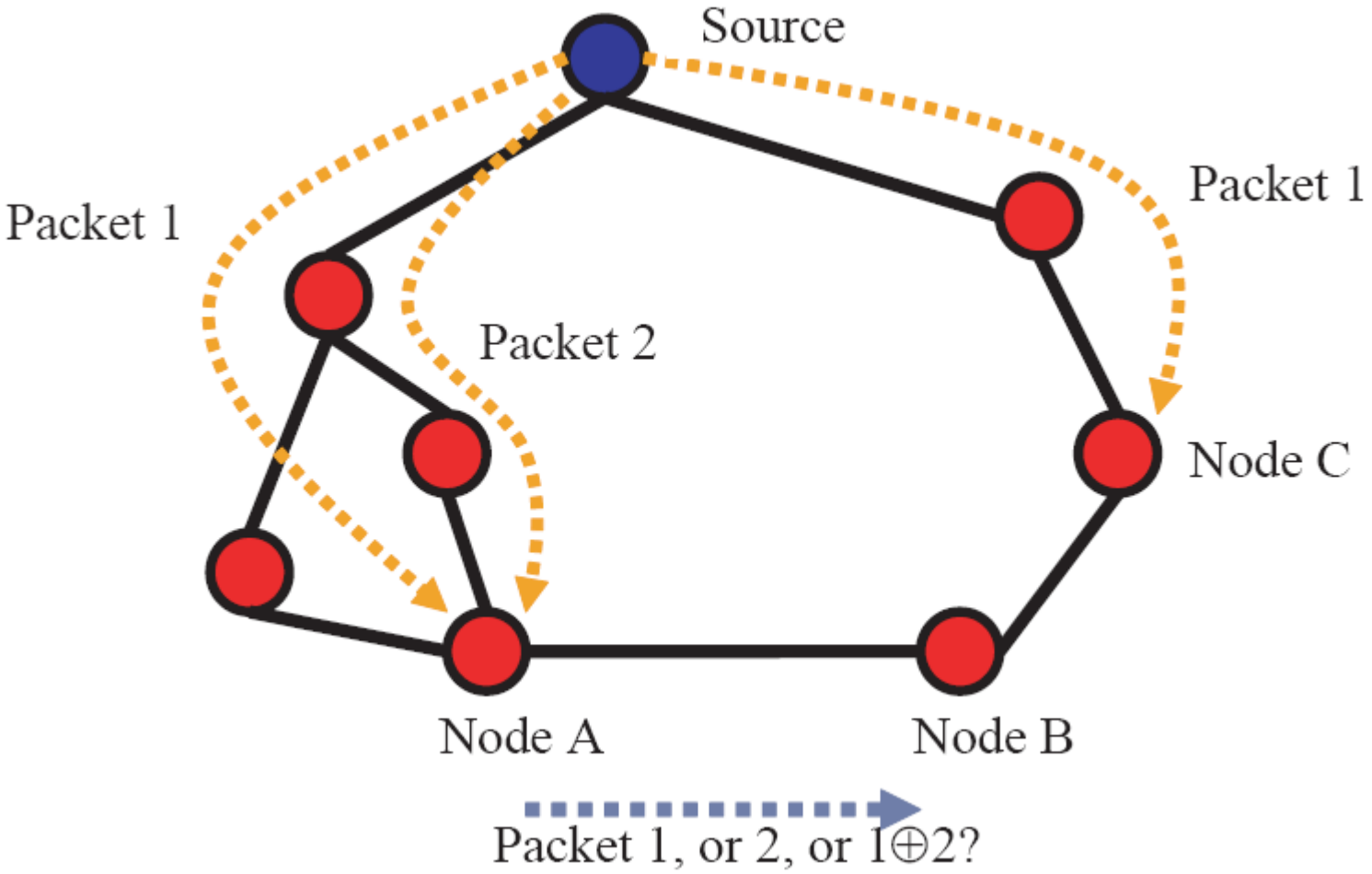
Coding theory for CDNs

- Erasure codes
  - Data is divided into k packets
  - Transformed into n>k packets such that any k packets can reconstruct the original data (erasure codes)
  - Reed-Solomon or Tornado codes
- Implementing a digital fountain
  - “fountain” keeps transmitting these n packets
  - Downloaders can join at any time, can catch any k of the packets (perhaps from neighbors) and leave

Network coding

- Not only server does encoding but also the clients
- A huge, practically unlimited number of different packets are floating around, generated by clients concurrently
- Any k of these packets is enough for decoding
- Possible coding approach: linear combination over finite fields
  - All codes are linear combinations of the original packets
  - Clients create new linear combinations when they offer content
  - Decoding is solving a linear system of equations

Advantage of network coding



Only the number of packets counts, no worries about which packet to fetch

A possible protocol

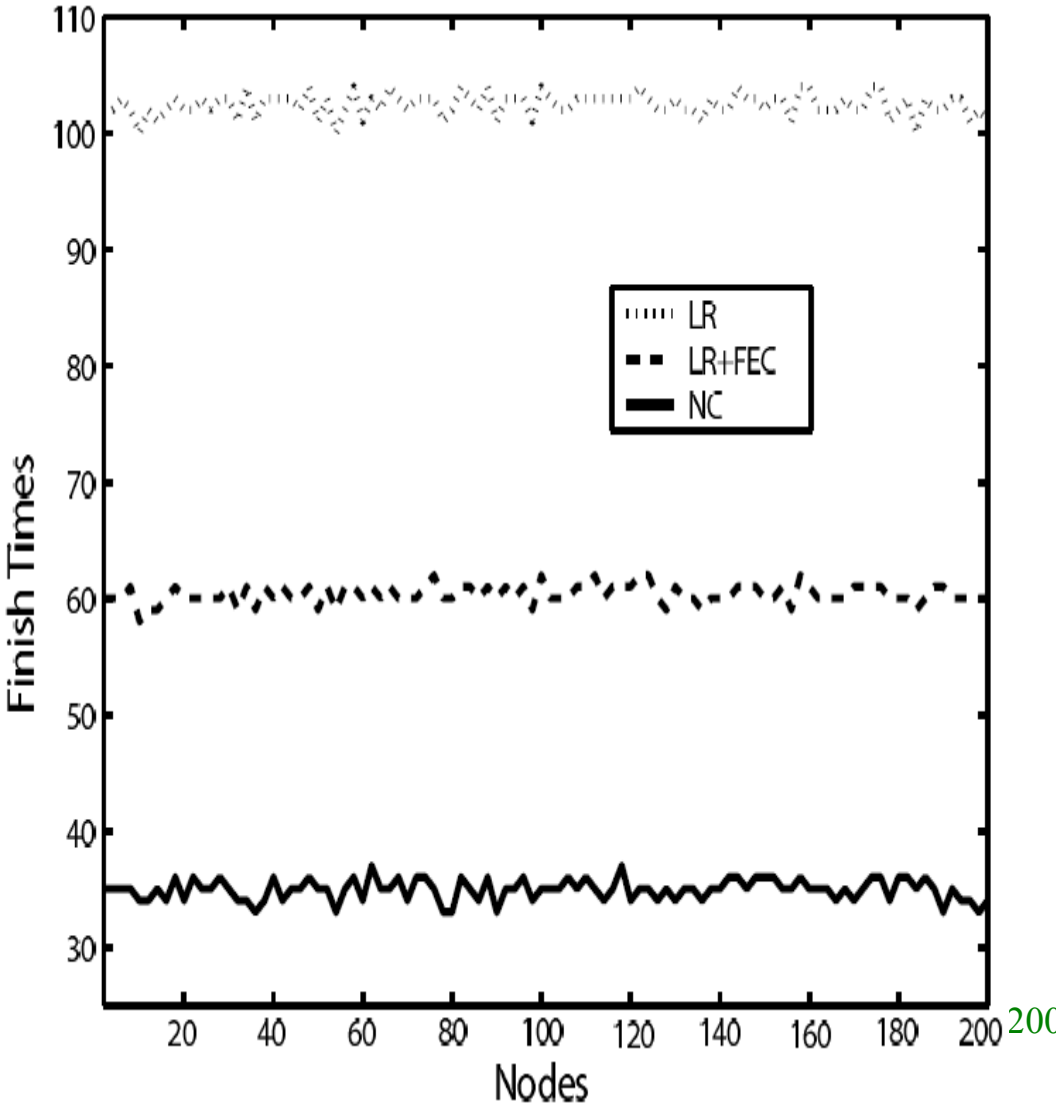
- Same as BitTorrent, only
  - Clients offer new random linear combinations for download and transfer the coefficients as well (low overhead)
  - There is no chunk selection problem
    - **No rare chunks can occur**
    - **No endgame problem**
    - **No topology bottleneck problem**
    - **No data loss problem due to catastrophic failure**
- Same incentive mechanisms too (tit-for-tat), but with explicit accounting (no more upload than download)

Experimental results

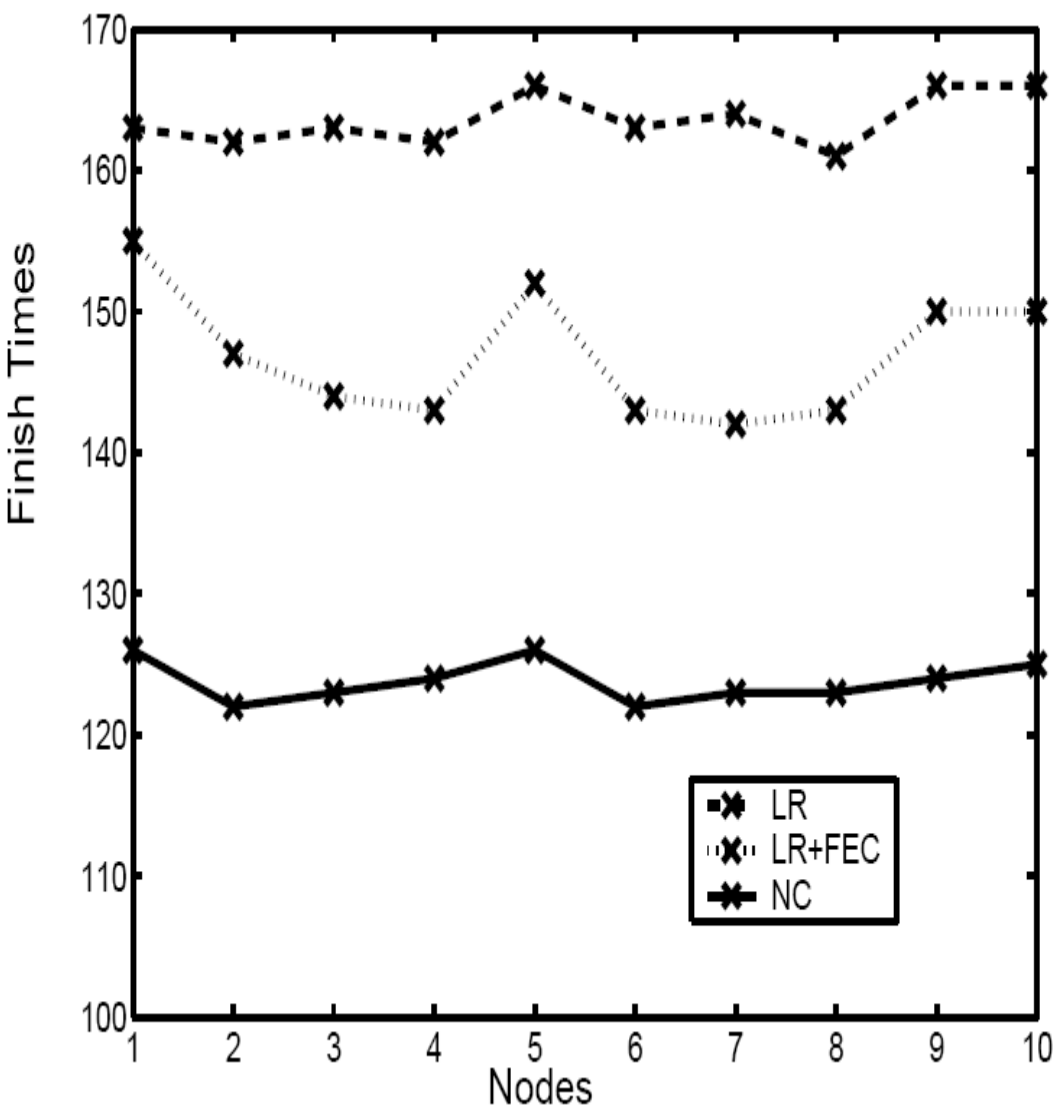
- Three algorithms
  - Local rarest chunk selection (LR) (similar to BitTorrent)
  - Local rarest combined with server encoding
  - Network encoding
- Network size is 200 (small!)
- Neighbors is max 6 (small!)
- Different scenarios
  - Clustered topology, heterogeneity, dynamism (If random network and homogeneous static peers, then the strategies are very similar)

Clustering and heterogeneity

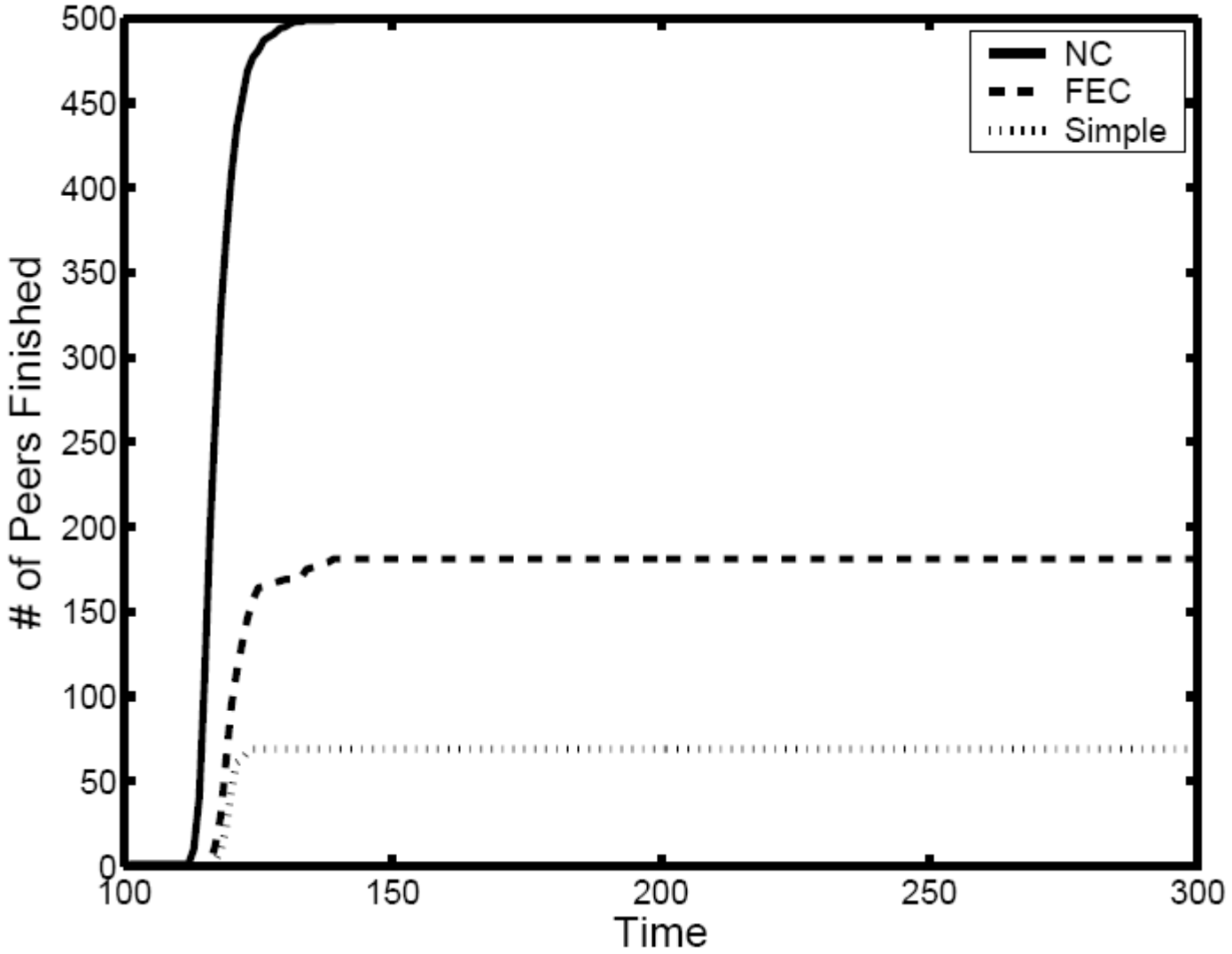
Two clusters: 100 nodes each



10 fast nodes (4x faster) 190 slow nodes



Server availability



Server leaves after serving all chunks plus 5% extra chunks, nodes immediately leave when finished

server coding needs 10-15%, no coding 20-30% extra chunks to achieve full coverage

Final note

- BitTorrent is in fact quite good in practice
  - No network bottlenecks occur because a random network is maintained
  - Rarest chunk policy is very good (combined with initial random chunk and end-game strategies)
  - Heterogeneity might be an issue (in practice low capacity nodes simply go away, as we saw...)
- A convincing study is still to be written with larger scale systems and a more complete BT implementation

Media streaming

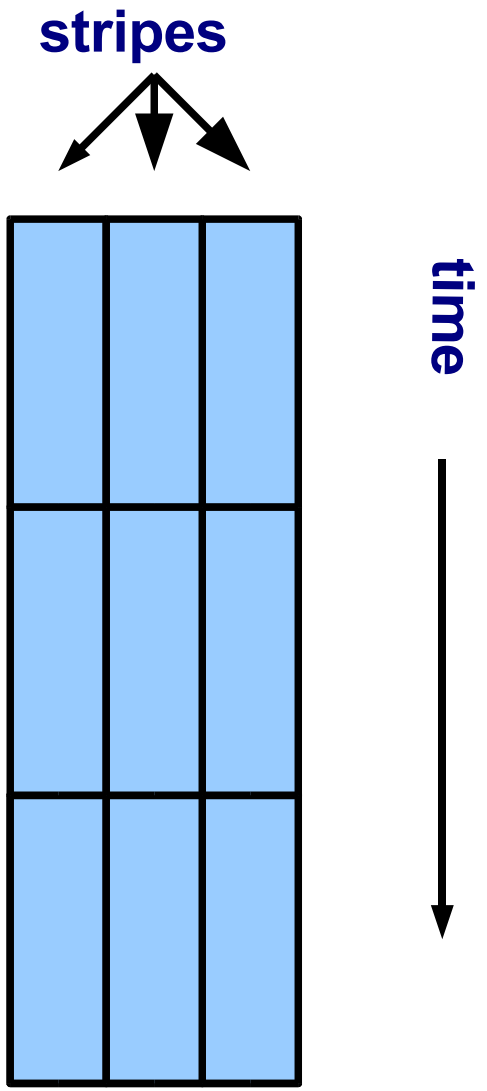
- Similar to distribution of large files but time is important
  - Packets must have low delay
  - If we do not get a packet for some time, we forget about it
- Classification as before
  - Dedicated router infrastructure (Cisco, etc)
  - Dedicated application layer overlay (Akamai, etc)
  - P2P cooperative approaches
- We look at SplitStream and Bullet (both P2P)

Multiple description coding

- We have seen erasure codes for large file distribution
  - Here any k packets were enough for decoding, but k-1 packets is of not much help
- Multiple description code (MDC) is similar
  - k packets are enough for decoding
  - Less than k packets can be used to approximate content
    - **Similar to progressive encoding, only order of packets is insignificant**
    - **Useful for multimedia (video, audio) but not for other data**

Multiple description coding

- Media streaming applications often use MDC in some form because
  - Loosing a packet results in no interruption, only quality degradation
  - Lower bandwidth nodes simply ask for < k packets
- Streams can be sliced into parallel “stripes” that are MDC encoded



Trees: not optimal

- The most natural way of cooperative media streaming is through broadcast trees
- Trees have problems though, esp in end-to-end approaches
  - Vulnerable to failure (no cycles)
  - Bandwidth strictly decreases towards leaves
  - Difficult to create optimal tree (and it is important to do so)
  - Leaves do not contribute in a cooperative setting

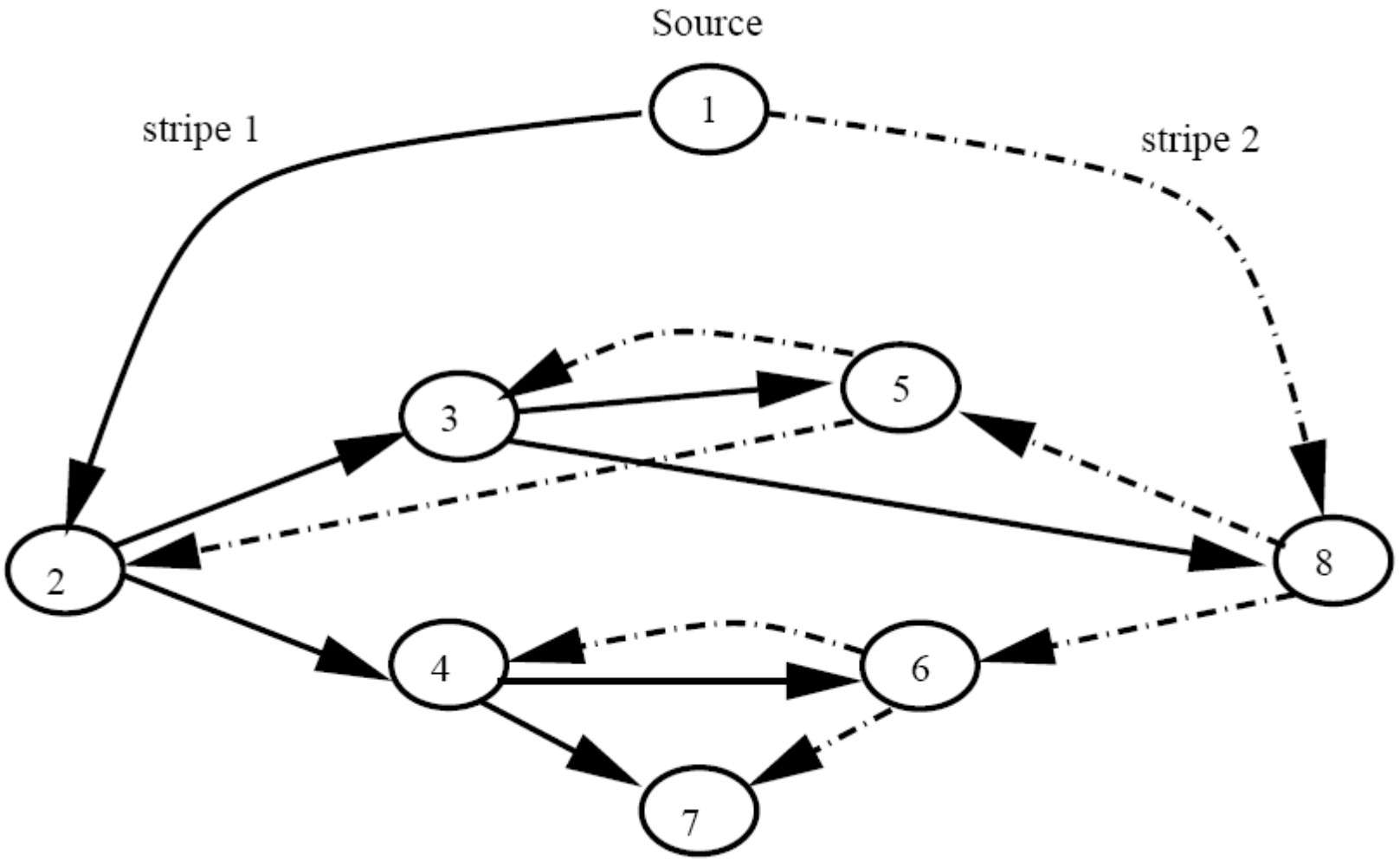
Solving the problems with trees

- Use multiple trees
- Use a tree but also use a mesh for cooperation
- Axe them (Chainsaw, IPTPS 2005)
  - We do not discuss this here, although remarkable
- In the following we look at
  - SplitStream that uses multiple trees
  - Bullet that uses the union of a mesh and a tree

SplitStream

- Basic idea
  - Split the stream into k stripes (perhaps with MDC encoding)
  - For each stripe create a multicast tree such that the forrest
    - **Contains interior-node-disjoint trees**
    - **Respects nodes' individual bandwidth constraints**
- Approach
  - Use Scribe (and some hacks) to create the forrest
  - Scribe is on top of Pastry

Illustration of SplitStream



The forrest construction problem

- A constraint satisfaction problem
  - All nodes have incoming capacity requirements (number of stripes they need) and outgoing capacity limits
  - There is one or more source for each stripe
  - We have to construct a weighted directed acyclic distribution graph (forrest) that respects these constraints
- An observation: such a forrest exists if
  - Sum of incoming capacity is less then or equal to the sum of outgoing capacity over the nodes and
  - All nodes that have large outgoing than incoming capacities must posess (receive or originate) all stripes

Constructing the forrest: scribe

- Scribe works over Pastry
  - Mutlicast groups are identified by an ID
  - Tree is definied by the route towards the ID in the Pastry network
  - Join: route towards the ID, connect to first member as child
- Basic idea
  - All k stripes are assigned a group ID, and Scribe is used to create mutlicast trees
  - This does not necessarily satisfy constraints

Constructing the forrest

- Additional tricks for constraint satisfaction
  - Group IDs start with a different letter: interior-node-disjoint forrest
  - If a node has too many children
    - **"Push-down" approach: joining node looks for a parent further down the tree, or if not found, in the "spare capacity group"**
  - Spare capacity group
    - **Scribe group that contains nodes that can take more children**
- Algorithm always succeeds if all nodes want to receive all stripes or suceeds with a high probability as a function of spare capacity and minimal incoming capacity

Bullet

- Basic idea
  - Use a multicast tree as a basis
  - In addition each node continuously looks for peers to download from
  - In effect, the overlay is a tree combined with a random network
- Approach
  - A service (ranSub) that provides random peers
  - A mechanism to select “good” peers
  - Low level transfer protocol (to replace TCP)

Bullet: RanSub

- Two phases
  - Collect phase: using the tree, membership info is propagated upwards (random sample and subtree size)
  - Distribution phase: moving down the tree, all nodes are provided with a random sample from the entire tree, or from the non-descendant part of the tree, etc.
- Nodes in the network receive random peers this way end select those that seem to be most useful

Bullet

- When selecting a peer, first a similarity measure is calculated
  - Based on “summary-sketches”
- Before exchange missing packets need to be identified
  - Bloom filter of available packets is exchanged (usual false positive issue)
  - Old packets are removed from the filter (to keep the size of the set constant)
- Periodically re-evaluate senders (how useful they are)
  - If needed, senders are dropped and new ones are requested

Some comments

- Tree is needed
  - Because of RanSub: but other sampling services can be used that do not rely on trees
  - To maximize diversity of packets in the network: but rarest first chunk selection in BT does the same, besides, with encoding techniques, it is irrelevant
- So is the tree needed?
- Isn't the protocol unnecessarily complex trying to explicitly control things that are “for free” in simpler approaches?
  - Eg in BT through the local-rarest-first strategy

References

- C Gkantsidis and P R Rodriguez. Network coding for large scale content distribution. In INFOCOM 2005, pp 2235–2245, 2005.
- M Freedman, E Freudenthal, and D Mazières. Democratizing content publication with Coral. In NSDI '04, 2004.
- M. Izal, G. Urvoy-Keller, E.W. Biersack, P.A. Felber, A. Al Hamra, and L. Garcés-Erice. Dissecting bittorrent: Five months in a torrent's lifetime. In Passive and Active Network Measurement, LNCS 3015, pages 1–11. Springer, 2004.
- M Castro, P Druschel, A-M Kermarrec, A Nandi, A Rowstron, and A Singh. Splitstream: high-bandwidth multicast in cooperative environments. In SOSP'03, pages 298–313, New York, NY, USA, 2003
- B Cohen. Incentives build robustness in bittorrent. In P2PECON, 2003.
- D Kostic, A Rodriguez, J Albrecht, and A Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In SOSP'03, pages 282–297

Gossip Algorithms

Introduction

- Gossip-like phenomena are commonplace
  - human gossip
  - epidemics (virus spreading, etc)
  - computer epidemics (malicious agents: worms, viruses, etc)
  - phenomena such as forest fires, branching processes and diffusion are all similar mathematically
- In computer science, epidemics are relevant
  - for security (against worms and viruses)
  - for designing useful protocols (we look at this here)

Introduction

- originally for information dissemination in a very simple but efficient and reliable way
- later the term has been extended to many local probabilistic and periodic protocols
- we will introduce a simple common skeleton and look at
  - information dissemination
  - topology construction
  - aggregation

Introduction

- the push-pull model is sown
- the active thread initiates communication (push) and receives peer state (pull)
- the passive thread mirrors this behavior

```
do once in each T time units at
a random time
    p = selectPeer()
    send state to p
    receive state_p from p
    state = update(state_p)
```

active thread

```
do forever
    receive state_p from p
    send state to p
    state = update(state_p)
```

passive thread

Information dissemination (broadcast)

- state: set of updates
- selectPeer: a random peer from the network
  - very important component, we get back to this soon
- update: add the received updates to the local set of updates
- some notes
  - implementations take care of details to optimize bandwidth usage (check which updates are needed, etc)
  - propagation of one given update can be limited (max k times or with some probability, etc)

Performance of gossip

- various mathematical results are available
  - epidemiological models (virus spreading)
  - percolation theory, complex networks, etc
- underlying network (that is, the implementation of selectPeer) plays a key role
- in a random network
  - push-pull gossip spreads approximately exponentially fast
  - gossip (that is, random networks...) is extremely robust to benign failure (node failure and link failure)

References

- Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87), pages 1–12, Vancouver, British Columbia, Canada, August 1987. ACM Press.
- Eugster, P. T., Guerraoui, R., Kermarrec, A.-M., and Massoulie, L. 2004. Epidemic information dissemination in distributed systems. IEEE Computer 37, 5 (May), 60–67.
- A.-M. Kermarrec and M. van Steen, editors, Special issue of ACM SIGOPS Operating Systems Review on Gossip Protocols, (probably 2007, in press).

Peer Sampling

- A key method is selectPeer in all gossip protocols (determines performance and reliability)
- In earliest works all nodes had a global view to select a random peer from
  - scalability and dynamism problems
- Scalable solutions are available to deal with this
  - random walks on fixed overlay networks
  - dynamic random networks

Random walks on networks

- if we are given any fixed network, we can sample the nodes with any arbitrary distribution with the Metropolis algorithm:

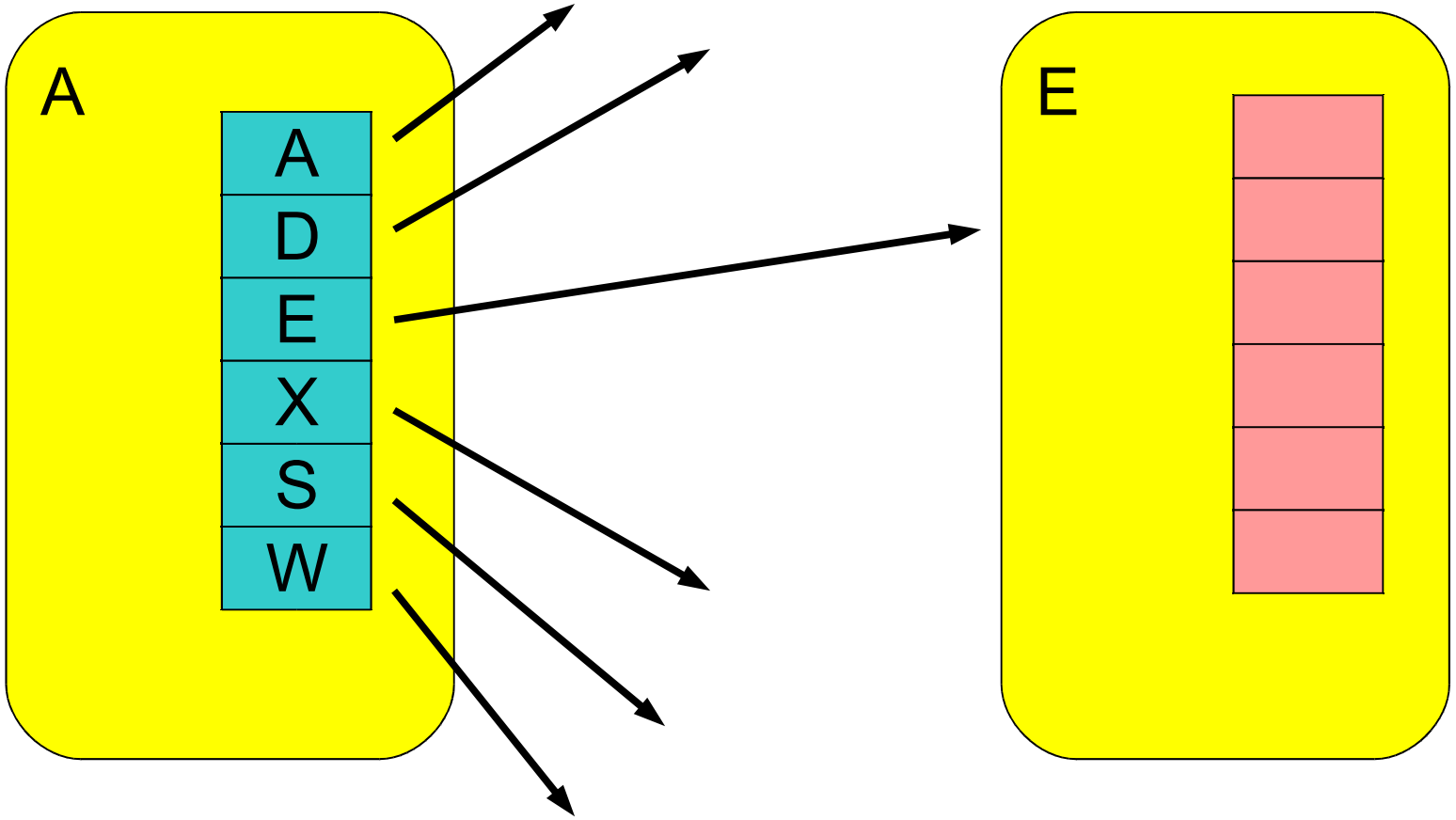
$$P_{i,j} = \begin{cases} \frac{1}{2} \cdot \frac{1}{d_i} & \text{if } \frac{\pi(i)}{d_i} \leq \frac{\pi(j)}{d_j}; \\ \frac{1}{2} \cdot \frac{1}{d_j} \cdot \frac{\pi(j)}{\pi(i)} & \text{if } \frac{\pi(i)}{d_i} > \frac{\pi(j)}{d_j}. \end{cases}$$

- This Markov chain has stationary distribution π where d_i is the degree of node i (undirected graph)

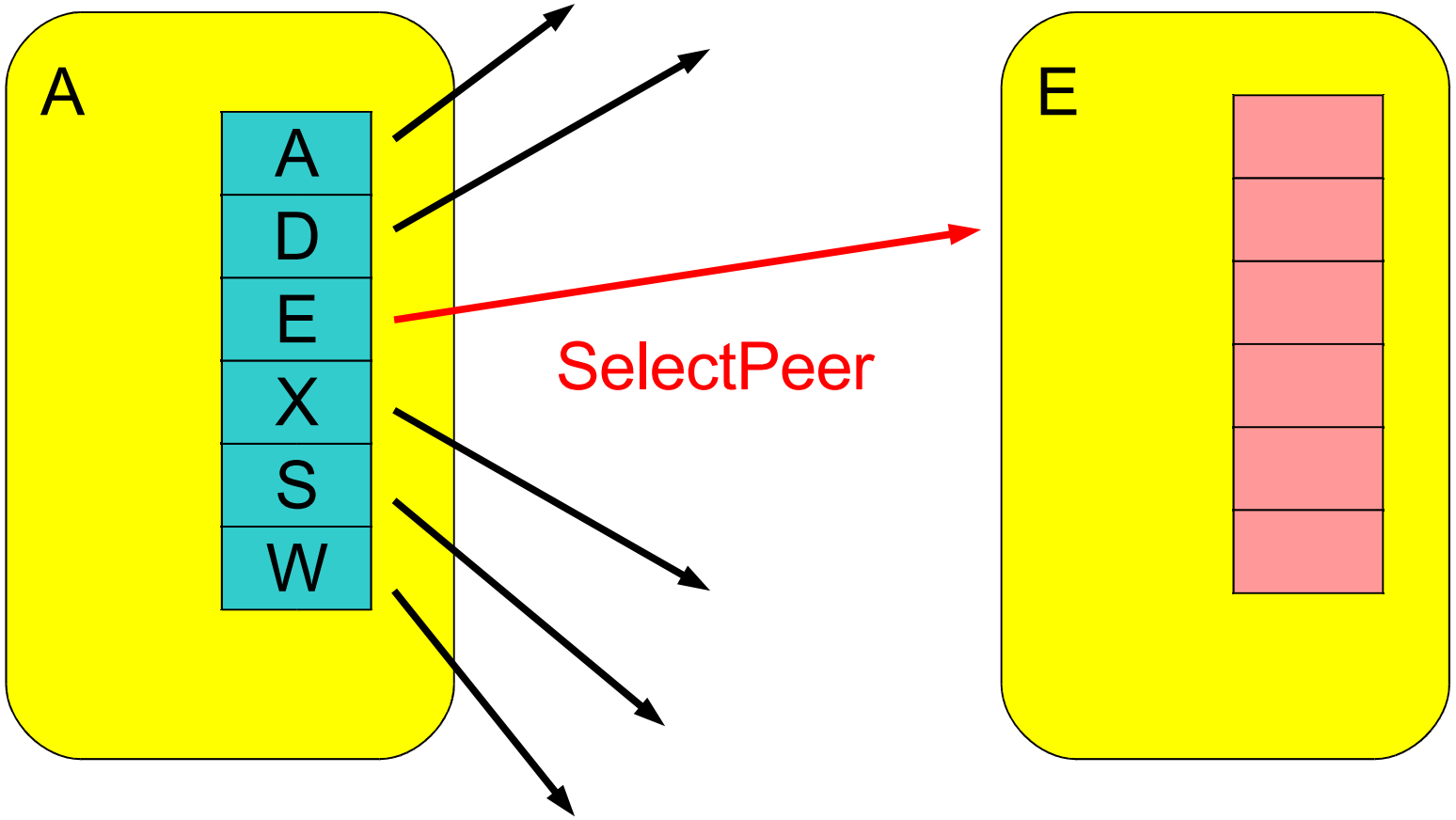
Gossip based peer sampling

- basic idea: random peer samples are provided by a gossip algorithm: the peer sampling service
- The peer sampling service uses itself as peer sampling service (bootstrapping)
  - no need for fixed (external) network
- state: a set of random overlay links to peers
- selectPeer: select a peer from the known set of random peers
- update: for example, keep a random subset of the union of the received and the old link set

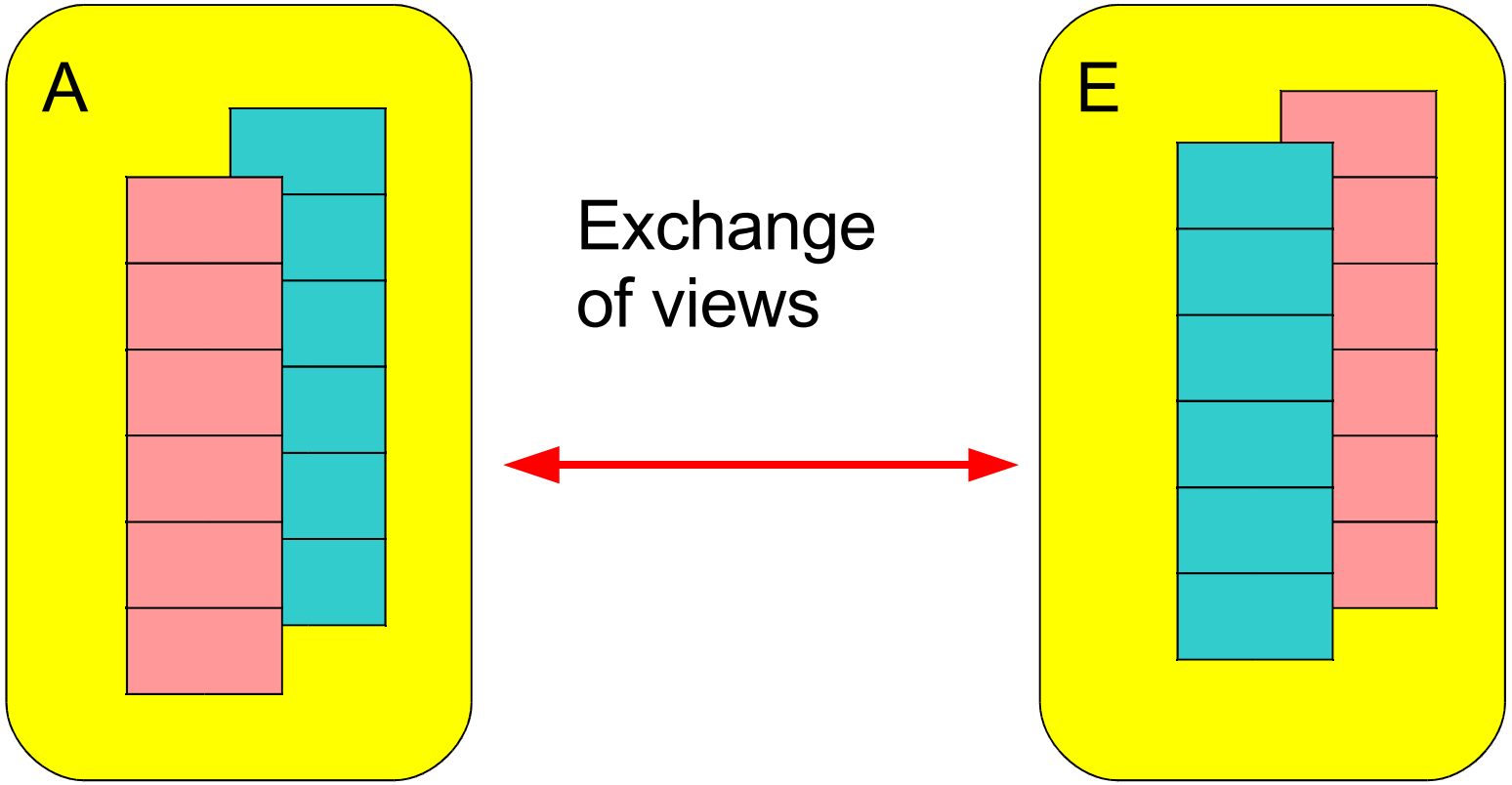
Gossip protocols for topology management



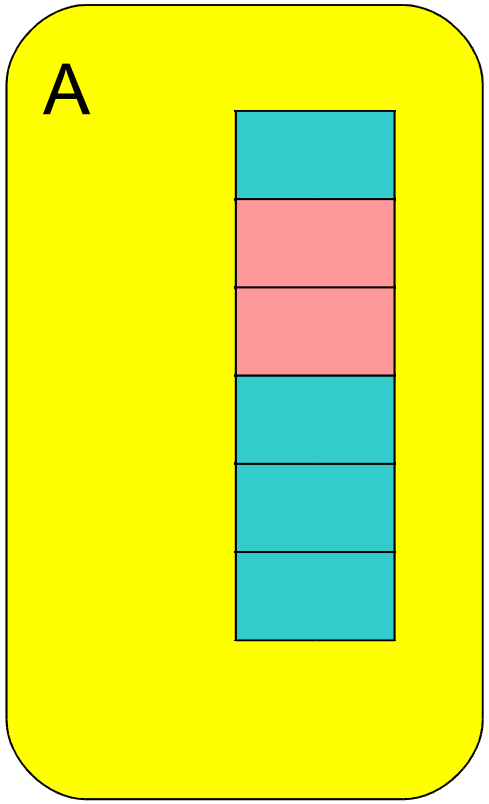
Gossip protocols for topology management



Gossip protocols for topology management

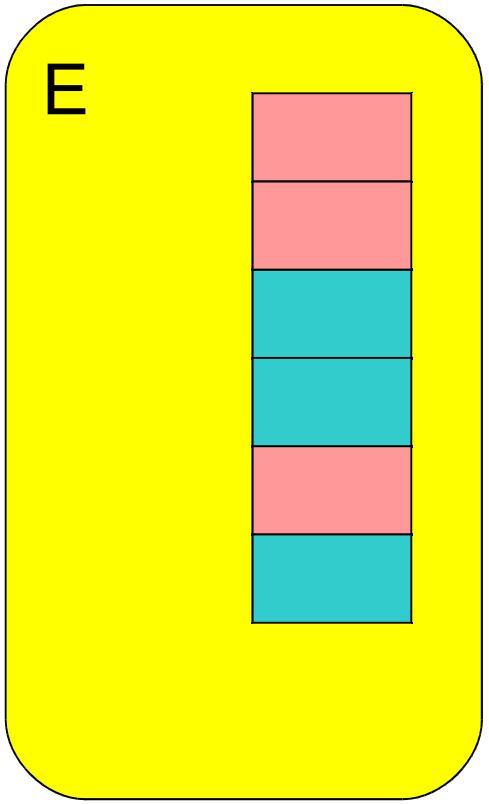


Gossip protocols for topology management



Both sides apply update

thereby redefining topology



Gossip based peer sampling

- in reality a huge number of variations exist
  - timestamps on the overlay links can be taken into account: we can select peers with newer links, or in update we can prefer links that are newer
- these variations represent important differences w.r.t. fault tolerance and the quality of samples
  - the links at all nodes define a random-like overlay that can have different properties (degree distribution, clustering, diameter, etc)
  - turns out actually not really random, but still good for gossip

References

- Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In Hans-Arno Jacobsen, editor, Middleware 2004, volume 3231 of Lecture Notes in Computer Science, pages 79–98. Springer-Verlag, 2004. (journal version: ACM TOCS 2007 aug)
- Zhong, M., Shen, K., and Seiferas, J. 2005. Non-uniform random membership management in peer-to-peer networks. In Proc. of the IEEE INFOCOM. Miami, FL.

Gossip based topology management

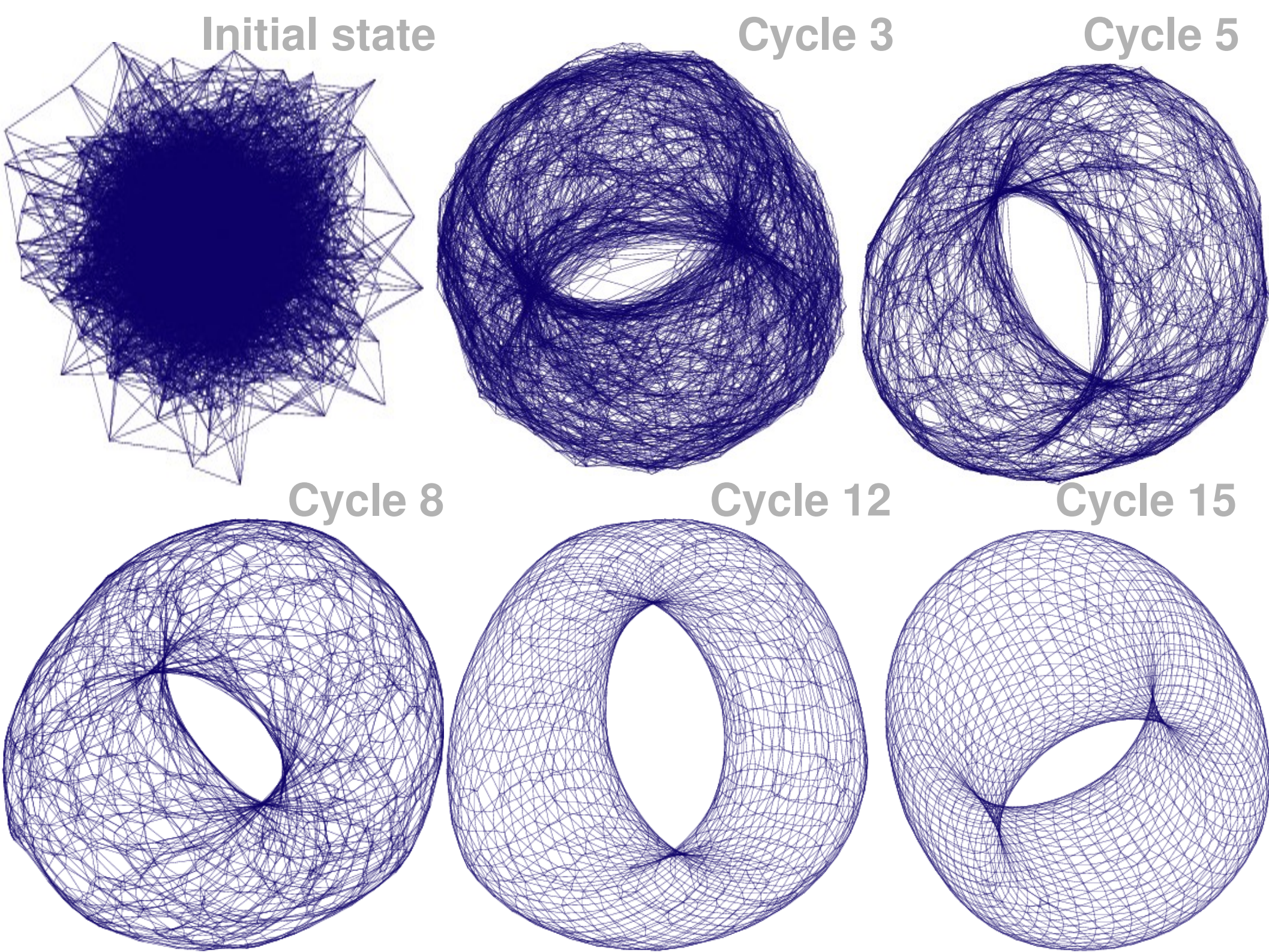
- We saw we can build random networks. Can we build any network with gossip?
- Yes, many examples
  - proximity networks
  - DHT-s (Bamboo DHT: maintains Pastry structure with gossip inspired protocols)
  - semantic proximity networks
  - etc

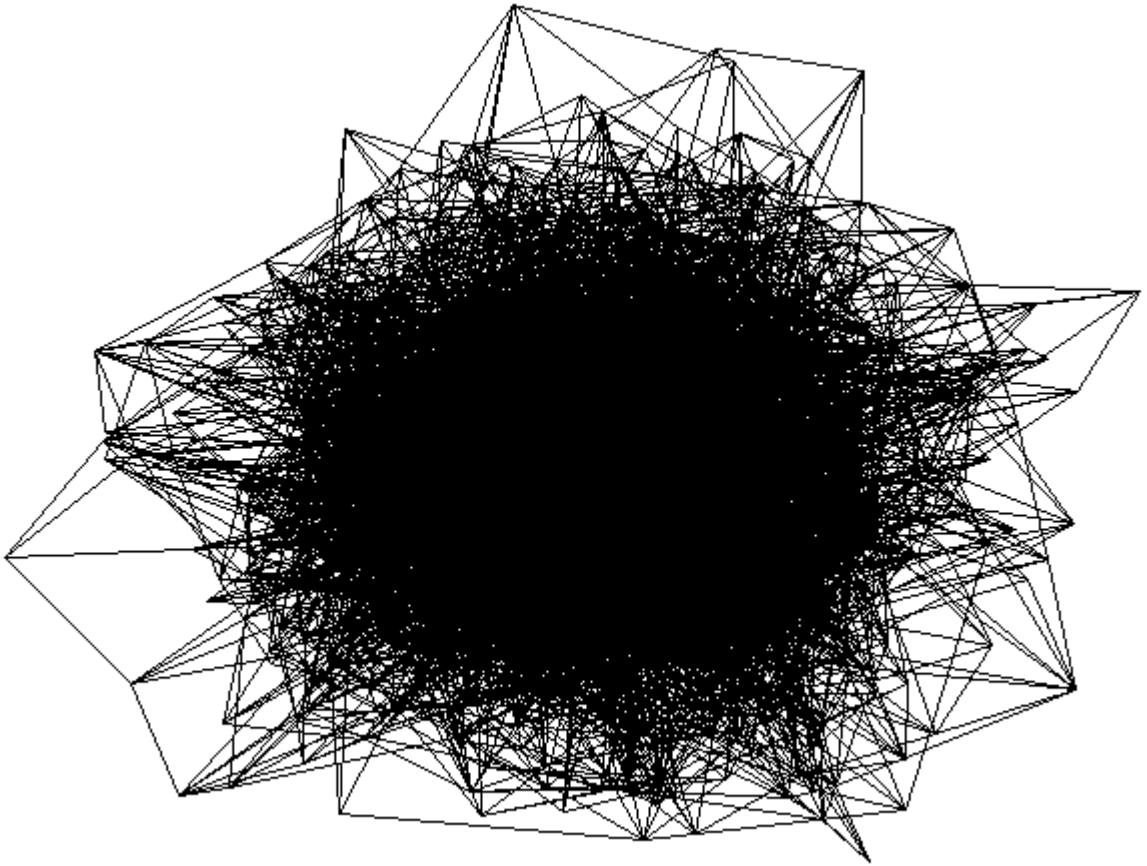
T-Man

- T-MAN is a protocol that captures many of these in a common framework, with the help of the ranking method:
  - ranking is able to order any set of nodes according to their desirability to be a neighbor of some given node
  - for example, based on hop count in a target structure (ring, tree, etc)
  - or based on more complicated criteria not expressible by any distance measure

Gossip based topology management

- basic idea: random peer samples are provided by a gossip algorithm: the peer sampling service
- The peer sampling service uses itself as peer sampling service (bootstrapping)
  - no need for fixed (external) network
- state: a set of overlay links to peers
- selectPeer: select the peer from the known set of peers that ranks highest according to the ranking method
- update: keep those links that point to nodes that rank highest


Initial state
Cycle 3
Cycle 5
Cycle 8
Cycle 12
Cycle 15



References

- Márk Jelasity and Ozalp Babaoglu. T-Man: Gossip-based overlay topology management. In Sven A. Brueckner, Giovanna Di Marzo Serugendo, David Hales, and Franco Zambonelli, editors, Engineering Self-Organising Systems: Third International Workshop (ESOA 2005), Revised Selected Papers, volume 3910 of Lecture Notes in Computer Science, pages 1–15. Springer-Verlag, 2006.
- Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. Proceedings of the USENIX Annual Technical Conference, June 2004.
- Laurent Massoulie, Anne-Marie Kermarrec, and Ayalvadi J. Ganesh. Network awareness and failure resilience in self-organising overlay networks. In Proceedings of the 22nd Symposium on Reliable Distributed Systems (SRDS 2003), pages 47–55, Florence, Italy, 2003.
- Spyros Voulgaris and Maarten van Steen. Epidemic-style management of semantic overlays for content-based searching. In Jose C. Cunha and Pedro D.Medeiros, editors, Proceedings of Euro-Par, number 3648 in Lecture Notes in Computer Science, pages 1143–1152. Springer, 2005.

Aggregation

- Calculate a global function over distributed data
  - eg average, but more complex examples include variance, network size, model fitting, etc
- usual structured/unstructured approaches exist
  - structured: create an overlay (eg a tree) and use that to calculate the function hierarchically
  - unstructured: design a stochastic iteration algorithm that converges to what you want (gossip)
- we look at gossip here

Implementation of aggregation

- state: current approximation of the average
  - initially the local value held by the node
- selectPeer: a random peer (based on peer sampling service)
- updateState(s_1,s_2)
  - $(s_1+s_2)/2$: result in averaging
  - $(s_1 s_2)^{1/2}$: results in geometric mean
  - $\max(s_1,s_2)$: results in maximum, etc

Illustration of averaging

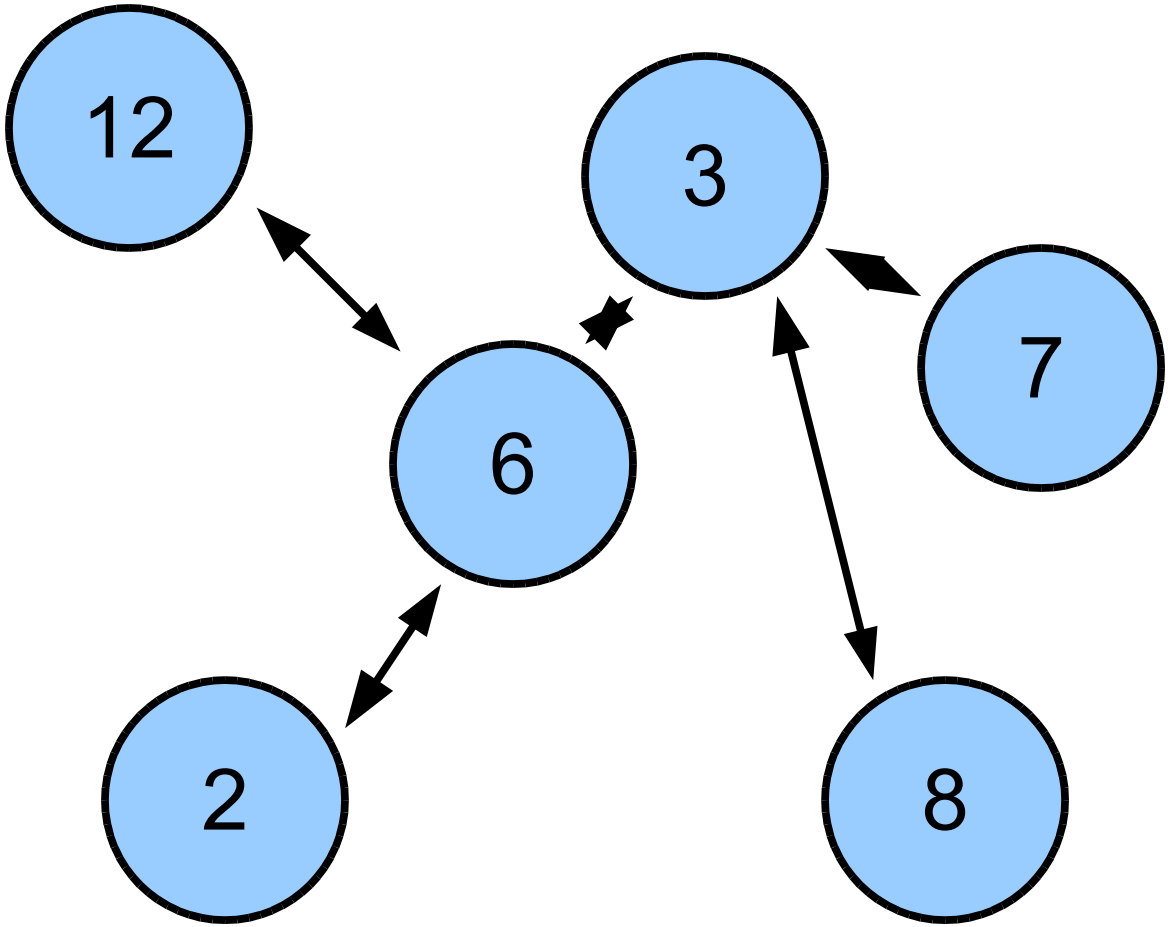


Illustration of averaging

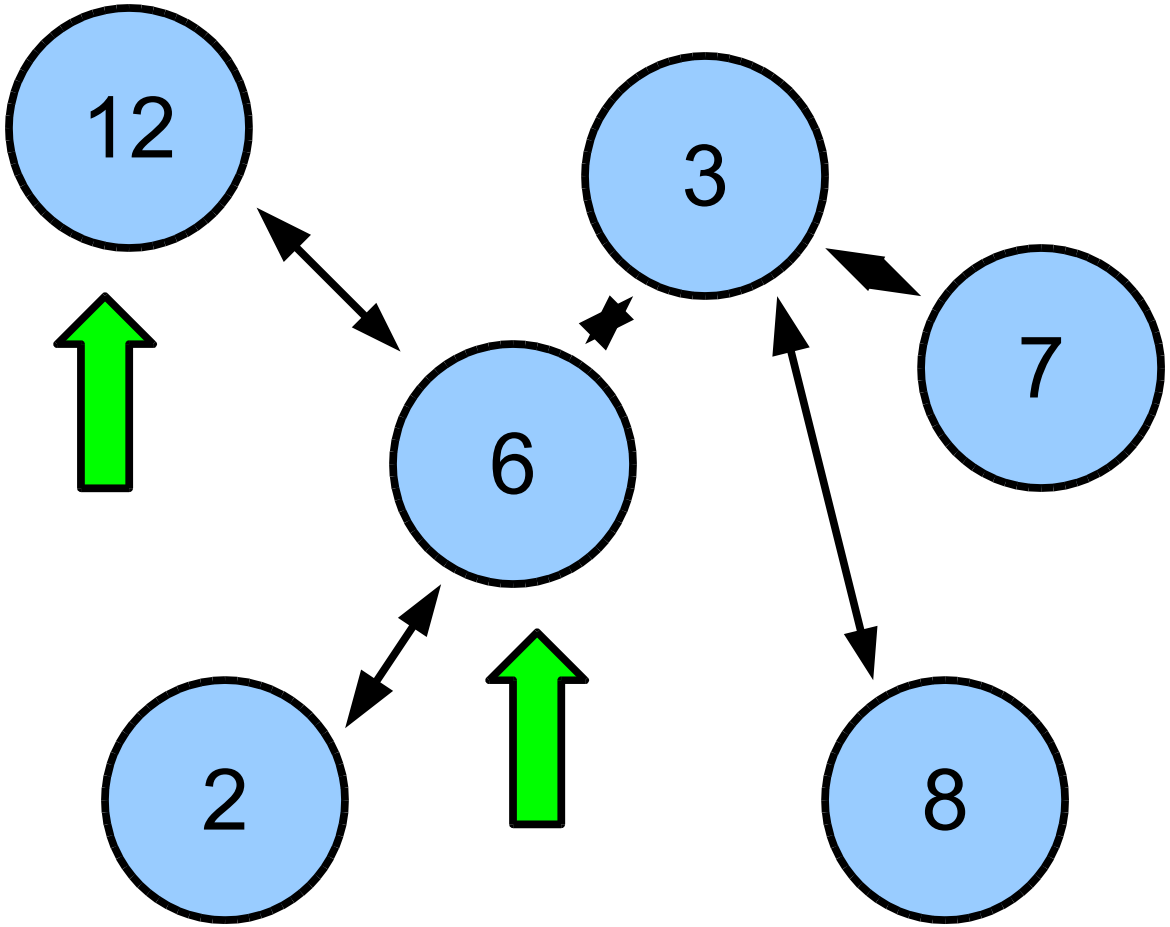
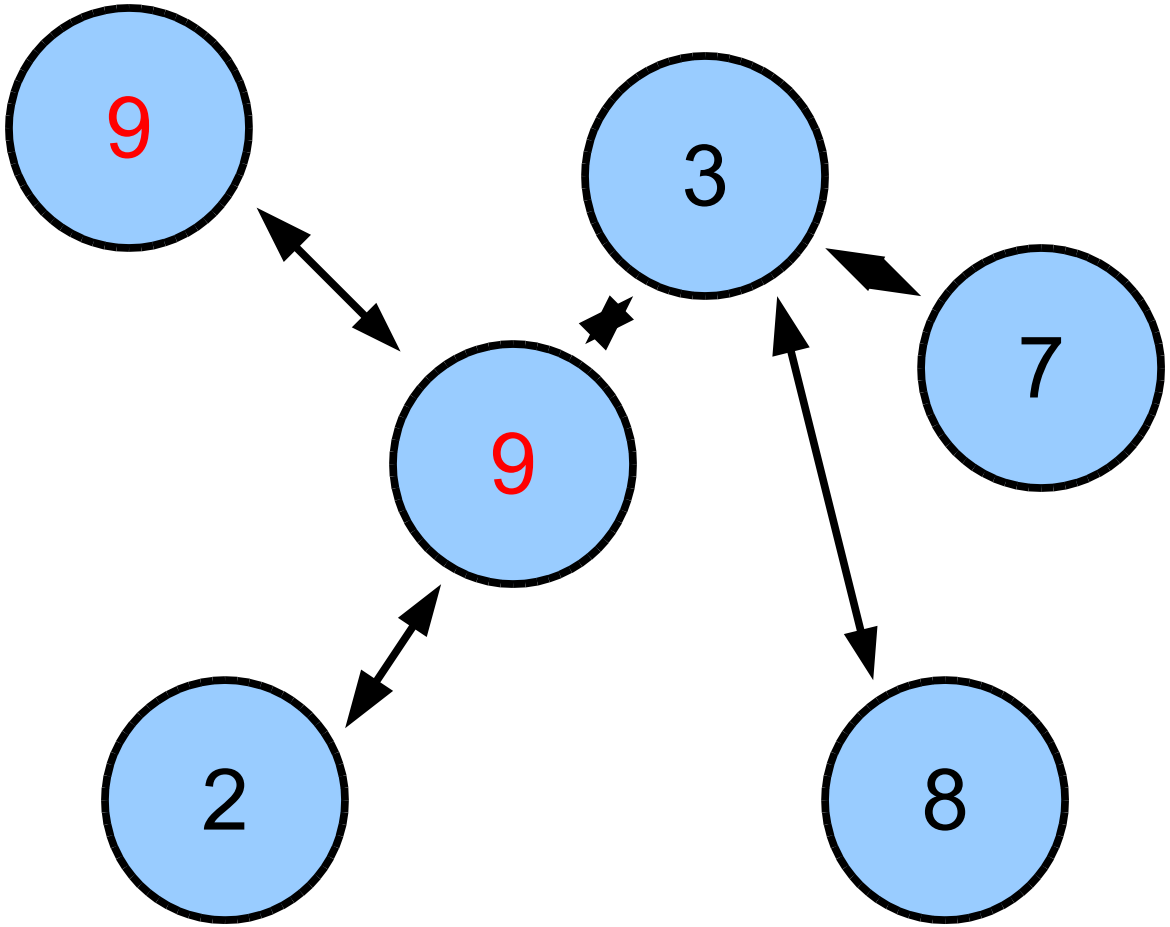


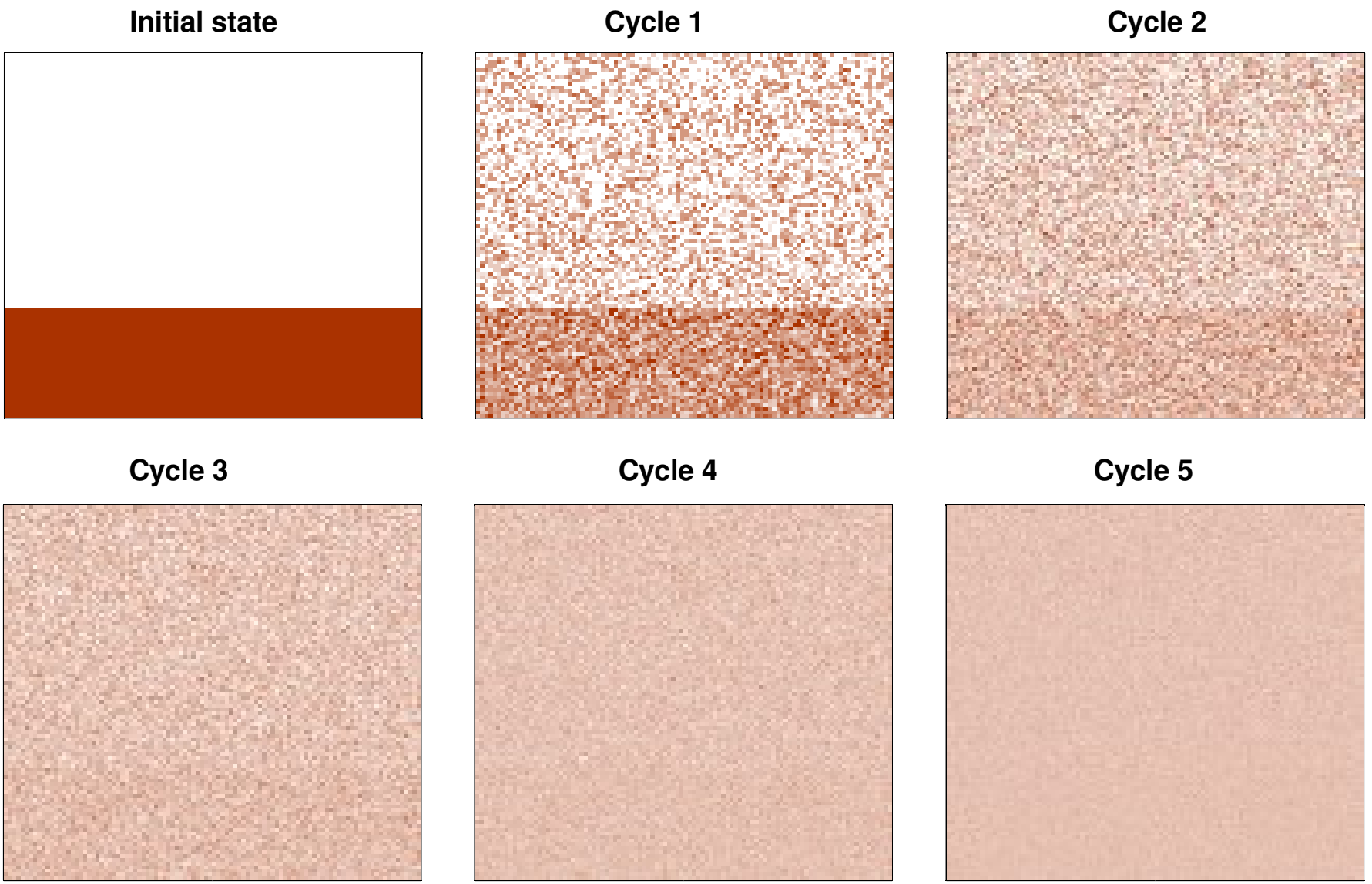
Illustration of averaging



Improvements

- Tolerates asymmetric message loss (only push or pull) badly
- Tolerates overlaps in pairwise exchanges badly
- [Kempe et al 2003] propose a slightly different version
  - all nodes maintain s (sum estimate) and w (weight)
  - estimate is s/w
  - only push: send (s/2,w/2), and keep s=s/2, w=w/2
- several other variations exist

Illustration of averaging



References

- Kempe, D., Dobra, A., and Gehrke, J. 2003. Gossip-based computation of aggregate information. In Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS'03). IEEE Computer Society, 482–491.
- Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. ACM Transactions on Computer Systems, 23(3):219–252, August 2005.
- Robbert van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Transactions on Computer Systems, 21(2):164–206, May 2003.

Outlook

- Gossip is similar to many other fields of research that also have some of the following features:
  - periodic, local, probabilistic, symmetric
- examples include
  - swarm systems, cellular automata, parallel asynchronous numeric iterations, self-stabilizing protocols, etc

This tutorial is available from

http://www.inf.u-szeged.hu/~jelasity/talks/saso07tutorial.pdf

Backup Slides

- details on [Demers et al, 1989]
  - propagation speeds
  - rumor mongering
  - spatial gossip
- Astrolabe

Epidemic Database Updates

- Problem
  - Xerox corporate Internet, replicated databases
  - Each database has a set of keys that have values (along with a time stamp)
  - Goal: all databases are the same, in the face of key updates, removals and additions
  - Updates are made locally and have to be replicated at all sites (300 sites)
- Solution in 1986
  - Anti-entropy and remailing
  - Didn’t work due to huge amount of traffic

Anti-Entropy

- basic idea: pairwise exchange of new updates
- state: the local database
- selectPeer: select a random peer
- update: resolve differences between the two databases
- some theoretical notes
  - easy to see that eventually all databases get all updates
  - expected time to achieve that is logarithmic (pushpull is fastest)

End-phase convergence of anti-entropy

- Pull
  - p_i is the proportion of not infected nodes in cycle i

$$p_{i+1} = p_i^2$$

- Push (slower in the end phase)

$$p_{i+1} = p_i \left(1 - \frac{1}{N}\right)^{N(1-p_i)} \approx p_i e^{-1}$$

Rumor spreading

$$\frac{ds}{dt} = -si$$

$$\frac{di}{dt} = si - \frac{1}{k}(1-s)i$$

$$\rightarrow s = e^{-(k+1)(1-s)}$$

- Rumor spreading
  - Push gossiping, but stop spreading info with probability 1/k if unsuccessful infection attempt (become removed)
  - s: susceptible, i: infective, r: removed
- Eg if k=1, 20% miss the gossip, if k=2, 6% miss it

Some other rumor mongering algorithms

- Some modifications
  - Blind vs feedback: blind is removed with pr. 1/k irrespective of success
  - Counter vs random: counter counts k unsuccessful attempts, random is removed with 1/k probability after each unsuccessful attempt
  - Push vs pull
    - **Push: always $s=e^{-m}$ where s is residue and m is avg number of messages sent by a node (Nm messages are sent altogether, to random targets)**
    - **Pull: better residue, but generates traffic even when there are no updates**

Some empirical results (1000 nodes)

Counter k	Residue s	Traffic m	Convergence t_{ave}	t_{last}
1	0.176	1.74	11.0	16.8
2	0.037	3.30	12.1	16.9
3	0.011	4.53	12.5	17.4
4	0.0036	5.64	12.7	17.5
5	0.0012	6.68	12.8	17.7

Feedback+ Counter+ push

Counter k	Residue s	Traffic m	Convergence t_{ave}	t_{last}
1	0.960	0.04	19	38
2	0.205	1.59	17	33
3	0.060	2.82	15	32
4	0.021	3.91	14.1	32
5	0.008	4.95	13.8	32

Blind+ Random+ push

Counter k	Residue s	Traffic m	Convergence t_{ave}	t_{last}
1	3.1×10^{-2}	2.70	9.97	17.63
2	5.8×10^{-4}	4.49	10.07	15.39
3	4.0×10^{-6}	6.09	10.08	14.00

Feedback+ Counter+ pull

Combining anti-entropy and rumor mongering

- Rumor mongering is used to spread updates
- Anti-entropy is run infrequently to make sure all updates are spread with pr. 1
- When anti-entropy finds an undelivered update: redistribution
  - Redistribution is done via rumor mongering
- [Originally, both primary spreading and redistribution was by email, but costs are prohibitive]

Spacial Gossip

- So far: random contacts
  - This is not good for underlying network traffic
  - Need to take proximity into account
- Spacial gossip: getPeer is biased according to distance of the peer: selecting node i is proportional to d^{-a} where d is the distance of i
- If underlying topology is linear, then expected traffic per link:

$$T(n) = \begin{cases} O(n), & a < 0; \\ O(n/\log n), & a = 1: \\ O(n^{2-a}). & 1 < a < 2; \\ O(\log n). & a = 2; \\ O(1), & a > 2 \end{cases}$$

Spatial Gossip

- a=2 is the best
  - Best tradeoff between speed and traffic
  - Probability is proportional to $1/d^2$
- Generalize to non-linear case
  - Q(d): cumulative number of sites at most at distance d
  - Probability proportional to $1/Q(d)^2$
- Smoothing out pathological topologies
  - Order all sites according to distance
  - Treat it as a linear structure

Astrolabe (middleware)

- Organizes hosts into a domain hierarchy (like DNS)
- Provides online monitoring service based on aggregation; a sort of data mining
- Fully decentralized through gossip
- Allows online configuration of monitoring capabilities (new things to observe, etc)
- Provides an API to applications
- Actually implemented
  - Security, firewalls, etc taken care of

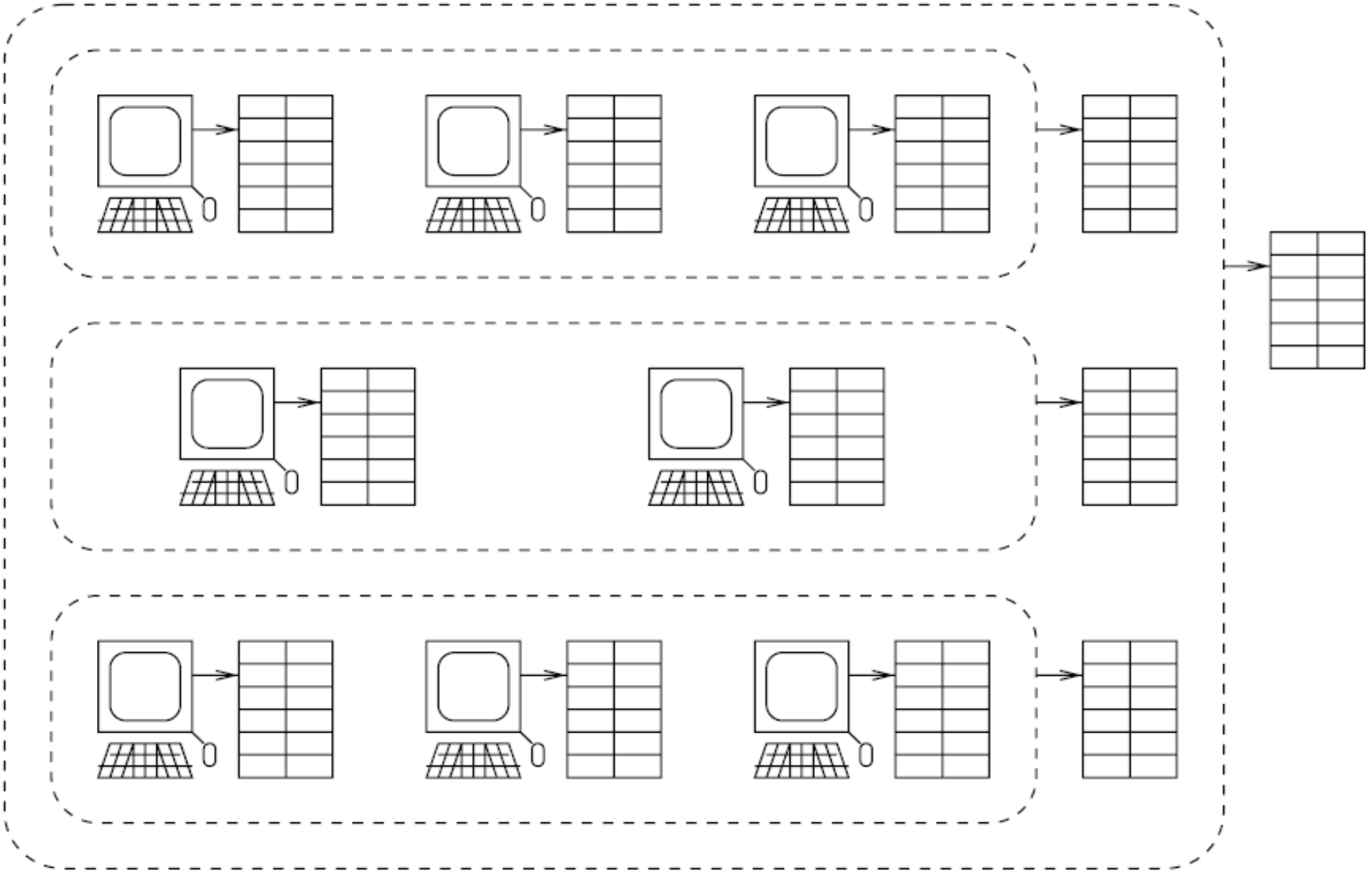
Analogy with DNS

- DNS
  - Directory service based on hierarchical domains
  - Lately more functionality
    - **Round robin DNS, server records, etc**
  - Updates are slow, and vulnerable
- Astrolabe also hierarchical but
  - More efficient
  - More robust
  - More generic
    - **arbitrary info about a domain**
    - **Collected online real time, in a configurable way**

Aggregation as Key Abstraction

- Aggregation is summarizing info
  - Over the entire system or within domains
  - It is of small size (not listing, only summary (O(1)))
- For example
  - Average, maximum, count, etc of some values
- Info is stored in (small) databases: MIBs
  - Management information base
- Aggregation is expressed by a simplified SQL language
- Aggregates are proactively updated at each level

Schematic view of Astrolabe



Astrolabe API

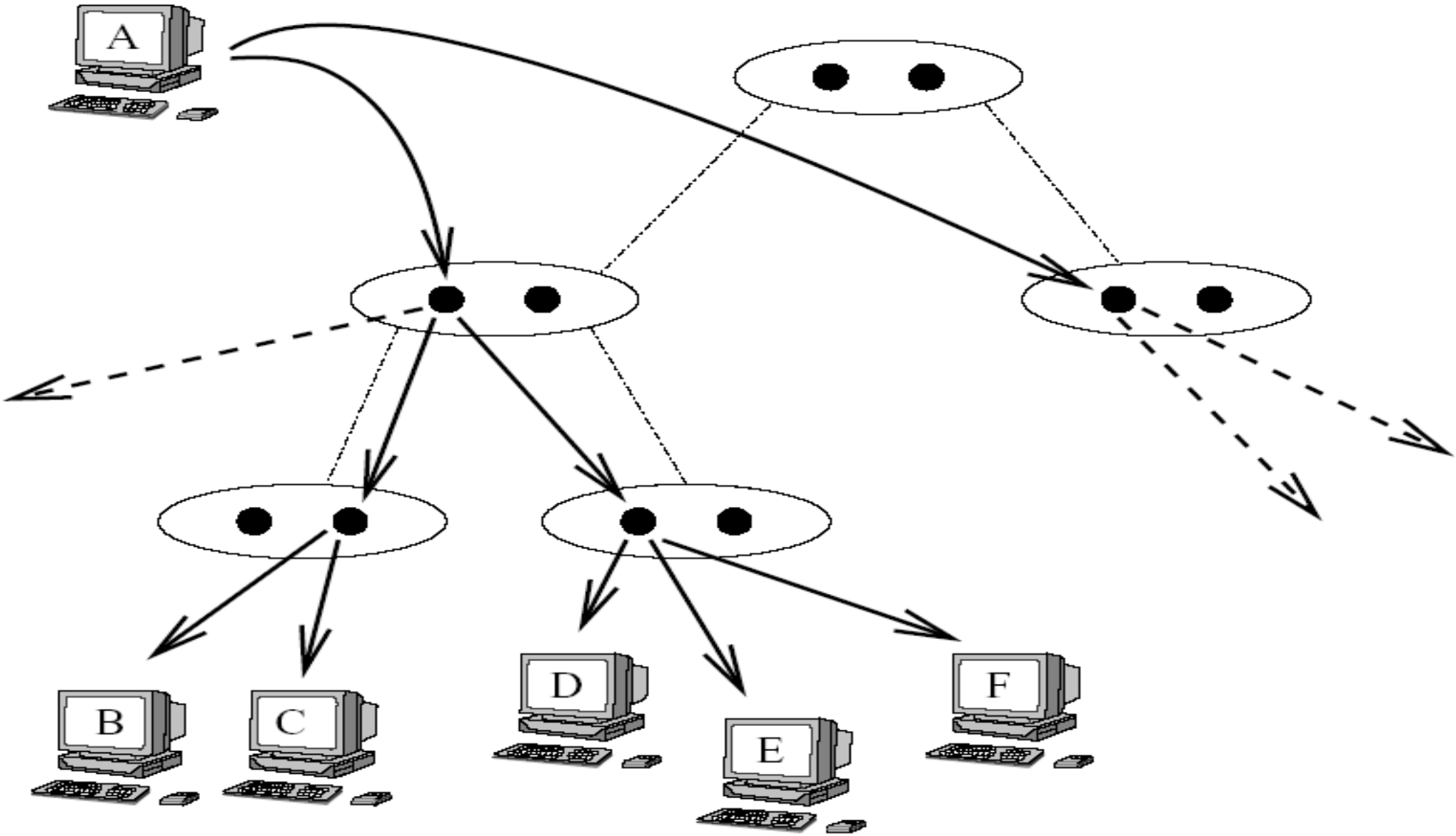
Method	Description
find_contacts(time, scope)	search for Astrolabe agents in the given *scope*
set_contacts(addresses)	specify addresses of initial agents to connect to
get_attributes(zone, event_queue)	report updates to attributes of *zone*
get_children(zone, event_queue)	report updates to zone membership
set_attribute(zone, attribute, value)	update the given attribute

- Can be accessed locally at an Astrolabe host or remotely through RPC
- scope: well defined subset of the tree
- zone: subtree (or leaf)
- updates only on leaf (virtual child zone)

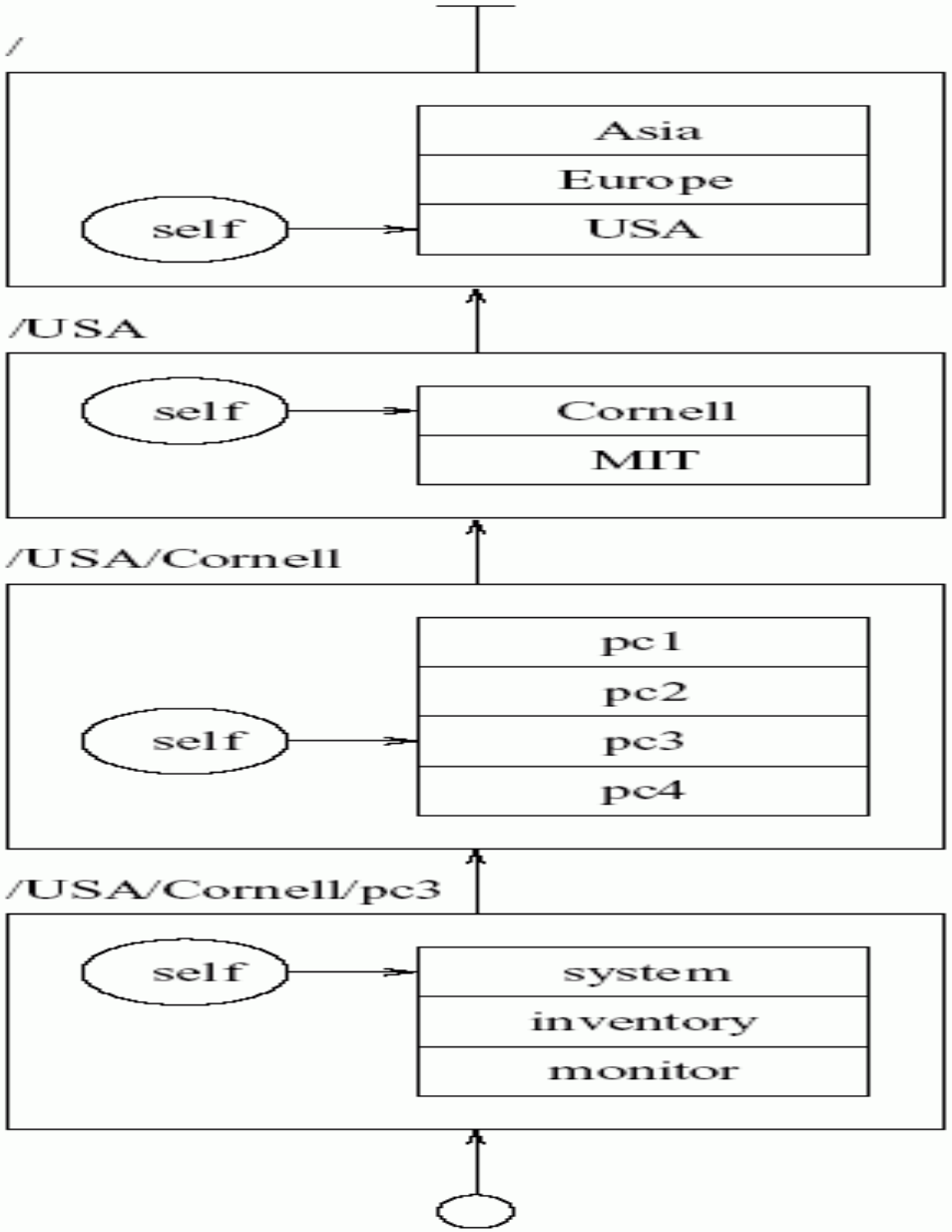
Example Applications

- P2P information diffusion: selectCast
  - Multicast to multicast groups
  - Each zone aggregates members of a group
    - **eg SELECT FIRST(2,game) AS game ORDER BY rate**
  - This way an overlay is superimposed that is used to multicast
  - Having two selected members at each zone allows for redundancy
- Note that the underlying Astrolabe infrastructure takes care of keeping all this up-to-date, scalable and robust

Schematic view of SelectCast



Implementation



- Each agent maintains a copy of the chain MIBs up to the root
- It also replicates the MIBs of all child zones of all the zones in this chain
- So zones are purely virtual and are replicated over all members

Compulsory Attributes

- ID
  - the local zone name within the parent zone
- Issued
  - the timestamp of last update of this MIB
- Contacts
  - Representatives for this zone (who will gossip)
- Nmembers
  - Number of members in the zone
- Servers
  - Small set of agents that implement the API

Gossip

- This set of MIBs is replicated (refreshed) through gossip
- For all zones separately
  - There is a gossip rate (cycle length)
  - Contacts for a zone pick a sibling zone at random
  - Initiate gossip with a contact of the selected zone
  - They run an anti-entropy step (regarding their own level and up)
- Note that most communication is done between sibling leaf nodes

Other issues

- Membership management
  - If a given zone's MIB is not refreshed for some time, it is removed
  - Joins are dealt with
    - **Setting a contact node explicitly**
    - **Or doing IP broadcast, etc**
- Communication
  - Issues with firewalls
    - **Application level gateways (ALGs), etc**
- Security
  - Through certificates
    - **Each zone has a certificate authority (CA)**

References

- Robbert van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Transactions on Computer Systems, 21(2):164–206, May 2003. (doi:10.1145/762483.762485)
- Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87), pages 1–12, Vancouver, British Columbia, Canada, August 1987. ACM Press.